COORDINATED LOCAL LEARNING ALGORITHMS FOR CONTINUOUSLY ADAPTIVE NEURAL SYSTEMS

A Dissertation in
Information Sciences & Technology
by
Alexander G. Ororbia II

© 2018 Alexander G. Ororbia II

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2018
The dissertation of Alexander G. Ororbia II was reviewed and approved* by the following:

C. Lee Giles  
David Reese Professor of Information Sciences & Technology  
Dissertation Co-Advisor, Co-Chair of Committee

David Reitter  
Associate Professor of Information Sciences & Technology  
Dissertation Co-Advisor, Co-Chair of Committee

Vasant Hanovar  
Professor & Edward Frymoyer Chair of Information Sciences & Technology

Daniel Kifer  
Associate Professor of Computer Science & Engineering  
External Committee Member

Mary Beth Rosson  
Associate Dean of Information Sciences & Technology

*Signatures are on file in the Graduate School.
Abstract

It is common statistical learning practice to build models, nowadays largely connectionist models, on very large, static, and fully annotated datasets of identically and independently distributed samples. However, the nature of this setup raises several questions that uncover the brittleness of the models fit to these datasets. What if the data contains few, if any, labels? Labels are generally difficult to come by and require intensive human labor if a dataset is to be fully and properly given ground truth. Alternatively, what if the data is sequential in nature and contains dependencies that span large gaps of time? In the task of modeling the characters of text, a model needs to learn how to spell words as well as how to arrange them in a coherent order in order to produce a meaningful sentence. Or finally, what if the distribution to be learned is dynamic and samples are presented over time, either drawn from a stream or a sequence of task datasets? A model must learn to predict the future well and yet retain previously acquired knowledge. In these settings, traditional machine learning approaches no longer directly apply. Motivated by this issue, we will ultimately address the issue of lifelong learning and the nature of systems that adapt themselves to such distributions.

The goal of this thesis is to propose a new family of algorithms that will enable connectionist systems to operate in the lifelong learning setting. In this dissertation, we will specifically: 1) propose a class of deep neural architectures, as well as their learning and inference procedures, that are capable of learning from data with few class labels, 2) develop a novel framework for incorporating simple and effective forms of long-term memory into recurrent neural networks so that they are better able to capture longer-term dependencies in sequential data, and 3) propose a new family of learning algorithms inspired and motivated by neuro-cognitive theory and develop an architecture that can learn without unfolding over time. Finally, we will investigate the challenging problem of sequential and cumulative learning. The work carried forth in this dissertation is meant to serve as a crucial stepping stone towards tackling the greatest challenge facing machine learning and artificial intelligence research efforts—developing an agent that can continually learn.
# Table of Contents

List of Figures ix

List of Tables xiii

Acknowledgments xv

Chapter 1

**Introduction**

1.1 Towards Agents That Learn Continuously 1
1.2 Contributions of this Thesis 4
1.3 Overview 5

Chapter 2

**Parallel Distributed Processing, Revisited** 8

2.1 The Parallel Distributed Processing Framework 9
2.2 Learning Problem Formulations 11
    2.2.1 Supervised Learning 11
    2.2.2 Unsupervised and Representation Learning 11
    2.2.3 Sequence Learning 13
2.3 Computing Updates to Parameters 14
    2.3.1 Back-propagation of Errors 14
        2.3.1.1 Unrolling: Back-propagation through Time 16
    2.3.2 Random Feedback Alignment 18
    2.3.3 Recirculation and Target Propagation 20
    2.3.4 Contrastive Divergence 23
    2.3.5 The Wake-Sleep Algorithm 24
    2.3.6 On Parameter Optimization 25
2.4 Theoretical Inspirations 26
    2.4.1 Predictive Coding 26
    2.4.2 Implicit Learning Theory 28
# Chapter 3

## Deep Hybrid Architectures

<table>
<thead>
<tr>
<th>3.1 Introduction</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Learning when Category Labels are Scarce</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1 The Semi-supervised Learning Problem</td>
<td>32</td>
</tr>
<tr>
<td>3.2.2 The Generative Statistical Approach</td>
<td>33</td>
</tr>
<tr>
<td>3.2.3 A Special Case: Unsupervised Pretraining</td>
<td>34</td>
</tr>
<tr>
<td>3.2.4 The Dual Discriminative-Generative Approach</td>
<td>35</td>
</tr>
<tr>
<td>3.3 Learning Greedily and Pseudo-Jointly</td>
<td>35</td>
</tr>
<tr>
<td>3.3.1 The Multi-Block Harmonium</td>
<td>36</td>
</tr>
<tr>
<td>3.3.1.1 Model Structure</td>
<td>36</td>
</tr>
<tr>
<td>3.3.1.2 Calculating Parameter Updates</td>
<td>37</td>
</tr>
<tr>
<td>3.3.2 The Stacked Boltzmann Expert Network</td>
<td>39</td>
</tr>
<tr>
<td>3.3.3 The Multi-Input Autoassociator</td>
<td>41</td>
</tr>
<tr>
<td>3.3.3.1 Model Structure</td>
<td>41</td>
</tr>
<tr>
<td>3.3.3.2 Calculating Parameter Updates</td>
<td>42</td>
</tr>
<tr>
<td>3.3.4 The Hybrid Stacked Denoising Autoencoders Model</td>
<td>43</td>
</tr>
<tr>
<td>3.3.5 Ensembling of Layer-Wise Experts</td>
<td>44</td>
</tr>
<tr>
<td>3.3.6 Greedily Learning a Stack of Experts</td>
<td>45</td>
</tr>
<tr>
<td>3.3.7 The Bottom-Up Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>3.3.8 The Bottom-Up-Top-Down Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>3.3.8.1 Incorporating Top-Down Signals</td>
<td>45</td>
</tr>
<tr>
<td>3.3.8.2 Pseudo-Joint Learning</td>
<td>48</td>
</tr>
<tr>
<td>3.4 Learning via Coordinated Local Signals</td>
<td>49</td>
</tr>
<tr>
<td>3.4.1 The Deep Hybrid Boltzmann Machine</td>
<td>49</td>
</tr>
<tr>
<td>3.4.1.1 Model Structure</td>
<td>49</td>
</tr>
<tr>
<td>3.4.1.2 Heuristic Weight Doubling</td>
<td>51</td>
</tr>
<tr>
<td>3.4.1.3 Approximate Inference</td>
<td>51</td>
</tr>
<tr>
<td>3.4.1.4 Calculating Parameter Updates</td>
<td>53</td>
</tr>
<tr>
<td>3.4.2 The Deep Hybrid Denoising Autoencoder</td>
<td>55</td>
</tr>
<tr>
<td>3.4.2.1 Model Structure</td>
<td>55</td>
</tr>
<tr>
<td>3.4.2.2 Approximate Inference</td>
<td>55</td>
</tr>
<tr>
<td>3.4.2.3 Calculating Parameter Updates</td>
<td>56</td>
</tr>
<tr>
<td>3.4.3 The Hybrid Loss Function</td>
<td>56</td>
</tr>
<tr>
<td>3.5 The Multi-Level Semi-Supervised Prior</td>
<td>58</td>
</tr>
<tr>
<td>3.6 Experiments</td>
<td>58</td>
</tr>
<tr>
<td>3.6.1 Data</td>
<td>59</td>
</tr>
<tr>
<td>3.6.1.1 Image Classification Data</td>
<td>59</td>
</tr>
<tr>
<td>3.6.1.2 Document Categorization Data</td>
<td>59</td>
</tr>
<tr>
<td>3.6.2 Evaluation of Greedy &amp; Bottom-Up Approaches</td>
<td>60</td>
</tr>
</tbody>
</table>
4.6.2 The Multimodal Neural Architecture
4.7 Experiments: Multimodal Language Modeling
4.7.1 Model Analysis
4.7.2 Conditional Sampling
4.8 Other Applications of the Delta-RNN
4.8.1 Image Compression: Learned Iterative Decoding
4.8.2 Data Synthesis for User Privacy
4.9 Conclusions

Chapter 5
Reducing the Discrepancy between Local Representations
5.1 Introduction
5.2 Problems with Back-Propagation of Errors
5.3 Discrepancy Reduction: Credit Assignment through Local Error Signals
5.4 The Local Representation Alignment Algorithm
5.4.1 Notation
5.4.2 Setting the targets.
5.4.3 Divide and Conquer: The Computation Subgraph
5.4.4 Instantiation: The Multilayer Perceptron
5.4.4.1 Handling Non-Differentiable Activations
5.4.4.2 Overcoming Poor Initializations
5.4.4.3 Overcoming Exploding and Vanishing Gradients
5.5 Experiments: Properties of LRA
5.5.1 Effect of Manifold-Walking
5.5.2 Robustness to Initialization
5.5.3 Training from Null Initialization
5.5.4 Stochastic Networks
5.6 Reformulating LRA using Error-Units
5.6.1 Error-Driven LRA
5.6.2 Improving Difference Target Propagation
5.6.3 A Comment on the Efficiency of LRA-E and DTP
5.7 Experiments: Evaluation of LRA-E and DTP-σ
5.7.1 Classification Performance
5.8 On Unfolding
5.9 Conclusions

Chapter 6
Adaptive Learning by Aligning Representations
6.1 Introduction

vii
List of Figures

2.1 Back-propagation of errors applied to a three-layer MLP. The blue lines indicate the reverse propagation calculation. Note that $z^0 = x$, the input datum, for notational consistency. The blue dash-dotted lines mark the global feedback pathway that back-propagation of errors depends upon to find updates to parameters in the lower-level of the model. ........ 16

2.2 Back-propagation through time applied to a simple Elman RNN. Unrolling occurs over three time steps and the instantaneous task is to predict the data point at time $t = 4$. ................. 18

2.3 Random feedback alignment (RFA) and direct feedback alignment (DFA) applied to a three-layer MLP. The blue lines indicate the reverse propagation calculation. Note that $z^0 = x$, the input datum. Since explicit weight matrices are used to transmit the error signals down to each layer, we now replace the original symbol for identifying the partial derivative of the global loss with respect to each layer pre-activation, $\frac{\partial L^3(\cdot)}{\partial h^3}$, with a symbol to represent displacement instead, since the transmitted error is not a proper derivative. .................. 19

3.1 Architecture of the SBEN model. The flow of data through the system is indicated by the numbered arrows. Given a sample $x$, the dash-dotted arrow indicates obtaining an estimated label by using the current layer’s conditional via Equation 3.11 (i.e., Step # 1, dash-dotted red arrow). The latent representation is computed using this proxy label and the data vector via Equation 3.8 (i.e., Step # 2, dashed green arrow). This procedure is repeated recursively, replacing $x$ with $h_n$. ................. 40

3.2 A 3-layer deep hybrid Boltzmann machine architecture. Note the bi-directional nature of the weights, meaning most sets of variables require both a bottom-up & top-down calculation. .................. 50
3.3 The DHBM and DHDA architectures compared side-by-side with their layer-wise predecessors, the SBEN and the HSDA. The flow of information to gather layer-wise statistics is indicated by the arrows (which represent an operation, such as matrix-multiplication followed by an element-wise non-linearity), with the appropriate layer parameter matrix and the vector at the arrow’s origin), which are numbered according to the sequential computational steps taken to calculate them. Arrows (or operations) in the same computational step are numbered the same and point to resulting activation values to be calculated. $v$ corresponds to the recognition network’s ($Q_{rec}$) initial guess for the mean-field while $\mu$ represents the actual mean-field statistic. Horizontal dotted arrows indicate a copy operation.

3.4 Online error (y-axis) of 3-SBEN, 3-HSDA, & HRBM (or 1-SBEN) evaluated every 100 time steps (x-axis). Each curve reported is a 4-trial mean of the lowest validation error model.

3.5 Mean CV generalization performance as a function of labeled sample subset size (using 200 features).

3.6 Online learning on the Waveform (top) and LED streams (bottom panels). Left panels zoom in to the first 50,000 iterations. 10% labeled data (sampled uniformly 0-20% in each trial).

4.1 The Delta-RNN computation graph, unfolded over time. The learnable gates $\gamma$ and $\beta$ control how much influence the previous state and the currently computed data-dependent state have on computing the model’s next hidden state.

4.2 L1 norm of deltas between consecutive states of model trained on Penn Treebank plotted over words of example sentences. A simple polynomial trend-line (dashed red) was fit to the bar heights in order to illustrate the informative “bumps” of each sample sentence. The main observation is that the norm is, in general, lower for low-information content words, such as the article “the”, and higher for informative words, such as “government”.

4.3 The multimodal $\Delta$-RNN, unrolled over time. The gray-dashed connections represent the identity connections that carry over the slow-moving state while the dash-dotted black lines represent the next-step predictions made by the model. Solid black lines correspond to synaptic weight matrices (labeled accordingly).

4.4 Comparison of learning curves for the $\Delta$-RNNs in each language (English, German, Spanish).

57 64 68 70 81 96 102 106
5.1 How the LRA algorithm is applied to a multilayer perceptron (MLP). A single computation subgraph is selected in the red dashed box. Note how \textit{LRA-fdbk} can easily handle non-differentiable functions, $f(\cdot)_d$, since these can be viewed as being housed within the computation subgraph. The short-circuit error feedback connection simply wires directly to the input pre-activation variables. .................................................. 117

5.2 Drop in test-set error during first few epochs of training tanh networks on the MNIST and Fashion MNIST image datasets. ................................. 127

5.3 Third-level filters acquired after 1 epoch under each learning algorithm: (a) backprop, (b) LRA-diff, (c) LRA-fdbk. ................................. 128

5.4 Generalization of learning algorithms used to train deep and thin networks of either sigmoid (top) or tanh (bottom) units on MNIST (left) and Fashion MNIST (right). Initial weights were sampled: $w_{i,j} \sim \mathcal{N}(\mu = 0, \sigma^2 = 0.025)$. Note that DTP, FA, and BP often fail or are unstable with this initialization when training deep sigmoidal networks on Fashion MNIST. ................................................................. 130

5.5 Average depth of the credit assignment carried out by LRA on w.r.t. deep sigmoidal and tanh networks, compared against backprop’s fixed depth. 133

5.6 t-SNE plots of the various latent spaces acquired by the sparse rectifier network trained with backprop (top) or LRA-fdbk (bottom) on Fashion MNIST. ................................................................. 137

5.7 In (a), we compare the angle between the updates calculated by LRA-E and backprop. In (b), we show how the total discrepancy, as measured in an LRA-trained MLP, evolves during training, alongside the output loss. 143

5.8 The two main phases of Local Representation Alignment for a given $(y, x)$. ................................................................. 144

5.9 Visualization of the topmost hidden layer extracted from an MLP trained by Direct Feedback Alignment (DFA), Equilibrium Propagation (Equil-Prop), adaptive noise Difference Target Propagation (DTP-$\sigma$), and error-driven Local Representation Alignment (LRA-E). ................................. 149

5.10 Validation accuracy of \textit{DTP} vs improved \textit{DTP-$\sigma$} as a function of epoch. 150

6.1 Error-driven computation where one pair of cells in layer 1 communicate the discrepancy backwards to an earlier cell in layer 2 to correct its representation. Solid lines represent the initial, feedforward pass of information while dashed lines represent the information flow along the structural feedback connections, which feed into an error unit that computes the corrected state for $z^2_1$. ................................................................. 158
6.2 A two-layer TNCN continually correcting itself as it processes visual sensory stimuli over time. Dash-dotted lines represent a state correction and dotted lines indicate a span of time. ........................................ 159

6.3 Model performance evaluated on the streaming version of the bouncing ball problem. Depicted, for all of the models in terms of prequential squared error, is (a) the next-step prediction, and (b) the prediction horizon, or 5 steps of closed-loop prediction. ........................................ 171

6.4 Two variations of sequential task types. Note that shared components (in blue and black) and task-specific components (in red). ......................... 176

6.5 The complementary system defined under the memory consolidation—in this setup, a discriminative network, i.e., the network/graph to the left, serve as a task model and temporary memory while another, generative network, i.e., the network to the right, serves as a form of longer-term memory. The generative model actively refreshes the discriminative model while also trying to store statistical structure regarding the data generating process itself. The generative model will, in some sense, serve as the approximate “inversion” of the discriminative model. ......................... 183

6.6 Reconstructions, on a randomly sampled test example, tracked across each new learned task. At the very top of each column is the original sample. Each row shows the reconstruction of the selected example after learning a new task. ...................................................... 186

6.7 Test retention results on the four-task cumulative learning problem. . . 188
List of Tables

3.1 Generalization results on the CAPTCHA simulated dataset. Classification results reported as 10-trial averages with single standard deviation from the mean. .................................................. 62
3.2 Generalization results on the Stanford OCR dataset. Classification results reported as 10-trial averages with single standard deviation from the mean. 62
3.3 Text categorization results on the WEBKB dataset. ................................. 63
3.4 WEBKB categorization results on 1% of the training data labeled (8 examples per class), rest unlabeled (i.e., 5-fold means with standard error of the mean). ................................................. 69
3.5 20NewsGroup data-set categorization results on 1% of the training data labeled (8 examples per class), rest unlabeled (i.e., 5-fold means with standard error of the mean). ................................................. 69
3.6 Top-most words that the SBEN (BUTD) model associates with the 4 NewsGroup meta-topics. ................................................................. 69

4.1 Test-set results on the Penn Treebank word-level and character-level language modeling tasks. Note that “impl.” means implementation. 90
4.2 Test-set negative log likelihoods while holding number of parameters approximately constant. Subword modeling tasks on Penn Treebank and IMDB. ................................................................. 92
4.3 Word-level language modeling text perplexities on several language modeling datasets. ................................................................. 97
4.4 Generalization performance of language models trained and evaluated on linguistic data only (L), full: trained and evaluated on multimodal linguistic and visual data (LV), and, blind: trained on multimodal data (LV) but evaluated on language only (L). ...................... 107
4.5 Decoder analysis: Word query similarity test. ........................................... 107
4.6 Some captions generated by the Multimodal ∆-RNN in English. ........... 109
5.1 Training and generalization error (%) of deep sigmoid or tanh networks, trained with various learning algorithms, including backprop and our proposed LRA-fdbk. ........................................ 131
5.2 Generalization error (%) of various networks trained with LRA-fdbk, from null initialization. We also report, next to each “best error”, the end of which epoch the model reached this level of generalization. Note that, for the output of the internal subgraphs, LWTA-1 means tanh was used and LWTA-2 means hard-tanh was used. ......................... 136
5.3 Generalization error (%) of various stochastic binary networks. Errors are calculated using model posterior probabilities averaged over $M = 100$ samples. ................................................. 138
5.4 MNIST supervised classification results. .................................. 148
5.5 Fashion MNIST supervised classification results. ..................... 148
5.6 Effect of update rule on LRA when training a 3-layer MLP on MNIST. 148

6.1 Test performance on the bouncing ball generative modeling problem. To the left is model test squared error using original dataset originally proposed in [1]. To the right is the performance (also squared error) of models trained on the two-ball version of the problem and evaluated on different test-sets with different number of balls. ................. 167
6.2 Test performance on the Penn Treebank character language modeling problem. .................................................. 168
6.3 Generalization accuracy of a network trained on different classification subsets over time. This amounts to the task metric matrix $R$ in [2]. ... 185
6.4 Lifelong generalization metrics, based on task performance matrix $R$ [2], for the networks trained on the task set of Table 6.3. For all metrics, higher values are better. ............................................. 185
6.5 Test reconstruction error of a network trained on different classification subsets over time. This amounts to a reconstruction error matrix (each slot is the squared error). ..................................... 186
6.6 Cumulative learning generalization metrics, based on the task performance matrices in Figure 6.4. For all metrics, higher values are better. 188
Acknowledgments

Lee Giles has been a mentor and an advisor mentor to me these past five years. I especially thank him for bringing me on to be a part of his lab at Penn State, when I was only a young engineer who just wanted to work on complex adaptive systems. It was that first summer under him where I coded up my first Hopfield network and became fascinated by its limited memory capacity, so starting my crazy journey into the world of connectionism. I am grateful for all of the beers at Whiskers he bought me, his countless stories and bits of wisdom, the opportunity to work with him in creating and co-teaching some of the first deep learning courses and seminars to be offered at Penn State, and for him simply letting me find my own way as a researcher. He gave me the opportunity to deeply pursue the things in neural networks that captivated me the most, starting with Boltzmann machines, with little worry over how I was to be funded. For that, I am eternally indebted to him.

David Reitter has also been my advisor throughout my time at Penn State. In addition to his advice and guidance, I thank him for putting up with my stubbornness and for consistently pushing me to build my “pipeline”. I learned a great deal about cognition and language through him, and I am grateful to him for looking out for me and also funding my way when I needed it. Whether it was a long car chat about research agenda setting on the way to SBP-BRIMS in DC, editing my papers and explaining how I can improve my language/message delivery, or just watching me sketch out my crazy computation graphs all over his office whiteboard (right next to the upcoming paper deadlines!), he’s given me valuable encouragement and support.

I would like to thank my other committee members, Daniel Kifer and Vasant Honavar, as well. I am grateful to Vasant for challenging me to not forget my statistical learning theory and to deeply consider the type of researcher I want to be and the research community I want to contribute to. As for Daniel, collaborating with him proved to be one of my best learning experiences at Penn State. His focus on and excitement over the mathematical underpinnings and workings behind a proposed idea, whether it was related to a regularizer or a weird learning algorithm quirk, are the kind of qualities I want to bring into the work I do with students in the future.
I gratefully acknowledge the funding received towards my Ph.D. from the National Science Foundation and the Alfred P. Sloan Foundation. Furthermore, I am grateful for the Bunton Waller Scholarship, the Jordan Rednor Fellowship, and the NSF-funded Integrative Graduate Education and Research Training (IGERT) Big Data Social Science Ph.D. Training Program. The findings and conclusions of this dissertation do not necessarily reflect the view of these funding agencies and programs.

Burt Monroe gets extra thanks for all of his patience with me as I crazily ran around semester after semester pestering him about research results. I also thank him for bringing me aboard to IGERT. I additionally thank Dr. David Miller for his advice and comments throughout our collaborations, his prowess and mathematical rigour in pattern recognition is inspiring.

A special thanks goes much farther back in time to Dr. Steve Guattery, my undergraduate mentor at Bucknell University. His early and enthusiastic encouragement to pursue graduate studies, in a time when not much other encouragement was being given to me, was especially important in affirming my decision to continue on in academia. I will bring the same understanding and encouragement to the next generation of students as a mentor and guide myself.

I am also grateful to Dr. Andrew McCallum, as it was a pleasure to work with him and his lab, including Ari Kobren, Emma Strubell, Patrick Verga, Benjamin Roth, and Nicolas Monath. I thank David Belanger for getting me to look at BIDMat—this Scala linear algebra would become the backbone of the statistical learning library that backs this very dissertation.

I am indebted to the Interactions LLC, New Jersey team for making my summer internship an excellent experience, which was full of comradery and passion. I especially thank Patrick Haffner, Ryan Price, and Srinivas Bangalore for their mentorship and, of course, imparting to me the many tricks-of-the-trade of building large-scale, neural-based speech recognition systems.

From Université de Montréal, I express my gratitude to Yoshua Bengio for many useful discussions over the years, whether chatting in person or him readily responding to my emails, and Joelle Pineau and Aaron Courville for their advice and encouragement as Iulian Serban and I worked to get our piece-wise latent variable neural model published. I learned a lot from my collaboration with my friend Iulian. I would say that those late nights at the whiteboard, working through the partial derivatives (pesky Dirac delta functions included!), were well spent.

I would like to express my gratitude to Tomas Mikolov for the many conversations we had regarding recurrent networks and for the time he took to work and collaborate with me. Furthermore, I pay thanks to Dr. Hugo Larochelle, whose fast and informative responses to my wave of emails, full of useful advice from a deep learning expert, proved immensely helpful.

Many thanks goes out to the members of the Intelligent Systems Laboratory/CiteSeer
group, including Jian Wu, Kyle Williams, Madian Khabsa, Sagnik Ray Choudhury, and Hung-Hsuan Chen. I thank them for showing me the ropes. I am further grateful that I got the chance to work with Ankur Mali. My only wish is that I had been able to collaborate with him sooner in my life—if we thought of it, we engineered it. Additional thanks goes to the members of the Applied Cognitive Science, including Christopher Dancy II, Jeremy Cole, and Yang Xu, for very useful discussions.

I also want to acknowledge Eric Obeysekare, Jamie Reep, Kahina Ghanem, Bill Aurite, and Jake Weidman. Each of you, in different ways, helped me to get through the tough times during the long arduous ride that is graduate school. I am grateful to each of you.

And Jake, we’ve come a long way since that summer when we were just starting our research here at Penn State, many years ago. We have shared countless good times and helped each other push through when we didn’t think we would make it. Though there were many doubts along the way, now I can finally say to you: We made it, man. We made it.
Dedication

To my father, Alexander G. Ororbia, my mother, Cheryl M. Ororbia, and my brother, Maximilian E. Ororbia. If it were not for you guys listening to me, supporting me, and encouraging me during all of the highs and lows of the research process, I do not think I would have made it this far.

And, father, I appreciate the many edits and all of the feedback that you gave me as I worked to bring this dissertation to life. More importantly, I am grateful that you got to witness this Golden Braid be woven, from beginning to end.
Chapter 1  |
Introduction

1.1 Towards Agents That Learn Continuously

Designing a machine that can learn across tasks is an important and challenging open problem for machine learning and Artificial Intelligence (AI). However, statistical learning programs today perform well on very constrained, narrowly-defined tasks yet struggle and fail in non-stationary, one-shot [3], and zero-shot [4] learning environments. Even with respect to a single, isolated task, in order to generalize well, a model, even those based on artificial neural networks, requires a tremendous amount of data, particularly labeled data.

Humans, in contrast, do not learn in this isolated, narrow way. Instead, a human agent retains what it has learned in the past and uses whatever prior information it has acquired to solve future problems. When we encounter a novel situation or problem, we are able to quickly discern what parts are not new, i.e., what we have seen in old situations/contexts. This is only possible because, in humans, knowledge is accumulated, reused, and adapted in tackling new problems, dramatically reducing the number of samples needed to learn new concepts or skills.

It is also important to observe that humans learn largely without much supervision. This is for good reason—it would be impossible to create correct labels for every possible task or problem that a learner would have to solve in his/her lifetime. This is further compounded by the fact that real-world distributions can often be perceived as changing or evolving, quickly making any old labels irrelevant. However, the more a human agent learns, the more knowledgeable and more effective at learning she/he becomes. In short, real-world learning involves exploiting the relationships across tasks and situations due
to the fact that objects in nature are related and interconnected. It is highly unlikely that something is completely unrelated to anything else.

Therefore, the essence of lifelong, never-ending, or continual machine learning is about mimicking this learning process in humans.

Take, for example, the problem domain of natural language. Here, it is easy to observe the role that continual learning plays. Words and phrases largely have the same meaning across domains and linguistic tasks, governed by the grammatical rules and syntactic structure, e.g., a sharing of expressions, meanings, and syntax. In short, we do not need to re-learn a language, such as English, every time a new language problem is encountered, e.g., in studying a new subject such as Artificial Intelligence, we do not need to first re-learn English—all that is necessary is understanding the new concepts and words specific to Artificial Intelligence itself since English itself still functions the same even in the presence of a new topic. In addition, nearly all problems in language are related and affect each other in some fashion. This ensures that learning can occur across tasks. For example, in one task, the learner might uncover that a computer is a product/entity, and in another task, the learner might identify that a product object has a price and that the adjective “expensive” describes the price attribute of a product. These are two different bits of knowledge, but when combined, allow the learner to properly parse the following sentence: “the operating speed of the computer is very fast, but it is very expensive”. The learner can identify that “it” refers to the computer and not “operating speed” and can further extract the idea that “operating speed” pertains to an attribute of the computer itself. A bit of knowledge can be anything that is now known but was once unknown, e.g., the learned parameters of a next-step predictive language model count as a form of knowledge. Furthermore, knowledge is diverse, and therefore, for a lifelong learning agent to be effective at adapting to new tasks, a large quantity of knowledge is required. This means that, over time, the agent needs to learn from a large number of tasks across a wide variety of domains in order to accumulate diverse knowledge. Since any single one previous task might only contain a small relevant bit of knowledge that is relevant to a new task (in the worst case, nothing about the old task informs solving the new problem), having knowledge constructed from many diverse tasks increases the agent’s chance of quickly and satisfactorily solving a future task. Being able to pick and choose the appropriate bits of knowledge extracted from previous tasks greatly facilitates learning in the future.

Neural architectures, though widely successful in a vast array of narrow, isolated
learning tasks [5], greatly struggle when adapted to the problem of continual learning. In short, although such connectionist systems are loosely inspired by the human brain, each time a neural architecture is trained on a new task, it *catastrophically* forgets the knowledge it acquired on previous tasks. While many remedies and heuristics have been proposed to alleviate this issue, artificial neural networks still suffer from this particular form of forgetting when faced with a sequence of tasks, whereas in humans, forgetting is, at most, gradual.

In order to face the catastrophic forgetting problem, this thesis will challenge the very workhorse behind training neural networks—back-propagation of errors [6]. Beyond practical issues, such as the issue of unstable gradients [7], one central problem with back-propagation is that it is highly unlikely to be implemented in the human brain. Currently, there is a great dearth of evidence to suggest that learning representations of a human agent’s environment is done through back-propagation [8]. Given this, then it might be fruitful to explore more biologically-motivated alternatives to back-propagation.

This thesis advocates that the path towards true and effective lifelong learning machines involves rethinking how we structure and learn the parameters of neural architectures. Specifically, instead of optimizing a single, global objective that is grounded in the input, as we do when using back-propagation of errors, a neural system should instead reduce discrepancy. In summary, this dissertation puts forth that such a system should be able to:

1. Search for representations that better explain the environment, also known as target representations. This leads to the concept of *higher-level* objectives.

2. Directly minimize the mismatch between its current representations and target representations.

The above criteria describe what we call coordinated local learning rules, which can be linked to one of the most promising, modern theories of how learning is done in the brain—*predictive coding*. We furthermore show that the first step of reducing discrepancy lies in learning to how to *simultaneously discriminate and generate*. The resultant hybrid model will then allow us to introduce mechanisms for memory refresh, which we will show allows the neural system to consolidate old knowledge by learning how to not forget. This will be particularly important when learning across a sequence of tasks. Finally, with respect to streams of data, we will show that discrepancy reduction can be extended to beyond simply learning parameters and further used as a mechanism
for continual error-correction, a mechanism also strongly motivated by the theory of predictive coding. When combined with a simple principle–fast and slow states–meant to encourage the improved capture of longer-term dependencies in data sequences, we arrive at a novel architecture, the Temporal Neural Coding Network, which is meant to learn continuously from temporal data streams, such as video or natural language conversations.

1.2 Contributions of this Thesis

The prominent contributions of this thesis are the following:

1. Deep neural architectures, and the various algorithms needed to search for suitable parameters, are developed when the amount of labels/annotations are scarce but unlabeled data is plentiful. Experiments show that these deep hybrid models generalize better than several competitive approaches proposed at the time of their introduction.

2. A framework is proposed for learning recurrent network models that are better able to capture longer-term dependencies in data sequences and a vastly simpler and effective model, the $\Delta$-RNN, is derived. Experimental results show that this model can perform comparably or even outperform current, popular complex gated recurrent network models on the task of language modeling. Several other applications of the $\Delta$-RNN are also presented, showing the promise of this model to solve a wider variety of problems in Machine Learning including image compression and data synthesis.

3. A family of coordinated local learning algorithms is developed based on the proposed learning principle of this thesis, discrepancy reduction. In addition to being more neuro-cognitively plausible than back-propagation of errors, various incarnations of Local Representation Alignment, a novel algorithm we derive under this family, can even train highly nonlinear networks composed of discrete and stochastic units as well as units not previously integrable into the hidden layers of a neural system, such as the softmax activation.

4. Local Representation Alignment is shown to be capable of training temporal neural architectures on sequential data, without unrolling the network back through time,
one of the most biologically implausible aspects of back-propagation of errors when applied to recurrent neural networks. A novel directed generative neural network is proposed, the Temporal Neural Coding Network, that not only uses Local Representation Alignment to update parameters, but also uses its target computation scheme to error correct itself continuously, allowing it to adapt to even a stream of data where the distribution might evolve with time.

5. We show that Local Representation Alignment mitigates the loss of previously acquired knowledge in a neural system used to learn across multiple tasks when, at minimum, boundaries separating tasks are known. The use of error-feedback weights, whether random or learned, are evaluated in lifelong classification and a task we call lifelong autoencoding (or auto-association).

6. When task boundaries are not clearly defined and classes are accumulated incrementally, we propose an algorithm called the Memory-Consolidation Algorithm that learns, through Local Representation Alignment, a dual generative-discriminative system that can actively refresh its task memory. Significant retention of prior task performance is observed.

1.3 Overview

This thesis could be divided into multiple, general parts.

After developing some background in Chapter 2, the first part includes Chapters 3 and 4, which could be considered early work for this thesis that played an important role in developing some of the key ideas that are prominent in the later chapters. These ideas include: 1) the need for globally coordinated local learning, 2) the value of a hybrid goal where generation and discrimination objectives are satisfied simultaneously, and 3) a stateful neural model’s ability to capture longer-term dependencies in the data that can be improved by decomposing the state into a mixture of a slowly-changing memory and a fast-changing proposal that is data-driven.

The second part encompasses Chapters 5 and 6. This part takes the above ideas and focuses on developing a family of algorithms that can replace back-propagation of errors and investigates if these alternatives mitigate catastrophic forgetting. The key ideas in this section include: 1) learning in a neural system can be reformulated as the act of discrepancy reduction, or progressively cutting down on the mismatch
between the system’s internal representations of the world and target representations that better describe the world, 2) adaptation in the face of data sequences, even some with non-stationarity behavior, can be achieved through continual error-correction, where learning and inference are intertwined, and 3) when learning across sequences of tasks, information retention can be significantly improved not only by using a coordinated local learning algorithm but by also learning a complementary discriminative-generative model, where the generative component can refresh the discriminative facet by capturing useful structure in both the input data and the task/discriminative network itself.

The chapters are broken down as follows:

1. **Chapter 2: Parallel Distributed Processing, Revisited.** In this chapter, we provide a brief technical overview of all of the concepts that underpin the background behind the remaining chapters of this thesis. We start by revisiting the classical Parallel Distributed Processing theoretical framework and then we will review the basic machine learning problem formulations that lay the groundwork for later chapter problems. We will then continue with a brief technical review of back-propagation of errors as well as alternative learning procedures. Finally, we discuss the relevant theoretical concepts from predictive coding and implicit theory that have inspired several aspects of this thesis. Note that any further background material that is chapter-specific will be reviewed at the appropriate time within the relevant chapter.

2. **Chapter 3: The Deep Hybrid Learning Framework.** This chapter presents a framework for deep hybrid neural models that learn to discriminate and generate simultaneously. Semi-supervised learning scenarios are the domain in which these models are applied.

3. **Chapter 4: The Differential State Framework for Capturing Long-Term Dependencies.** In this chapter, a general way to derive state-based neural architectures is developed. Beyond showing how all modern gated neural architectures are special cases of this framework, we derive a vastly simpler recurrent neural network called the $\Delta$-RNN, which is shown to be competitive with far more complex models in a variety of sequential modeling applications.

4. **Chapter 5: Reducing the Discrepancy between Local Representations.** This chapter will present and develop a family of learning algorithms based on the
principle of discrepancy reduction. Comparisons to back-propagation and other more biologically-plausible alternative procedures will be made on static, non-temporal tasks such as classification.

5. **Chapter 6: Adaptive Learning by Aligning Representations.** In this chapter, we will investigate how the Local Representation Alignment algorithms developed in the previous chapter (Chapter 5) work in the face of several continual learning tasks. We will develop the Temporal Neural Coding Network, heavily inspired by the theory of predictive coding, and show how its continual error-correcting ability helps it adapt to time-varying data. Then, we will examine the effect that Local Representation Alignment has on networks when attempting to learn from sequences of tasks as well as cumulatively learn. Finally, we will present the Memory Consolidation algorithm, which particularly combines Local Representation Alignment with the idea of dual discriminative-generative models that earn how to not forget old information on previously seen tasks.

6. **Chapter 7: Conclusions.** This chapter will contain a brief summary of our contributions and describe future work.
Chapter 2  |  Parallel Distributed Processing, Revisited

This chapter will review the technical background and historic content needed to approach the ideas in this thesis. Furthermore, this chapter will begin to formulate one of the critical ideas that form part of the backbone of this thesis–local learning. We will start by reviewing the Parallel Distributed Processing (PDP) framework [9], one of the first theoretical frameworks proposed for unifying the wide variety of connectionist models. Starting from the PDP framework highlights that this thesis is concerned with merging ideas from Cognitive Science and Machine Learning in articulating the machinery by which distributed representations may be automatically acquired. Following this, some of the generic, basic problem formulations in statistical learning will be described and formalized, since these problems will appear throughout the various chapters of this thesis and will structure the experiments conducted.

In terms of learning algorithms (or update rules), back-propagation of errors, the quintessential algorithm for computing updates to the parameters of artificial neural networks, will be described. Beyond this, we will carefully review an array of important alternative procedures that either form part of the inspiration for the central ideas of this thesis and characterize some part of the algorithmic frameworks developed in later chapters. Note these algorithms will be reframed in the notation and perspective of this thesis to ensure consistency and clarity.

Finally, important neuro-cognitive concepts from the theory of predictive coding and implicit learning, which have motivated and informed most of the work done to support this thesis, will be elucidated.
2.1 The Parallel Distributed Processing Framework

To develop an understanding of the vast array of connectionist architectures that have been proposed over the decades, a general guiding framework is necessary to view such models from a unified and rigorous perspective. In 1986, Rumelhart et al. developed the Parallel Distributed Processing (PDP) framework [9], which aimed to describe the common, foundational elements in all connectionist models as well as their respective training algorithms. This framework, despite its age, still serves as a useful and important way for understanding modern neural models and learning algorithms, as many of these architectures share many similarities with their cousins of the 1980’s or build upon the pioneering work of the time, such as the Deep Belief Network [10]. As recent work has reported state-of-the-art advancing results in tasks ranging from computer vision to audio signal processing, viewing these neural-based architectures from the PDP framework allows us to understand how early principles have led to many scientific advancements in the context of ever-increasing computational power and massive parallelism (something not present at the time PDP was first introduced).

A connectionist model under the view of the PDP framework is a computational system that is inspired by the way the human brain processes information. The framework rests on the idea that neuronal connections are distributed in a parallel fashion, allowing for the emergence of higher order behavior from the activities of simple networked units or agents. The PDP framework serves to describe all similar models or architectures composed of uniform units that aim to describe mental phenomena, such as modeling states via numerical vector representations, or simply sets of neuronal activation values.

A PDP model must represent information in a non-local, distributed fashion, store memory and knowledge implicitly in the connections between processing elements, and learning happens through the incremental or gradual adjustment of neuron-to-neuron synaptic strengths. Models that embody the assumptions of PDP [11] are composed of the following:

1. A set of processing elements.
2. A state of activation.
3. An output function for units (such as the sigmoidal non-linearity).
4. Connectivity patterns among units.
5. A rule for propagating signals through network connections.

6. An activation rule for aggregating incoming inputs to a given unit to produce a resulting state for a given unit.

7. A learning rule for modifying connections via experiences.

8. An environment in which such a network system learns from and operates in.

Since its introduction, the core elements the PDP framework describes that are common to all connectionist models have remained the same, making it still a useful lens through which to view modern neural architectures, especially those that have seen recent success resulting from the composition of multiple layers of processing units. With the advancement of hardware for computational tasks as well as improvements in large-scale machine learning paradigms [12], larger PDP architectures can be investigated to process copious amounts of data in an attempt to uncover useful, underlying information regarding its unknown distribution and ultimately build better categorization or predictive models.

There are many architectures that embody the PDP framework’s key components, proposed over the many decades of research. While this review will not cover all variants, the aim will be to sample a few models that are representative of the central learning paradigms: associative learning, which entails supervised learning to perform either discriminative or pattern completion tasks (i.e., discriminative learning), and regularity discovery, generally involving an unsupervised procedure for building good generative models of the data itself (i.e., generative learning). Examining these two classical paradigms will support the notion of hybrid learning later in Chapter 3. We will also make sure to review the problem of sequence learning and representation learning.

As described in the PDP framework, connectionist architectures require a learning or training phase in order to acquire knowledge of the underlying statistical distribution of the data. A learning algorithm is the key element of a PDP model that can transform a network of randomly initialized parameters (such as biases or connection weights) to one capable of generalizing from a population sample to previously unseen examples. While results reported in modern literature use varying approaches to train networks, many of them have roots in or are simply modern implementations of classical techniques.
2.2 Learning Problem Formulations

2.2.1 Supervised Learning

The task of supervised learning can be posed as estimating the conditional probability distribution $p(y|x)$, where we associate a value $x \in \mathcal{X}$ (the input space) with some target $y \in \mathcal{Y}$ (the output space).\(^1\) This is typically done through maximum likelihood estimation to find optimal model parameters $\Theta$ for a parametrized family of distributions $p(y|x; \Theta)$.

In a typical setup, the dataset $D = D_{lab}$, of $N$ examples, or, specifically, $D_{lab} = \{(y_i, x_i)\}_{i=1}^N \sim p(y, x)$. The goal is to learn a function $f : \mathcal{X} \mapsto \mathcal{Y}$ parametrized by $\Theta$, or $f_{\Theta}(\cdot)$. Note that $f \mapsto \mathcal{F}$ where $\mathcal{F}$ is the hypothesis space. The key assumption behind supervised learning is that the training set consists of a sample independent and identically distributed pairs of $(y_i, x_i)$.

We define a loss function, $L : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}_{\geq 0}$, to measure how well our parametrized mapping fits the training data. If a model predicts $\hat{y}$ for a particular $(y_i, x_i)$, the loss would be $\mathcal{L}(y_i, \hat{y}_i)$. If we take an empirical risk minimization approach to finding an optimal predictor, we are optimizing the negative log likelihood as our loss: $\mathcal{L}(y, \hat{y}) = -\log p(y|x)$. If we take a structural risk minimization approach, we incorporate a regularization penalty into the learning process to control the bias/variance tradeoff. This regularization penalty, or prior, is placed on the parameters of our function leading to the loss: $\mathcal{L}(y, \hat{y}, \Theta) = -\log p(y|x, \Theta) - \log p(\Theta)$ (the last term also contains the control parameter $\lambda$, which typically becomes concrete once a prior distribution over $\Theta$ is defined).

So far, this section has only described discriminative approaches to finding an optimal predictor. However, if we can define the joint probability of $p(y, x)$, then the loss function (from an empirical risk point-of-view) becomes: $\mathcal{L}(y, \hat{y}) = -\sum_i \log p(y_i, x_i)$. This approach falls under the category of generative learning.

2.2.2 Unsupervised and Representation Learning

Unsupervised learning is focused on extracting information from a distribution automatically (without the need for human intervention). Some typical forms of unsupervised

\(^1\)Note that all vectors/matrices in this thesis are to be taken as strictly column-major orientation. This also aligns with the software library that was written in support of the content of this thesis.
learning include density estimation and denoising samples that are drawn from some (unknown) distribution.

Another, more classical, view of unsupervised learning is that it seeks the “best” representation of the data, or one that preserves as much information as possible about x while obeying the constraint that the representation is simpler than x itself. Simpler can refer to representations that are:

1. Lower-dimensional: “compressing” as much information about x as possible into a smaller representation,

2. Sparse: embedding the data into a representation that has mostly zeros as entries (usually of higher-dimensionality to avoid discarding information),

3. Independent: disentangling the underlying sources of variation in the data’s distribution leaving the dimensions of the representation statistically independent.

Much of the work developed in this thesis falls under the realm of representation-learning with the goal of learning, “…transformations of the data that make it easier to extract useful information when building classifiers or other predictors” [13]. Shallow learning methods, which require extensive prior human knowledge and large, labeled datasets, have been argued to be limited in terms of learning functions that violate restrictive assumptions such as smoothness and locality [14]. Moreover, architectures with a single unobserved layer require an exponentially increasing number of units to accurately learn complex distributions that deeper architectures, composed of multiple layers of non-linearity, potentially can. Deeper models “exploit the unknown structure” of the input data distribution to generate high-level features that are invariant to most variations in the training examples and yet preserve as much information regarding the input as possible [15]. Whether discriminative or generative, these architectures all learn, albeit differently, internal, higher-level representations of the input distribution in their hidden layers, a common property unifying all connectionist models.

Generally, it has been shown that multiple layers of non-linear functions can better approximate complex functions and unknown distributions of data [10,16]. This powerful notion that depth helps with efficiency and compactness in representation of complex functions can itself be traced back to established concepts in circuit theory [17]. As universal function approximators [18], MLPs can be seen as trainable associators of input patterns (i.e., data samples represented as feature vectors) to output pattern activations (i.e., class label vectors or target variables).
2.2.3 Sequence Learning

Sequence learning, or modeling, in contrast to the standard supervised and unsupervised setups described earlier, is concerned with estimating the next sequence element $y^t$ given past inputs. This can be formulated as maximizing the log likelihood: $\log p(y^t|x^1,\cdots,x^t)$ (or minimizing the negative log likelihood). The full loss being optimized by a sequential neural model is ultimately:

$$L(\{y_1, \cdots, y_\tau\}, \{x_1, \cdots, x_\tau\}) = \sum_t L_t(y_t|x_t) = -\sum_t \log p_\theta(y_t|\{x_1, \cdots, x_t\}) \quad (2.1)$$

which is called the sequence loss. The convention $y^t$ is used to translate the idea of supervised learning, when data was simply available as an unordered collection of $(y_i, x_i)$ pairs.

We can make the above modeling task look more clearly like a next-step prediction problem, such as in language modeling, by simply performing the substitutions $y_t = x_t$ and $x_t = x_{t-1}$, where $x_0 = \emptyset$, the null input (to represent a start of a sequence). This means we are attempting to learn a graphical model of the joint distribution $p(x_1, \cdots, x_\tau)$ decomposed as follows:

$$p(x_1, \cdots, x_\tau) = \prod_{t=1}^\tau p(x_t|x_{t<}) = \prod_{t=1}^\tau p(x_t|x_1, \cdots, x_{t-1}). \quad (2.2)$$

This means that Equation 2.1, obtained by taking the negative of the logarithm of Equation 2.2, can be written as:

$$\log p_\theta(x_1, \cdots, x_\tau) = -\sum_{t=1}^\tau \log p_\theta(x_t|x_{t<}) \quad (2.3)$$

where we note that there are no dependencies among the outputs $y_t$, which would require a different specification of the sequence probability distribution. In other words, outputs $y$ are conditionally independent given the sequence of $x$ data points.
2.3 Computing Updates to Parameters

In this section, we will now review back-propagation of errors, the core algorithm used to compute updates to the synaptic weights. In addition, we will also review and describe a variety of alternative algorithms important to the background of this thesis, some of which have been argued to more plausible in terms of neuro-biology. These include feedback alignment, recirculation and target propagation, as well as Contrastive Divergence and Wake-Sleep.

For the sake of uniformity and clarity, we will describe how each algorithm calculates parameters updates in the context of a feedforward network, or multilayer perceptron (MLP), for a supervised learning task (see Section 2.2.1). We will further restrict ourselves to the single-sample case, or mini-batches of size one, for further simplicity, although these algorithms naturally extend to using more than one sample in parallel.

To be consistent with notation of the later chapters of this thesis, the pre-activities of the MLP at layer $\ell$ are denoted as $h^\ell$ while the post-activities, or the values output by the non-linearity $\phi_\ell(\cdot)$, are denoted as $z^\ell$. The target variable at the output units, $z^L$, is denoted $y^L_z$. Note that in order to map directly to the supervised learning problem described earlier, take $y^L_z = y$, $z^L = \hat{y}$, and $\mathcal{L}^L(y^L_z, z^L) = \mathcal{L}^L(y, \hat{y})$. For this example network, $L = 3$ which means the model will be defined by the parameters $\Theta = \{W^1, W^2, W^3\}$, or three sets of synaptic weight matrices.\(^2\)

Forward propagation of activities, starting with the data $z^0 = x$ at the input layer, in the MLP is defined as follows:

\begin{align*}
  h^1 &= W_1 z^0, & z^1 &= \phi_1(h^1) \\
  h^2 &= W_2 z^1, & z^2 &= \phi_2(h^2) \\
  h^3 &= W_3 z^0, & z^3 &= \phi_3(h^3).
\end{align*}

2.3.1 Back-propagation of Errors

If we assume an L2-norm output loss, or squared error cost function, at the output layer, we derive the updates for the synaptic weight matrices by employing the chain rule of calculus. The gradient of the output loss with respect to the topmost weight matrix, $W^3$,

\(^2\)Biases $c^\ell$ are omitted for clarity.
is:
\[
\Delta W_3 \propto - \frac{\partial L^3(y^3, z^3)}{\partial W_3} = - \frac{\partial L^3(y^3, z^3)}{\partial h^3} \frac{\partial h^3}{\partial W_3} = - \frac{\partial L^3(y^3, z^3)}{\partial h^3}(h^2)^T \\
= - \left( \frac{\partial L^3(y^3, z^3)}{\partial z^3} \otimes \frac{\partial z^3}{\partial h^3} \right)(h^2)^T = - \left( e^3 \otimes \phi'(h^3) \right)(h^2)^T
\]

where \((\cdot)^T\) denotes the matrix transpose operator and \(\otimes\) denotes the Hadamard product. Furthermore, observe that \(e^3 = L^3(y^3, z^3)\), which means that we can think of the first derivative of the cost function with respect to the output units as a set of error units, an important idea that will play a major role throughout this thesis. To find the updates for the lower-level matrices, \(W^1\) and \(W^2\), we continue recursively applying the chain rule using the gradient of the loss with respect to \(h^3\) and working our way down. As a result, we obtain:
\[
\Delta W_2 \propto - \frac{\partial L^3(y^3, z^3)}{\partial W_2} = - \frac{\partial L^3(y^3, z^3)}{\partial h^2} \frac{\partial h^2}{\partial W_2} = - \left( \frac{\partial L^3(y^3, z^3)}{\partial h^3} \frac{\partial h^3}{\partial z^2} \frac{\partial z^2}{\partial h^2} \right) \frac{\partial h^2}{\partial W_2} \\
= - \left( \left( W_3 \right)^T \frac{\partial L^3(y^3, z^3)}{\partial h^3} \otimes \phi'(h^3) \right)(h^1)^T
\]

\[
\Delta W_1 \propto - \frac{\partial L^3(y^3, z^3)}{\partial W_1} = - \frac{\partial L^3(y^3, z^3)}{\partial h^2} \frac{\partial h^2}{\partial z^1} \frac{\partial z^1}{\partial h^1} \frac{\partial h^1}{\partial W_1} \\
= - \left( \left( W_2 \right)^T \frac{\partial L^3(y^3, z^3)}{\partial h^2} \otimes \phi'(h^1) \right)(z^0)^T
\]

where \(z^0 = x\), or the input pattern vector. Figure 2.1 illustrates how the derivations above work with respect to the underlying computation graph. One key item to observe in Figure 2.1 is the feedback pathway that starts from the global cost function and works its way down towards the very bottom of the graph.

While we have shown how to compute the synaptic weight updates using the chain rule for an MLP, the same algorithm can be applied to any computation graph, provided that it is acyclic. This includes very long chains of operations, such as those that make up deep autoencoders or unfolded recurrent networks, which can create new problems as the depth of credit assignment increases. This issue will be discussed at length in Chapter 5.
Figure 2.1. Back-propagation of errors applied to a three-layer MLP. The blue lines indicate the reverse propagation calculation. Note that $z^{0} = x$, the input datum, for notational consistency. The blue dash-dotted lines mark the global feedback pathway that back-propagation of errors depends upon to find updates to parameters in the lower-level of the model.

2.3.1.1 Unrolling: Back-propagation through Time

In order to learn from sequence data, e.g., text data consisting of English sentences, one could adapt the MLP by working within a fixed-size context window (or bounded history) as was done in early neural probabilistic language models [19, 20]. However, a more powerful neural architecture, the recurrent neural network [21] or recurrent network (RNN), can be used to process variable-length sequences. The RNN is essentially a connectionist state-machine, or stateful process, since computation is function of the model’s recent memory of its previous states, as summarized in a context vector, and the current presentation of a data point at any given time $t$.

To update the internal weights of an RNN given sequence data, back-propagation of errors is used with one modification—the model must be unrolled or unfolded $T$ steps back in time ($T$ being the number of ordered data points in a given sequence). In short, we must undo the internal cycle by transforming the RNN into one very (structurally) deep MLP. This MLP does not contain any more parameters than the original RNN since

$$L^3 = ||e^3||_2^2$$

$$z^3 = \phi_3(h^3)$$

$$h^3 = W_3^T z^2$$

$$z^2 = \phi_2(h^2)$$

$$h^2 = W_2^T z^1$$

$$z^1 = \phi_1(h^1)$$

$$h^1 = W_1^T z^0$$

$$z^0 = x$$

$$e^3 = (z^3 - y^3_3)$$

$$\frac{\partial L^3}{\partial h^3}$$

$$\Delta W_3$$

$$\Delta W_2$$

$$\Delta W_1$$
we will copy the parameters to each step in time we unroll the RNN, effectively sharing parameters and only changing the input datum at each time step. When backprop is applied to the unfolded RNN, the resulting learning process is usually referred to as back-propagation through time [22, 23]. In practice, in order to maintain computational tractability, a sample sequence will be broken up into sub-sequences or chunks of length \( K \), where \( K < T \), and the RNN will be unfolded over each chunk instead process time-varying data in chunks, i.e, truncated back-propagation through time. This creates a very deep feedforward network, with each input of each time step being fed into the unfolded graph and the underlying parameters being copied at each time step. This parameter-sharing over time circumvents the need for a number of model parameters that increases with the length of each sequence, which would be highly impractical from a computer memory storage perspective.

As a simple example, consider an Elman RNN with a linear output layer, defined by the following set of equations:

\[
\begin{align*}
    z_t &= \phi^h (W x_t + V z_{t-1}), \\
    \tilde{y}_t &= \phi^o (U z_t)
\end{align*}
\] (2.10)

with parameters \( \Theta = \{W, V, U\} \) (biases omitted) and \( \phi^h(\cdot) \) and \( \phi^o(\cdot) \) are the hidden and output post-activation functions, respectively. For the case of unrolling, imagine we are predicting the fourth data point of a single sequence \( y_3 = x_4 \) given the previous two data points \( (x_1, x_2, x_3) \) and we use are using the mean squared error as our loss. Finding the derivative of the loss at time \( t = 4 \) with respect to the input-to-hidden weights \( W \) and the recurrent weights \( V \) would be:

\[
\begin{align*}
    \frac{\partial L(y_3, \tilde{y}_3)}{\partial W} &= \sum_{t=1}^{3} \frac{\partial L(y_3, \tilde{y}_3)}{\partial \tilde{y}_3} \otimes \frac{\partial \tilde{y}_3}{\partial z_3} \otimes \frac{\partial z_3}{\partial z_t} \otimes \frac{\partial z_t}{\partial W} \quad (2.11) \\
    \frac{\partial L(y_3, \tilde{y}_3)}{\partial V} &= \sum_{t=1}^{3} \frac{\partial L(y_3, \tilde{y}_3)}{\partial \tilde{y}_3} \otimes \frac{\partial \tilde{y}_3}{\partial z_3} \otimes \frac{\partial z_3}{\partial z_t} \otimes \frac{\partial z_t}{\partial V} \quad (2.12)
\end{align*}
\]

where \( \otimes \) denotes the Hadamard (or elementwise) product. while the derivative of the loss with respect to the output weights (in this example) is simply:

\[
\frac{\partial L(y_3, \tilde{y}_3)}{\partial U} = \frac{\partial L(y_3, \tilde{y}_3)}{\partial \tilde{y}_3} \otimes \frac{\partial \tilde{y}_3}{\partial U}. \quad (2.13)
\]

Since parameters are shared over each time step, we will need to sum the contributions
of each gradient over the unrolling length. See Figure 2.2 for a graphical depiction of this example.

2.3.2 Random Feedback Alignment

Random feedback alignment (RFA) [24], and its variants, notably direct feedback alignment (DFA) [25], attack one of the problems of back-propagation of errors—the weight-transport problem.

Feedback alignment [26] and its variants [25, 27], have shown that random feedback weights can deliver useful teaching signals. This random form of back-propagation has also been used to develop an event-driven variation of the learning rule suitable for neuromorphic implementations of neural networks [28]. More importantly, feedback alignment algorithms resolve the weight-transport problem, which has been one criticism of back-propagation [29, 30], by showing that coherent learning is possible with asymmetric forward and backward pathways. Rather, the error back-projection pathways need not be the transposition of the weights used to carry out forward propagation, and the learning process can instead be viewed as the alignment of feedforward weights with feedback weights.

Since RFA is essentially back-propagation of errors but with the removal of the
\[ L^2 = ||e||^2 \]

\[ z^3 = \phi_3(h^3) \]

\[ y^3 = \phi_3(h^3) \]

\[ \Delta W_3 \propto - \left( E_3 \frac{\partial L^3}{\partial h^3}(y^3, z^3) \otimes \phi'(h^3) \right) (h^1)^T \]  

\[ \Delta W_1 \propto - \left( E_2 \frac{\partial L^3}{\partial h^2}(y^3, z^3) \otimes \phi'(h^1) \right) (z^0)^T. \]  

In the case of the DFA algorithm, the weight updates are simply:

\[ \Delta W_2 \propto - \left( E_3 \frac{\partial L^3}{\partial h^3}(y^3, z^3) \otimes \phi'(h^3) \right) (h^1)^T \]  

\[ \Delta W_1 \propto - \left( E_2 \frac{\partial L^3}{\partial h^2}(y^3, z^3) \otimes \phi'(h^1) \right) (z^0)^T. \]  

Figure 2.3. Random feedback alignment (RFA) and direct feedback alignment (DFA) applied to a three-layer MLP. The blue lines indicate the reverse propagation calculation. Note that \( z^0 = x \), the input datum. Since explicit weight matrices are used to transmit the error signals down to each layer, we now replace the original symbol for identifying the partial derivative of the global loss with respect to each layer pre-activation, \( \frac{\partial L}{\partial h^l} \), with a symbol to represent displacement instead, since the transmitted error is not a proper derivative.

symmetric weight constraint for transmitting error signals back along the network, its weight updates, for \( W_1 \) and \( W_2 \), look quite similar to those of Equations 2.8 and 2.9, and given as follows:

\[ \Delta W_2 \propto - \left( E_3 \frac{\partial L^3}{\partial h^3}(y^3, z^3) \otimes \phi'(h^3) \right) (h^1)^T \]  

\[ \Delta W_1 \propto - \left( E_2 \frac{\partial L^3}{\partial h^2}(y^3, z^3) \otimes \phi'(h^1) \right) (z^0)^T. \]  

In the case of the DFA algorithm, the weight updates are simply:

\[ \Delta W_2 \propto - \left( E_3 \frac{\partial L^3}{\partial h^3}(y^3, z^3) \otimes \phi'(h^3) \right) (h^1)^T \]  

\[ \Delta W_1 \propto - \left( E_2 \frac{\partial L^3}{\partial h^2}(y^3, z^3) \otimes \phi'(h^1) \right) (z^0)^T. \]  

19
For both RFA and DFA, the weight updates for the topmost synaptic weight matrix $W_3$ are the same as in back-propagation of errors (Equation 2.7).

### 2.3.3 Recirculation and Target Propagation

Recirculation [31, 32] is an old idea and was originally proposed as a simpler alternative to training auto-associators (or autoencoders) with back-propagation of errors. The core procedure works by essentially using two forward passes through the encoder to create signals for learning. The intuition is that the second forward pass is used to inform the higher of two layers about the effect that it had on the lower-level layer. Training the autoencoder block then amounts to: 1) using the input $x$ to the encoder as the target for the decoder’s reconstruction $\hat{x}$, and 2) using the encoder’s output (or initial encoded value) $z$ as the target for the second forward pass through the encoder, or $\hat{z}$.

We can use the above basic idea and propose extending it to learn a deeper network, in particular, our three-layer MLP example. To utilize recirculation, the MLP is extended by adding the decoder weights $\{V_1, V_2, V_3\}$ to the original parameters $\Theta$. The decoders for the extended model are defined as:

\[
\begin{align*}
\hat{h}^2 &= V_3z^3, \\
\hat{z}^2 &= \phi^v_3(\hat{h}^2) \\
\hat{h}^1 &= V_2z^2, \\
\hat{z}^1 &= \phi^v_2(\hat{h}^1) \\
\hat{h}^0 &= V_1z^1, \\
\hat{z}^0 &= \phi^v_0(\hat{h}^0)
\end{align*}
\]

(2.18) (2.19) (2.20)

the structure of which will be examined again in Chapter 5, in the context of an improved variation of an algorithm called Difference Target Propagation.

While the algorithm was proposed for autoencoders (with a light description at the end of the original paper as to how this would apply to models with multiple hidden layers), it says nothing as to how this procedure would work with MLPs. We consider two ways in which this might work. 1) Calculate the update for $W_3$ using Equation 2.7 and do not form an autoencoder for the topmost layer, 2) Alter the update that would be given by recirculation for the topmost weights by replacing the encoder’s output from the first pass with the actual target variable. This would also allow the algorithm to work in the semi-supervised setting, a problem which will be the subject of Chapter 3. After propagating the activities through the network as normal, from bottom to top, we use the decoder at each layer and attempt to reconstruct the bottom layer’s activity. This new
decoded value is then run through the encoder a second time to get a new input value for
the decoder. Using this basic procedure, we obtain the feedforward weight updates as
follows:

\[
\Delta W_3 \propto \left( (z^3 = y) - \tilde{z}^3 \right) (\tilde{z}^2)^T
\]  \hspace{1cm} (2.21)
\[
\Delta W_2 \propto \left( z^2 - \tilde{z}^2 \right) (\tilde{z}^1)^T
\]  \hspace{1cm} (2.22)
\[
\Delta W_1 \propto \left( z^1 - \tilde{z}^1 \right) (\tilde{z}^0)^T
\]  \hspace{1cm} (2.23)

and obtain the decoding/generative weight updates as follows:

\[
\Delta V_3 \propto \left( z^2 - \tilde{z}^2 \right) (z^3)^T
\]  \hspace{1cm} (2.24)
\[
\Delta V_2 \propto \left( z^1 - \tilde{z}^1 \right) (z^2)^T
\]  \hspace{1cm} (2.25)
\[
\Delta V_1 \propto \left( (z^0 = x) - \tilde{z}^0 \right) (z^1)^T
\]  \hspace{1cm} (2.26)

where we notice in these particular update rules that we do not require the derivative
of the post-activation functions of either the encoder or decoder output units. More
importantly, one notices that the above update rules are purely local—each encoder and
decoder is only concerned with perturbations within its own subgraph (a concept we will
define in Chapter 5) within the network.

Another important idea of recirculation is to ensure that the units in the block have a
high-level of “regression”, which ensures that the original input values are not simply
replaced by the reconstructed values. The equation given in the original paper actually
shows that this is implemented through a convex combination of the original input to the
encoder and the reconstructed input as follows:

\[
\bar{x} = \lambda \otimes x + (1 - \lambda) \tilde{x}
\]  \hspace{1cm} (2.27)

where \( \lambda \) is to a value close to one, ensuring that the new input to the encoder, for the
second pass, is not too different from the initial input value. This interpolation ensures
the upper layer measures the effect of a small change to the input. While recirculation
requires that the encoder and decoder are symmetric in order to work properly, the
learning procedure is self-aligning, and as a result, unmatched weights will ultimately
tend towards symmetry.

Target propagation [33, 34] builds on the same idea that recirculation does—a learning
signal might be created by instead calculating targets rather than loss gradients at each layer. This allows for the development of local learning rules, removing the need for a global error pathway to carry error derivatives across (and thus side-stepping the vanishing gradient problem). Furthermore, some connections can be made between target propagation and Spike-Time-Dependent-Plasticity [35]. The general approach in target propagation is that the feedback weights are trained to learn the inverse of the feedforward mapping. This has also been roughly applied to training recurrent network models [36] but still requires unrolling the computation graph along the length of the sequence. Target propagation also requires a few other things in order to work, notably that every layer in the network model must be an autoencoder and that a linear correction term is used to account for the imperfections of auto-encoders (which can obstruct learning).

The target propagation algorithm revolves around the concept of the function inverse—if we had access to the inverse of the network of forward propagations, we could compute which input values at the lower levels of the network would result in better values at the top that would please the global cost. In essence, we would use the inverse to propagate back along the network the target value and then update each layer to move closer to this target value. So long as we have access to the inverse of the functions used for each layer, we can use any non-linear activation, including those that are discrete-valued. Under simple conditions, when all the layer objectives are combined, target propagation could yield updates with the same sign as the updates obtained by back-propagation [37].

In implementing target propagation as a connectionist model, one constructs a stack of autoencoders, where the decoder of each approximates the inverse function for that layer. A layer’s target value for its encoder is the value generated at that layer by its corresponding decoder. Since the encoder is supposed to be the inverse of the decoder, when a lower layer adapts towards its target value, it will cause the layers higher-up to do so as well, creating the a feedback pathway for learning. Furthermore, to ensure that the encoder is receiving useful targets, the values produced by the decoder should be justifiably better than the currently inferred values of the encoder. The idea of denoising is then reintroduced into the learning procedure—noise is added to the input of decoders and targets for the decoders are the clean values produced by the encoders. The reconstructions of the decoder are then more probable as expected and, as a result, the resultant targets might prove useful in helping the encoder learn separate information from noise as it tries to disentangle the factors of variation at each layer to help the layers above. Through a layerwise training approach, target propagation tries to ensure
that each layer is individually a useful contractive autoencoder, and targets are created through a denoising variation of the recirculation algorithm. A linear correction term is also introduced to help correct for imperfections in the auto-association process.

The idea of learning locally is slowly becoming prominent in the training artificial neural networks, with other recent proposals including kickback [38], which was derived specifically for regression problems. Decoupled neural interfaces [39] also operate locally, but take the approach of learning a predictive model of error gradients (and inputs) instead of trying to use local information to estimate an update to weights. As a result, this procedure allows layers of the underlying model to be trained independently. Target propagation and its variations represents a strong push towards using local, more biologically plausible, rules to learn neural systems. Later, this thesis will treat target propagation as an instance of a more general family of algorithms, the family of Discrepancy Reduction algorithms, and will derive far simpler and more effective procedures for computing synaptic weight updates within a local learning framework.

2.3.4 Contrastive Divergence

One early, biologically-plausible algorithm proposed for learning certain connectionist graphical models, e.g. restricted Boltzmann machines (RBMs), was Contrastive Divergence [40]. Through a Markov Chain Monte Carlo (MCMC) sampling process, or rather, alternating block Gibbs sampling, “fantasy” data vectors are generated from the model after initializing a Markov chain from a clamped vector drawn from the empirical distribution / training data. By propagating the data up to the latent variables and back down to the visible units, the model is run to thermal equilibrium, and a sample is taken at special state is (serving as an approximation of the model’s internal distribution) and used in what reduces to a simple subtraction step for computing parameter updates. One interpretation of the learning process underlying Contrastive Divergence is that it is attempting to minimize the distance, specifically the Kullback-Leibler Divergence [40], between two probability distributions—the model’s fantasy distribution and the data’s real underlying distribution).

While Contrastive Divergence is technically minimizing a different objective than the actual negative log likelihood of $p(x)$, even ignoring a “tricky term” in the proxy objective function itself, in practice this approximation of an approximated estimator appears to work well in practice. This training procedure can be made even less computationally
expensive by running the Gibbs chain for a few steps \((CD-k)\) or even just a single step \((CD-1)\) and computing the gradient from this biased sample, as the variance introduced by not running the model to equilibrium is not significant enough to make the learning process ineffective. Further improvements in generative learning may be obtained through algorithmic alternatives such as Parallel Tempering, which exploits the advantages offered by Persistent Gibbs Sampling, alternatively referred as Stochastic Maximum Likelihood. In essence, one simply maintains one or more continuously running chains in the background, using either random chain swapping or an expectation calculated across these “fantasy particles”, to facilitate a broader and faster exploration of the RBM’s internal configuration state space [41].

However, one key drawback of using the Contrastive Divergence training algorithm is the inability to track the log likelihood objective, which requires computing the intractable partition function \(Z\). Expensive approximations of the partition function, such as Annealed Importance Sampling, or metrics like reconstruction cross entropy can be used as proxies for tracking actual progress, but do not reflect how well or how poorly an RBM model is learning.

Many other algorithms for learning neural networks take inspiration from Contrastive Divergence, including the variational walkback algorithm [42] and equilibrium propagation [43]. Contrastive Hebbian learning (CHL) [44–46] works similarly to Contrastive Divergence in that it is ultimately computing parameter updates using a positive phase and a negative phase, trying to make the negative phase (or the “fantasies”) more similar to the positive phase (which is the state of the model clamped at the data).

In the next chapter, we will present the implementation of Contrastive Divergence in greater detail, given that it is one of the first important algorithms we experimented with when studying neural architectures in the course of this thesis.

**2.3.5 The Wake-Sleep Algorithm**

The wake-sleep algorithm [47, 48] was originally proposed as a simple way to learn two opposite directed generative models. One model was used to generate samples while the other one was used for inference. The “wake phase” of the algorithm entailed presenting a data point to the inference model, propagating the activities forward, and then updating the generative network, at each layer, to make it more likely to generate the values found in the inference network. The “sleep phase” involved sampling the generative model
to obtain a “fantasy” and the inference model was trained to infer the fantasy sample’s causes. In this algorithm, the updates are purely local (but not greedy).

The up-down back-fitting algorithm [49] was proposed as a contrastive version of the wake-sleep algorithm and was used to fine-tune Deep Belief Networks (after a greedy, layer-wise construction phase). In this variation of the wake-sleep algorithm, the problem of mode averaging is circumvented. In wake-sleep, it is possible for the inference weights to pick a particular mode of the layer above and stick to it, even if other modes would be useful in generating the data. This will hinder the effectiveness of the downward pass, sleep phase, in training the recognition weights since no other modes would be recovered. In the up-down algorithm, instead of starting the generative pass using a completely random sample input to the topmost layer of the generative model, we bias the sample at the top by running a Gibbs sampler for a few steps using the associative memory formed by the top-most RBM of the network.

The idea of using a network to infer the states of the generative model will also come back in various forms throughout this thesis. One somewhat modern incarnation of this idea is neural variational inference (NVIL)\(^3\). Variational autoencoders [50], a class of models that can be learned with NVIL, operate by minimizing the Kullback-Leibler Divergence between a prior distribution, such as a unit Gaussian or piecewise multimodal distribution [51], and a posterior distribution, represented by a multivariate Gaussian parametrized by a neural network, over a set of latent variables. One drawback to models learned via NVIL, as discussed in [52], is that the generative model is constrained by the quality of the task-dependent recognition model used to infer the posterior. This drawback will be circumvented in Chapter 6, where it will be shown that a directed model can be learned without a complementary inference model in the context of predicting time-varying data.

### 2.3.6 On Parameter Optimization

It should be noted that while it is not the focus of this thesis, parameter optimization of varying forms is used throughout all of the experiments. This thesis makes the distinction between optimization procedures and update calculation procedures, specifically referring to the later often as *update rules* in accordance with early literature related to artificial neural networks. Update rules range from the delta-rule (or updates found through

\(^3\)This also shares relevance as part of the model’s cost function is measuring discrepancy in latent space, which is one key idea that will be described in Chapter 5.
back-propagation of errors or reverse-mode differentiation) to Hebbian learning [53].

Once updates are found using any of the algorithms described in this chapter, parameters must be moved by the (weighted) displacement suggested by the update rule. The most basic algorithm for moving parameters is (stochastic) gradient descent [54, 55], which entails multiplying the update matrix by a single (global) scalar coefficient. This scalar is often referred to as the learning rate or step-size. However, modern update rules take the idea of the learning rate and extend it to the case where there is one learning rate per scalar parameter value. Update rules of this sort, also referred to in this thesis as adaptive update rules (and some of the literature as adaptive learning rates), include AdaGrad [56], AdaDelta [57], and Adam [58]. Many of the experiments in this dissertation will employ either simple gradient descent, RMSprop [59], or Adam.

2.4 Theoretical Inspirations

In this section, we will conclude by describing concepts from two neuro-cognitive theories, namely predictive coding and implicit learning, that have inspired various aspects of this dissertation and guided the development of the various models and learning procedures proposed. The descriptions put forth are primarily to familiarize the reader with some of the motivation behind this thesis and later chapters will describe further details and connections to these theoretical/computational concepts as needed.

2.4.1 Predictive Coding

Predictive coding theories posit that the brain is in a continuous process of creating and updating hypotheses that predict the sensory input it receives, directly influencing conscious experience [60]. Models of sparse coding [61] and predictive coding [62] embody the idea that the brain is a directed generative model where the processes of generation (top-down mechanisms) and inference (bottom-up mechanisms) are intertwined [63] and interact to perform a sort of iterative inference of latent variables/states. Furthermore, nesting the ideas of predictive coding within the Kalman Filter framework [64] can create dynamic models that handle time-varying data. Many variations and implementations of predictive coding have been developed [65–67]. Some approaches, such as predictive sparse decomposition [68], attempt to speed up the iterative inference by introducing an inference network, but this again, creates a problem similar to that of variational
inference—the generative model is constrained by the quality of the approximate inference model.

One key concept behind predictive coding that our own work embodies is that, for a multi-level objective to work well, each layer of a neural architecture would need an error feedback mechanism that could communicate the needs of the layer below it. If the learning signals are moved closer to the layers themselves, the error connections can directly transmit the information to the right representation units. Very importantly, this allows us to side-step the vanishing gradient problem that plagues pure back-propagation, where the internal layers of the architecture are trying to satisfy an objective that they only indirectly influence.

If we were to compare the updates from predictive coding’s local learning approach to back-propagation, the updates would still ascend/descend towards a similar objective, just not the steepest ascent/descent, so long as they were within 90 degrees of the direction given by back-propagation.\(^4\) However, since steepest ascent/descent is a greedy form of optimization, updates from a more localized approach might lead to vastly better generalization results [69]. However, it is important to emphasize that learning cannot be purely local (no information from other parts of the network model), or a coherent model might not be learnt. This problem of coherence will come up again in the next chapter, characterized as the problem of *shifting distributions*, where layer-wise representations, such as those closer to the input nodes of a Boltzmann-based model, change too much throughout learning (or simply have not had enough time to settle to more stable patterns) and confuse the layers farther away from the input, ultimately leading to unstable or ineffective acquisition of distributed representations that usefully describe the data.

Considering the problem of shifting representations above and the importance of more coherent local learning, this thesis will put forth and develop the idea of *coordinated local learning* throughout. Coordinated local learning means that higher-level, or more abstract, layers (which are farther away structurally from the input nodes) essentially direct lower-level layers (which are closer to the input nodes) in what patterns they should be extracting. A lower-level feature detector might be able to find different aspects of structure in its input since multiple patterns might satisfy its layer-wise objective. However, this detector will only locate the right bit of structure needed for the whole model to make sense, at any time-step, if a higher-level layer signals what pattern it should be looking for. Since greedy layer-wise approaches build the model from the

\(^4\)We will explicitly measure this angle for one of our key proposed algorithms later in Chapter 5.
bottom-up, freezing the learned lower-level parameters, this coordination is impossible to achieve, which will be a problem for some of the earlier models proposed in Chapter 3.

Another important concept from the work in predictive coding this dissertation will emphasize and exploit is encouraging neuronal competition and sparsity. Intra-layer competition among neuronal units is important (for reasons we will describe in later chapters) and we follow in the spirit of sparse coding [61] by often enforcing sparsity in neuronal activation patterns through a penalty or constraint.

2.4.2 Implicit Learning Theory

While there is no clear consensus on exactly what implicit learning is in humans, this thesis adopts the idea that implicit learning is different from explicit in that no knowledge is consciously accessed. Research on implicit learning that different areas of the brain, particularly working memory and attention, are involved in explicit learning as opposed to implicit learning [70]. In short, there are some things that a human agent learns without being to consciously describe or understand what was learned, with these skills and functions are largely stored in implicit forms of memory, such as procedural memory.

There are many, even common, examples of implicit learning such as learning how to ride a bike. A classic, well-studied example is artificial grammar learning where a human subject is presented with a string of letters, which follows the rules of an artificially constructed grammar, and then is tasked with separating out new strings as either belong to the artificial grammar studied or in a dustbin category. While the human subjects are able to successfully distinguish between examples from the artificial grammar and those that are not, they are unable to verbalize or explain how or why they separated the strings the way they did [71].

Sequence learning represents a large set of general examples of implicit learning, where human agents are able to learn the underlying structure of series or time-varying data but without understanding or being able to elucidate what exactly it is that they learn [72]. In language, people process sentences that follows the rules of syntax and make use of proper context much faster than those that do not and, furthermore, are fast in predicting what word will be uttered next in a sequence of words and quickly able to generate series of words that follow linguistic constraints [73].

The learning procedures and corresponding neural systems developed in this thesis could be considered as very high-level, abstract connectionist models of implicit learning.
especially with respect to sequential information. As such, much of the knowledge acquired by these neural models might only end up being stored in implicit memory components, again, such as in procedural memory. When tackling the more difficult problems in the later chapters of this thesis, especially those in Chapter 6, we will be tackling tasks that could still be considered as falling under implicit learning. However, a true never-ending agent would need forms of explicit memory (such as declarative memory) where retention of actual facts or bits of knowledge become important in facilitating higher cognitive function—this is a subject that is beyond the scope of this thesis and will be addressed in future work.
Chapter 3  |  Deep Hybrid Architectures

3.1 Introduction

One central principle advocated in this thesis is that neural architectures should be trained using local learning and optimization procedures. This premise was motivated in large part by the early successes reported in pre-training neural architectures using a greedy, unsupervised approach [10]. The model parameters uncovered in this pre-training phase would then be later fine-tuned on datasets with few labels or annotations. By greedily constructing a deep network layer by layer, effectively solving a sequence of local optimization problems, one could learn the parameters of networks deeper than one or two hidden layers which had proven nearly impossible historically. In what was an overall two-stage approach, one would:

1. Construct a generative model, such as a deep belief network (DBN), layer by layer, on unlabeled samples, up to \( L \) layers

2. Initialize the input-to-hidden and hidden-to-hidden parameter matrices of a multilayer perceptron (MLP) of depth \( L \) and add (at least) one hidden-to-output matrix of parameters, randomly initialized. The final set of parameters for this newly constructed MLP would then be fine-tuned jointly on the target labeled dataset.

In this way, one could train a multi-layered architecture, where each processing layer, in some sense, could be thought of as one level of abstraction of the input space.

Though this two-stage recipe made training artificial neural networks with more than two hidden layers effective for the first time, it suffered from several key issues. Notably, there was no principled way to determine how the unsupervised learning stage would
affect the performance of the subsequent fine-tuning stage. This meant that we would need to train a neural system “blindly” and hope that the parameters uncovered through unsupervised generative modeling would be useful for downstream supervised tasks with limited labeled data points.

To address this problem, we require a framework that can take advantage of the strong regularization benefit afforded by learning an expressive generative model of the input space while providing a clear way of tracking the progress on the target discriminative task(s) of interest. The simplest and most effective way to do this is learn both types of model parameters simultaneously rather than sequentially. The contribution of this chapter is a series of increasingly less greedy training procedures for training deep neural networks on datasets where the number of labels is significantly reduced, starting from a purely bottom-up view and ultimately introducing various forms of top-down feedback to guide the learning process. The resulting family of algorithms falls are unified by what this thesis defines as the deep hybrid learning framework, which takes a generative modeling approach to semi-supervised learning and extends it to efficiently learning architectures of multiple levels of abstraction. This framework is formalized under the multi-level semi-supervised prior hypothesis.

The deep hybrid algorithms of this chapter are particularly useful for semi-supervised learning situations where annotations are particularly scarce and expensive to obtain. This chapter will present results demonstrating the merit of these learning algorithms as the proportion of samples labeled accurately is decreased significantly and furthermore investigate the differences in the representations acquired by the final models, which, due to the formulation of the multi-objective function being optimized, can span the spectrum from entirely discriminative to entirely generative and entirely supervised to entirely unsupervised. This chapter will also present some experimental results when the problem is made a great deal more difficult and samples, unlabeled and labeled, arrive in an online fashion in the form of data stream. Throughout the development of these semi-supervised learning algorithms, the idea of local optimization and the need for global coordination without an explicit feedback pathway will emerge. This idea will play a central role to be developed further in Chapters 4 and 5.
3.2 Learning when Category Labels are Scarce

In the purely supervised learning scenario, to train effective neural-based classifiers composed of many parameters, massive quantities of properly labeled data points are required. However, for many applications, unlabeled data, such as images, documents, web-pages, and audio clips, are readily available while high-quality annotations are often difficult to come by. A intelligent system that could extract useful statistical structure from unlabeled data, with only minimal human guidance, would be ideal.

Semi-supervised online learning not only addresses practical applications, but it also reflects some challenges of human category acquisition [74]. Consider the case of a child learning to discriminate between object categories and mapping them to words, given only a small amount of explicitly labeled data, e.g., the mother pointing to the object, and a large portion of unsupervised learning, where the child comprehends an adult’s speech or experiences positive feedback for his or her own utterances regardless of their correctness. The original argument in this respect is applied to grammar (e.g., Chomsky, 1980; Pullum & Scholz, 2002). While neural networks are not necessarily models of actual cognitive processes, semi-supervised models can show learnability and illustrate possible constraints inherent to the learning process.

3.2.1 The Semi-supervised Learning Problem

In the typical inductive semi-supervised learning setup, we are given dataset \( D = D_{lab} \cup D_{unlab} \) of \( N \) samples, which can be decomposed into a labeled subset \( D_{lab} = \{(y_i, x_i)\}_{i=1}^{l} \sim p(y, x) \) and an unlabeled subset \( D_{unlab} = \{(x_{i+l})_{i=l+1}^{l+l} \sim p(x) \). From this data, our goal is to learn a prediction function \( f : \mathcal{X} \mapsto \mathcal{Y} \) (\( f \mapsto \mathcal{F} \)), where \( \mathcal{F} \) is the hypothesis space, parametrized by \( \Theta \), which will also be represented by \( f_{\Theta}(\cdot) \). \( x \in \mathcal{X} \) is a sample of the input space and \( y \in \mathcal{Y} \) is its corresponding target, or sample of the output space, which can be discrete in the case of classification and continuous in the case of regression. \( p(y, x) \) is the joint distribution over the labeled samples and \( p(x) \) is the marginal distribution, or simply the input distribution (when we integrate or sum over all possible target variables). Learning effectively in this setting means we must uncover a prediction function that predicts the target variables of unseen future examples that a function learned from the labeled or unlabeled data alone. Note that this problem setup is different from the transductive semi-supervised learning scenario, where the interest is
only in exclusively making accurate predictions on the unlabeled subset $D_{unlab}$ and not in generalizing to future data points.

One key idea from statistical learning theory, taken from the work of [77], that informs the objective functions designed for semi-supervised learning is the implicit ordering of the unlabeled samples. If this implicit ordering happens to rank the target predictor $f^*$ near the top, less labeled data will be required to learn $f^*$. Additionally, this ordering can be thought of as a prior over $F$. A learning algorithm must have some way of handling this implicit order and hunt for the best predictor using this ordering.

For effective semi-supervised learning, we also make two additional assumptions, related to consistency, about the data. The first is local consistency, which posits that nearby data points are likely to have the same label. The second is global consistency, which states that data points from the same structure (either a cluster or underlying manifold) are likely to have the same target variable. These two assumptions, taken together with the implicit ordering prior, serve as the foundation for the semi-supervised learning prior hypothesis and its deeper extension, which will be described shortly.

### 3.2.2 The Generative Statistical Approach

Semi-supervised learning can be done using a purely generative approach, in which a model is constructed by assuming a form of the joint distribution $p(y, x|\Theta) = p(y|\Theta)p(x|y, \Theta)$. $p(y|\Theta).$ $p(y|\Theta).$ The class prior distribution over $\mathcal{Y}$ can be assumed to be something such as the multinomial distribution while $\mathcal{X}$ can be assumed to be multivariate Gaussian for the class-conditional distribution $\Theta)p(x|y, \Theta)$ as in [78, 79]. The predictor can then be found using Bayes rules:

$$f(x|\Theta) = \arg \max_y p(y|x, \Theta) = \arg \max_y \frac{p(y, x|\Theta)}{\sum_{y'} p(y', x|\Theta)}$$  (3.1)

where the implicit ordering in the unlabeled data is given by the large-to-small ordering of the log likelihoods of possible $\Theta$ over the unlabeled data. This log likelihood ordering can be found by using:

$$\log p(\{x_i\}_{i=l+1}^{l+u}|\Theta) = \sum_{i=l+1}^{l+u} \log \left( \sum_{y \in \mathcal{Y}} p(y, x_i|\Theta) \right)$$  (3.2)
which means the highest ranked $f_{\Theta}$ is the function, which defines the underlying generative model parametrized by $\Theta$, that best fits the unlabeled data. One key issue with this approach is that the assumed distributions for the class prior and class conditionals must be correct, otherwise a mismatch will yield a poor fit.

The form of the objective function we finally arrive at when dealing with both labeled and unlabeled data is as follows:

$$\arg \max_{\Theta} \left[ \log p(\{y_i, x_i\}_{i=1}^l|\Theta) + \lambda \log p(\{x_i\}_{i=l+1}^{l+u}|\Theta) \right]$$  \hspace{1cm} (3.3)

where $\lambda$ is a coefficient that controls the effect that the search for the function over the unlabeled data log likelihood ranking has on the overall objective. The generative model learned using this non-convex, multi-objective approach will be one that fits both the labeled and unlabeled data well. Traditional approaches under this framework resort to Expectation-Maximization and other related optimization methods to find reasonable local maxima.

### 3.2.3 A Special Case: Unsupervised Pretraining

Instead of solving the above problem directly, one could instead aim to learn the input distribution exclusively using the unlabeled data (and perhaps also the labeled data instances but without their respective labels). In this situation, one first learns $p(x)$ and then modifies and finetunes the parameters acquired by then learning $p(y, x)$ or $p(y|x)$. This is known as pretraining which can yield great success in training deep architectures for discriminative tasks.

Unsupervised pre-training can help construct an architecture composed of many layers of feature detectors [80]. Even though the ultimate task is discriminative, a generative architecture, such as the Deep Belief Network (DBN) or Stacked Denoising Autoencoder (SDA), may be first used to initialize the parameters of a multi-layer perceptron (MLP) which then is fine-tuned to the supervised learning task [10].

However, learning the parameters of the unsupervised architecture is quite difficult, often with little to no grasp of the final influence the generative parameters will have on the final discriminative model [81, 82]. These architectures often feature many hyperparameters that affect generalization performance, quickly creating a challenging tuning problem for their human users.
3.2.4 The Dual Discriminative-Generative Approach

We can augment the above generative approach to semi-supervised learning by also directly learning a parametrized conditional predictor for the labeled data subset, \( p(y|x) \). Essentially, we combine the above generative modeling problem with a discriminative learning problem (taken from the fully supervised scenario) and attempt to find a model that solves both problems jointly [83]. In this situation, the generative and discriminative models both share some of the same parameters in \( \Theta \), which will regularize the function learned even further. The objective function then becomes:

\[
\arg \max_{\Theta} \left[ \gamma \log p(\{y_i|x_i\}_{i=1}^l|\Theta) + \beta \log p(\{y_i,x_i\}_{i=1}^l|\Theta) + \alpha \log p(\{x_i\}_{i=l+1}^{l+u}|\Theta) \right]
\]

(3.4)

where \( \gamma, \beta, \) and \( \alpha \) are coefficients that we will introduce in order to control the effect of each term of the tri-objective function, thus introducing three hyper-parameters that need to be tuned on validation data. We can reduce the number of meta-parameters to two by taking a convex combination of the two terms that operate on the labeled data and set \( \gamma = (1 - \lambda) \) and \( \beta = \lambda \). This formulation emphasizes a trade-off between the focus given to discriminative modeling and the focus given to generative modeling during the learning process, governed by the value of \( \lambda \).

3.3 Learning Greedily and Pseudo-Jointly

The above statistical learning framework, while powerful when applied and implemented correctly, has limitations. First and foremost, in order to learn \( p(y,x) \), one must be able to say something about its form, requiring prior knowledge of how the data is distributed. In addition, it is typical to construct fully observable models or those with only a single layer of latent variables, given the increasing difficulty and complexity of learning a hierarchy of latent variables.

In this section, we define two simple, flexible neural models that make only weak assumptions about the underlying joint distribution. Furthermore, we will propose their natural extensions to deep hierarchies of latent or hidden variables.
3.3.1 The Multi-Block Harmonium

3.3.1.1 Model Structure

The multi-block harmonium is a bidirectional graphical model that permits fast inference due to its special bipartite structure. We define the multi-block harmonium to be a generalization of the classical harmonium, or restricted Boltzmann machine (RBM), that describes \( N \) separate blocks of observed variables, \( \{x^1, \cdots, x^N\} \), using a single set of latent variables \( h \). Each block of observed variables corresponds to a different input distribution. Since the harmonium is an energy-based model, the energy function of the multi-block generalization would simply be:

\[
E(y, x^1, \cdots, x^N, h) = -\left(h^T W^1 x^1 + b_1^T x^1 + \cdots + h^T_N W_N x^N + b_N^T x^N\right) - c^T h
\]  

(3.5)

where we assign each observed variable block a unique set of parameters \( \{W^n, b^n\} \) and only require a single bias vector \( c \) for the latent variables \( h \). The multi-block model assigns the probability to the tuple \( (x^1, \cdots, x^N, h) \) as follows:

\[
p(x^1, \cdots, x^N, h) = \frac{e^{-E(x^1,\cdots,x^N,h)}}{Z}, \text{ with } p(x^1, \cdots, x^N) = \frac{1}{Z} \sum_h e^{-E(x^1,\cdots,x^N,h)} \]  

(3.6)

where the joint distribution over \( N \) blocks of variables can be found by marginalizing out the latent variables \( h \). \( Z \) is the partition function (or normalization constant), which ensures that the value assignment under the harmonium graphical model is a valid probability distribution, which is fully defined as \( Z = \sum_{(x^1,\cdots,x^N,h)} e^{-E(x^1,\cdots,x^N,h)} \).

The hybrid (dual-winged) harmonium, or the hybrid restricted Boltzmann machine (HRBM) [81] (also sometimes referred to as the ClassRBM), is a special case of the multi-block generalization, where only two distinct blocks of variables, \( \{x^1, x^2\} \), are to be described by the latent variable set \( h \). Specifically, we define the first observed block to be \( x^1 = e_y \), which is the one-hot (or 1-of-\( C \)) encoding \( e_y = (1_{i=y})^C_{i=1} \) of label \( y \), and \( x^2 = x \). This specification means that the multi-block harmonium is to learn the joint distribution \( p(y, x) \) for the supervised learning scenario. If the categories defined in \( \mathcal{Y} \) are mutually exclusive, we may assume a categorical or multinoulli distribution and employ the softmax as the activation function for reconstructing \( e_y \). If the categories are
not mutually exclusive categories, as in multi-label tasks, we can assume an independent Bernoulli distribution for each category and employ the logistic sigmoid as the activation function. The energy function of the HRBM is simply:

$$E(y, x, h) = -h^T W x - b^T x - c^T h - d^T e_y - h^T U e_y$$

(3.7)

which is derived from the multi-block generalization when we set $W_1 = U$ and $b_1 = d$, $W_2 = W$ and $b_2 = b$, and $W_3 = W_4 = \cdots = W_N = 0$ and $b_3 = b_3 = \cdots = b_N = 0$.

In the HRBM, it is often not possible, just as it is for any harmonium, to compute the joint $p(y, x, h)$ or the marginal $p(y, x)$ due to the intractable partition function $Z$. In order to exactly calculate $Z$, a summation over all possible configurations of the observed and latent variables is required. However, since the HRBM is bipartite, or, in other words, has no intra-layer edges, we may leverage block Gibbs sampling to draw samples of the HRBM’s latent variable layer given the current state of the visible layer (composed of $x$ and $e_y$) and vice versa. This yields conditioning equations for various layers of the model as follows:

$$p(h|y, x) = \prod_j p(h_j|y, x), \text{ with } p(h_j = 1|y, x) = \sigma(c_j + U_j y + \sum_i W_{ji} x_i)$$

(3.8)

$$p(x|h) = \prod_i p(x_i|h), \text{ with } p(x_i = 1|h) = \sigma(b_i + \sum_j W_{ji} h_j)$$

(3.9)

$$p(y|h) = \frac{e^{d_y + \sum_j U_j y h_j}}{\sum_{y^*} e^{d_{y^*} + \sum_j U_{y^*} h_j}}$$

(3.10)

where $\sigma(v) = 1/(1 + e^{-v})$, the logistic link function. Furthermore, to perform classification directly using the HRBM, one uses the definition of the model’s free energy function $F(y, x)$ to compute the conditional distribution:

$$p(y|x) = \frac{e^{-F(y, x)}}{\sum_{y^* \in \{1, \ldots, C\}} e^{-F(y^*, x)}}$$

(3.11)

where $-F(y, x) = (d_y + \sum_j \log(1 + \exp(c_j + U_{jy} + \sum_i W_{ji} x_i)))$.

### 3.3.1.2 Calculating Parameter Updates

The hybrid harmonium is trained leveraging a compound objective loss function that balances a discriminative objective $L_{disc}$ and generative objective $L_{gen}$, defined as follows:
Algorithm 1: Contrastive Divergence: Single update for HRBM generative objective.

Input: training sample \((y_t, x_t)\), HRBM current model parameters \(\Theta\)

// Note that “\(a \leftarrow b\)” indicates assignment and “\(a \sim b\)” indicates \(a\) is sampled from \(b\)

function COMPUTEGENERATIVEGRADIENT(yt, xt, \(\Theta\))

\[
\text{if } y_t = \emptyset \text{ then } \quad \triangleright \text{Obtain a pseudo-label for the unlabeled sample.}
\]

\[
y_t \sim p(y|xt) \quad \text{// Conduct Positive Phase}
\]

\[
y^0 \leftarrow y_t, x^0 \leftarrow x_t, \tilde{h}^0 \leftarrow \sigma(c + Wx^0 + Ue_{y^0})
\]

\[
\text{// Conduct Negative Phase}
\]

\[
\begin{align*}
    h^0 &\sim p(h|y^0, x^0), y^1 \sim p(y|h^0), x^1 \sim p(x|h^0) \\
    \tilde{h}^1 &\leftarrow \sigma(c + Wx^1 + Ue_{y^1})
\end{align*}
\]

\[
\text{// Compute Gradient Update}
\]

\[
\text{for } \theta \in \Theta \text{ do }
\]

\[
\triangledown \leftarrow \frac{\partial}{\partial \theta} \mathbb{E}(y^0, x^0, \tilde{h}^0) - \frac{\partial}{\partial \theta} \mathbb{E}(y^1, x^1, \tilde{h}^1)
\]

return \(\triangledown\)

\[
L_{\text{disc}}(\mathcal{D}_{\text{lab}}) = -\sum_{t=1}^{\left|\mathcal{D}_{\text{lab}}\right|} \log p(y|xt) \quad L_{\text{gen}}(\mathcal{D}_{\text{lab}}) = -\sum_{t=1}^{\left|\mathcal{D}_{\text{lab}}\right|} \log p(y_t, xt)
\]

(3.12) \hspace{1cm} (3.13)

where \(\mathcal{D}_{\text{lab}} = \{(y_t, x_t)\}\) is, again, the labeled training dataset. The gradient for \(L_{\text{disc}}\) may be computed directly, following the general form

\[
\frac{\partial \log p(y_t|\mathbf{x})}{\partial \theta} = -\mathbb{E}_{h|y_t, x_t} \left[ \frac{\partial}{\partial \theta} \left( \mathbb{E}(y_t, x_t, h) \right) \right] + \mathbb{E}_{y, h|x} \left[ \frac{\partial}{\partial \theta} \left( \mathbb{E}(y, x, h) \right) \right]
\]

(3.14)

implemented via direct formulation (see [81] for details) and possibly by adding a mechanism for regularization through either drop-out or drop-connect [84]. The generative gradient follows the form:

\[
\frac{\partial \log p(y_t, \mathbf{x})}{\partial \theta} = -\mathbb{E}_{h|y_t, x_t} \left[ \frac{\partial}{\partial \theta} \left( \mathbb{E}(y_t, x_t, h) \right) \right] + \mathbb{E}_{y, h|x} \left[ \frac{\partial}{\partial \theta} \left( \mathbb{E}(y, x, h) \right) \right]
\]

(3.15)

and, although intractable for any \((y_t, \mathbf{x}_t)\), is approximated via contrastive divergence [40], where the intractable second expectation is replaced by a point estimate using one Gibbs sampling step (after initializing the Markov Chain at the training sample).

In the semi-supervised context, where \(\mathcal{D}_{\text{lab}}\) is small but a large, unlabeled dataset \(\mathcal{D}_{\text{unlab}}\) is available, the HRBM can be further extended to train with an unsupervised
objective $L_{\text{unsup}}$, where negative log-likelihood is optimized according to

$$L_{\text{unsup}}(D_{\text{unlab}}) = - \sum_{t=1}^{|D_{\text{unlab}}|} \log p(x_t).$$  \hspace{1cm} (3.16)$$

The gradient for $L_{\text{unsup}}$ can be simply computed using the same contrastive divergence update for $L_{\text{gen}}$ but incorporating an extra step at the beginning by sampling from the model’s current estimate of $p(y|u)$ for an unlabeled sample $u$. This form of the generative update could be viewed as a form of self-training or Entropy Regularization [85]. The pseudo-code for the online procedure for computing the generative gradient (either labeled or unlabeled example) for a single HRBM is shown in Algorithm 1.

The full multi-objective function for the HRBM is the following weighted summation:

$$L_{\text{semi}}(D_{\text{lab}}, D_{\text{unlab}}) = \gamma L_{\text{disc}}(D_{\text{lab}}) + \alpha L_{\text{gen}}(D_{\text{lab}}) + \beta L_{\text{unsup}}(D_{\text{unlab}})$$ \hspace{1cm} (3.17)$$

where $\alpha$ and $\beta$ are coefficient handles designed to explicitly control the effects that the generative gradients have on the HRBM’s learning procedure. The additional coefficient $\gamma$ is introduced as a means to directly control the effect of the discriminative gradient in model training. Setting $\gamma = 0$ leads to constructing a purely generative model of $D_{\text{lab}}$ and $D_{\text{unlab}}$, and further setting $\beta = 0$ leads to purely supervised generative modeling of labeled dataset $D_{\text{lab}}$. If the target task is classification, then $\gamma$ may be set to any value in $(0, 1]$ (for simplicity, we chose $\gamma = 1$, although future work shall investigate building models with values of this coefficient that shift the balance to models that favor generative features slightly). These free parameters, though making model selection more challenging, offer an explicit means of controlling the extent to which the final parameters discovered are influenced by generative learning [81], much in contrast to simple generative pre-training of neural architectures.

### 3.3.2 The Stacked Boltzmann Expert Network

When multiple hybrid harmoniums are stacked, the resultant architecture will be called a Stacked Boltzmann Expert Network (SBEN) model. To compose an $N$-layer SBEN (or $N$-SBEN), one follows the same greedy, layer-wise procedure of a deep belief network (DBN). However, unlike DBN’s, where stacking RBM’s warrants only a direct feedforward operation (since RBM’s contain only a single set of inputs), modifications must be made to account for the architectural design of the HRBM graphical model. In order to
unify the SBEN architecture while respecting HRBM building block design, one must combine Equations 3.8 and 3.11 to properly compute intermediate data representations during training and prediction (See Figure 3.1). This gives rise to the architecture as depicted in Fig. 3.1 where the representation for the layer above cannot be computed without first obtaining an estimate of the current layer’s $p(y|x)$.

Each HRBM layer of the SBEN is greedily trained using the frozen latent representations of the one below, generated by using lower level expert’s inputs and predictions. The generative objectives (for both unlabeled and labeled samples) used to train the model can be viewed as a form of data-dependent regularization acting on the discriminative learning gradient of each layer. One key advantage of SBEN training is that each layer’s discriminative progress may be tracked directly, since each layer-wise expert is capable of direct classification using Equation 3.11 to compute the conditional $p(y|h_{\text{below}})$. Note that setting the number of hidden layers equal to 1 recovers the original HRBM architecture (a 1-SEBN). One may notice some similarity with the partially supervised, layer-wise procedure of [10] where a simple softmax classifier was loosely coupled with each RBM of a DBN. However, this only served as a temporary mechanism.

---

1One may sample from this prediction vector as one would for hidden activations, however, we found that simply using this mean in forward propagation step yielded best results.
for pre-training whereas the SBEN leverages the more unified framework of the HRBM during and after training. Note that inputs to the SBEN, like the DBN, can be trivially extended \[86, 87\]

A useful property of the SBEN model is that it also contains a generative model “facet” due to its hybrid nature. One could treat this facet as a directed, top-down generative network and generate fantasy samples for specific, clamped class units. In addition, one could generate class-specific probabilistic scores for input features by adapting the procedure in [84] and, since each layer contains class units, potentially uncover the SBEN hierarchy extracted from the data.

### 3.3.3 The Multi-Input Autoasssociator

#### 3.3.3.1 Model Structure

Instead of a fully specified bidirectional graphical model, a far simpler building block could be considered, which we will call the multi-block autoassociator, or multi-block autoencoder. While not a proper graphical model specification, the autoencoder structure will also attempt to describe multiple blocks of variables but with a set of hidden processing elements instead of latent variables. Furthermore, the autoencoder is a directed model, and will instead attempt to learn the reconstruction function of a block set \((x^1, \cdots, x^M)\) from a latent space \(h\). The general model can be decomposed into an encoding function and a decoding function. The encoder is defined as follows:

\[
h = f_\theta(\tilde{x}^1, \cdots, \tilde{x}^N) = \phi(W_1\tilde{x}^1 + \cdots + W_N\tilde{x}^N + c) \tag{3.18}
\]

which represents a nonlinear map of the input blocks to the hidden layer. Note that \(\tilde{x}\) is a corrupted version of \(x\), which is created via the stochastic mapping \(\tilde{x}_t \sim q_D(\tilde{x}_t|x)\), where \(q_D\) is a function that adds noise to the input vector, e.g., randomly masking entries by setting them to zero under a given probability. To perform reconstruction, one would design a single decoder per block, as follows:

\[
x^1 = g^1_\theta(h) = \phi^1(U_1h + b_1), \quad \cdots, \quad x^N = g^N_\theta(h) = \phi^N(U_Nh + b_N) \tag{3.19}
\]

where each decoder can be tailored to its block’s specific distribution by using an appropriate nonlinearity \(\phi(\cdot)\).

The hybrid denoising autoencoder (hDA) can be created from the multi-block autoas-
sociator framework defined by Equations 3.18 and 3.19. As in the HRBM, we define the first observed block to be \( x^1 = e_y \) and \( x^2 = \tilde{x} \). The encoding parameters are \((W_1, W_2, c)\) and the decoding parameters are \((U_1, U_2, b_1, b_2)\). In the autoencoder structure (since no constraint on weight symmetry is required as is the case for harmoniums), it is also possible to tie the weights, \( U_1 = (W_1)^T \) and \( U_2 = (W_2)^T \), where again \( T \) is the transpose operator, to significantly reduce the number of parameters needed to model the data. This structure represents a directed version of the hybrid harmonium defined in the previous section, explicitly specified as follows:

\[
\begin{align*}
    h &= f_\theta(e_y, \tilde{x}) = \phi(W_1 e_y + W_2 \tilde{x} + c) \\
    y &= g^1_\theta(h) = \phi^1(U_1 h + b_1) \\
    x &= g^2_\theta(h) = \phi^2(U_2 h + b_2)
\end{align*}
\]

(3.20) \hspace{1cm} (3.21) \hspace{1cm} (3.22)

where \( y \) (Observe that there is no corrupted version of the target variable vector unlike in the case of \( x \).) is the posterior probability vector produced by the decoder \( g^1_\theta \). The probability \( p(y|h) \) may be selected from \( y \) using the index of the desired class \( y \). One can further simplify the hDA by altering the encoding function to only be a function of the corrupted input block \( \tilde{x} \), or \( f_\theta(\tilde{x}) \). In this case, the hDA would only learn to reconstruct \( x \) and simply predict \( y \) from the hidden representation \( h \). The resulting model would then be:

\[
\begin{align*}
    h &= f_\theta(\tilde{x}) = \phi(W \tilde{x} + c) \\
    y &= g^1_\theta(h) = \phi^1(U_1 h + b_1) \\
    x &= g^2_\theta(h) = \phi^2((W_1)^T h + b_2)
\end{align*}
\]

(3.23) \hspace{1cm} (3.24) \hspace{1cm} (3.25)

where we use parameter-tying for the input block reconstruction model \( U_2 = (W_1)^T \). In this formulation, one can interpret the architecture as a fusion of a single hidden-layer MLP with a single hidden-layer autoencoder, where both models depend on a shared set of input-to-hidden synaptic weights. Alternatively, we could also view the hDA as a variant of the classical reverberating network [88].

3.3.3.2 Calculating Parameter Updates

To compute the updates to the autoassociator parameters we resort to reverse-mode differentiation (or back-propagation of errors). If we take the simplified variant of the
model, where we remove the dependency of the hidden state on the value of the label units, we can calculate the discriminative gradient as we would in a single hidden layer MLP. In this case, we assume a categorical (or multinoulli) distribution over the label units and calculate the gradients of the log likelihood function with respect to model parameters, $\frac{\partial \log p(y_t|x)}{\partial \theta}$.

The generative objective $L_{gen}$, however, proceeds somewhat differently:

$$L_{gen}(D_{lab}) = -\sum_{t=1}^{n_{lab}} x_t \log \hat{x}_t + (1 - x_t) \log(1 - \hat{x}_t)$$

$$L_{unsup}(D_{unlab}) = -\sum_{t=1}^{n_{unlab}} x_t \log \hat{x}_t + (1 - x_t) \log(1 - \hat{x}_t)$$

This is the cross-entropy of two independent multivariate Bernoulli distributions, or the cross-entropy loss. Unlike the HRBM, training an hDA under this generative criterion is notably simpler since back-propagation of errors may be used. A full hDA is trained using the weighted, tri-objective framework described as in the previous section, $L_{semi}$ (Equation 3.17), where its unlabeled objective $L_{unsup}$ uses the same cross-entropy function as $L_{gen}$ but operates on samples drawn from $D_{unlab}$.

The simple hDA, as defined by Equations 3.20, 3.21, and 3.22, differs from the HRBM complement in not only gradient calculation but also in that its unsupervised components do not require a corresponding sample of the model’s estimate of $p(y_t|u_t)$ for an unlabeled sample $u_t$. This is advantageous since generative gradients are computed independently of the existence of a label, saving computational time and avoiding one drawback of self-training schemes: reinforcement of incorrect predictions through model-generated pseudo-labels. If, however, we want to create a directed graph structure more in line with the HRBM graphical model, we must use the first set of Equations 3.23, 3.24, and 3.25.

### 3.3.4 The Hybrid Stacked Denoising Autoencoders Model

In the same greedy, layer-wise fashion as the SBEN, the $N$-layer HSDA ($N$-HSDA) may be composed by stacking hDA’s. By replacing the procedure for generative gradients (Algorithm 1) and the discriminative gradient with the appropriate autoencoder cross-entropy back-propagation alternatives and substituting Equations 3.11 and 3.8 with Equation 3.18 (for computing hidden activities in $COMPUTELATENTREPRESENTATION$ of
Algorithm 2), one may build an HSDA using Algorithm 2 (see Section 3.3.8.1). The most useful property of the HSDA is that required computation for training and prediction may be reduced since dimensionality of each latent representation in auto-encoder architectures can be gradually decreased for upper levels of the network.

One may notice that previously proposed architectures [89, 90] may be recovered from our framework by manipulating the coefficients $\gamma$, $\alpha$, and $\beta$ in the $L_{semi}$ objective function for the HSDA. Both these studies made use of dual-gradient models, which either focused on a hybrid objective that balanced a discriminative and weighted generative objective on a single sample (where the objective collapsed into a single generative objective when no label was available) [90] or where a generative objective was used as the primary objective and combined with a weighted discriminative objective [89]. Since our HSDA architecture can be viewed as a more general formulation of these original models, it is also amenable to their own particular extensions (such as feature growth/pruning, alternative input units for handling different types of data, etc.).

### 3.3.5 Ensembling of Layer-Wise Experts

Both the SBEN and the HSDA models, in addition to unique strengths, possess the interesting property where each layer, or expert, of the model is capable of classification given the appropriate latent representation of the data. This implies that the model is ensemble-like in its very nature but differs from standard ensemble methods where many smaller models are horizontally aggregated using well-established schemes such as boosting [91] or majority voting. Traditional feedforward models simply propagate data through the final network to obtain an output prediction from its penultimate layer for a given $x_t$. In contrast, these hybrid models are capable of a producing a label $y^n_t$ at each level $n$ for $x_t$, resulting from their layer-wise multi-objective training.

To vertically aggregate layer-wise expert outputs, we experimented with a variety of schemes in development, but found that computing a simple mean predictor, $p(y|x)_{ensemble}$ worked best, defined as:

$$p(y|x)_{ensemble} = \frac{1}{N} \sum_{n=1}^{N} p(y|x)_n$$  \hspace{1cm} (3.28)

This ensembling scheme allows for all components of the hybrid model to play a role in classification of unseen samples, perhaps leveraging acquired discriminative “knowledge” at their respective level of abstraction in the model hierarchy to ultimately improve
final predictive performance. This scheme exploits our model’s inherent layer-wise
discriminative ability, which stands as an alternative to coupling helper classifiers [10]
or using “companion objectives” [92] to solve potential exploding gradients in deep
convolution networks for object detection.

3.3.6 Greedily Learning a Stack of Experts

The simplest way to learn an SBEN or HSDA is greedily–learn one building model,
either an HRBM or hDA, one at a time and progressively construct the final model layer
by layer. Each time a new building block model is instantiated and placed on top of
the current stack of experts, the parameters of the lower-level models are frozen and no
longer modified during training. One way to instantiate this procedure is to simply treat
the current \( L - 1 \) models as a fixed, nonlinear transformation and simply run the dataset
through this transform to obtain the higher-level representations needed for training the
current building block at \( L \). This procedure is summarized for an SBEN in Algorithm 2
of Section 3.3.8.1.

3.3.7 The Bottom-Up Algorithm

We can relax the greedy constraint of learning each layer one-by-one, and instead
implement a stack of \( L \) building block models up front. In this case, the first phase of
\( N \)-SBEN or \( N \)-HSDA learning consists of a bottom-up pass where each layerwise model
can be trained using the compound objective function, Equation 3.17. Data samples are
propagated up the model to the layer targeted for layer-wise training using the basic
construction schema outlined in Algorithm 2 in the following section.

3.3.8 The Bottom-Up-Top-Down Algorithm

3.3.8.1 Incorporating Top-Down Signals

Although efficient, the bottom-up procedure, especially the first variant described, which
means that the gradients are computed for each layer-wise HRBM independent of gra-
dient information from other layers of the model. One way we propose to introduce
a degree of joint training of parameters is to incorporate a second phase that adjusts
the SBEN parameters via a modified form of back-propagation. Such a routine can
further exploit the SBEN’s multiple predictors (or entry points) where additional error
Algorithm 2 Greedy, layer-wise construction of an $N$-SBEN, where $N$ is the desired number of layers of latent variables.

**Input:** $D_{\text{train}}$, $D_{\text{unlab}}$, learning rate $\lambda$ and hyper-parameters $\gamma, \alpha, \beta, \text{numSteps}$, and initial model parameters $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_N\}$

**function** CONSTRUCTMODEL($D_{\text{train}}, D_{\text{unlab}}, \lambda, \gamma, \alpha, \beta, \text{numSteps}, \Theta$)

$D_{\text{train}}^{n} \leftarrow D_{\text{train}}$, $D_{\text{unlab}}^{n} \leftarrow D_{\text{unlab}}$ \(\triangleright\) Initialize subsets to low-level representations

for $\Theta_n \in \Theta$ do

$t \leftarrow 0$

while $t \leq \text{numSteps}$ do

$(y_t, x_t) \sim D_{\text{train}}^{n}$ \(\triangleright\) Draw sample from $D_{\text{train}}^{n}$ without replacement

$(u_t) \sim D_{\text{unlab}}^{n}$ \(\triangleright\) Draw sample from $D_{\text{unlab}}^{n}$ without replacement

$(\nabla_{\text{disc}}, \nabla_{\text{gen}}, \nabla_{\text{unsup}}) \leftarrow \text{UPDATELAYER}(y_t, x_t, u_t, \Theta_n)$

$\Theta_n \leftarrow \Theta_n + \lambda(-\gamma \nabla_{\text{disc}} + \alpha \nabla_{\text{gen}} + \beta \nabla_{\text{unsup}})$, $t \leftarrow t + 1$

$D_{\text{train}}^{h} \leftarrow \emptyset$, $D_{\text{unlab}}^{h} \leftarrow \emptyset$

for $(y_t, x_t) \in D_{\text{train}}^{n}$ do \(\triangleright\) Compute latent representation dataset for $D_{\text{train}}^{n}$

$D_{\text{train}}^{h} \leftarrow \text{COMPUTELATENTREPRESENTATION}(y_t, x_t, \Theta_n)$

for $(\emptyset, u_t) \in D_{\text{unlab}}^{n}$ do \(\triangleright\) Compute latent representation dataset for $D_{\text{unlab}}^{n}$

$D_{\text{unlab}}^{h} \leftarrow \text{COMPUTELATENTREPRESENTATION}((\emptyset, u_t, \Theta_n)$

$D_{\text{train}}^{n} \leftarrow D_{\text{train}}^{n}$, $D_{\text{unlab}}^{n} \leftarrow D_{\text{unlab}}^{h}$

**function** UPDATELAYER($y_t, x_t, u_t, \Theta_n$)

$\nabla_{\text{disc}} \leftarrow \text{COMPUTEDISCRIMINATIVEGRADIENT}(y_t, x_t, \Theta_n)$ \(\triangleright\) See [81] for details

$\nabla_{\text{gen}} \leftarrow \text{COMPUTEGENERATIVEGRADIENT}(y_t, x_t, \Theta_n)$ \(\triangleright\) See Algorithm 1

$\nabla_{\text{unsup}} \leftarrow \text{COMPUTEDiscriminativeGrad}(\emptyset, u_t, \Theta_n)$ \(\triangleright\) See Algorithm 1

return $(\nabla_{\text{disc}}, \nabla_{\text{gen}}, \nabla_{\text{unsup}})$

**function** COMPUTELATENTREPRESENTATION($y_t, x_t, \Theta_n$)

$y_t^{h} \leftarrow p(y_t | x_t, \Theta_n)$ \(\triangleright\) Equation 3.11 under the layerwise model

$h_t \sim p(h_t | y_t^{h}, x_t, \Theta_n)$ \(\triangleright\) Equation 3.8 under the layerwise model

return $(y_t, h_t)$

signals may be computed and aggregated while signals are reverse-propagated down the network. We hypothesize that holistic fine-tuning ensures that discriminative information is incorporated into the generative features being constructed in the bottom-up learning step. Furthermore, errors from experts above are propagated down to lower layers, which were initially frozen during the greedy, bottom-up training phase.

Fine-tuning in the context of training an SBEN is different from using a pre-trained MLP that is subsequently fine-tuned with back-propagation. First, since the SBEN is a more complex architecture than an MLP, pre-initializing an MLP would be insufficient given that one would be tossing potentially useful information stored in the SBEN’s class.
Algorithm 3 Top-down fine-tuning of an $N$-SBEN (ensemble back-propagation). Note that “$\cdot$” indicates a Hadamard product, $\xi$ is an error signal vector, the prime superscript indicates a derivative (i.e., $\sigma'$ means derivative function of the sigmoid), and $\bar{z}$ is the symbol for linear pre-activation values.

**Input:** $(x_t, y_t) \in \mathcal{D}$, learning rate $\lambda$ and model parameters $\Theta = \{\Theta_1, \Theta_2, \ldots, \Theta_N\}$

**function** FINETUNE MODEL $(x_t, y_t), \lambda, \Theta$

$\Omega \leftarrow \emptyset, x_n \leftarrow x_t, y_n \leftarrow \emptyset$  \hspace{1cm} $\triangleright$ Initialize list of layer-wise model statistics & variables

// Conduct feed-forward pass to collect layer-wise statistics
for $\Theta_n \in \Theta$ do

$(h_n, z_n, y_n^h, x_n) \leftarrow$ COMPUTE LAYERWISE STATISTICS $(x_n, \Theta_n)$

$\Omega_n \leftarrow (h_n, z_n, y_n^h, x_n), x_n \leftarrow h_n, y_n \leftarrow y_n^h$

// Conduct error back-propagation pass to adjust layer-wise parameters
$\xi_t \leftarrow \emptyset$

for $l \leftarrow N, l ---, \text{while } l \geq 1$ do

$(h_l, z_l, y_l^h, x_l) \leftarrow \Omega[l]$  \hspace{1cm} $\triangleright$ Grab relevant statistics for layer $l$ of model

if $i = N$ then

$(\nabla_{\text{disc}}, \xi_t) \leftarrow$ COMPUTE DISCRIMINATIVE GRADIENT $(y_t, x_l, \emptyset, h_n, \bar{z}, \Theta_l)$

else

$\xi_t \leftarrow \xi_t \cdot \sigma'(\bar{z}_t)$

$(\nabla_{\text{disc}}, \xi_t) \leftarrow$ COMPUTE DISCRIMINATIVE GRADIENT $(y_t, x_l, \xi_t, h_n, \bar{z}, \Theta_l)$

$\Theta_n \leftarrow \Theta_n - \lambda(\nabla_{\text{disc}})$

**function** COMPUTE LAYERWISE STATISTICS $(x_t, \Theta_n)$

$y_t^h \leftarrow p(y_t|x_t, \Theta_n)$  \hspace{1cm} $\triangleright$ Equation 3.11 under the layerwise model

$\bar{z} \leftarrow c + Wx_t + Ue_{y_t}$  \hspace{1cm} $\triangleright$ Can re-use $\bar{z}$ to perform next step

$h_n \sim p(h_t^h | x_t, \Theta_n)$  \hspace{1cm} $\triangleright$ Equation 3.8 under the layerwise model

**return** $(h_t, \bar{z}, y_t^h, x_t)$

**function** COMPUTE DISCRIMINATIVE GRADIENT $(y_t, x_l, \xi_t, h_n, \bar{z}, \Theta_l)$

$o \leftarrow p(y_t | h_n(\Theta_t)), \xi \leftarrow \text{softmax}'(o) \cdot -(y_t/o)$

$\nabla U \leftarrow \xi h_T^T, \nabla_d \leftarrow \xi, \xi \leftarrow U\xi, \xi \leftarrow \xi \cdot \sigma'(\bar{z})$

if $\xi_t \neq \emptyset$ then

$\xi \leftarrow \xi \cdot \xi_t$

$\nabla W \leftarrow \xi x_T^T, \nabla_c \leftarrow \xi, \nabla b \leftarrow 0, \nabla U \leftarrow \nabla U + (\xi e_{y_t}^T), \xi \leftarrow W^T\xi$

**return** $(\nabla \leftarrow (\nabla W, \nabla U, \nabla b, \nabla c, \nabla d), \xi)$

filters (and corresponding class bias vectors) of each layer-wise expert (i.e., $U$ and $d$). Second, merely using the SBEN as an intermediate model ignores the fact the SBEN can already perform classification directly. To avoid losing such information and to fully exploit the model’s predictive ability, we adapt the back-propagation algorithm for
Algorithm 4 The Bottom-Up-Top-Down training procedure for learning an $N$-SBEN.

**Input:** $(x_t, y_t) \in D_{train}$, $(u_t) \in D_{unlab}$, rates $\lambda$ & $\beta$, $\bar{p}$, & parameters $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_N\}$

**function** bottomUpTopDown($(y_t, x_t, u_t, \lambda, \beta, \Theta)$)

- APPLYBOTTOMUPPASS($y_t, x_t, u_t, \lambda, \gamma = 0, \alpha = 1, \beta, \Theta$) $\triangleright$ See [93]
- // Up to two calls can be made to the top-down tuning routine
- FINETUNEMODEL($x_t, y_t, \lambda, \Theta$) $\triangleright$ See Algorithm 3 for details
- $v_t \leftarrow p_{\text{ensemble}}(y|x, \Theta_n)$ $\triangleright$ Calculate pseudo-label probability using Equation 3.28
- **if** $\max[v_t] > \bar{p}$ **then**
  
  $v_t \leftarrow \text{TOONEHOT}(v_t)$ $\triangleright$ Convert to 1-hot vector using argmax of model conditionals

- FINETUNEMODEL($u_t, v_t, \lambda, \Theta$)

training MLP’s to operate on the SBEN, which we shall call ensemble back-propagation since the fine-tuning method propagates error derivatives down the network from many points of entry. Ensemble back-propagation is described in Algorithm 3.

### 3.3.8.2 Pseudo-Joint Learning

The two learning procedures described above may be combined to form the so-called Bottom-Up-Top-Down (BUTD) training algorithm for fully training a deep hybrid architecture. This means that the single bottom-up modification step is followed by a single top-down joint fine-tuning step using the ensemble back-propagation procedure defined in Algorithm 3 for each training time step.

A full top-down phase can consist of up to two calls to the ensemble back-propagation procedure. One is used to jointly modify the SBEN’s parameters with respect to the sample taken from $D_{train}$. A second one is potentially needed to tune parameters with respect to the sample drawn from $D_{unlab}$. For the unlabeled sample, if the highest class probability assigned by the SBEN (using Equation 3.28) is greater than a pre-set threshold (i.e., $\max[p_{\text{ensemble}}(y|u)] > \bar{p}$), a pseudo-label is created for that sample by converting the model’s mean vector to a one-hot encoding. The probability threshold $\bar{p}$ for the potential second call to the ensemble back-propagation routine allows us to incorporate a tunable form of pseudo-labeling [85] into the Bottom-Up-Top-Down learning algorithm.

The high-level view of the BUTD learning algorithm is depicted in Algorithm 4.
3.4 Learning via Coordinated Local Signals

The generative models used for pre-training [10] do not provide “global coordination between the different levels” [33], that is, they are not optimized globally. This is, empirically, sub-optimal [94], and identified as a cause of “shifting representations” [95], where upper layers of a multi-level model are updated using immature latent representations from below. This can lead to unstable learning behavior or worsened generalization performance. Greedily built models can be later jointly tuned, leading to the Deep Boltzmann Machine (DBM, [96]) or the Wake-Sleep algorithm for Deep Belief Networks (DBNs, [97, 98]). Yet, the original (pre-)training difficulty remains.

How can we avoid pre-training or multi-phase learning? One way, while still exploiting the power of representation learning, is to instead solve the hybrid learning problem: force a model to balance multiple supervised and unsupervised learning objectives in a principled manner, at the same time. This approach provided a more practical process for model construction when compared to traditional pre-training, including the ability to track the influence generative learning goals had on the target task objective (minimizing negative log likelihood of the conditional posterior).

However, the models developed so far in this chapter still suffer from some key issues: 1) parameters are learned in a layer-wise fashion, which means that these models are susceptible to the “shifting representations” problem, and 2) the prediction step of these hybrid architectures naively involves basic vertical aggregation.

3.4.1 The Deep Hybrid Boltzmann Machine

3.4.1.1 Model Structure

In this section, we present the deep hybrid Boltzmann machine (DHBM, architecture depicted in Figure 3.2), which is the jointly trained variant of the SBEN (much like the deep Boltzmann machine, or DBM, is the jointly trained variant of the DBN).

Like the SBEN, one could view a DHBM as a stack of HRBM experts. However, the design of the DHBM begins to deviate strongly when the idea of top-down feedback is introduced—the latent variable layer of each HRBM is now conditioned on both the latent variable layer of the expert immediately below and the latent variable layer of the expert immediately above. Furthermore, the aggregate prediction of the full neural system is no longer a naive ensemble average of the layerwise conditional probability estimates.
A three-layer DHBM (or 3-DHBM) is depicted in Figure 3.2, though as indicated by the dotted lines in the figure, this definition easily extends to $L$ layers. With pattern vector input $x = (x_1, \cdots, x_D)$ and its corresponding target variable $y \in \{1, \cdots, C\}$, utilizing two sets of latent variables $h^1 = (h^1_1, \cdots, h^1_{H_1})$ and $h^2 = (h^2_1, \cdots, h^2_{H_2})$ and model parameters $\Theta^m = (W^1, U^1, W^2, U^2)$, the energy of a DHBM is:

\[
E(y, x, h^1, h^2; \Theta) = -h^1^T W^1 x - h^1^T U^1 e_y - h^2^T W^2 h^1 - h^2^T U^2 e_y. \tag{3.29}
\]

The probability that the 3-DHBM assigns to the 4-tuple $(y, x, h^1, h^2)$ is:

\[
p(y, x; \Theta) = \frac{1}{Z} \sum_{h^1, h^2} e^{(E(y, x, h^1, h^2))} \tag{3.30}
\]

where $Z$ is, again, the partition function that ensures a valid probability distribution. The visible and latent states of the 3-DHBM can be computed directly as follows:

\[
p(h^1|y, x, h^2) = \prod_j p(h^1_j|y, x, h^2), \text{ with } p(h^1_j = 1|y, x), \text{ so}
\]

\[
\mu^1_j = \phi^h(U^1_{jy} + \sum_i W^1_{ji}x_i + \sum_k W^2_{kj}h^1_k) \tag{3.31}
\]

\[
p(h^2|y, h^1) = \prod_k p(h^2_k|y, h^1), \text{ with } p(h^2_k = 1|h^1), \text{ so}
\]

\[
\mu^2_j = \phi^h(U^2_{ky} + \sum_j W^2_{kj}h^1_j) \tag{3.32}
\]

\[
p(x|h^1) = \prod_i p(x_i|h^1), \text{ with } p(x_i = 1|h), \text{ so}
\]
\[ \mu_j^0 = \phi^v(\sum_{j} W_{ji}^1 h_j^1) \] (3.33)

\[ p(y|h^1, h^2) = \frac{e^{\sum_{j} v_j h_j^1 + \sum_{j} u_j y h_j^2}}{\sum_{y} e^{\sum_{j} v_j y h_j^1 + \sum_{j} u_j y h_j^2}} \] (3.34)

where the activation \( \phi^v(v) = 1/(1 + e^{-v}) \) (the logistic sigmoid). \( y \) accesses a particular class filter from \( U^l \). To adapt the model to different types of input (i.e., continuous-valued variables), \( \phi^v(v) \), the post-activation function applied to the input units, can be switched to functions such as the rectified linear unit or the identity.

### 3.4.1.2 Heuristic Weight Doubling

Since the conditionals of both the DHBM and DHDA are mutually dependent, to initialize the mean-field, one can treat the network as a simple feedforward model and propagate the activities towards the topmost layer. However, if this heuristic approach is used, all of the weights must be doubled, in order to account for the missing top-down activities [96]. To get the model’s reconstruction of the input and target prediction, several steps of mean-field inference are then performed, cycling through Equations 3.31, 3.32, 3.33, and 3.34.

### 3.4.1.3 Approximate Inference

To better initialize the mean-field inference cycle without resorting to heuristics such as the weight doubling described above, one can instead augment the DHBM or DHDA with a co-model, or auxiliary network.\(^2\) This auxiliary model, in the case of the DHDA and DHBM, is an MLP with the same number of hidden layers (of the same dimensionality) as the target generative model. As in [99], this network, trainable via back-propagation of errors, function approximates the inference process underlying the DHBM or DHDA is used to obtain higher-quality initial states of the latent variables needed to properly run mean-field inference. As long as the assumption that the target generative model’s mean-field parameters do not change too much between each successive learning step holds, we can easily co-train this auxiliary network using samples generated from the mean-field process. Recognition networks have also recently become very popular in training directed generative models [100], allowing us to use variational inference to learn richer, more complex models of data.

\(^2\)Also referred to as a recognition or inference network.
To calculate the posterior \( p(h|x; \Theta) \), we need to use an approximating distribution \( Q(h|x; \mu) \), with mean-field parameters \( \mu \). Instead of maximizing the log likelihood, where the exact approach involves intractable computations, we can use our approximating distribution to instead maximize the variational lower bound of our model following a naive mean-field approach to inference:

\[
\log p(x; \Theta) \geq \sum_h Q(h|x; \mu) \log p(h|x; \Theta) + \mathcal{H}(Q) \tag{3.35}
\]

with \( \mathcal{H}(Q) \) representing the entropy functional. Note that this bound will become tight if and only if \( Q(h|x; \mu) = p(h|x; \Theta) \). Using the fixed-point formulas defined earlier (Equations 3.31, 3.32), we can define the fully factorized mean-field distribution to be:

\[
Q^{MF}(h|x; \mu) = \prod_{j=1}^{|h^1|} q^{MF}(h^1_j) \prod_{k=1}^{|h^2|} q^{MF}(h^2_k) \tag{3.36}
\]

where \( \mu = \{\mu^1, \mu^2\} \) are the mean-field parameters. Inference under the DHBM (and DHDA) requires searching for the value \( \mu \) that maximizes the variational lower bound defined by Equation 3.35 for each sample in the training set given current model parameters \( \Theta \). To find the mean-field values, we can cycle through fixed-point Equations 3.31, 3.32, which form the mean-field inference process. Once the mean-field values have been found, one can compute parameter updates to maximize the lower bound using stochastic approximation.

Using naive mean-field to obtain the necessary statistics in order to learn the hybrid models above is computationally expensive. Instead, a cheaper alternative is to define an inference network, \( Q^{rec}(h|x; \mu) \), with a separate set of parameters, \( \Theta^{rec} = \{R^1, R^2\} \), which are initialized with the same values of the generative model, \( \Theta \), before learning begins. \( Q^{rec}(h|x; \mu) \) is defined over the two sets of latent variables as:

\[
Q^{rec}(h|x; \mu) = \prod_{j=1}^{|h^1|} q^{rec}(h^1_j) \prod_{k=1}^{|h^2|} q^{rec}(h^2_k) \tag{3.37}
\]

where, for \( l = 1, 2 \), \( q^{rec}(h^l_i) = v^l_i \). The recognition network can then activate each
layer of hidden units in a single forward propagation phase:

\[ v_j^1 = \phi \left( 2 \sum_{i=1}^{|x|} R_{ij}^1 v_i \right), \quad \text{and,} \quad v_k^2 = \phi \left( \sum_{j=1}^{|h^1|} R_{jk}^2 v_j^1 \right) \tag{3.38} \]

noting the recognition weights are multiplied by a factor of 2 (except for the recognition parameters of the topmost layer) in order to account for missing top-down input. The modified inference process then runs as follows:

1. Run the inference network given a current sample to guess the hidden layers of the model, \( v = \{v^1, v^2\} \)
2. Initialize the mean-field parameters at the guessed values \( \mu = v \)
3. Run the mean-field process as defined earlier for \( K \) steps (even just a single step works well in practice)

During the learning phase, the above inference process also includes a fourth step that involves adapting the recognition network to make its initial guesses of the mean-field values better. To do so, we will need to minimize the Kullback-Leibler (KL) divergence between the mean-field posterior distribution \( Q_{MF}(h|x; \mu) \) and the factorial posterior distribution defined by \( Q_{rec}(h|x; \mu) \):

\[
KL(Q_{MF}(h|x; \mu)||Q_{rec}(h|x; \mu)) = -\sum_i \mu_i \log v_i - \sum_i (1 - \mu_i) \log(1 - v_i) + C, \tag{3.39}
\]

the gradients of which (with respect to recognition parameters) can be calculated using reverse-mode differentiation. As learning progresses, since the recognition model is trained jointly with the generative model, it will progressively get better at guessing the fixed-point solutions to the mean-field equations defined earlier. At test-time, the hybrid architecture can also quickly generate predictions of the target variable \( y \) by first using the recognition network to infer the latent states of the system and then running a step of mean-field to then generate a prediction of both the input and target variables.

### 3.4.1.4 Calculating Parameter Updates

One simple estimator for calculating the gradients of a hybrid architecture only makes use of two sets of layerwise statistics obtained from the recognition network and running the
Algorithm 5 The estimator for calculating gradients using mean-field Contrastive Divergence for the DHBM. $<>_N$ denotes calculating an expectation over $N$ samples.

Input: $(y, x)$ mini-batch of $N$ samples, parameters $\Theta = \{W_1, W_2, U^1, U^2\}$, initial mean-field statistics $\mu = \{\mu_1, \mu_2\}$ (perhaps those initially given by an inference network $Q^{rec}$), and $M$, or number of Markov chains.

function CALCPARAMGRADIENTS($(y, x), \Theta, \mu, M$)

// Run block Gibbs samplers (one step each)
for each particle $m = 1$ to $M$ do
    Sample $(\tilde{x}_{t+1,m}, \tilde{h}_{t+1,m}, \tilde{y}_{t+1,m})$ given $(\tilde{x}_{t,m}, \tilde{h}_{t,m}, \tilde{y}_{t,m})$.

// Calculate updates to parameters via Contrastive Divergence
$\Delta W^1 \leftarrow (< \mu^1(x)^T >_N - < \tilde{h}^1(\tilde{x})^T >_M )$
$\Delta W^2 \leftarrow (< \mu^2(\mu^1)^T >_N - < \tilde{h}^2(\tilde{h}^1)^T >_M )$
$\Delta U^1 \leftarrow (< \mu^1(y)^T >_N - < \tilde{h}^1(y)^T >_M )$
$\Delta U^2 \leftarrow (< \mu^2(y)^T >_N - < \tilde{h}^2(Y)^T >_M )$

return $\{W^1, W^2, U^1, U^2\}$

mean-field equations for a single step. This results in a set of “positive phase” statistics, which are the data vectors that result from clamping model’s visible units to the input data, and “negative phase” statistics, which result from a single step of the model’s free-running mode. One may then use these two sets of statistics to calculate parameter gradients to move the hybrid model towards more desirable optima using a procedure similar to the one described in the earlier sections of this chapter.

Alternatively, a potentially better estimator of the gradients for the hybrid architecture could employ a approximate maximum likelihood approach to learning which would encourage better mixing rates, or the minimum number of steps before the Markov chain’s distribution is close to its stationary distribution with respect to total variation distance. The key idea behind this procedure is to maintain a set of multiple persistent Gibbs sampling chains in the background from which we may sample during each update of the hybrid architecture.

Using the inference process described in the previous section, we will have obtained a set of mean-field statistics $\mu = \{\mu_1, \mu_2\}$. We start by initializing a set of Markov chains randomly with $M$ fantasy particles, or $\{\tilde{x}_0, \tilde{h}_0, \tilde{y}_0\}, \cdots, \{\tilde{x}_M, \tilde{h}_M, \tilde{y}_M\}$. Note that $y$ in this section represents the one-hot encoded representation of label index $y$. 

54
3.4.2 The Deep Hybrid Denoising Autoencoder

3.4.2.1 Model Structure

The structure of the Deep Hybrid Denoising Autoencoder (DHDA), the joint variant of
the HSDA described earlier, shares many of the same ideas motivating the design of the
DHBM, including the integration of top-down feedback. However, instead of learning via
a Boltzmann-based approach, we shall learn a stochastic encoding and decoding process
through multiple HDA layers jointly.

A 3-layer version of the DHDA is specified by the following set of encoding
$\{f^1_\theta(\hat{x}, \hat{h}^2), f^2_\theta(\hat{h}^1)\}$ and decoding ($g_\theta(\hat{h}^1)$) functions:

\[
\begin{align*}
    h^1 &= f^1_\theta(\hat{x}, \hat{h}^2) = \phi(W^1\hat{x} + (W^2)^T\hat{h}^2) \quad (3.40) \\
    h^2 &= f^2_\theta(\hat{h}^1) = \phi(W^2\hat{h}^1) \quad (3.41) \\
    \hat{x} &= g_\theta(\hat{h}^1) = \phi((W^1)^T\hat{h}^1) \quad (3.42)
\end{align*}
\]

where $\{W^1, W^2\}$ are weight matrices connecting input $x$ to $h^1$ and $h^1$ to $h^2$ respectively
(the superscript $T$ denotes a matrix transpose operation). The output function $o_\theta(y|h^1, h^2)$
needed to generate predictions is calculated via Equation 3.34 much like the DHBM
model. Like the HSDA, the DHDA uses a stochastic mapping function
$\hat{h}^l \sim q_D(\hat{h}^l|h)$ to
corrupt input vectors, i.e., randomly masking entries under a given probability. Since the
DHDA does not use its parameters to model a joint distribution, fewer matrix operations
are required which facilitates faster inference as compared to the DHBM.

3.4.2.2 Approximate Inference

To calculate layer-wise activation values for the DHDA, one follows a process similar
to that of the DHBM. Under mean-field inference, we cycle through Equations 3.40,
3.41, 3.42, and Equation 3.34, initializing the mean-field using the inference network
defined for the DHBM. Instead of sampling each layer-wise set of activations (such as
treating each layer as a vector of Bernoulli distributions as in the DHBM), we apply the
corruption function to the input variables to each fixed-point equation.

\footnote{Note that $h^0 = x$.}
3.4.2.3 Calculating Parameter Updates

Once the denoised layerwise activations are computed, instead of using Contrastive Divergence to compute parameter updates, back-propagation of errors is employed to calculate gradients. The reconstruction cross entropy loss can be used at each layer, requiring us to compute updates by using the measured error between the denoised activation and original (uncorrupted) activation (such as that provided by the very first step of mean-field applied to the recognition network-initialized statistics). The full specification of the learning procedure can be found in [101].

3.4.3 The Hybrid Loss Function

In this section, we will briefly review and cover the approximate global objectives that the DHBM and the DHDA models are ultimately trying to optimize. These objectives are similar in spirit to those of the SBEN and HSDA respectively, but now the key difference is that the DHBM and DHDA are jointly trained to make predictions of the output class targets without the need for an external vertical ensembling mechanism (as was done to compute a mean prediction of the various layerwise experts of the HSDA or SBEN). Figure 3.3 shows the full gamut of models that make up the family of deep hybrid architectures. This includes the SBEN, HSDA, DHBM, and the DHDA.

In short, the DHDA is designed to minimize the following hybrid loss function:

\[ L(D_{lab}, D_{unlab}) = \alpha \left( - \sum_{t=1}^{\left| D_{lab} \right|} \log p(y_t | x_t) + L_{CE}(D_{lab}) \right) + \beta \left( - \sum_{t=1}^{\left| D_{unlab} \right|} \log p(y'_t | x_t) + L_{CE}(D_{unlab}) \right) \] (3.43)

where \( y' \) is a pseudo-label generated for an unlabeled sample given the current DHDA model and \( D_{lab} \) is the set of labeled samples and \( D_{unlab} \) is the set of unlabeled samples. \( \alpha \) and \( \beta \) are coefficient handles that control the effects that supervised and unsupervised gradients have on the model’s learning procedure, respectively. The loss function \( L_{CE} \) is reconstruction cross entropy, defined as

\[ L_{CE}(D) = - \sum_{t=1}^{\left| D \right|} x_t \log z_t + (1 - x_t) \log (1 - z_t). \] (3.44)
Figure 3.3. The DHBM and DHDA architectures compared side-by-side with their layer-wise predecessors, the SBEN and the HSDA. The flow of information to gather layer-wise statistics is indicated by the arrows (which represent an operation, such as matrix-multiplication followed by an element-wise non-linearity), with the appropriate layer parameter matrix and the vector at the arrow’s origin), which are numbered according to the sequential computational steps taken to calculate them. Arrows (or operations) in the same computational step are numbered the same and point to resulting activation values to be calculated. \( v \) corresponds to the recognition network’s \( (Q_{\text{rec}}) \) initial guess for the mean-field while \( \mu \) represents the actual mean-field statistic. Horizontal dotted arrows indicate a copy operation.

The DHDA’s loss (Equation 3.43) reflects the fact the model couples a conditional model learning objective with that of a reconstruction model (dealing with data input directly without label information). This is similar in spirit to the HSDA presented earlier. It is important to note that while we show a global objective that can be used to track model performance, each layer actually makes of a local cross entropy objective (described in detail in [101] in order to make gradient transmission easier.

The DHBM, on the other hand, minimizes the following hybrid objective:

\[
\mathcal{L}(D_{\text{lab}}, D_{\text{unlab}}) = -\alpha \sum_{t=1}^{D_{\text{lab}}} \log p(y_t, x_t) - \beta \sum_{t=1}^{D_{\text{unlab}}} \log p(x_t). \tag{3.45}
\]

The DHBM’s loss (Equation 3.45) maps to the strong multi-level semi-supervised hypothesis given it directly models a joint density, augmented by modeling the input distribution of unlabeled data as well, to yield a useful conditional one.\(^4\)

\(^4\)The DHBM’s hybrid loss function could be augmented with a pair of direct discriminative gradients that additionally minimize \( -\sum_{t=1}^{D} \log p(y|\mathbf{x}_t) \) for both \( D_{\text{lab}} \) and \( D_{\text{unlab}} \) using the ensemble back-propagation algorithm proposed in [102]. This could further improve performance but we did not explore it this thesis.
3.5 The Multi-Level Semi-Supervised Prior

Our motivation for developing hybrid models comes from the semi-supervised learning prior hypothesis of [103], where learning aspects of \( p(x) \) improves the model’s conditional \( p(y|x) \). So long as there is some relationship between \( p(x) \) and \( p(y|x) \), a learner can make use of information afforded by cheaply obtained unlabeled samples in tandem with expensive labeled ones.

In [95], this hypothesis was extended to a multi-level form in order to explain the inner workings and expressiveness of deep hybrid neural architectures, \( L \) layers deep. The Hybrid Stacked Denoising Autoencoders (HSDA) model, a stack of single-layer MLPs coupled with single-layer auto-associators, directly embodies the multi-level view of the semi-supervised learning prior hypothesis, specifically, the “weak multi-level semi-supervised learning hypothesis”\(^5\). According to this hypothesis, learning something about the marginal \( p(h_l) \) along with \( p(y|h_l) \) for \( l = [0, L] \), will improve predictive performance on \( p(y|x) \). The Deep Hybrid Denoising Autoencoder, our proposed joint version of the HSDA presented in Section 3.3.4, also embodies this variant of the multi-level semi-supervised learning hypothesis.

An alternative model is the Stacked Boltzmann Experts Network (SBEN) model, a stack of hybrid restricted Boltzmann machines, where instead each layer-wise expert models a joint distribution at level \( l, p(y, h_l) \). In the SBEN, aggregating the resulting \( p(y|h_l) \) for \( l = [0, L] \), ultimately improves predictive performance on \( p(y|x) \). This we call the “strong multi-level semi-supervised learning hypothesis”\(^6\). The DHBM, our proposed joint version of the SBEN, likewise embodies this hypothesis.

The empirical results that follow will provide some initial positive support of these two variant hypotheses, as each are embodied in the form of one of the various neural architectures described throughout this chapter.

3.6 Experiments

We will now present experimental results on several classification problems in both optical character recognition and document categorization. The experiments will test

---

\(^5\)“Weak” refers to the possibility that \( p(x) \) may be loosely related to \( p(y|x) \), if at all. Learning \( p(x) \) may or may not help with prediction.

\(^6\)“Strong” refers to the fact we know \( p(y|x) \) is related to \( p(y, x) \) (i.e., \( p(y|x) = p(y, x)/p(x) \)). Thus learning the joint will yield information relevant to the conditional.
each and every construction procedure described in this chapter when attempting to learn SBENs or HSDAs for semi-supervised classification tasks.

### 3.6.1 Data

#### 3.6.1.1 Image Classification Data

**CAPTCHA:** This is a synthetic stochastic process implemented for this thesis to generate $28 \times 28$ pixel CAPTCHA images of single characters based on the CAGE model\(^7\). One of 26 English characters may be generated, of either lower or upper-case form in various font, orientations, and scales. The only feature processing applied to the CAPTCHA samples is pixel binarization.

**Stanford OCR:** This dataset contains 52,152 $16 \times 8$ binary pixel images labeled as 1 of 26 letters of the English alphabet. Training ($\sim 2\%$ of source), validation ($\sim 1.9\%$), unlabeled ($\sim 19.2\%$), and test sets ($\sim 77\%$) were created via random sampling without replacement. Furthermore, it was ensured that examples from each class appeared in roughly equal quantities in both the training and validation subsets.

**MNIST:** This dataset\(^8\) contains $28 \times 28$ images with gray-scale pixel feature values in the range of $[0, 255]$. The only preprocessing applied to this set of samples is to normalize the feature values to the range of $[0, 1]$, which is done simply by dividing the pixel values by 255.

#### 3.6.1.2 Document Categorization Data

**WebKB:** The first was the pre-processed WEBKB\(^9\) text collection (i.e., font formatting, stop words removed, terms stemmed, and words with length less than 3 removed) \([104]\), which contains pages from a variety of universities (Cornell, Texas, Washington, and Wisconsin and miscellaneous pages from others). The 4-class classification problem defined by this dataset is to determine if a web-page can be identified as one belonging to a Student, Faculty, Course, or a Project. The dataset was already pre-partitioned into a training set (2,803 samples) and a test set (1,396 web pages), so using the same sampling scheme as the OCR data, we built from the training split a smaller training ($\sim 20.2\%$) and validation ($\sim 14.2\%$) subset, and put the rest into the unlabeled set ($\sim 62.5\%$),

---

\(^7\)https://akiraly.github.io/cage/index.html

\(^8\)Publicly available at the URL: http://yann.lecun.com/exdb/mnist/.

\(^9\)http://www.cs.cmu.edu/ webkb/
discarding 87 document vectors that contained less than 2 active features. The test set contained 1,344 samples, after discarding 52 samples with less than 2 active features. The low-level feature representations were further simplified by using the top 2000 words in the corpus and binarizing the document term vectors.

**Twenty NewsGroups:** The small-scale version of the 20NewsGroup data-set was used for the experiments of this chapter. \(^{10}\) This text corpus contains 16,242 total samples and was already pre-processed to contain only 100 terms. Documents were represented binary-occurrence vectors with tags for the four top-most highest level domains or meta-topics in the newsgroups array.

### 3.6.2 Evaluation of Greedy & Bottom-Up Approaches

#### 3.6.2.1 Models and Baselines

The HSDA and SBEN models were compared to the non-linear, shallow HRBM, which can also be regarded as a \(1\)-SBEN. For a simpler classifier, the incremental version of Maximum Entropy was implemented (which, as explained in [105], is equivalent to a softmax classifier), or *MaxEnt*. Furthermore, the Pegasos SVM was implemented (*SVM*) [106] and extended to follow a proper multi-class scheme [107]. This is the online formulation of the Support Vector Machine, trained via sub-gradient descent on the primal objective followed by a projection step (note that for simplicity we built the linear-kernel version of the model, which is quite fast). Evaluating the Pegasos SVM algorithm in the following experiments allows us to compare our deep semi-supervised models against the incremental version of a strong linear-kernel classifier. To provide context with previously established deep architectures also learnable in a 1-phase fashion like our own, we present results for a simple sparse Rectifier Network, or *Rect*. \(^{11}\) Note that we extended all shallow classifiers and the Rectifier Network to leverage self-training so that they may also learn from unlabeled examples. To do so, we implemented a scheme similar to that of [85] and used a classifier’s estimate of \(p(y|\mathbf{u})\) for an unlabeled sample. However, a 1-hot proxy encoding using the *argmax* of model’s predictor was only created for such a sample if \(\max[p(y|\mathbf{u})] > \bar{p}\). We found that by explicitly controlling pseudo-labeling through \(\bar{p}\) we could more directly improve model performance.

\(^{10}\)The exact data-set can downloaded from: http://www.cs.nyu.edu/~roweis/data.html.

\(^{11}\)Model implementations were computationally verified for correctness when applicable. Since discriminative objectives entailed using an automatic differentiation framework, we checked gradient validity via finite difference approximation.
3.6.2.2 Model Selection

Model selection was conducted using a parallelized multi-setting scheme, where a configuration file for each model was specified, describing a set of hyper-parameter combinations to explore (this is akin to a course-grained grid search, where points of model evaluation are set manually a priori). For the HSDA, SBEN, HRBM, and Rect we varied model architectures, exploring under-complete, complete, and over-complete versions, as well as the learning rate, $\alpha$, and $\beta$ coefficients (holding $\gamma$ fixed at 1.0). If a model was trained using its stochastic form (i.e., HRBM, SBEN, or HSDA), to ensure reproducible model behavior, we ran it in feedforward mean-field, where no sampling steps were taken when data vectors were propagated through a network model when collecting layer-wise predictions (we also found that this yielded lowest generalization error). For the SVM, we tuned its slack variable $\lambda$. The rectifier network’s training also involved using a L2 regularization penalty (0.002), initialization of hidden biases to small positive values ($|N(0, 0.25)|$) [108], and the use of the improved leaky rectifier unit [109].

For the SVM, $\lambda$ was varied in the interval $[0.0001, 0.5]$ while the learning rate for all other models was varied in $[0.0001, 0.1]$. For the HRBM, SBEN, & HSDA, $\beta$ was explored in the interval $[0.05, 0.1]$, and for HRBM, SBEN, & HSDA, $\alpha$ was explored in $[0.075, 1.025]$. The threshold $\bar{p}$ was varied in $[0.0, 1.0]$ and the number of latent layers $N$ for deeper architectures was explored in $[2, 5]$ where we delineate the optimal number with the prefix “N-”.

3.6.2.3 Measuring Performance

For all finite dataset experiments, model performance is reported on the test set using the model with lowest validation-set error found during the training step. Generalization performance was evaluated by calculating classification error, precision, recall, and F-Measure, where F-Measure was chosen to be the harmonic mean of precision and recall, $F1 = 2(\text{precision} \cdot \text{recall})/(\text{precision} + \text{recall})$.

Since the creation of training, validation, and unlabeled subsets was controlled through a seeded random sampling without replacement process, the procedure described above composes a single trial. For the Standford OCR and CAPTCHA datasets, the results we report are 10-trial averages with a single standard deviation from the mean, where each trial used a unique seed value.
Table 3.1. Generalization results on the CAPTCHA simulated dataset. Classification results reported as 10-trial averages with single standard deviation from the mean.

<table>
<thead>
<tr>
<th></th>
<th>Error</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxEnt</td>
<td>0.475±0.010</td>
<td>0.535±0.011</td>
<td>0.524±0.010</td>
<td>0.522±0.010</td>
</tr>
<tr>
<td>SVM</td>
<td>0.461±0.011</td>
<td>0.564±0.010</td>
<td>0.537±0.011</td>
<td>0.526±0.011</td>
</tr>
<tr>
<td>2-Rect [85, 108]</td>
<td>0.365±0.011</td>
<td>0.651±0.011</td>
<td>0.634±0.011</td>
<td>0.627±0.013</td>
</tr>
<tr>
<td>HRBM [81]</td>
<td>0.368±0.009</td>
<td>0.643±0.010</td>
<td>0.631±0.009</td>
<td>0.629±0.009</td>
</tr>
<tr>
<td>5-SBEN</td>
<td>0.324±0.008</td>
<td>0.681±0.009</td>
<td>0.675±0.008</td>
<td>0.671±0.009</td>
</tr>
<tr>
<td>5-HSDA</td>
<td>0.359±0.011</td>
<td>0.650±0.011</td>
<td>0.640±0.011</td>
<td>0.633±0.011</td>
</tr>
</tbody>
</table>

Table 3.2. Generalization results on the Stanford OCR dataset. Classification results reported as 10-trial averages with single standard deviation from the mean.

<table>
<thead>
<tr>
<th></th>
<th>Error</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxEnt</td>
<td>0.425±0.009</td>
<td>0.508±0.006</td>
<td>0.563±0.005</td>
<td>0.512±0.006</td>
</tr>
<tr>
<td>SVM</td>
<td>0.428±0.008</td>
<td>0.504±0.004</td>
<td>0.582±0.011</td>
<td>0.510±0.007</td>
</tr>
<tr>
<td>3-Rect [85, 108]</td>
<td>0.387±0.009</td>
<td>0.549±0.009</td>
<td>0.592±0.014</td>
<td>0.548±0.011</td>
</tr>
<tr>
<td>HRBM [81]</td>
<td>0.399±0.019</td>
<td>0.565±0.009</td>
<td>0.606±0.016</td>
<td>0.552±0.014</td>
</tr>
<tr>
<td>3-SBEN</td>
<td>0.333±0.009</td>
<td>0.602±0.009</td>
<td>0.668±0.009</td>
<td>0.610±0.012</td>
</tr>
<tr>
<td>3-HSDA</td>
<td>0.399±0.012</td>
<td>0.546±0.007</td>
<td>0.601±0.012</td>
<td>0.537±0.009</td>
</tr>
</tbody>
</table>

3.6.2.4 Optical Character Recognition Performance

Two experiments were conducted. The first experiment uses the Stanford OCR dataset. The second experiment makes use of the CAPTCHA stochastic process—which was used in two ways: 1) create a finite dataset of 16,000 samples with (∼ 3.125% in training, ∼ 3.125% in validation, ∼ 31.125% in unlabeled, and ∼ 62.25% in test) and perform an experiment similar to the OCR dataset, and 2) use the process as a controllable data-stream, which allows for compact storage of a complex distribution of image samples.

With respect to the optical character recognition tasks, observe that hybrid incremental architectures have, in general, lower error as compared to non-hybrid ones. In the CAPTCHA experiment (Table 3.1), both the SBEN and HSDA models reduced prediction error over the SVM by nearly 30% and 22% respectively. Furthermore, both models consistently improved over the error the HRBM, with the SBEN model reducing error by ∼ 12%. In the OCR dataset (Table 3.2), the SBEN improves over the HRBM by more than 16% and the SVM by more than 22%. In this case, the HSDA only marginally improves over the SVM model (∼ 6%) and equal to that of an HRBM, the poor performance we attribute to a coarse search through a meta-parameter space window as opposed to an exhaustive grid search.
Table 3.3. Text categorization results on the WEBKB dataset.

<table>
<thead>
<tr>
<th></th>
<th>Error</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxEnt</td>
<td>0.510</td>
<td>0.386</td>
<td>0.387</td>
<td>0.384</td>
</tr>
<tr>
<td>3-SBEN</td>
<td>0.210</td>
<td>0.788</td>
<td>0.770</td>
<td>0.769</td>
</tr>
<tr>
<td>SVM</td>
<td>0.524</td>
<td>0.404</td>
<td>0.378</td>
<td>0.387</td>
</tr>
<tr>
<td>3-HSDA</td>
<td>0.219</td>
<td>0.757</td>
<td>0.780</td>
<td>0.765</td>
</tr>
</tbody>
</table>

Algorithm 6 Online variant of layer-wise construction of a deep hybrid architecture.

Input: \((y_t, x_t) \in D_{train}, (u_t) \in D_{unlab}\), learning rate \(\lambda\), hyper-parameters \(\gamma, \alpha, \beta, \Theta\).

function CONSTRUCT Model\((y_t, x_t, u_t, \lambda, \gamma, \alpha, \beta, \Theta)\)

\[ x^h_t \leftarrow x_t, u^h_t \leftarrow u_t \] \quad \triangleright \text{Initialize samples to low-level representations}

for \(\Theta_n \in \Theta\) do

\[ (\nabla_{disc}, \nabla_{gen}, \nabla_{unsup}) \leftarrow \text{UPDATE LAYER}(y_t, x^h_t, u^h_t, \Theta_n) \]

\[ \Theta_n \leftarrow \Theta_n + \lambda(-\gamma \nabla_{disc} + \alpha \nabla_{gen} + \beta \nabla_{unsup}), t \leftarrow t + 1 \]

// Compute latent representation of data samples

\[ (y_t, x^h_t) \leftarrow \text{COMPUTE LATENT REPRESENTATION}(y_t, x^h_t, \Theta_n) \]

\[ (\emptyset, u^h_t) \leftarrow \text{COMPUTE LATENT REPRESENTATION}(\emptyset, u^h_t, \Theta_n) \]

3.6.2.5 Text Classification Performance

With respect to the WEBKB dataset, over the MaxEnt model (which slightly outperformed the SVM itself), we observed a \(\sim 57\%\) improvement in error for the HSDA and \(\sim 58\%\) for the SBEN (Table 3.3). Note that the rectifier network is competitive, however, in both image-based experiments, the SBEN model outperforms it by more than 11\% on CAPTCHA and nearly 14\% on OCR.

3.6.2.6 Learning Models Online

In the online learning setting, samples from \(D_{unlab}\) may not be available at once and instead are available at a given rate in a stream for a single time instant (we chose to experiment with one example presented at a given iteration and only constant access to a \(|D_{train}| = 500\)). In order to train a deep architecture in this setting, while still exploiting the efficiency of a greedy, layer-wise approach, one may remove the “freezing” step of Algorithm 2 and train all layers dis-jointly in an incremental fashion as opposed to a purely bottom-up approach. Using the same sub-routines as depicted in Algorithm 2, this procedure may be implemented as shown in Algorithm 6, effectively using a single bottom-up pass to modify model parameters. This approach adapts the training of hybrid
architectures, such as the SBEN and HSDA, to the online learning setting.

As evidenced by Fig. 3.4, it is possible to train the layer-wise experts of a multi-level hybrid architecture simultaneously and still obtain a gain in generalization performance over a non-linear, shallow model such as the HRBM. The HRBM settles at an online error of 0.356 whereas the 5-HSDA reaches an error of 0.327 and the 5-SBEN an error of 0.319 in a 10,000 iteration sweep. Online error was evaluated by computing classification error on the next 1,000 unseen samples generated by the CAPTCHA process.

While the simultaneous greedy training used in this experiment allows for unified construction of a deep hybrid model when faced with a data stream, we note that instability may occur in the form of “shifting representations”. This is where an upper level model is dynamically trained on a latent representation of a lower-level model that has not yet settled since it has not yet seen enough samples from the data distribution.

### 3.6.3 Evaluation of the Bottom-Up-Top-Down Algorithm

For text-based classification, a dominating model is the support vector machine (SVM) [110] with many useful innovations to yet further improve its discriminative performance [111]. When used in tandem with prior human knowledge to hand-craft good features, this simple architecture has proven effective in solving practical text-based tasks, such as academic document classification [112]. However, while model construction may be fast (especially when using a linear kernel), this process is costly in that it requires a great deal
of human labor to annotate the training corpus. The hybrid approaches developed in this chapter provide a means for improving classification performance when labeled data is in scarce supply, learning structure and regularity within the text to reduce classification error incrementally.

We next investigated the performance of the BUTD algorithm discussed in Section 4 in the context semi-supervised text categorization. Model performance was evaluated on the WebKB data-set and the small-scale version of the 20NewsGroup data-set. Setup and preprocessing details can be found earlier in this chapter.

For both data-sets, we evaluated model generalization performance using a stratified 5-fold cross-validation (CV) scheme. For each possible train/test split, we automatically partitioned the training fold into separate labeled, unlabeled, and validation subsets using stratified random sampling without replacement. Generalization performance was evaluated by estimating classification error, average precision, average recall, and average F-Measure, where F-Measure was chosen to be the harmonic mean of precision and recall,

\[ F_1 = \frac{2(\text{precision} \cdot \text{recall})}{\text{precision} + \text{recall}}. \]

### 3.6.3.1 Models and Baselines

We evaluated the BUTD version of our model, the 3-SBEN,BUTD, as described in Algorithm 2. For simplicity, the number of latent variables at each level of the SBEN was held equal to the dimensionality of the data (i.e., a complete representation). We compared this model trained with BUTD against a version utilizing only the bottom-up phase (3-SBEN,BU) as in Ororbia et al. (2015). Both SBEN models contained 3 layers of latent variables.

We compared the proposed models against an array of baseline classifiers. We implemented an incremental version of Maximum Entropy, or MaxEnt-ST (which, as explained in Sarikaya et al., 2014, is equivalent to a softmax classifier). Furthermore, we used our implementation of the Pegasos algorithm (SVM-ST) [106] which was extended to follow a proper multi-class scheme [107]. This is the online formulation of the SVM, trained via sub-gradient descent on the primal objective followed by a projection step (for simplicity, we opted to use a linear-kernel). Additionally, we implemented a semi-supervised Bernoulli Naive Bayes classifier (NB-EM) trained via Expectation-Maximization as in [113]. We also compared our model against the HRBM [114] (effectively a single layer SBEN), which serves as a powerful, non-linear shallow classifier in of itself, as well as a 3-layer sparse deep Rectifier Network [108], or Rect, composed
of leaky rectifier units.

All shallow classifiers (except NB-EM and the HRBM) were extended to the semi-supervised setting by leveraging a simple self-training scheme in order to learn from unlabeled data samples. The self-training scheme entailed using a classifier’s estimate of \( p(y|u) \) for an unlabeled sample and, if \( \max[p(y|u)] > \bar{p} \), we created a 1-hot proxy encoding using the \( \text{argmax} \) of model’s predictor, where \( \bar{p} \) is a threshold meta-parameter. Since we found this simple pseudo-labeling approach, similar in spirit to [85], to improve the results for all classifiers, and thus we report all results utilizing this scheme. 12 All classes of models (SBEN, HRBM, Rect, SVM-ST, MaxEnt-ST, NB-ST) were subject to the same model selection procedure described in the next section.

3.6.3.2 Model Selection

Model selection was conducted using a parallelized multi-setting scheme, where a configuration file for each model was specified, describing a set of hyper-parameter combinations to explore (this is akin to a course-grained grid search, where the points of model evaluation are set manually a priori). For the SBEN’s, we varied the learning rate \((0.01, 0.25]\) and \(\beta\) coefficient \(\{0.1, 1.0\}\) and experimented with stochastic and mean-field versions of the models 13 (we found that mean-field did slightly better for this experiment and thus report the performance of this model). The HRBM’s meta-parameters were tuned using a similar set-up to [81] with learning rate varied in \((0.01, 0.25]\), \(\alpha\) in \((0.1, 0.5]\), and \(\beta\) in \(\{0.01, 0.1\}\). For the SVM-ST algorithm, we tuned its slack variable \(\lambda\), searching in the interval \([0.0001, 0.5]\), for MaxEnt-ST its learning rate in \([0.0001, 0.1]\), and for \(\bar{p}\) of all models (shallow and deep) that used pseudo-labeling we searched the interval \([0.1, 1.0]\). All models of all configurations were trained for a 10,000 iteration sweep incrementally on the data and the model state with lowest validation error for that particular run was used. The SBEN, HRBM, and Rect models were also set to use a momentum term of 0.9 (linearly increased from 0.1 in the first 1000 training iterations) and the Rect model made use of a small \(L1\) regularization penalty to encourage additional hidden sparsity. For a data-set like the 20NewsGroup, which contained a number of unlabeled samples greater than training iterations, we view

---

12All model implementations were computationally verified for correctness when applicable. Since most discriminative objectives followed a gradient descent optimization scheme and could be realized in an automatic differentiation framework, we checked gradient validity via finite difference approximation.

13Mean-field simply means no sampling steps were taken after computing probability vectors, or “means” in any stage of the computation.
our schema as simulating access to a data-stream, since all models had access to any
given unlabeled example only once during a training run.

### 3.6.3.3 Text Classification Performance

Our first experiment, using the WebKB data-set, explored classification error as a function
of labeled data subset cardinality (Figure 3.5). In this setup, we repeated the stratified
cross-fold scheme for each possible labeled data subset size, comparing the performance
of the SVM model against 3-SBEN,BU (blue dotted curve) and 3-SBEN,\textit{BUTD} (green
dash-dotted curve). Note that as the number of labeled examples increases (which entails
greater human annotation effort) all models improve, nearly reaching 90% accuracy.
However, while the performance difference between models becomes negligible as the
training set becomes more supervised, as expected, it is in the less scarce regions of the
plot we are interested in. For small proportions, both variants of the SBEN outperform
the SVM, and furthermore, the SBEN trained via full \textit{BUTD} can reach lower error,
especially for the most extreme scenario where only 8 labeled examples per class are
available. We notice a bump in the performance of \textit{BUTD} as nearly the whole training
set becomes labeled and posit that since the \textit{BUTD} involves additional pseudo-labeling
steps (as in the top-down phase), there is greater risk of reinforcing incorrect predictions
in the pseudo-joint\footnote{The phrase “pseudo-joint” is used to differentiate a model that has all its parameters trained jointly from
our own, where only the top-down phase of \textit{BUTD} introduces any form of joint parameter modification.} tuning of layerwise expert parameters. For text collections where
most of the data is labeled and unlabeled data is minimal, only a simple bottom-up pass
is needed to learn a good hybrid model of the data.

The next set of experiments was conducted with only 1% of the training sets labeled.
As shown in Tables 3.4 and 3.5, the deep hybrid architecture trained via \textit{BUTD} outper-
forms all other models with respect to all performance metrics. While the SBEN trained
with simply an online bottom-up performs significantly better than the SVM model, we
note a further reduction of error using our proposed \textit{BUTD} training procedure. The
additional top-down phase serves as a mechanism for unifying the layer-wise experts,
where error signals for both labeled and pseudo-labeled examples increase agreement
among all model layer experts.

For the 20NewsGroup data-set, Ia simple experiment was conducted to uncover some
of the knowledge acquired by our model with respect to the target categorization task. We
applied the mechanism from [81] to extract the variables that are most strongly associated
Figure 3.5. Mean CV generalization performance as a function of labeled sample subset size (using 200 features).

with each of the clamped target variables in the lowest layer of a BUTD-trained SBEN. The top-scored terms associated with each class variable are shown in Table 3.6, using the 10 hidden nodes most highly triggered by the clamped class node, in a model trained on all of the 20NewsGroup data using a model configuration determined from CV results for the 20NewsGroup data-set reported in this chapter. Since the SBEN is a composition of layer-wise experts each capable of classification, we note that this procedure could be applied to each level to uncover which unobserved variables are most strongly associated with each class target. We speculate that this could serve as the basis for uncovering the model’s underlying learnt hierarchy of the data and useful for knowledge extraction, a subject for future work in analyzing black box neural models such as our own.

3.6.4 Evaluation of Joint Hybrid Training

To test the performance of our proposed jointly trained hybrid architectures, we make use of the Massive Online Analysis framework (MOA, [115], 2010) to simulate the process
of online learning, where the learner does not have access to a static, finite data-set of i.i.d. labeled samples but rather is presented with mini-batches of samples (we chose $N = 20$ samples at time) generated from a continuous non-stationary flow of non-i.i.d data. We extended the evolving processes to the semi-supervised setting by enforcing that generated mini-batches are mixed in the sense that, on average, only 10% of the samples were labeled at any time. To learn effectively, the learner must make use of these labeled samples as best as it can while also self-training from unlabeled samples.

Specifically, the LED and Waveform streams were leveraged, in particular the versions that featured customizable feature concept drift and injection of feature noise to make
learning more difficult over time. The LED stream task entails predicting 1 of 10 possible digits on an LED panel given 24 noisy features (percentage of noise applied was set to 10%, and 4 features in the space experienced concept drift). The Waveform task involves predicting 1 of 3 waveform types given 40 noisy real-valued attributes (these features were additionally normalized to lie in the range $[0, 1]$, and 10 features in this space experienced concept drift). For both data-streams, we investigate model performance over a 1,000,000 iteration sweep.

We compare 4 models: 1) the rectifier network trained via pseudo-labeled back-propagation of [85], 2) a sigmoidal DHBM trained via MF-CD, 3) a sigmoidal DHBM trained via SAP (using 10 fantasy particles), and 4) a sigmoidal DHDA trained via MF-BP with corruption probability $p = 0.15$. For both streams, all models consisted of complete-representation architectures (the LED stream used $24 - 24 - 24 - 24 - 24 - 10$
and the Waveform stream used \(40 - 40 - 40 - 40 - 40 - 40 - 3\). All models used a drop-out probability of \(p = 0.5\), a learning rate \(\lambda = 0.051\), \(\alpha = 1.0\), and \(\beta = 0.1\) (we note that adaptive hyper-parameter schedules would be highly suitable to this learning setting and will be investigated in future work).

The results we report are 10-trial averages of each model’s following the prequential method for evaluating statistical learning models for streaming scenarios. Specifically, we make use of prequential error with a weighted forgetting factor \(\alpha_{err} = 0.995\) (following the discussion of admissible values in [116], 2012). This error metric is calculated (at time \(i\)) via the following equation:

\[
P_\alpha(i) = \frac{\sum_{k=1}^{i} \alpha^{i-k} L(y_k, \hat{y}_k)}{\sum_{k=1}^{i} \alpha^{i-k}}, \text{ with } 0 \ll \alpha \leq 1.
\] (3.46)

[116] showed that metrics with forgetting mechanisms were more appropriate for evolving data stream settings (as compared to simple predictive sequential error). In particular, the one we use in this work is memoryless and thus advantageous to use in streaming settings over other alternatives such as sliding window prequential error. For each model, we show an error curve for the conditions when, on average, only 10% of samples of a mini-batch at given time-step are labeled (Figure 3.6.4).

For both the LED and Waveform data-stream experiments, our proposed hybrid architectures consistently outperform the pseudo-labeled MLP. However, we note that it is the \texttt{DHBM\_MF} that performs the best (and not the \texttt{DHBM\_SAP}), maintaining a consistently lower error in the face of both evolving data-streams (with the DHDA also performing reasonably well). We speculate that the reason the SAP estimator does not help improve (and in fact, appears to hurt) performance is tied to the evolutionary nature of the input distribution itself. SAP uses additional fantasy particles to better explore the DHBM’s fantasy distribution and generally exhibit improved results over 1-step Contrastive Divergence in static data-set settings (where multiple epochs are possible). However, since our input changes with time, it becomes more challenging to manipulate the model’s fantasy distribution to match the input distribution. The simpler mean-field \texttt{CD} approach, where the negative phase is simply 1 step away from the positive data-driven one, may simply facilitate an easier learning problem and thus allow for better adaptation of model parameters to handle changes in the input distribution.

Nonetheless, while all models ultimately experience fluctuations in error as the distributions change, we see that the \texttt{DHBM\_MF} is even able to begin recovering in
the Waveform experiment (roughly at around 60,000 iterations). We observe that in the long-run, the MLP model ultimately performs quite poorly. While the MLP is also semi-supervised, one reason behind its failure may be simply that its design is to make use of information useful for learning a discriminative mapping (or rather a conditional model $p(y|x)$). This is the case for both of its gradients (supervised and unsupervised). We argue that the key to our models’ success is that they attempt to directly learn information about the input distribution itself in addition to conditional information. The information afforded from including a generative perspective of the data seems to better equip hybrid architectures for input distributions that change with time than those without.

Additionally, it is likely that at certain time steps mini-batches presented to a learner are entirely unlabeled (as in our setting). It is here that semi-supervised models that also attempt to model the input distribution directly have an advantage over those that do not. In this case, the pseudo-labeled MLP, which relies on the strength of its supervised gradient, is more likely to succumb to the problem most often associated with self-training schemes: reinforcing incorrect predictions.

### 3.7 Conclusions

This chapter developed a family of deep hybrid neural models designed to handle learning problems when labels are scarce. These architectures, unified under multi-level, semi-supervised prior hypothesis, include the Stacked Boltzmann Expert Network (SBEN), the Hybrid Stacked Denoising Autoencoders (HSDA) model, the deep hybrid Boltzmann machine (DHBM), and the deep hybrid denoising autoencoder (DHDA). The key idea behind these models is to balance the goal of learning a generative model of the data while simultaneously extracting discriminative regularity to perform useful classification. The multi-objective function allows for explicit control over generative and discriminative objectives and the learning framework facilitates easy tracking of discriminative model performance.

Several methods for constructing these deep hybrid architectures were presented, including one that is greedy and layer-wise, another that is bottom-up and layer-wise but not greedy, an approach called the *Bottom-Up-Top-Down* learning algorithm for pseudo-joint training of parameters, and a full joint optimization approach.

Following is a brief summary of the advantages and disadvantages of using the hybrid connectionist models developed in this chapter for semi-supervised learning.
3.7.1 Advantages

Deep hybrid architectures combine the efficient greedy, layer-wise construction of [117] with a multi-objective learning approach. To avoid a two-phase procedure, the goal of learning a generative model of the data is balanced with extracting discriminative regularity to perform useful classification. More importantly, this approach facilitates more explicit control over the multiple objectives involved. Additionally, we presented a vertical aggregation scheme, *layer-wise ensembling*, for generating predictions that exploit discriminative knowledge acquired at all levels of abstraction defined by the architecture’s hierarchical form. The framework allows for explicit control over generative and discriminative objectives as well as a natural scheme for tracking layer-wise learning. More importantly, these models can be constructed incrementally, may potentially able to adapt to “in-the-wild” samples, and are capable of more fully embracing the “...unreasonable effectiveness of data” [118].

3.7.2 Disadvantages (and Solutions)

Though promising, these models are not without potential limitations. First, there is the danger of “shifting representations” if using Algorithm 6 for online learning. To combat this, samples could be pooled into mini-batch matrices before computing gradients and minimize some of the noise of online error-surface descent. Alternatively, all layer-wise experts could be extended temporally to Conditional RBM-like structures, potentially improving performance as in [119]. Second, additional free parameters were introduced that require tuning, creating a more challenging model selection process for the user. This may be alleviated with a parallelized, automated approach, however, a model that adapts its objective weights during the learning process would be better, altering its hyper-parameters in response to error progress on data subsets. Our frameworks may be augmented with automatic latent unit growth for both auto-encoder [90] and Boltzmann-like variants [120] or perhaps improved by “tying” all layer-wise expert outputs together in a scheme like that in [121].
Chapter 4  The Differential State Framework for Capturing Long-Term Dependencies

4.1 Introduction

The previous chapter focused on the value of dual discriminative-generative models and introduced the concept of globally coordinated local learning rules. In this chapter, we now turn our attention to a different but very important mechanism that an adaptive neural system will require if it is to ultimately learn continually—the ability to learn longer-term dependencies in order sequences of data points. Capturing useful information across long time lags is a critical and difficult problem for temporal neural models. Existing, popular architectures, such as Long-Short-Term-Memory (LSTM, [122]) or the Neural Turing Machine [123], that try to address the issue are often complex and costly to train.

This chapter will develop the Differential State Framework (DSF), one possible way to alleviate the difficulty of capturing longer-term dependencies in data sequences. DSF is designed with the goal of deriving simple and high-performing temporal neural models and furthermore unifies previously introduced gated architectures. DSF models maintain longer-term memory by learning to interpolate between a fast-changing data-driven representation and a slowly changing, implicitly stable state. We will derive a very simple neural temporal model using DSF—the Delta-RNN—which, as we will show empirically, can perform comparably to or even outperform in some instances, complex gated architectures such as the LSTM. Notably, the Delta-RNN requires hardly any
more parameters than a classical, simple recurrent network—giving us the insight that longer-term memory might be emulated in simple yet powerful modifications of a neural model’s internal state.

## 4.2 On Longer-Term Memory

Recurrent neural networks are increasingly popular models for sequential data. The simple recurrent neural network (RNN) architecture [124] is, however, not suitable for capturing longer-distance dependencies. Architectures that address this shortcoming include the Long Short-Term Memory (LSTM, [125]), the Gated Recurrent Unit (GRU, [126, 127]), and the structurally constrained recurrent network (SCRN, [128]). While these can capture some longer-term patterns (20 to 50 words), their structural complexity makes it difficult to understand what is going on inside. One exception is the SCRN architecture, which is by design simple to understand. It shows that the memory acquired by complex LSTM models on language tasks does correlate strongly with simple weighted bags-of-words. This demystifies the abilities of the LSTM model to a degree: while some authors have suggested that the LSTM understands the language and even the thoughts being expressed in sentences [129], it is arguable whether this could be said about a model that performs equally well and is based on representations that are essentially equivalent to a bag of words.

One property of recurrent architectures that allows for the formation of longer-term memory is the self-connectedness of the basic units: this is most explicitly shown in the SCRN model, where one hidden layer contains neurons that do not have other recurrent connections except to themselves. Still, this architecture has several drawbacks: one has to choose the size of the fully connected and self-connected recurrent layers, and the model is not capable of modeling non-linearities in the longer-term memory component.

In this chapter, we aim to increase representational efficiency, i.e., the ratio of performance to acquired parameters. We simplify the model architecture further and develop several variants under the Differential State Framework, where the hidden layer state of the next time step is a function of its current state and the delta change is computed by the model. We do not present the Differential State Framework as a model of human memory for language. However, we point out its conceptual origins in Surprisal Theory [130–132], which posits that the human language processor develops complex expectations of future words, phrases, and syntactic choices, and that these
expectations and deviations from them (surprisal) guide language processing, e.g., in reading comprehension. How complex the models are (in the human language processor) that form the expectation is an open question. The cognitive literature has approached this with existing parsing algorithms, probabilistic context-free grammars, or n-gram language models. We take a connectionist perspective. The Differential State Framework proposes to not just generatively develop expectations and compare them with actual state changes caused by observing new input; it explicitly maintains gates as a form of high-level error correction and interpolation. An instantiation, the Delta-RNN, will be evaluated as a language model, and we will not attempt to simulate human performance such as in situations with garden-path sentences that need to be reanalyzed because of costly initial mis-analysis.

4.3 The Differential State Framework

In this section, we will describe the proposed Differential State Framework (DSF) as well as several concrete implementations one can derive from it.

4.3.1 Slow and Fast States

The most general formulation of the architectures that fall under DSF distinguishes two forms of the hidden state. The first is a fast state, which is generally a function of the data at the current time-step and a filtration (or summary function of past states). The second is a slow state, or data-independent state. This concept can be specifically viewed as a composition of two general functions, formally defined as follows:

\[
\begin{align*}
    h_t &= g_\Theta(x_t, M_{t-1}) \\
    &= f_\psi[g_\theta(x_t, M_{t-1}), M_{t-1}]
\end{align*}
\]  

(4.1)

where \(\Theta = \{\theta, \psi\}\) are the parameters of the state-machine and \(M_{t-1}\) is the previous latent information the model is conditioned on. In the case of most gated architectures, \(M_{t-1} = h_{t-1}\), but in some others, as in the SCRN or the LSTM, \(M_{t-1} = \{h_{t-1}, c_{t-1}\}\)\(^1\) or could even include information such as de-coupled memory, and in general will be updated as symbols are iteratively processed. We define \(g_\theta(\cdot)\) to be any, possibly

\(^1\)c_t refers to the “cell-state” as in [122].
complicated, function that maps the previous hidden state and the currently encountered data point (e.g. a word, subword, or character token) to a real-valued vector of fixed dimensions using parameters $\theta$. $f \psi(\cdot)$, on the other hand, is defined to be the outer function that uses parameters $\psi$ to integrate the fast-state, as calculated by $g \theta(\cdot)$, and the slowly-moving, currently un-transformed state $h_{t-1}$. In the sub-sections that follow, we will describe simple formulations of these two core functions and, later in this chapter, we will show how currently popular architectures, like the LSTM and various simplifications, are instantiations of this framework. The specific structure of Equation 4.1 was chosen to highlight that we hypothesize the reason behind the success of gated neural architectures is largely because they have been treating the next-step prediction tasks, like language modeling, as an interaction between two functions. One inner function focuses on integrating observed samples with a current filtration to create a new data-dependent hidden representation (or state “proposal”) while an outer function focuses on computing the difference, or “delta”, between the impression of the sub-sequence observed so far (i.e., $h_{t-1}$) with the newly formed impression. For example, as a sentence is iteratively processed, there might not be much new information (or “surprisal”) in a token’s mapped hidden representation (especially if it is a frequently encountered token), thus requiring less change to the iteratively inferred global representation of the sentence. However, encountering a new or rare token (especially an unexpected one) might bias the outer function to allow the newly formed hidden impression to more strongly influence the overall impression of the sentence, which will be useful when predicting what token/symbol will come next. Later in this chapter, we will present a small demonstration using one of the trained word-models to illustrate the intuition just described.

In the sub-sections to follow, we will describe the ways we chose to formulate $g \theta(\cdot)$ and $f \psi(\cdot)$ in the experiments of this chapter. The process we followed for developing the concrete implementations of $g \theta(\cdot)$ and $f \psi(\cdot)$ involved starting from the simplest possible form using the fewest (if any) possible parameters to compose each function and testing it in preliminary experiments to verify its usefulness.

It is important to note that Equation 4.1 is still general enough to allow for future design of more clever or efficient functions that might improve the performance and

---

$^2$One way to extract a “sentence representation” from a temporal neural language model would be to simply to take the last hidden state calculated upon reaching a symbol such as punctuation (e.g., period or exclamation point). This is sometimes referred to as encoding variable-length sentences or paragraphs to a real-valued vector of fixed dimensionality.
long-term memory capabilities of the framework. More importantly, one might view the parameters $\psi$ that $f_\psi(\cdot)$ uses as possibly encapsulating structures that can be used to store explicit memory-vectors, as is the case in stacked-based RNNs [133, 134] or linked-list-based RNNs [134].

### 4.3.2 Instantiating The Delta Recurrent Neural Network

#### 4.3.2.1 Forms of the Outer Function

Keeping $g_\theta(\cdot)$ as general as possible, here we will describe several ways one could design $f_\psi(\cdot)$, the function meant to decide how new and old hidden representations will be combined at each time step. We will strive to introduce as few additional parameters as necessary and experimental results will confirm the effectiveness of our simple designs.

One form that $f_\psi(\cdot)$ could take is a simple weighted summation, as follows:

$$h_t = f_\psi[g_\theta(x_t, h_{t-1}), h_{t-1}] = \Phi(\gamma[g_\theta(x_t, h_{t-1})] + \beta h_{t-1})$$ (4.2)

where $\Phi(\cdot)$ is an element-wise activation applied to the final summation and $\gamma$ and $\beta$ are bias vectors meant to weight the fast and slow states respectively. In Equation 4.2, if $\gamma = \beta = 1$, no additional parameters have been introduced making the outer function simply a rigid summation operator followed by a non-linearity. However, one will notice that $h_{t-1}$ is transmitted across a set of fixed identity connections in addition to being transformed by $g_\theta(\cdot)$.

While $\gamma$ and $\beta$ could be chosen to be hyper-parameters and tuned externally (as sort of per-dimension scalar multipliers), it might prove to be more effective to allow the model to learn these coefficients. If we introduce a vector of parameters $r$, we can choose the fast and slow weights to be $\gamma = (1 - r)$ and $\beta = (r)$, facilitating simple interpolation. Adding these negligibly few additional parameters to compose an interpolation mechanism yields the state-model:

$$h_t = \Phi((1 - r) \otimes g_\theta(x_t, h_{t-1}) + r \otimes h_{t-1}).$$ (4.3)

Note that we define $\otimes$ to be the Hadamard product. Incorporating this interpolation mechanism can be interpreted as giving the Differential State Framework model a flexible
mechanism for mixing various dimensions of its longer-term memory with its more localized memory. Interpolation, especially through a simple gating mechanism, can be an effective way to allow the model to learn how to turn on/off latent dimensions, potentially yielding improved generalization performance, as was empirically shown by [135].

Beyond fixing $r$ to some vector of pre-initialized values, there are two simple ways to parametrize $r$:

\[ r = \frac{1}{1 + e^{\beta r}} \text{, or} \]

\[ r = \frac{1}{1 + e^{\beta(W x_t + b_r)}} \] (4.5)

where both forms only introduce an additional set of learnable bias parameters, however Equation 4.5 allows the data at time step $t$ to interact with the gate and thus takes into account additional information from the input distribution when mixing stable and local states together. Unlike [135], we constrain the rates to lie in the range $[0, 1]$ by using the logistic link function, $\sigma(v) = \frac{1}{1 + e^{-v}}$, which will transform the biases into rates much like the rates of the SCRN. We crucially choose to share $W$ in this particular mechanism for two reasons: 1) we avoid adding yet another matrix of input to hidden parameters and, much to our advantage, reuse the computation of the linear pre-activation term $W x_t$, and 2) additionally coupling the data pre-activation to the gating mechanism will serve as further regularization of the input-to-hidden parameters (by restricting the amount of learnable parameters, much as in classical autoencoders). Two error signals, $\frac{\partial r}{\partial W}$ and $\frac{\partial r_t}{\partial W}$, now take part in the calculation of the partial derivative $\frac{\partial L(y_t, x_{t+1})}{\partial W}$ ($y_t$ is the output of the model at $t$).

Figure 4.1 depicts the architecture using the simple late-integration mechanism.

### 4.3.2.2 Forms of the Inner Function

When a concrete form of the inner function $g_\theta(\cdot)$ is chosen, we can fully specify the Differential State Framework. We will also show, later in this chapter, how many other commonly-used RNN architectures can, in fact, be treated as special cases of this general framework defined under Equation 4.1.

Starting from Equation 4.2, if we fix $\gamma = 1$ and $\beta = 0$, we can recover the classical Elman RNN, where $g_\theta(x_t, h_{t-1})$ is a linear combination of the projection of the current data point and the projection of the previous hidden state, followed by a non-linearity
However, if we also set $\beta = 1$, we obtain a naive way to compute a delta change of states. Specifically, the simple-RNN’s hidden state, where $\Phi(v) = v$ (the identity function), is:

\[
\begin{align*}
    h_t &= \gamma \otimes g_\theta(x_t, h_{t-1}) + \beta \otimes h_{t-1} \\
    h_t &= 1 \otimes \phi(Vh_{t-1} + Wx_t + b) + 0 \otimes h_{t-1} \\
    h_t &= \phi(Vh_{t-1} + Wx_t + b)
\end{align*}
\]

(4.6)

where $h_t$ is the hidden layer state at time $t$, $x_t$ is the input vector, and $\theta = \{W, V\}$ contains the weight matrices. In contrast, the simple Delta-RNN, where instead $\phi(v) = v$, we have:

\[
    h_t = \Phi(Vh_{t-1} + Wx_t + b).
\]

(4.7)

Thus, the state can be implicitly stable, assuming $W$ and $V$ are initialized with small values and $\phi(\cdot)$ allows this by being partially linear. For example we can choose $\phi(\cdot)$ to be the linear rectifier (or initialize the model so to start out in the linear regime of the hyperbolic tangent). In this case, the simple Delta-RNN does not need to learn anything to maintain the state constant over time.

Preliminary experimentation with this simple form (Equation 4.7) often yielded unsatisfactory performance. This further motivated the development of the simple interpolation mechanism presented in Equation 4.3. However, depending on how one chooses the non-linearities, $\phi(\cdot)$ and $\Phi(\cdot)$, one can create different types of interpolation.

Using an Elman RNN for $g_\theta(x_t, h_{t-1})$ as in Equation 4.6, substituting into Equation 4.3 can create what we propose as the “late-integration” state model:

\[
\begin{align*}
    z_t &= g_\theta(x_t, h_{t-1}) \\
    &= \phi(Vh_{t-1} + Wx_t + b), \text{ and,} \\
    h_t &= \Phi((1 - r) \otimes z_t + r \otimes h_{t-1}).
\end{align*}
\]

(4.8)

where $\Phi(\cdot)$ could be any choice of activation function, including the identity function. This form of interpolation allows for a more direct error propagation pathway since gradient information, once transmitted through the interpolation gate, has two pathways: through the non-linearity of the local state (through $g_\theta(x_t, h_{t-1})$) and the pathway
Figure 4.1. The Delta-RNN computation graph, unfolded over time. The learnable gates $\gamma$ and $\beta$ control how much influence the previous state and the currently computed data-dependent state have on computing the model’s next hidden state.

composed of implicit identity connections.$^3$

When using a simple Elman RNN, we have essentially described a first-order Delta-RNN. However, historically, second-order recurrent neural architectures have been shown to be powerful models in tasks such as grammatical inference [137] and noisy time-series prediction [138] as well as incredibly useful in rule-extraction when treated as finite-state automata [139, 140]. Very recently, [141] showed that the gating effect between the state-driven component and data-driven components of a layer’s pre-activations facilitated better propagation of gradient signals as opposed to the usual linear combination. A second-order version of $g_\theta(x_t, h_{t-1})$ would be highly desirable, not only because it further mitigates the vanishing gradient problem that plagues back-propagation through time (used in calculating parameter gradients of neural architectures), but because the form introduces negligibly few additional parameters. We do note that the second-order form we use, like in [141], is a rank-1 matrix approximation of the actual tensor used in [139, 140].

We can take the late-integration model, Equation 4.9, and replace, similar to [137], $z_t$ with:

$$z_t = \phi(V h_{t-1} \otimes W x_t + b)$$

---

$^3$Late-integration might remind the reader of the phrase “late fusion”, as in the context of [136]. However, [136] was focused on merging the information from an external bag-of-words context vector with the standard cell state of the LSTM.
or a more general form [141]:

\[
\begin{align*}
    d_t^1 &= \alpha \otimes V_d h_{t-1} \otimes x_t \\
    d_t^2 &= \beta_1 \otimes V_d h_{t-1} + \beta_2 \otimes x_t \\
    z_t &= \phi(d_t^1 + d_t^2 + b),
\end{align*}
\] (4.11)

where we note that \( z_t \) can be a function of any arbitrary incoming set of information signals that are gated by the last known state. The Delta-RNN will ultimately combine this data-driven signal \( z_t \) with its slow-moving state. More importantly, observe that even in the most general form (Equation 4.11), only a few further bias vector parameters, \( \alpha \), \( \beta_1 \), and \( \beta_2 \) are required.

### 4.3.2.3 Regularizing the Delta-RNN

Regularization is often important when training large, over-parametrized models. To control for overfitting, approaches range from structural modifications to impositions of priors over parameters [142]. Commonly employed modern approaches include drop-out [143] and variations [144] or mechanisms to control for internal covariate drift, such as Batch Normalization [145] for large feedforward architectures. In this thesis, we investigate the effect that drop-out will have on the Delta-RNN’s performance.\(^4\)

To introduce simple (non-recurrent) drop-out to the framework, our preliminary experiments uncovered that drop-out was most effective when applied to the inner function \( g(\cdot) \) as opposed to the outer function’s computed delta-state. For the full Delta-RNN, under drop-out probability \( p_{\text{drop}} \), this would lead to the following modification:

\[
    h_t = \Phi((1 - r) \otimes DROP(g_\theta(x_t, h_{t-1}), p_{\text{drop}}) + r \otimes h_{t-1}).
\] (4.12)

\( DROP(x, p_{\text{drop}}) = x \otimes (\sim \text{B}(1, p_{\text{drop}})) \) is the drop-out operator which masks its input argument with a binary vector sampled from \( H \) independent Bernoulli distributions.

\(^4\)In preliminary experiments, we also investigated incorporating layer normalization [146] into the Delta-RNN architecture. We did not find observe noticeable gains using layer normalization over drop-out, and thus only report the results of drop-out.
4.3.2.4 Parameter Complexity of the Delta-RNN

Assuming a single hidden layer language model, with $H$ hidden units and $V$ input units (where $V$ corresponds to the cardinality of the symbol dictionary), a full late-integration Delta-RNN that employs a second-order $g_\theta(x_t, h_{t-1})$ (Equation 4.11), has only $((H \ast H) + 2(H \ast V) + 5H + V)$ parameters, which is only slightly larger than a classical RNN with only $((H \ast H) + 2(H \ast V) + H + V)$ parameters. This stands in stark contrast to the sheer number of parameters required to train commonly-used complex architectures such as the LSTM (with peephole connections), with $4(H \ast H) + 8(H \ast V) + 4H + V)$ parameters, and the GRU, with $3(H \ast H) + 4(H \ast V) + 3H + V)$ parameters.

4.3.3 Deriving Other Gated Architectures from DSF

The Differential State Framework offers a way to unify previous proposals for gated neural architectures, especially those used in next-step prediction tasks like language modeling, and explore directions of improvement. Since we will ultimately compare our proposed Delta-RNN to these architectures, we will next present how to derive several key architectures from our general form, such as the Gated Recurrent Unit and the Long Short Term Memory. More importantly, we will introduce them in the same notation / design as the Delta-RNN and highlight the differences between previous work and our own through the lens of $f_\psi(\cdot)$ and $g_\theta(x_t, M_{t-1})$.

Simple models, largely based on the original Elman RNN [124], have often been shown to perform quite well in language modeling tasks [147, 148]. The Structurally Constrained Recurrent Network (SCRN, [128]), an important predecessor and inspiration for this work, showed that one fruitful path to learning longer-term dependencies was to impose a hard constraint on how quickly the values of hidden units could change, yielding more “stable” long-term memory. The SCRN itself is very similar to a combination of the RNN architectures of [149, 150]. The key element of its design is the constraint that part of recurrent weight matrix must stay close to the identity, a constraint that is also satisfied by the Delta-RNN. These identity connections (and corresponding context units that use them) allow for improved information travel over many time-steps and can even be viewed as an exponential trace memory [150]. Residual Networks, though feed-forward in nature, also share a similar motivation [151]. Unlike the SCRN, the

$^5H$ counts the hidden bias, the full interpolation mechanism $r_t$ (Equation 4.5), and the second-order biases, $\{\alpha, \beta_1, \beta_2\}$. 

83
proposed Delta-RNN does not require a separation of the slow and fast moving units, but instead models this slower time-scale through implicitly stable states.

The Long Short Term Memory (LSTM, [125]) is arguably the currently most popular and often-used gated neural architecture, especially in the domain of Natural Language Processing. Starting from our general form, Equation 4.1, we can see how the LSTM can be deconstructed, where setting $c_t = g_\theta(x_t, M_{t-1})$, yields:

$$h_t = f_\psi[g_\theta(x_t, M_{t-1}), M_{t-1}]$$

$$h_t = r_t \otimes \Phi(c_t), \text{ where,}$$

$$r_t = \sigma(W_r x_t + V_r h_{t-1} + U_r c_t + b_r)$$

where $M_{t-1} = \{h_{t-1}, c_{t-1}\}$, noting that $c_{t-1}$ is the cell-state designed to act as the constant error carousal in mitigating the problem of vanishing gradients when using back-propagation through time. A great deal of recent work has attempted to improve the training of the LSTM, often by increasing its complexity, such as through the introduction of so-called “peephole connections” [152]. To compute $c_t = g_\theta(x_t, M_{t-1})$, using peephole connections, we use the following set of equations:

$$c_t = f_t \otimes c_{t-1} + i_t \otimes z_t, \text{ where,}$$

$$z_t = \Phi(W_z x_t + V_z h_{t-1} + b_z),$$

$$i_t = \sigma(W_i x_t + V_i h_{t-1} + U_i c_{t-1} + b_i),$$

$$f_t = \sigma(W_f x_t + V_f h_{t-1} + U_f c_{t-1} + b_f).$$

The Gated Recurrent Unit (GRU, [126, 127]) can be viewed as one of the more successful attempts to simplify the LSTM. We see that $f_\psi(\cdot)$ and $g_\theta(\cdot)$ are still quite complex, requiring many intermediate computations to reach an output. In the case of the outer mixing function, $f_\psi(\cdot)$, we see that:

$$h_t = \Phi(\gamma g_\theta(x_t, h_{t-1}) + \beta h_{t-1})$$

$$\gamma = r_t \text{ and } \beta = (1 - r_t), \text{ where,}$$

$$r_t = \sigma(V_r h_{t-1} + W_r x_t + b_r)$$

noting that the state gate $r_t$ is also a function of the RNN’s previous hidden state and introduces parameters specialized for $r$. In contrast, the Delta-RNN does not use an
extra set of input-to-hidden weights, and more directly, the pre-activation of the input projection can be reused for the interpolation gate. The inner function of the GRU, \( g_\theta(x_t, h_{t-1}) \), is defined as:

\[
g_\theta(x_t, h_{t-1}) = \phi(V_h(q_t \otimes h_{t-1}) + W_h x_t + b_h)
\]

\[
q_t = \sigma(V_q h_{t-1} + W_q x_t + b_q)
\]

where \( \phi() \) is generally set to be the hyperbolic tangent activation function. A mutated architecture (MUT, [153]) was an attempt to simplify the GRU somewhat, as, much like the Delta-RNN, its interpolation mechanism is not a function of the previous hidden state but is still largely as parameter-heavy as the GRU, only shedding a single extra parameter matrix, especially since its interpolation mechanism retains a specialized parameter matrix to transform the data. The Delta-RNN, on the other hand, shares this with its primary calculation of the data’s pre-activation values. The Minimally Gated Unit (MGU, [154]) is yet a further attempt to reduce the complexity of the GRU by merging its reset and update gates into a single forget gate, essentially using the same outer function under the GRU defined in Equation 4.16, but simplifying the inner function \( g_\theta(x_t, h_{t-1}) \) to be quite close to the Elman-RNN but conditioned on the forget gate as follows:

\[
g_\theta(x_t, h_{t-1}) = \phi(V_h(r_t \otimes h_{t-1}) + W_h x_t + b_h).
\]

While the MGU certainly does reduce the number of parameters, viewing it from the perspective of our general Delta-RNN framework, one can see that it still largely uses a \( g_\theta(x_t, h_{t-1}) \) that is rather limited (only the capabilities of the Elman-RNN). The most effective version of our Delta-RNN emerged from the insight that a more powerful \( g_\theta(x_t, h_{t-1}) \) could be obtained by (approximately) increasing its order, which requires a few more bias parameters, and nesting it within a non-linear interpolation mechanism that will compute the delta-states. Our framework is general enough to also allow designers to incorporate functions that augment the general state-engine with an external memory to create architectures that can exploit the strengths of models with decoupled memory architectures [155–157] or data-structures that serve as memory [134, 158].

A final related, but important, strand of work uses depth (i.e., number of processing layers) to directly model various time-scales, as emulated in models such as the hierarchical/multi-resolutional recurrent neural network (HM-RNN) [159]. Since the Delta-RNN is designed to allow its interpolation gate \( r \) to be driven by the data, it is
possible that the model might already be learning how to make use of boundary information (word boundaries at the character/sub-word level, sentence boundaries as marked by punctuation at the word-level). The HM-RNN, however, more directly attacks this problem by modifying an LSTM to learn how to manipulate its states when certain types of symbols are encountered. (This is different from models like the Clockwork RNN that require explicit boundary information [160].) One way to take advantage of the ideas behind the HM-RNN would be to manipulate the Differential State Framework to incorporate the explicit modeling of time-scales through layer depth (each layer is responsible for modeling a different time-scale). Furthermore, it would be worth investigating how the HM-RNN’s performance would change when built from modifying a Delta-RNN instead of an LSTM.

4.4 Learning Parameters

Let \( w_1, \ldots, w_T \) be a variable-length sequence of \( T \) symbols (such as words that would compose a sentence \( n \in N \)). In general, the distribution over the variables follows the graphical model:

\[
P_\theta(w_1, \ldots, w_T) = \prod_{t=1}^{T} P_\Theta(w_t|w_{<t}),
\]

where \( \Theta = \{ \psi, \theta \} = \{ V, W, R, b, b_r, \alpha, \beta_1, \beta_2 \} \) are the model parameters (of a full Delta-RNN).

No matter how the hidden state \( h_t \) is calculated, in this chapter, it will ultimately be fed into a maximum-entropy classifier\(^6\) defined as:

\[
P(w, h_t) = P_\Theta(w|h_t) = \frac{\exp(w^T R h_t)}{\sum_{w'} \exp((w')^T R h_t)},
\]

To learn parameters for any of our models, we optimize with respect to the sequence negative log likelihood:

\[
\mathcal{L} = -\sum_{i=1}^{N} \sum_{t=1}^{T} \log P_\Theta(w_t|h)_n,
\]

\(^6\)Note that the bias term has been omitted for clarity.
where we index a sentence of $N$ total sentences by its estimated probability $P_{\Theta}(w_1, \ldots, w_T)_n$. Model parameters, $\Theta = \{\theta, \psi\}$, of the Delta-RNN are learned under an empirical risk minimization framework. We employ back-propagation of errors (or rather, reverse-mode automatic differentiation with respect to this negative log likelihood objective function) to calculate gradients and update the parameters using the method of steepest gradient descent. For all experiments conducted, we found that the ADAM adaptive learning rate scheme \[161\] (followed by a Polyak average \[162\] for the subword experiments) yielded the most consistent and near-optimal performance. We therefore use this set-up for optimization of parameters for all models (including baselines), unless otherwise mentioned. For all experiments, we unroll computation graphs $T$ steps in time (where $T$ varies across experiments/tasks), and, in order to approximate full back-propagation through time, we carry over the last hidden from the previous mini-batch (within a full sequence). More importantly, we found that by using the derivative of the loss with respect to the last hidden state, we can improve the approximation and thus perform one step of iterative inference \[7\] to update the last hidden state carried over. We ultimately used this proposed improved approximation for the sub-word models (since in those experiments we could directly train all baseline and proposed models in a controlled, identical fashion to ensure fair comparison).

For all Delta-RNNs experimented with, the output activation of the inner function $g(\cdot)$ was chosen to be the hyperbolic tangent. The output activation of the outer function $f(\cdot)$ was set to be the identity for the word and character benchmark experiments and the hyperbolic tangent for the subword experiments (these decisions were made based on preliminary experimentation on sub-sets of the final training data). The exact configuration of this implementation involved using the late-integration form, either the un-regularized (Equation 4.9) or the drop-out regularized (Equation 4.12) variant, for the outer function and Equation 4.11.

We compare our proposed models against a wide variety of un-regularized baselines, as well as several state-of-the-art regularized baselines for the benchmark experiments. These baselines include the LSTM, GRU, SCRN, and the MGU. The goal is to see if our proposed Delta-RNN is a suitable replacement for complex gated architectures and can capture longer term patterns in sequential text data.

\[7\] We searched the step-size $\lambda$ over the values \{0.05, 0.1, 0.15\} in all of the experiments.
4.5 Experiments: Language Modeling

Language modeling is an incredibly important next-step prediction task, with applications in downstream applications in speech recognition, parsing, and information retrieval. As such, we will focus on experiments on this task domain to gauge the efficacy of our Differential State Framework framework, noting that the Delta-RNN framework might prove useful in, for instance, machine translation [163] or light chunking [164]. Beyond improving language modeling performance, the sentence (and document) representations iteratively inferred by our architectures might also prove useful in composing higher-level representations of text corpora, a subject we will investigate in future work.

4.5.1 Datasets

4.5.1.1 Penn Treebank

The Penn Treebank corpus [165] is often used to benchmark both word and character-level models via perplexity or bits-per-character, and thus we start here.\(^8\) The corpus contains 42,068 sentences (971,657 tokens, average token-length of about 4.727 characters) of varying length (the range is from 3 to 84 tokens, at the word-level).

4.5.1.2 IMDB

The large sentiment analysis corpus [166] is often used to benchmark algorithms for predicting the positive or negative tonality of documents. However, we opt to use this large corpus (training consists of 149,714 documents, 1,875,523 sentences, 40,765,697 tokens, average token-length is about 3.4291415 characters) to evaluate our proposed Delta-RNN as a (subword) language model. The IMDB data-set serves as a case when the context extends beyond the sentence-level in the form of actual documents.

4.5.1.3 TagMyNews

We use the TagMyNews news article corpus.\(^9\) This dataset contains many text snippets where each is a small document containing multiple phrases/sentences as well as an article title. The data is split into a training set of 27,404 articles, a development set of

\(^8\)To be directly comparable with previously reported results, we make use of the specific pre-processed train/valid/test splits found at http://www.fit.vutbr.cz/imikolov/rnnlm/.

\(^9\)Data is publicly available at http://acube.di.unipi.it/tmn-dataset/.
2600 articles, and a test set of 2600 articles. Pre-processing in terms of tokenization is carried out in the same fashion as in word-level Penn Treebank. A vocabulary of 18, 417 unique words was created after replacing words that occurred less than a single time (replaced with an unknown token symbol, $<$ unk $>$).

### 4.5.1.4 Text8

For this dataset, which is a fairly large dataset, we follow the pre-processing scheme of [128]. The original dataset was created by taking the first 100 million characters from a Wikipedia dump. This corpus is split into a training set containing the first 99 million characters and the last 1 million characters are used as a test set to measure generalization. A vocabulary of 44K unique words is created after substituting words that occur less than five times with an unknown token symbol, $<$ unk $>$.

### 4.5.2 Word & Character-Level Benchmarks

The first set of experiments allows us to examine our proposed Delta-RNN models against reported state-of-the-art models. These reported measures have been on traditional word and character-level language modeling tasks—we measure the per-symbol perplexity of models. For the word-level models, we calculate the per-word perplexity (PPL) using the measure $PPL = \exp \left[ -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log P_{\theta}(w_t|h) \right]$. For the character-level models, we report the standard bits-per-character (BPC), which can be calculated from the log likelihood using the formula: $BPC = -\frac{1}{(N \log(2))} \sum_{i=1}^{N} \sum_{t=1}^{T} \log P_{\theta}(w_t|h)$.

Over 100 epochs, word-level models with mini-batches of 64 (padded) sequences. (Early stopping with a lookahead of 10 was used.) Gradients were clipped using a simple magnitude-based scheme [7], with the magnitude threshold set to 5. A simple grid-search was performed to tune the learning rate, $\lambda = \{0.002, 0.001, 0.0005, 0.0002\}$, as well as the size of the hidden layer $H = \{500, 1000, 1500\}$. Parameters (non-biases) were initialized from zero-mean Gaussian distributions with variance tuned, $\sigma = \{0.1, 0.01, 0.005, 0.001\}$\(^{10}\). The character-level models, on the other hand, were updated using mini-batches of 64 samples over 100 epochs. (Early stopping with a

\(^{10}\)We also experimented with other initializations, most notably the identity matrix for the recurrent weight parameters as in [177]. We found that this initialization often worsened performance. For the activation functions of the first-order models, we experimented with the linear rectifier, the parametrized linear rectifier, and even our own proposed parametrized smoothened linear rectifier, but found such activations lead to less-than-satisfactory results.
Table 4.1. Test-set results on the Penn Treebank word-level and character-level language modeling tasks. Note that “impl.” means implementation.

**Penn Treebank: Word Models**

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Penn Treebank: Word Models</th>
<th>PPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Gram [128]</td>
<td></td>
<td>141</td>
</tr>
<tr>
<td>NNL [167]</td>
<td></td>
<td>140.2</td>
</tr>
<tr>
<td>N-Gram + cache [128]</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>RNN [168]</td>
<td></td>
<td>129</td>
</tr>
<tr>
<td>RNN [167]</td>
<td></td>
<td>124.7</td>
</tr>
<tr>
<td>LSTM [128]</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>SCR [128]</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>LSTM [169]</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>MI-RNN ([141], our impl.)</td>
<td></td>
<td>109.2</td>
</tr>
<tr>
<td>Delta-RNN (present work)</td>
<td></td>
<td>100.324</td>
</tr>
<tr>
<td>Delta-RNN, dynamic #1 (present work)</td>
<td></td>
<td>93.296</td>
</tr>
<tr>
<td>Delta-RNN, dynamic #2 (present work)</td>
<td></td>
<td>90.301</td>
</tr>
<tr>
<td>LSTM-recurrent drop [170]</td>
<td></td>
<td>87.0</td>
</tr>
<tr>
<td>NR-dropout [171]</td>
<td></td>
<td>78.4</td>
</tr>
<tr>
<td>V-dropout [144]</td>
<td></td>
<td>73.4</td>
</tr>
<tr>
<td>Delta-RNN-drop, static (present work)</td>
<td></td>
<td>84.088</td>
</tr>
<tr>
<td>Delta-RNN-drop, dynamic #1 (present work)</td>
<td></td>
<td>79.527</td>
</tr>
<tr>
<td>Delta-RNN-drop, dynamic #2 (present work)</td>
<td></td>
<td>78.029</td>
</tr>
</tbody>
</table>

**Penn Treebank: Character Models**

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Penn Treebank: Character Models</th>
<th>BPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-discount N-gram [172]</td>
<td></td>
<td>1.48</td>
</tr>
<tr>
<td>RNN + stabilization (Krueger et al., 2016)</td>
<td></td>
<td>1.48</td>
</tr>
<tr>
<td>Linear MI-RNN [141]</td>
<td></td>
<td>1.48</td>
</tr>
<tr>
<td>Clockwork RNN [160]</td>
<td></td>
<td>1.46</td>
</tr>
<tr>
<td>RNN [172]</td>
<td></td>
<td>1.42</td>
</tr>
<tr>
<td>GRU [173]</td>
<td></td>
<td>1.42</td>
</tr>
<tr>
<td>HF-MRNN [172]</td>
<td></td>
<td>1.41</td>
</tr>
<tr>
<td>MI-RNN [141]</td>
<td></td>
<td>1.39</td>
</tr>
<tr>
<td>Max-Ent N-gram [172]</td>
<td></td>
<td>1.37</td>
</tr>
<tr>
<td>LSTM [170]</td>
<td></td>
<td>1.356</td>
</tr>
<tr>
<td>Delta-RNN (present work)</td>
<td></td>
<td>1.347</td>
</tr>
<tr>
<td>Delta-RNN, dynamic #1 (present work)</td>
<td></td>
<td>1.331</td>
</tr>
<tr>
<td>Delta-RNN, dynamic #2 (present work)</td>
<td></td>
<td>1.326</td>
</tr>
<tr>
<td>LSTM-norm stabilizer [170]</td>
<td></td>
<td>1.352</td>
</tr>
<tr>
<td>LSTM-weight noise [170]</td>
<td></td>
<td>1.344</td>
</tr>
<tr>
<td>LSTM-stochastic depth [170]</td>
<td></td>
<td>1.343</td>
</tr>
<tr>
<td>LSTM-recurrent drop [170]</td>
<td></td>
<td>1.286</td>
</tr>
<tr>
<td>RBN [174]</td>
<td></td>
<td>1.32</td>
</tr>
<tr>
<td>LSTM-zone out [170]</td>
<td></td>
<td>1.252</td>
</tr>
<tr>
<td>H-LSTM + LN [175]</td>
<td></td>
<td>1.25</td>
</tr>
<tr>
<td>TARDIS [176]</td>
<td></td>
<td>1.25</td>
</tr>
<tr>
<td>3-HM-LSTM + LN [159]</td>
<td></td>
<td>1.24</td>
</tr>
<tr>
<td>Delta-RNN-drop, static (present work)</td>
<td></td>
<td>1.251</td>
</tr>
<tr>
<td>Delta-RNN-drop, dynamic #1 (present work)</td>
<td></td>
<td>1.247</td>
</tr>
<tr>
<td>Delta-RNN-drop, dynamic #2 (present work)</td>
<td></td>
<td>1.245</td>
</tr>
</tbody>
</table>

(lookahead of 10 was used.) The parameter initializations and grid-search for the learning rate and hidden layer size were the same as for the word models, with the exception of
the hidden layer size, which was searched over $H = \{500, 1000, 1500, 2000\}$\textsuperscript{11}.

A simple learning rate decay schedule was employed: if the validation loss did not decrease after a single epoch, the learning rate was halved (unless a lower bound on the value had been reached). When drop-out was applied to the Delta-RNN (\textit{Delta-RNN-drop}, we set the probability of dropping a unit to $p_{\text{drop}} = 0.15$ for the character-level models and $p_{\text{drop}} = 0.5$ for the word level models. We present the results for the un-regularized and regularized versions of the models. For all of the Delta-RNNs, we furthermore experiment with two variations of dynamic evaluation, which facilitates fair comparison to compression algorithms, inspired by the improvements observed in [167]. \textit{Delta-RNN-drop, dynamic #1} refers to simply updating the model sample-by-sample after each evaluation, where in this case, we update parameters using simple stochastic gradient descent [167], with a step-size $\lambda = 0.005$. We develop a second variation of dynamic evaluation, \textit{Delta-RNN-drop, dynamic #2}, where we allow the model to first iterate (and update) once over the validation set and then finally the test-set, completely allowing the model to “compress” the Penn Treebank corpus. These two schemes are used for both the word and character-level benchmarks. It is important to stress the BPC and PPL measures reported for the dynamic models follow a strict “test-then-train” online paradigm, meaning that each next-step prediction is made before updating model parameters.

The standard vocabulary for the word-level models contains 10K unique words (including an unknown token for out-of-vocabulary symbols and an end-of-sequence token)\textsuperscript{12} and the standard vocabulary for the character-level models includes 49 unique characters (including a symbol for spaces). Results for the word-level and for the character-level models are both reported in Table 4.1.

\textbf{4.5.3 Sub-word Language Modeling}

We chose to measure the negative log likelihood of the various architectures in the task of \textit{subword modeling}. Subwords are particularly appealing not only in that the input distribution is of lower dimensionality but, as evidenced by the positive results of [172], sub-word/character hybrid language models result in improved performance

\textsuperscript{11}Note that $H = 2000$ would yield nearly 4 million parameters, which was our upper bound on total number of parameters allowed for experiments in order to be commensurable with the work of [141], which actually used $H = 2048$ for all Penn Treebank models.

\textsuperscript{12}We use a special “null” token (or zero-vector) to mark the start of a sequence.
Table 4.2. Test-set negative log likelihoods while holding number of parameters approximately constant. Subword modeling tasks on Penn Treebank and IMDB.

<table>
<thead>
<tr>
<th>PTB-SW</th>
<th>Performance</th>
<th>IMDB-SW</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN</td>
<td># Params: 1, 272, 464, NLL: 1.8939</td>
<td>RNN</td>
<td># Params: 499, 176, NLL: 2.1691</td>
</tr>
<tr>
<td>SCRN</td>
<td># Params: 1, 268, 604, NLL: 1.8420</td>
<td>SCRN</td>
<td># Params: 496, 196, NLL: 2.2370</td>
</tr>
<tr>
<td>MGU</td>
<td># Params: 1, 278, 692, NLL: 1.8694</td>
<td>MGU</td>
<td># Params: 495, 444, NLL: 2.1312</td>
</tr>
<tr>
<td>MI-RNN</td>
<td># Params: 1, 267, 904, NLL: 1.8441</td>
<td>MI-RNN</td>
<td># Params: 495, 446, NLL: 2.1741</td>
</tr>
<tr>
<td>GRU</td>
<td># Params: 1, 272, 404, NLL: 1.8251</td>
<td>GRU</td>
<td># Params: 499, 374, NLL: 2.1551</td>
</tr>
<tr>
<td>LSTM</td>
<td># Params: 1, 274, 804, NLL: 1.8412</td>
<td>LSTM</td>
<td># Params: 503, 664, NLL: 2.2080</td>
</tr>
<tr>
<td>Delta-RNN</td>
<td># Params: 1, 268, 154, NLL: 1.8260</td>
<td>Delta-RNN</td>
<td># Params: 495, 570, NLL: 2.1333</td>
</tr>
</tbody>
</table>

Over pure character-level models. Sub-word models also enjoy the advantage held by character-level models when it comes to handling out-of-vocabulary words, avoiding the need for an “unknown” token. Research in psycholinguistics has long suggested that even human infants are sensitive to word boundaries at an early stage (e.g., [178]), and that morphologically complex words enjoy dedicated processing mechanisms [179]. Subword-level language models may approximate such an architecture. Consistency in subword formation is critical in order to obtain meaningful results [172]. Thus, we design our sub-word algorithm to partition a word according to the following scheme:

1. Split on vowels (using a predefined list)
2. Link/merge each vowel with a consonant to the immediate right if applicable
3. Merge straggling single characters to subwords on the immediate right unless a subword of shorter character length is to the left.

This simple partitioning scheme was designed to ensure that no subword was shorter than two characters in length. Future work will entail designing a more realistic subword partitioning algorithm. Subwords below a certain frequency were discarded, and combined with 26 single characters to create the final dictionary. For Penn Treebank, this yielded a vocabulary of 2405 symbols was created (2,378 subwords + 26 characters + 1 end-token). For the IMDB corpus, after replacing all emoticons and special non-word symbols with special tokens, we obtained a dictionary of 1926 symbols (1899 subwords + 26 single characters + 1 end-token). Results for all sub-word models are reported in Table 4.2.

Specifically, we test our implementations of the LSTM\(^\text{13}\) (with peephole connections as described in [180]), the GRU, the MGU, the SCRN, as well as a classical Elman

---

\(^{13}\)We experimented with initializing the forget gate biases of all LSTMs with values searched over \{1, 2, 3\} since previous work has shown this can improve model performance.
network, of both 1st and 2nd-order [137, 141]. Subword models were trained in a similar fashion as the character-level models, updated (every 50 steps) using mini-batches of 20 samples but over 30 epochs. Learning rates were tuned in the same fashion as the word-level models, and the same parameter initialization schemes were explored. The notable difference between this experiment and the previous ones is that we fix the number of parameters for each model to be equivalent to that of an LSTM with 100 hidden units for PTB and 50 hiddens units for IMDB. This ensures a controlled, fair comparison across models and allows us to evaluate if the Delta-RNN can learn similarly to models with more complicated processing elements (an LSTM cell versus a GRU cell versus a Delta-RNN unit). Furthermore, this allows us to measure parameter efficiency, where we can focus on the value of actual specific cell-types (for example, allowing us to compare the value of a much more complex LSTM memory unit versus a simple Delta-RNN cell) when the number of parameters is held roughly constant.

4.5.3.1 Thinking about the Delta Intuition

With respect to the word and character-level benchmarks, we see that the Delta-RNN outperforms all previous, un-regularized models, and performs comparably to regularized state-of-the-art. As documented in Table 4.2, we further trained a second-order, word-level RNN (MI-RNN) to complete the comparison, and remark that the second-order connections appear to be quite useful in general, outperforming the SCRN and coming close to that of the LSTM. This extends the results of [141] to the word-level. However, the Delta-RNN, which also makes use of second-order units within its inner function, ultimately offers the best performance and performs better than the LSTM in all experiments. In both Penn Treebank and IMDB subword language modeling experiments, the Delta-RNN is competitive with complex architectures such as the GRU and the MGU. In both cases, the Delta-RNN nearly reaches the same performance as the best performing baseline model in either data-set (i.e., it nearly reaches the same performance as the GRU on Penn Treebank and the MGU on IMDB). Surprisingly, on IMDB, a simple Elman network is quite performant, even outperforming the MI-RNN. We argue that this might be the result of constraining all neural architectures to only a small number of parameters for such a large data-set, a constraint we intend to relax in future work.

The Delta-RNN is far more efficient than a complex LSTM and certainly a memory-augmented network like TARDIS [176]. Moreover, it appears to learn how to make appropriate use of its interpolation mechanism to decide how and when to update its hid-
hidden state in the presence of new data.\textsuperscript{14} Given our derivations earlier in this chapter, one could argue that nearly all previously proposed gated neural architectures are essentially trying to do the same thing under the Differential State Framework. The key advantage offered by the Delta-RNN is that this functionality is offered directly and cheaply (in terms of required parameters).

It is important to contrast these (un-regularized) results with those that use some form of regularization. \cite{171} reported that a single LSTM (for word-level Penn Treebank) can reach a PPL of $\sim 80$, but this was achieved via dropout regularization \cite{143}. There is a strong relationship between using dropout and training an ensemble of models. Thus, one can argue that a single model trained with dropout actually is not a single model, but an implicit ensemble (see also \cite{143}). An ensemble of twenty simple RNNs and cache models did previously reach PPL as low as 72, while a single RNN model gives only 124 \cite{167}. \cite{171} trained an ensemble of 38 LSTMs regularized with dropout, each with 100x times more parameters than the RNNs used by \cite{167}, achieving PPL 68. This is arguably a small improvement over 72, and seems to strengthen our claim that dropout is an implicit model ensemble and thus should not be used when one wants to report the performance of a single model. However, the Delta-RNN is amenable to regularization, including drop-out. As our results show, when simple drop-out is applied, the Delta-RNN can reach much lower perplexities, even similar to the state-of-the-art with much larger models, especially when dynamic evaluation is permitted. This even extends to very complex architectures, such as the recently proposed TARDIS, which is a memory-augmented network (and when dynamic evaluation is used, the simple Delta-RNN can outperform this complex model). Although we investigated the utility of simple drop-out, our comparative results suggest that more sophisticated variants, such as variational drop-out \cite{144}, could yield yet further improvement in performance.

What are the lessons to be learned from the Differential State Framework? First, and foremost, we can obtain strong performance in language modeling with a simpler, more efficient (in terms of number of parameters), and thus faster, architecture. Second, the Delta-RNN is designed from the interpretation that the computation of the next hidden state is the result of a composition of two functions: one inner function decides how to “propose” a new hidden state while the outer function decides how to use this new proposal in updating the previously calculated state. The data-driven interpolation

\textsuperscript{14}At greater computational cost, a somewhat lower perplexity for an LSTM may be attainable, such as the perplexity of 107 reported by \cite{169} (see Table 4.1). However, this requires many more training epochs and precludes batch training.
mechanism is used by the model to decide how much impact the newly proposed state has in updating what is likely to be a slowly changing representation. The SCRN, which could be viewed as the predecessor to the Delta-RNN framework, was designed with the idea that some constrained units could serve as a sort of cache meant to capture longer-term dependencies. Like the SCRN, the Delta-RNN is designed to help mitigate the problem of vanishing gradients, and through the interpolation mechanism, has multiple pathways through which the gradient might be carried, boosting the error signal’s longevity down the propagation path through time. However, the SCRN combines the slow-moving and fast-changing hidden states through a simple summation and thus cannot model non-linear interactions between its shorter and longer term memories, furthermore requiring tuning of the sizes of these separated layers. On the other hand, the Delta-RNN, which does not require special tuning of an additional hidden layer, can non-linearly combine the two types of states in a data-dependent fashion, possibly allowing the model to exploit boundary information from text, which is quite powerful in the case of documents. The key intuition is that the gating mechanism only allows the state proposal to affect the maintained memory state only if the currently observed data-point carries any useful information. This warrants a comparison, albeit indirect, to Surprisal Theory. This “surprisal” proves useful in iteratively forming a sentence impression that will help to better predict the words that come later.

With respect to the last point made, we briefly examine the evolution of a trained Delta-RNN’s hidden state across several sample sentences. The first two sentences are hand-created (constrained to use only the vocabulary of Penn Treebank) while the last one is sampled from the Penn Treebank training split. Since the Delta-RNN iteratively processes symbols of an ordered sequence, we measure the L1 norm across consecutive pairs of hidden states. We report the (min-max) normalized L1 scores in Figure 4.2 and observe that, in accordance with our intuition, we can see that the L1 norm is lower for high-frequency words (indicating a smaller delta) such as “the” or “of” or “is”, which are words generally less informative about the general subject of a sentence/document. As this qualitative demonstration illustrates, the Delta-RNN appears to learn what to do with its internal state in the presence of symbols of variable information content.

\[ L_{1\text{score}} = \frac{L_1(h_{t-1}, h_t) - \min(L_{1\text{seq}})}{\max(L_{1\text{seq}}) - \min(L_{1\text{seq}})} \]

\(^{15}\)If we calculate the L1 norm, or Manhattan distance, for every contiguous pair of state vectors across a sequence of length \( T \) and \( h_0 \) is the state calculated for the start/null token, we obtain the sequence of L1 measures \( L_{1\text{seq}} = \{L_1(h_0, h_1), ..., L_1(h_T, h_T)\} \) (the L1 for the start token is simply excluded). Calculating the score for any \( h_t \) (\( t \in T \)) is then as simple as performing min-max normalization, or \( L_{1\text{score}} = \frac{L_1(h_{t-1}, h_t) - \min(L_{1\text{seq}})}{\max(L_{1\text{seq}}) - \min(L_{1\text{seq}})} \).
Figure 4.2. L1 norm of deltas between consecutive states of model trained on Penn Treebank plotted over words of example sentences. A simple polynomial trend-line (dashed red) was fit to the bar heights in order to illustrate the informative “bumps” of each sample sentence. The main observation is that the norm is, in general, lower for low-information content words, such as the article “the”, and higher for informative words, such as “government”.
Table 4.3. Word-level language modeling text perplexities on several language modeling datasets.

<table>
<thead>
<tr>
<th>Model</th>
<th>Text8 (PPL)</th>
<th>TagMyNews (BPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>300</td>
</tr>
<tr>
<td>SCRN [128]</td>
<td>184.0</td>
<td>164.0</td>
</tr>
<tr>
<td>LSTM</td>
<td>193.0</td>
<td>159.0</td>
</tr>
<tr>
<td>Δ-RNN</td>
<td>191.6</td>
<td>157.9</td>
</tr>
</tbody>
</table>

4.5.4 Additional Next Step Prediction Results

In this section, we will present additional experiments in language modeling. Here we will apply the Δ-RNN to two more datasets, processing the text at the word-level. The Δ-RNN will be compared to the complex and popular LSTM to understand the differences in performance in different scenarios.

For text8 experiments, we tested the models on word-level language modeling, which also allowed us to directly compare to the Structurally Constrained Recurrent Network (SCRN) in [128]. Models were trained using the same setup as in described in the original SCRN paper [128]. The LSTM results we report are the result of using our own implementation and we successfully reproduce the results reported in [128] when using the same learning setup. With respect the TagMyNews corpus. For this task, we used a similar setup as for text8, with a few exceptions. A initial learning rate of $\lambda = 1.0$ was used and was halved each time validation loss increased (using a simple lookahead) and the Adam adaptive learning rate was employed when updating parameters. The L2 norms of the gradients were hard clipped to stay in the range of $[-0.2, 0.2]$.

In Table 4.3, we report word-level perplexity (PPL) for text8 and TagMyNews. Observe that the Δ-RNN still manages to outperform the SCRN and slightly outperforms the LSTM on text8, even though the model is dramatically simpler and faster when using the same hidden layer size. However, for TagMyNews, this is not the case. While the Δ-RNN does perform reasonably well on the dataset, it still lags behind the LSTM on the test-set by 8 points of perplexity. However, the LSTM is demonstrating a significant amount of overfitting (with a train perplexity of 35) while the Δ-RNN does not appear to overfit the data. This might indicate that perhaps the Δ-RNN was not optimized well enough, despite being trained under the same exact conditions as the LSTM. It might also be possible that the Δ-RNN requires a different design of its outer function, since in this experiment we only utilized the identity function (whereas other nonlinearities such as the hyperbolic tangent or linear rectifier might work better).
Since TagMyNews certainly contains longer-term dependencies than text8 or Penn Treebank, this might indicate that while the Δ-RNN can perform as well or even outperform a complex model like the LSTM when modeling sentences, the inner and outer mixing functions appear to struggle a bit more when faced with cross-sentence dependencies. If this is the case, a much better way to improve the performance of the Δ-RNN in this scenario would be to further modify the outer function, perhaps by using an external memory component that can buffer more information which might be what the LSTM’s cell-state is able to do on TagMyNews.

While it is discouraging to see that the Δ-RNN does not outperform the LSTM on the TagMyNews corpus, this chapter has already presented many language modeling results that demonstrate the opposite, positive outcome. What this should highlight is that no model will always be the best across all datasets, which is to be expected given the No Free Lunch Theorem [181]. However, the Δ-RNN still demonstrates that we can design longer-term memory neural systems with vastly simpler, and more interpretable, mechanisms (as compared to complex architectures like the LSTM). More importantly, the Δ-RNN should be extended to incorporate more powerful, and simply expressed, neuro-cognitive ideas about memory, which might facilitate improved generalization across an even wider array of problems. Next, we will investigate extending the Δ-RNN to incorporate image information in modeling natural language as well as highlight several other applications of the model where promising performance is achieved.

### 4.6 Grounding the Delta-RNN in Multiple Modalities

We now turn our attention to a harder variation of the language modeling problem explored so far in this chapter: language modeling under multiple modalities, or multimodal language modeling. Specifically, we will explore how well the Δ-RNN performs when trained to generate symbol sequences conditioned on images. Investigating how a language model’s performance is affected when incorporating visual information is motivated by the theory of situated (or embedded) cognition.

The theory of situated cognition postulates that a person’s knowledge is inseparable from the physical or social context in which it is learned and used [182]. Knowledge of language cannot be separated from its physical context, which allows words and sentences to be learned by grounding them in reference to objects or natural concepts on hand (see [183], [183], for a review). Nor can knowledge of language be separated from
its social context, where language is learned interactively through communicating with others to facilitate problem-solving. Simply put, language does not occur in a vacuum.

Yet, statistical language models, typically connectionist systems, are often trained in such a vacuum. Sequences of symbols, such as sentences or phrases composed of words in any language, such as English or German, are often fed into the model independently of any real-world context they might describe. In the classical language modeling framework, a neural model is tasked with a series of next-step prediction tasks, learning to predict a word based on a history of words it has seen so far. While these models learn a great deal of linguistic structure from these symbol sequences alone, acquiring the essence of basic syntax, it is highly unlikely that this approach can create models that acquire much in terms of semantics or pragmatics, which are integral to the human experience of language. How might one build neural language models that “understand” the semantic content held within the symbol sequences, of any language, presented to it?

In the following sections, we take a small step towards a model that understands language by training a neural architecture jointly on corresponding linguistic and visual data. From an image-captioning dataset, we create a multi-lingual corpus where sentences are mapped to the real-world images they describe. We ask how adding such real-world context at training can improve the performance of language models. We extend the Δ-RNN [52], the Long Short Term Memory (LSTM) [184] and the Gated Recurrent Unit (GRU) [185] to incorporate visual context information, creating a unified multi-modal connectionist architecture. We find that the models acquire more knowledge of language than if they were trained without corresponding, real-world visual context.

4.6.1 Background on Grounded Cognition

The Perceptual Symbol Systems theory holds that all of cognition, language, reasoning, and memory, is grounded in perceptual features [186]. Both behavioral and neuroimaging studies have found considerable evidence for the contribution of perceptual information to linguistic tasks [187]. Cognitive theory has long held that language is acquired jointly with perception through interaction with the environment, e.g., [188]. Cognitive models can account for bootstrapped learning of word meaning and syntax when language is paired with ambiguous and limited perceptual experience [189], and for the ability of children to rapidly acquire new words by inferring the referent from their physical environment [190].
A number of models of distributional semantics integrate word co-occurrence data extracted from a corpus with perceptual data, either to achieve a better model of language as it exists in the minds of humans [191, 192] or to improve performance on machine learning tasks such as object recognition [193], image captioning [194], or image search [195].

Integrating language and perception can facilitate language acquisition by allowing models to infer how a new word is used from the perceptual features of its referent [192]. Likewise, this integration allows models to infer the perceptual features of an unobserved referent from how a word is used in language [192]. As a result, language data can be used to improve object recognition by providing information about unobserved or infrequently observed objects [193].

By representing the referents of concrete nouns as arrangements of elementary visual features [196], [191] find that the visual features of nouns capture semantic typicality effects, and that a combined representation, consisting of both visual features and word co-occurrence data, more strongly correlates with human judgments of semantic similarity than representations extracted from a corpus alone. While modeling similarity judgments is distinct from the problem of predictive language modeling, we take this finding as evidence that visual perception informs semantics, which suggests there are gains to be had integrating perception with predictive language models.

While knowledge of concrete nouns benefits most directly from integrating perceptual data with language, verbs also benefit, as the perceptual features of verbs can be inferred from the features of the nouns they act upon [192], such that a model with access to perceptual features gains the ability to discriminate between actions afforded by a verb and actions that are not afforded by the verb.

Image Captioning [194,197,198] systems have shown promising results in generating captions by mapping between vision and language. However such models are restricted to a single language and can introduce irreversible corruption to a vision signal if trained jointly, since randomly initialized language parameters generate Gaussian noise that can harm contextual interaction information. If a jointly trained vision and language model is trained on multiple languages then each language introduces language specific noise that would corrupt visual information.

In contrast to prior work in machine learning, our goal in integrating visual and linguistic data is not to accomplish a task such as image search or image captioning that inherently requires a mapping between these two modalities. Rather, our goal is to
demonstrate that perceptual information is intrinsic to how humans process language, and as such, a language model that is trained on both visual and linguistic data will be a better model, consistently across languages, than a model trained on linguistic data alone.

Prior work in cognitive modeling has focused on models of distributional semantics that capture the similarity relations between words, e.g., [191, 192], whereas the model we propose here is a predictive language model.

Due to the ability of language models to probabilistically constrain input on the basis of preceding context and to classify linguistic material, these models play a central role in natural-language and speech processing applications. However, the psycholinguistic questions surrounding how people acquire and use linguistic knowledge are fundamentally different from the aims of machine learning. Using NLP-style language models to address psycholinguistic questions is a new approach that integrates well with the theory of predictive coding in cognitive psychology [62, 199]. For language processing this means that when reading text or comprehending speech, humans constantly anticipate what will be said next. This is a fast, implicit cognitive process that does not require symbol manipulation, but that can make use of the kind of sequence learning that recurrent neural models excel at. We do not propose such models as direct accounts of human language processing. Instead, our intent is to examine what can and cannot be learned with the addition of a non-linguistic modality (vision) at training time.

4.6.2 The Multimodal Neural Architecture

In designing our neural model, we start from the Differential State Framework (DSF, [52]), which unifies gated recurrent architectures under the general view that state memory is a simple parametrized mixture of “fast” and “slow” states. Our aim is to model sequences of symbols, such as the words that compose sentences, where at each time we process \( x_t \), or the one-hot encoding of a token.

One of the simplest models that can be derived from the DSF is the \( \Delta \)-RNN, which has been shown to outperform most complex neural models in next-step symbol prediction tasks [52]. The model, with parameters \( \Theta = \{ W, U, V, b, c, b_r, \beta_1, \beta_2, \alpha \} \), is defined as:

\[
\begin{align*}
\mathbf{d}^{\text{rec}}_{t} &= V \mathbf{h}_{t-1},
\mathbf{d}^{\text{dat}}_{t} &= W \mathbf{e}_{w,t}, \\
\mathbf{d}^1_{t} &= \alpha \otimes \mathbf{d}^{\text{rec}}_{t} \otimes \mathbf{d}^{\text{dat}}_{t}, \\
\mathbf{d}^2_{t} &= \beta_1 \otimes \mathbf{d}^{\text{rec}}_{t} + \beta_2 \otimes \mathbf{d}^{\text{dat}}_{t},
\end{align*}
\]
Figure 4.3. The multimodal ∆-RNN, unrolled over time. The gray-dashed connections represent the identity connections that carry over the slow-moving state while the dash-dotted black lines represent the next-step predictions made by the model. Solid black lines correspond to synaptic weight matrices (labeled accordingly).

\[
\begin{align*}
  z_t &= \phi_{hid}(d_1^t + d_2^t + b), \quad (4.23) \\
  h_t &= \Phi((1 - r) \otimes z_t + r \otimes h_{t-1}), \text{and,} \quad (4.24) \\
  r &= 1/(1 + \exp(-[d_{dat}^t + b_r])). \quad (4.25)
\end{align*}
\]

where \(e_{w,t}\) is the 1-of-k encoding of the word \(w\) at time \(t\). Note that \(\{\alpha, \beta_1, \beta_2\}\) are learnable bias vectors that modulate the internal multiplicative interactions and the rate gate \(r\) reuses the computed pre-activation term \(d_{dat}^t\). In contrast to the model originally trained in [52], the outer activation is the linear rectifier, \(\Phi(v) = max(0,v)\), instead of the identity or hyperbolic tangent, because we found that it worked much better. We set the inner activation function \(\phi_{hid}(v)\) to be \(tanh(v) = \frac{(e^{2v} - 1)}{(e^{2v} + 1)}\).

To integrate visual context information into the ∆-RNN, we fuse the model with a neural vision system (the general model is depicted in Figure 4.3), motivated by promising recent work done in automated image captioning [198]. We adopt a transfer learning approach and incorporate a state-of-the-art convolutional neural network into the ∆-RNN model, namely the Inception-v3 network [200].\(^\text{16}\) The parameters of the vision network

\(^\text{16}\)In preliminary experiments, we also examined VGGNet and a few other variations, but found that the Inception worked the best when it came to acquiring somewhat more general distributed representations of natural images.
are fixed. As our focus is on language modeling and how the addition of visual context can improve neural network performance on the task, fixing the vision system prevents any noise from the language model from potentially corrupting the vision model and damaging its distributed representations. We leave learning the vision system jointly with the language model as future work.

To obtain a distributed representation of an image from the Inception-v3 network, we extract the vector produced from the final max-pooling layer, $c$, after running an image through the model (note that this operation occurs right before the final, fully-connected processing layers which are usually task-specific parameters, such as in object classification). The $\Delta$-RNN can make use of the information in this visual context vector if we modify its state computation in one of two ways. The first way would be to modify its inner state function to be a linear combination of the data-dependent pre-activation, the filtration, and a learned linear mapping of $c$ as follows:

$$z_t = \phi_{hid}(d_{1t} + d_{2t} + M c + b)$$  \hspace{1cm} (4.26)$$

where $M$ is a learnable synaptic connections that connect the visual context representation with the inner state. The second way to modify the $\Delta$-RNN would be change its outer mixing function instead:

$$h_t = \Phi([[1 - r] \otimes z_t + r \otimes h_{t-1}] \otimes (Mc))$$  \hspace{1cm} (4.27)$$

Here we see the linearly-mapped visual context embedding interacts with the currently computation state through a multiplicative operation, allowing the visual-context to persist and work in a longer-term capacity. In either situation, using a parameter matrix $M$ frees us from having to set the dimensionality of the hidden state to be the same as the context vector produced by the Inception-v3 network.

### 4.7 Experiments: Multimodal Language Modeling

The experiments for the task of multimodal language modeling were conducted using the MS-COCO image-captioning dataset. Images in the dataset contain significant contextual information and also five human annotated captions per image. We extracted

---

17https://competitions.codalab.org/competitions/3221
all the five sentences from the dataset and created 5 different ground truth splits. We translated ground truth splits into German and Spanish splits using state of the art Google Translation API. To our knowledge, this represents the first Multi-lingual MSCOCO dataset on situated learning. We process the corpus at the word-level and obtain a 16.6K vocabulary for English, 33.2K for German and 18.2k for Spanish.

Much as in the normal task of language modeling, our primary concern is with the next-step prediction of words/tokens, which means the negative log likelihood and perplexity of the learned generative model are of high importance. This is different from the goals of machine translation or image captioning, which, in most cases, is concerned with a ranking of possible captions where one measures how similar the model’s generated sequences are to ground-truth target phrases.

Baseline results were obtained with neural language models of text alone. For the ∆-RNN, this meant implementing a model using the equations presented in the last section. To verify that the experiment generalizes beyond the specific architecture chosen, a Gated Recurrent Unit [185] and a Long Short Term Memory [184] were also trained. We compare these symbol-only baselines to the two variations of our proposed multimodal ∆-RNN, as described in the previous section.

The multimodal variant of the GRU, where the context information is directly integrated into its inner function, is defined as follows:

\[
d_c = M c \tag{4.28}
\]

\[
z_t = \sigma(W_z x_t + V_z h_{t-1}) \tag{4.29}
\]

\[
r_t = \sigma(W_r x_t + V_r h_{t-1}) \tag{4.30}
\]

\[
\hat{h}_t = \tanh(W_h^x x_t + V_h^r r_t \otimes h_{t-1}) \tag{4.31}
\]

\[
h_t = [z_t \otimes h_{t-1} + (1 - z_t) \otimes \hat{h}_t] \otimes d_c \tag{4.32}
\]

where we note the parameter matrix \( M \) that maps the visual context \( c \) into the GRU state effectively gates the outer function.\(^{18}\) The multimodal variant of the LSTM (with peephole connections) is defined as follows:

\[
d_c = M c \tag{4.33}
\]

\(^{18}\)In preliminary experiments, we tried both ways of integrating the visual context information as proposed before, Equations 4.26 and 4.27. We ultimately found the second formulation to give better performance.
\[ h_t = [r_t \otimes \Phi(c_t)] \otimes d_c, \] where,
\[ r_t = \sigma(W_r x_t + V_r h_{t-1} + U_r c_t + b_r) \] \hspace{1cm} (4.34)
\[ c_t = f_t \otimes c_{t-1} + i_t \otimes z_t, \] where,
\[ z_t = \Phi(W_z x_t + V_z h_{t-1} + b_z), \] \hspace{1cm} (4.36)
\[ i_t = \sigma(W_i x_t + V_i h_{t-1} + U_i c_{t-1} + b_i), \] \hspace{1cm} (4.38)
\[ f_t = \sigma(W_f x_t + V_f h_{t-1} + U_f c_{t-1} + b_f). \] \hspace{1cm} (4.39)

All models were trained to minimize the sequence loss of the sentences in the training split. The weight matrices of all models were initialized from uniform distribution, \(U(-0.1, 0.1)\), biases were initialized from zero, and the \(\Delta\)-RNN-specific biases \(\{\alpha, \beta_1, \beta_2\}\) were all initialized to one. Parameter updates calculated through backpropagation through time required unrolling the model over 49 steps in time. All symbol sequences were zero-padded and appropriately masked to ensure efficient mini-batching. Gradients were hard clipped at a magnitude bound of \(l = 2.0\). Over mini-batches of 32 samples, model parameters were optimized using simple stochastic gradient descent with a learning rate that starts at \(\lambda = 1.0\) and is halved if the perplexity, measured at the end of each epoch, goes up three or more times.

To determine if our multimodal language model actually captures knowledge that is different from a text-only language model, we evaluate each model twice. First, we compute the model perplexity on the test set using the sentences’ visual context vectors. Next, we compute the model perplexity on the test sentences by feeding in a null-vector to the multimodal model as the visual context. If the model did truly pick up some semantic knowledge that is not exclusively dependent on the conditioned context vector, its perplexity in the second setting, while naturally worse than the first setting, should still outperform the text-only baselines.

In Table 4.4, we report the model negative log likelihood (NLL) and per-word perplexity (PPL). PPL is a function of NLL, and is simply calculated using the measure:

\[
PPL = \exp \left[ - \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log P_{\Theta}(w_t|h) \right]
\]  

(4.40)

We observe that in all cases the multimodal models outperform their respective text-only baselines. More importantly, the multimodal models, when evaluated without the Inception-v3 representations on held-out samples, still perform better than the
Figure 4.4. Comparison of learning curves for the Δ-RNNs in each language (English, German, Spanish).

text-only baselines. This improvement in generalization can be attributed to the visual context information given to the model in the training data, enriching its distributed representations over word sequences with knowledge of actual objects as provided by the Inception-v3 vision system. Figure 4.4 shows the validation perplexity of the various Δ-RNN on each language as a function of the first 15 epochs of learning. We observe that throughout the learning process, the improvement in generalization afforded by the visual context $c$ is persistent.
Table 4.4. Generalization performance of language models trained and evaluated on linguistic data only (L), full: trained and evaluated on multimodal linguistic and visual data (LV), and, blind: trained on multimodal data (LV) but evaluated on language only (L).

<table>
<thead>
<tr>
<th>Model/Type</th>
<th>English</th>
<th>German</th>
<th>Spanish</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Test-NLL</td>
<td>Test-PPL</td>
<td>Test-NLL</td>
</tr>
<tr>
<td>Δ-RNN (L-L)</td>
<td>2.714</td>
<td>15.086</td>
<td>2.836</td>
</tr>
<tr>
<td>MM-Δ-RNN (full LV-LV)</td>
<td>2.645</td>
<td>14.086</td>
<td>2.777</td>
</tr>
<tr>
<td>MM-Δ-RNN (blind LV-L)</td>
<td>2.694</td>
<td>14.786</td>
<td>2.808</td>
</tr>
<tr>
<td>GRU (L-L)</td>
<td>2.764</td>
<td>15.871</td>
<td>2.854</td>
</tr>
<tr>
<td>MM-GRU (blind LV-L)</td>
<td>2.687</td>
<td>14.689</td>
<td>2.815</td>
</tr>
<tr>
<td>LSTM (L-L)</td>
<td>2.722</td>
<td>15.217</td>
<td>2.814</td>
</tr>
<tr>
<td>MM-LSTM (full LV-LV)</td>
<td>2.645</td>
<td>14.089</td>
<td>2.773</td>
</tr>
</tbody>
</table>

Table 4.5. Decoder analysis: Word query similarity test.

<table>
<thead>
<tr>
<th>Ocean</th>
<th>Kite</th>
<th>Subway</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δ-RNN</td>
<td>MM-Δ-RNN</td>
<td>Δ-RNN</td>
</tr>
<tr>
<td>surfing</td>
<td>boats</td>
<td>plane</td>
</tr>
<tr>
<td>sandy</td>
<td>beach</td>
<td>kites</td>
</tr>
<tr>
<td>filled</td>
<td>pier</td>
<td>airplane</td>
</tr>
<tr>
<td>beach</td>
<td>wetsuit</td>
<td>surfboard</td>
</tr>
<tr>
<td>market</td>
<td>cloth</td>
<td>planes</td>
</tr>
<tr>
<td>crowded</td>
<td>surfing</td>
<td>airplanes</td>
</tr>
<tr>
<td>topped</td>
<td>windsurfing</td>
<td>boats</td>
</tr>
<tr>
<td>plays</td>
<td>boardwalk</td>
<td>jet</td>
</tr>
<tr>
<td>cross</td>
<td>flying</td>
<td>aircraft</td>
</tr>
<tr>
<td>snowy</td>
<td>biplane</td>
<td>jets</td>
</tr>
</tbody>
</table>

4.7.1 Model Analysis

To further probe the differences between the text-only and multimodal models, we analyze the decoders of each. Specifically, we examine the parameter matrix $U$, which
is directly involved in calculating the logits (or logarithm of the output probabilities) of the underlying generative model. $U$ can be essentially thought of as “transposed embeddings”, an idea that has also been exploited to introduce further regularization into the neural language model learning process [201,202]. If we treat each row of this matrix (since we assume column-major orientation in implementation) as the learned embedding for a particular word, we can calculate its similarity to other column embeddings using cosine similarity.

In Table 4.5, we examine the top ten highest ranked words given several query terms, using the decoder parameter matrix. By observing the different sets of nearest-neighbors produced by the $\Delta$-RNN and the MM-$\Delta$-RNN, we can see that MM-$\Delta$-RNN appears to have learned to combine the visual context information with the token sequence information in its distributed representations. For example, in the case of “ocean”, we see that while the $\Delta$-RNN does associate some relevant terms, such as “surfing” and “beach”, nearly all of the terms the MM-$\Delta$-RNN associates are relevant to the query. The same situation is observed for “kite” and “subway”. In the case of “racket”, while the text-only baseline does mostly seem to associate sports terms, especially sports equipment like “bat”, the MM-$\Delta$-RNN is actually able to relate the query correctly to the correct sport, “tennis”.

### 4.7.2 Conditional Sampling

Another interesting way to see how visual context information influences the neural language architecture is to sample from the learned conditional generative model. While image-captioning generally focuses on ranking appropriate caption candidates, we intend to use the model to generate sentences using only the image for guidance. Sampling the learned generative model will allow us to gauge if the system can “explain”, in some fashion, what it sees. Table 4.6 lists examples generated by the trained English model. Another sampling approach we implemented is beam search, where, iteratively, $m$ best sentences are picked at time $t$ from a set of generated sentences of length $t + 1$. We experimented with a beam of size 13 and Table 4.6 shows the generated captions using this specific beam-search.
Table 4.6. Some captions generated by the Multimodal Δ-RNN in English.

<table>
<thead>
<tr>
<th>Caption</th>
</tr>
</thead>
<tbody>
<tr>
<td>a skateboarder and person in front of skyscrapers.</td>
</tr>
<tr>
<td>a person with skateboarder on air.</td>
</tr>
<tr>
<td>a person doing a trick with skateboarder.</td>
</tr>
<tr>
<td>a person with camera with blue background.</td>
</tr>
<tr>
<td>a food bowl on the table</td>
</tr>
<tr>
<td>a bowl full of food on the table</td>
</tr>
<tr>
<td>a green and red bowl on the table</td>
</tr>
<tr>
<td>a salad bowl with chicken</td>
</tr>
<tr>
<td>a dog on blue bed with blanket.</td>
</tr>
<tr>
<td>a dog sleeps near wooden table.</td>
</tr>
<tr>
<td>a dog sleeps on a bed.</td>
</tr>
<tr>
<td>a dog on some blue blankets.</td>
</tr>
</tbody>
</table>

4.8 Other Applications of the Delta-RNN

4.8.1 Image Compression: Learned Iterative Decoding

For lossy\textsuperscript{19} image compression systems, the Δ-RNN has proven to be a useful component in an algorithm called iterative refinement [203]. Iterative refinement is meant to be a neural-based approach to improving a standard image decoder’s reconstruction ability, especially when starting from low-quality settings. The proposed approach for nonlinear, iterative decoding can work with any encoder, employing self-connected memory units to make use of both causal and non-causal spatial context information to progressively reduce reconstruction error over a fixed number of steps.

The experiments of [203] show that the learned proposed estimators, usually those that employed the Δ-RNN as the underlying recurrent network model, obtained as much

\textsuperscript{19}Where it is not possible to perfectly reconstruct the original image from a compressed/quantized encoding of the source.
as a 0.8921 decibel (dB) gain over the standard JPEG algorithm and a 0.5848 dB gain over a state-of-the-art neural compression model [204].

4.8.2 Data Synthesis for User Privacy

The field of disclosure control originated in statistical agencies primarily interested in survey data and has since evolved and expanded to become a very popular field in computer and information sciences as well as statistics, known generally as data privacy. These methods seek to release altered versions of the data or statistics of interest by utilizing some form of perturbation. Guiding a variety of methods are two primary goals: first to minimize the risk of privacy loss from the release or data or statistics and second to maximize the information maintained in the perturbed release relative to the information contained in the original data set.

While applications have grown to include many data types, there is a noticeable lack of protection for free text generated on social media platforms. Specifically, [205] focused on Twitter data, which is a common data source for social science researchers. In [205], an application in privacy was explored, utilizing the ∆-RNN as part of the approach. The work showed through empirical tests a computational solution might be possible with the hope the work would serve as the basis for future work in the domain of text privacy.

4.9 Conclusions

In this chapter, we developed the Differential State Framework, which affords us a useful perspective on viewing computation in recurrent neural networks. Instead of recomputing the whole state from scratch at every time step, the Delta-RNN learns how to only update the current state. This seems to be better suited for many types of problems, especially those that involve longer term patterns where part of the recurrent network’s state should be constant most of the time.

Comparison to the currently widely popular LSTM and GRU architectures shows that the Delta-RNN can achieve similar or better performance on language modeling tasks, while being conceptually much simpler and with far fewer parameters. Comparison to the Structurally Constrained Recurrent Network (SCRN), which shares many of the main ideas and motivation, shows better accuracy and a simpler model architecture.
(since, in the SCRN, tuning the sizes of two separate hidden layers is required, and this model cannot learn non-linear interactions within its longer-term memory). Furthermore, we investigated how the performance of the $\Delta$-RNN improves when visual stimuli is presented to aid in the language generation process.

Finally, we showed applications of the $\Delta$-RNN beyond the task of language modeling, such as in image compression and text privacy. These non-language related tasks demonstrate the generality and promising potential of the simple neural temporal models learned under the Differential State Framework, namely, the $\Delta$-RNN.
Chapter 5 | Reducing the Discrepancy between Local Representations

5.1 Introduction

So far in this thesis, we have developed some basic building blocks, each motivated by a specific problem context, that will be helpful in composing neural systems that are to ultimately learn continuously. In this and the next chapter, we will extend and combine these ideas to create a family of learning algorithms that are not only suitable for training incremental neural models on either temporal problems or multi-task scenarios but also are neuro-cognitively motivated. With respect to the last point, we develop approaches to learning grounded in ideas from the theory of predictive coding, the neuro-computational theoretical we first described in Chapter 2.

In this chapter, we revisit the idea of coordinated local learning, which we first described in Chapter 3 when tackling the challenging problem of semi-supervised learning, and develop a family of algorithms unified under a principle we call discrepancy reduction. We’ll also show that this family includes procedures such as Difference Target Propagation, another biologically-motivated learning procedure for training neural architectures. But first we will start by re-examining back-propagation of errors, the modern workhorse algorithm used to train neural networks, and consider its central problems from both practical and neuro-cognitive perspectives.
5.2 Problems with Back-Propagation of Errors

Although neural architectures are inspired by our current neuro-scientific understanding of the human brain, the connections to the actual mechanisms that compose systems of natural neurons are often very loose, at best. More importantly, back-propagation of errors [206], the core algorithm behind training modern high-performing connectionist models, faces some of the strongest neuro-biological criticisms, argued to be a highly implausible way in which learning occurs in the human brain. Several of the most prominent issues with backprop include:

1. the “weight transport problem”, where the feedback weights use to carry error signals must be the transposes of the feedforward weights

2. forward propagation and backward propagation utilize different computations

3. the error gradients are stored separately from the activations.

These problems largely stem from the one critical component of backprop—the global feedback pathway needed for transporting error derivatives across the system.¹

Why does backprop need a global feedback pathway? This pathway is necessary given the design of modern supervised learning system—a loss function measures the error between an artificial neural system’s output units and some target (such as a class label) and the global pathway relates how the internal processing elements affect this error. One problem with this, especially when used to generatively model data, is that most of the learner’s time is spent on surface-level properties of the data and not on extracting latent structure necessary for generalization. This makes sense since the objective function elusively operates at the input/output space. A good example is in speech processing, where the log likelihood cost used leads the model to focus mostly on acoustic details rather than higher-level linguistic features.²

From a more practical machine learning perspective, when neural models are made deeper, and thus more complex, the error gradients must pass backward through many layers of computation. As a result of the additional multiplications, these gradients tend

---
¹Note that one could instead use a so-called “error-propagation network” to combat the last two issues presented above and create two global information pathways. However, there is no known biological mechanism that allows this additional error network to know the weights of the feedforward network it is serving.

to either explode or vanish [7]. In order to remedy this problem, we keep the values of the gradients within reasonable magnitudes by requiring the layers to behave sufficiently linearly to prevent saturation of neuronal post-activities, which would yield zero gradient. However, this fix can lead to undesirable extrapolation effects, creating the well-known problem of adversarial samples [207, 208]. This requirement of linearity also violates what we know about biological neurons, which interleave linear and non-linear operations. As argued by [209], if the brain were to use a feedback pathway as implemented by back-propagation, it would require precise knowledge of the derivatives of the non-linear activation functions, which is not possible since not all neurons are the same. To make matters worse, this linearity also hinders usage of other neuro-biological, non-linear mechanisms, such as lateral competition and saturation and does not allow us to utilize discrete-valued or stochastic activations (such as sampling a Bernoulli or Categorical distribution), even though we know that real neurons communicate with spikes (binary values) and not by continuous values.

When considering modern theories of the brain [62, 199, 210, 211], which posit that local computations occur at multiple levels of the somewhat hierarchical structure of natural neural systems, this global pathway should not be necessary to learn effectively. However, if we remove this global feedback pathway, we create a new problem—where would the targets then come from in order to create learning signals for the hidden processing elements? One thing is likely: we will no longer be able to rely on a loss function that operates primarily in the input space, which is at the core of supervised learning. This means that the learning approach we seek will require higher-level objectives, or objectives that operate at various levels of latent space, not just at the input/output space as in supervised systems.

In designing higher-level objectives, we can better encourage a neural system to find hidden/abstract structure in data. This approach directly connects to representation learning, better embodying one of the key assumptions behind unsupervised learning: by observing a stream of data points, it is possible to derive the predictable systemic relationships between variables \(^3\) as well as relationships between these relationships, i.e., more complicated, abstract patterns. Higher-level patterns are what a representation learning system seeks to uncover—latent variables, or intermediate concepts and features, that capture useful statistical regularities of the world that the intelligent system is embedded in.

\(^3\)These variables can be pixels in images of a video or the characters of a word in a sentence.
This chapter will develop various ways to find targets for the various hidden layers of internal processing elements of a neural system.

To this end, this chapter will develop a family of learning algorithms, unified under the principle of what we will call discrepancy reduction, that can find the targets mentioned earlier and create the error signals needed for updating the parameters of a neural system. These algorithms will embody the idea that coordinated local learning, a concept that we first sketched in Chapter 2 to develop better semi-supervised neural systems, can offer a viable, neuro-cognitive mechanism for effectively training connectionist models.

### 5.3 Discrepancy Reduction: Credit Assignment through Local Error Signals

The governing principle for learning behind this chapter we will refer to as discrepancy reduction. This principle dictates that, in the face of a stream of samples from the environment, a neural system must reduce the mismatch between its current representation of that environment, which could very well be a multi-level representation, and a better way of representing the environment, i.e., a target representation. This means that mismatch can occur at any and all levels of abstraction and, as a result, will mean that a function is needed to measure this mismatch.

As mentioned before, this chapter proposes a family of algorithms, the Discrepancy Reduction family, that offer computational mechanisms to perform the following two steps when learning from a sample (or mini-batch of samples):

1. Search for latent representations that better explain the input/output, also known as target representations. This facilitates the need for local (higher-level) objectives that will help guide the current latent representations towards better ones.

2. Reduce, as much as possible, the mismatch between a model’s currently “guessed” representations and target representations. The sum of the internal, local losses is also defined as the total discrepancy $L_E$ in a system, and can also be thought of a sort of pseudo-energy function.

This general two-step process forms the basis of what we call coordinated local learning rules. Computing targets with these kinds of rules should not require an actual pathway, as in back-propagation, and instead make use of top-down and bottom-up signals to create...
targets. This idea is particularly motivated by the theory of predictive coding [60], which claims that the brain is in a continuous process of creating and updating hypotheses to predict the sensory input through the use of top-down generation and mediation combined with bottom-up sensory feedback. We will explore several ways in which this hypothesis updating\(^4\) might happen through mechanisms such as: a) through error-correction in Local Representation Alignment (LRA), and b) through repeated encoding and decoding as in Difference Target Propagation (DTP).

Formally, in a \(L\)-layer system, we will require a set of \(N\) local objective functions, \(\mathcal{L}_E = \{\mathcal{L}_0(z_0, y_0^z), \ldots, \mathcal{L}_N(z_N, y_N^z)\}\(^5\) where \(L\) is the number of processing layers of the neural system. \(Z = \{z^0, \ldots, z^\ell, \ldots, z^N\}\) are the representations currently produced by the neural system and \(Y^Z = \{y^0_z, \ldots, y^\ell_z, \ldots, y^N_z\}\) is the set of better representations the system should move towards. What the system will now try to minimize is the total discrepancy, \(\mathcal{L}_E\), which accounts for not only error measurements in the output space but also those made in latent space. The full objective, total discrepancy, can be expressed as follows:

\[
\mathcal{L}_E(Z, Y^Z) = \sum_{\ell=0}^{N} \mathcal{L}_\ell(z^\ell, y^\ell_z). \quad (5.1)
\]

Since it is one grand summation of terms, one can see that by differentiating the total discrepancy with respect to each particular representation \(z^\ell\) there will be one set of error signals per layer of neurons. In short, we see that:

\[
\frac{\partial \mathcal{L}_E(Z, Y^Z)}{\partial z^\ell} = \frac{\partial \mathcal{L}_0(z^0_z, y^0_z)}{\partial z^\ell} + \ldots + \frac{\partial \mathcal{L}_\ell(z^\ell_z, y^\ell_z)}{\partial z^\ell} + \ldots + \frac{\partial \mathcal{L}_N(z^N_z, y^N_z)}{\partial z^\ell} \quad (5.2)
\]

since all of the partial derivatives disappear when considering particular layer of processing elements. It is very important to treat each layer \(z^\ell\) as non-differentiable in this case, otherwise, for any layer above \(\ell\), we would have derivative signals traveling back through the global pathway defined by the chain rule of calculus.

This chapter will present several simple metrics in order to define the local losses above, including one to encourage sparse representations of the data. However, defin-

\(^4\)We can also call this “local target creation”.

\(^5\)Note that it is not necessary for \(N = L\) given we might only need to measure losses at only key intermediate layers provided we have a mechanism for transmitting error to layers that have no losses operating on them. Furthermore, some local losses might not be necessary, such as \(\mathcal{L}_0\), which would operate at the input layer of sensory neurons, underneath which there are no further parameters to modify.
Figure 5.1. How the LRA algorithm is applied to a multilayer perceptron (MLP). A single computation subgraph is selected in the red dashed box. Note how LRA-fdbk can easily handle non-differentiable functions, \( f(\cdot) \), since these can be viewed as being housed within the computation subgraph. The short-circuit error feedback connection simply wires directly to the input pre-activation variables.

5.4 The Local Representation Alignment Algorithm

To simplify notation, we describe LRA in the context of a multilayer perceptron architecture. However, LRA naturally applies to any stacked neural architecture, including those that are recurrent. In the supplementary material we provide the general formulation of LRA such that it may be applied recurrent models as well.

5.4.1 Notation

Let \( z^{\ell-1} \) be the inputs, as computed in the feed-forward phase, to the nodes in layer \( \ell \). This means that \( z^\ell \) is also the output of layer \( \ell \) (hence we call it the post-activation). Let \( W_\ell \) be the weight matrix at layer \( \ell \) that multiplies the inputs \( z^{\ell-1} \). Let \( h^\ell \) be the pre-activation of layer \( \ell \) (i.e. \( h^\ell = W_\ell z^{\ell-1} \)). Let \( f^\ell \) be the vector of activation functions.
for layer $\ell$, so that $z^\ell = f^\ell(h^\ell)$.

Following TargetProp [214], during training, in each weight update iteration, for each layer $\ell$ we set a target $y^\ell_z$ for the feedforward post-activations $z^\ell$. Then we take a gradient descent step on the network parameters $W_\ell$ to move $z^\ell$ closer to $y^\ell_z$ by minimizing a layer-specific loss function $L_\ell(z^\ell, y^\ell_z)$. Thus we iteratively choose new targets, then refine weights, then choose new targets, etc. This setup is illustrated in Figures 5.1(a) and 5.1(b). However, unlike target propagation [214], LRA will not modify the network architecture.

5.4.2 Setting the targets.

We first explain how backpropagation fits into this framework. One way, noted by [214], is to set the target for the output of layer $\ell$ to be its current feed-forward value plus the gradient of the overall loss with respect to the output of layer $\ell$ (and the loss function is squared loss). This is a global view - the target of each layer is set to optimize the global loss. Instead, we present a more local view in that the target for layer $i$ is specifically set to help layer $\ell + 1$ reach its target. This view is the starting point that will allow us to consider the training of a deep network as training on computation subgraphs (see Figures 5.1(a), 5.1(b)) that can be optimized locally.

In our view of backprop as target prop, we use $L_2$ loss for each layer and set the target $y^\ell_z$ of layer $\ell$ so that the difference between its output $z^\ell$ and target $y^\ell_z$ is the gradient of the next layer’s loss with respect to $z^\ell$ (i.e. the direction that $z^\ell$ should move to achieve the steepest local change in the next layer’s output).

**Lemma 1** Consider a network with $n$ layers, input $x$, feedforward activations $z^\ell$ ($\ell = 1, \ldots, n$), output layer target $t$, and squared loss $L_\ell$ for layers $1, \ldots, \ell - 1$ (that is, except the top layer). Recursively set $y^n_z = t$ and (for $\ell = n - 1, \ldots, 1$) $y^\ell_z = z^\ell - \nabla z^\ell L_{\ell+1}(z^{\ell+1}, y^{\ell+1}_z)$ (where $z^{\ell+1}$ is considered a function of $z^\ell$). A simultaneous one-step gradient descent update to all the weight matrices $W_1, \ldots, W_n$ with respect to the layer-wise losses is equivalent to one step of back-propagation.

**Proof** It suffices to show that if $W_{\ell,j}$ is the weight vector for node $j$ in layer $\ell$, then the derivative of the loss for layer $\ell$ with respect to these weights is the same as the derivative

---

6During the weight update, for the purposes of computing derivatives, the current target $y^\ell_z$ is not treated as a function of the input weights (so that the feed-forward output moves closer to the desired targets rather than vice versa).
of the overall loss with respect to these weights: 
\[
\frac{d}{dW_{\ell,j}} L_\ell(z^\ell, y^\ell_z) \equiv \frac{d}{dW_{\ell,j}} \frac{1}{2} ||z^\ell - y^\ell_z||_2^2 = \nabla_{W_{\ell,j}} L_n(z^n, t)
\]
for each \(j\) and for \(\ell = 1, \ldots, n - 1\) (since the update to the weights \(W_n\) at the top layer is always a gradient descent update). We will prove the stronger statement, that for any \(k \geq \ell\), then 
\[
\frac{d}{dW_{\ell,j}} L_k(z^k, y^k_z) = \nabla_{W_{\ell,j}} L_n(z^n, t)
\]
for each \(j\) and for \(\ell = 1, \ldots, n - 1\)

In the base case, when \(k = n - 1\),
\[
\frac{d}{dW_{\ell,j}} L_{n-1}(z^{n-1}, y^{n-1}_z) = \frac{d}{dW_{\ell,j}} \frac{1}{2} ||z^{n-1} - y^{n-1}_z||_2^2 = \frac{d}{dW_{\ell,j}} \nabla_{z^{n-1}} L_n(z^n, t)
\]
by the chain rule. For the inductive step, we show that if the result is true for \(k\) (i.e. \(\frac{d}{dW_{\ell,j}} L_k(z^k, y^k_z) = \nabla_{W_{\ell,j}} L_n(z^n, t)\)) then it is also true for \(k-1\) (i.e. \(\frac{d}{dW_{\ell,j}} L_{k-1}(z^{k-1}, y^{k-1}_z) = \nabla_{W_{\ell,j}} L_n(z^n, t)\)). So

\[
\frac{d}{dW_{\ell,j}} L_{k-1}(z^{k-1}, y^{k-1}_z) = \frac{d}{dW_{\ell,j}} \frac{1}{2} ||z^{k-1} - y^{k-1}_z||_2^2 = \frac{d}{dW_{\ell,j}} \nabla_{z^{k-1}} L_k(z^k, y^k_z)
\]
by the chain rule and inductive hypothesis.

Hence mini-batch gradient descent can be viewed as, for every iteration, selecting a mini-batch, choosing targets for each layer to obtain a per-layer optimization problem, and partially optimizing the layer-wise loss (with one gradient step).

This view naturally leads to three ways to improve training:

- The target \(y^\ell_z\), intuitively, is a desired value for the output of layer \(\ell\) that will help layer \(\ell + 1\) lower its loss. Thus it is important to ensure that the target is actually representable by layer \(\ell\). Thus we look at the pre-activation \(h^\ell\) (i.e. inputs
to the nodes at layer $\ell$) and determine what values of $h_\ell^\ell$, when fed through the activation function of layer $\ell$, will help layer $\ell + 1$ reduce its loss. Hence we set target $y_\ell^\ell$ for the pre-activation $h_\ell^\ell$ and feed these pre-activation targets through the activation function to obtain the targets $y_z^\ell$ for layer $\ell$, as shown in Figures 5.1(a) and 5.1(b). The target value for $h_\ell^\ell$ can be set with one step of gradient descent. Thus we can set the target $y_\ell^h = h_\ell^\ell - \eta \nabla_{h_\ell^\ell} L_{\ell+1}(z_{\ell+1}^z, y_{\ell+1}^z)$ and then $y_z^\ell = f^\ell(y_\ell^h) = f^\ell \left( h_\ell^\ell - \eta \nabla_{h_\ell^\ell} L(z_{\ell+1}^z - y_{\ell+1}^z) \right)$.

- To choose better pre-activation targets $y_\ell^h$, we can perform multiple gradient descent steps for $y_\ell^h$ on the loss of the next layer $L_{\ell+1}(z_{\ell+1}^z, y_{\ell+1}^z)$ to find a more helpful value of the pre-activation target. This can be viewed as walking along the manifold of $z_\ell^\ell$ parametrized by $h_\ell^\ell$, to find a pre-activation target $y_\ell^h$ (and hence $y_z^\ell$) that would help the next layer reduce its loss. An alternative to gradient steps is to use feedback alignment to update the targets (instead of the standard approach of updating the weights [24, 25]).

- The per-layer loss can be customized for each layer. For example, the least squares loss can be replaced by the $L_1$ norm or the log-penalty (Cauchy).

After the targets are set, the weights are updated with one step of gradient descent (possibly using an adaptive learning rate rule, e.g., Adam or RMSprop). In designing to handle discrete-valued activations, which will be discussed later in this chapter, we further present a simpler way to finding better input pre-activation values for a given subgraph, employing the use of a “short-circuit” connection (only used during training) to form the error pathway instead of resorting to full reverse-mode differentiation of the subgraph’s local loss with respect to the input pre-activities.

### 5.4.3 Divide and Conquer: The Computation Subgraph

LRA aims to decompose the larger credit assignment problem in neural architectures into smaller, easier-to-solve sub-problems. With this in mind, we can view any neural architecture, or rather, its full, underlying operation graph, as a composition of “computation subgraphs”. A directed, acyclic computation graph may be decomposed into a set of smaller direct subgraphs, where a subgraph’s boundaries are defined as the set of input node variables and the set of output node variables.
To be more specific, the input to a subgraph of a multilayer neural architecture is the vector of pre-activation values, \( h_{\ell-1} \) computed from the subgraph below (unless this is the bottommost subgraph which means that the input is simply the raw feature vector). The output of the subgraph is simply the post-activation it computes as a function of its input, or simply, \( z_\ell \). Note that while we present our notation to imply that a computation subgraph only encapsulates two layers of actual processing elements, layers \( \ell - 1 \) and \( \ell \), the subgraph itself could be deep and include processing elements one decides are internal to the subgraph itself. This would allow us to house self-connected nodes inside the graph as well, assuming that the subgraph is temporal and is intended to compute given sequences of inputs/outputs. In choosing boundaries, one could distinguish which node layers are to be representations (or latent variables) and which node layers are simply inner computation elements, meant to support the representation nodes.

Figures 5.1(a) and 5.1(b) show one subgraph emphasized against the general computation structure inside the red dashed boundary lines (in the case of a feedforward model, the graph is simply a linked chain of operations). Note that this subgraph also includes a loss function \( L_\ell \) and the target nodes \( y^\ell \).

### 5.4.4 Instantiation: The Multilayer Perceptron

To make concrete the dynamics and implementation of LRA, we describe how the LRA algorithm works in the case of a feedforward neural architecture. Algorithm 7 depicts the full procedure. \( L^{vec}_\ell(z_\ell, y^\ell) \) simply means a vector of computed losses, one per sample (so in a mini-batch of \( m \) samples, we would obtain a vector of \((1 \times m)\) scalar loss values, assuming column-major orientation), as opposed to the full, summed loss \( L_\ell(z_\ell, y^\ell) \). In the event that mini-batches of samples are used, we create a so-called “depth mask” \( m \) to allow LRA to decide how deep the credit assignment should go on a per-sample basis. \( \otimes \) is used to denote the Hadamard product (or elementwise multiplication). When using the first displacement function of Algorithm 8, we refer to the resulting LRA procedure as \( LRA-diff \), to make clear that the full subgraph must be fully differentiable. If the second displacement function is used, we will refer the algorithm as \( LRA-fdbk \) to signify that error feedback connections are used to short-circuit the subgraph and avoid the need for full differentiability.

A multilayer perceptron, or feedforward network, is defined by the set of parameters \( \Theta = \{ W_1, b_1, \cdots, W_\ell, b_\ell, \cdots, W_L b_\ell \} \). Note that \( b_\ell \) is the bias vector applied to units
**Algorithm 7** The LRA Algorithm as applied to feedforward neural architectures.

**Input:** data \((x, t)\), step-size \(\lambda\), number steps \(K\), scalar norm constraints \(c_1\) and \(c_2\), halting criterion \(\epsilon\), step-size \(\beta\), model parameters \(\Theta = \{W_1, b_1, \ldots, W_\ell, b_\ell, \ldots, W_n, b_n\}\)

**Specify:** \(\{f_1(h), \ldots, f_\ell(h), \ldots, f_n(h)\}\), \(\{L_1(z, y), \ldots, L_\ell(z, y), \ldots, L_n(z, y)\}\), and, optionally, \(\{E_1, \ldots, E_\ell, \ldots, E_n\}\)

// \(y^h_\ell\): what we would like the input to the activation function at layer \(\ell\) to be
// \(y^z_\ell\): what we would like the output of activation function \(f_\ell\) at layer \(\ell\) to be
// \(h_\ell\): input to the activation function at layer \(\ell\) resulting from feed forward phase
// \(z_\ell\): output of the activation function at layer \(\ell\) resulting from feed forward phase
// \(\bar{h}_\ell\) indicates a temporary variable for \(h_\ell\) and \(\bar{z}_\ell\) indicates a temporary variable for \(z_\ell\)

**Function** [ComputeUpdateDirection] \(\{(x, t), \Theta, K, c, \beta, \epsilon\}\)

\(z_0 = x\)  \(\triangleright \) Run feedforward pass to get initial layer-wise statistics

for \(\ell = 1\) to \(n\) do

\(W_\ell, b_\ell, \leftarrow \Theta\)

\(h_\ell \leftarrow W_\ell z_{\ell-1} + b_\ell, y^h_\ell \leftarrow h_\ell, \bar{h}_\ell \leftarrow h_\ell\)

\(z_\ell \leftarrow f_\ell(h_\ell), y^z_\ell \leftarrow z_\ell, \bar{z}_\ell \leftarrow z_\ell\)

\(y^z_n \leftarrow t\)  \(\triangleright \) Override top-level target with correct target from training data

while \(\ell \geq 1\) do

\(m = L^{sec}_\ell(z_\ell, y^z_\ell) \geq \epsilon\)  \(\triangleright \) Create a depth mask based on local loss at \(\ell\)

\(z^\ell_\ell \leftarrow z_\ell \otimes m, y^{z^\ell}_\ell \leftarrow y^z_\ell \otimes m, \bar{z}_\ell \leftarrow \bar{z}_\ell \otimes m\)

if \(L_\ell(z_\ell, y^{z_\ell}_\ell) \geq \epsilon\) then

\(W_\ell, b_\ell, \leftarrow \Theta\)

// Calculate parameter update direction for layer \(\ell\), comparing initial guess to target

\(\nabla_{W_\ell} \leftarrow \frac{\partial L_\ell(z_\ell, y^{z_\ell}_\ell)}{\partial W_\ell}, \nabla_{b_\ell} \leftarrow \frac{\partial L_\ell(z_\ell, y^{z_\ell}_\ell)}{\partial b_\ell}\)

\(\nabla_{W_\ell} \leftarrow \text{Normalize}(\nabla_{W_\ell}, c_1), \nabla_{b_\ell} \leftarrow \text{Normalize}(\nabla_{b_\ell}, c_1)\)

\(k = 1\)  \(\triangleright \) Calculate representation target for layer \(\ell - 1\)

while \(k \leq K\) do

// Calculate pre-activation displacement

\(\Delta_{h_{\ell-1}} \leftarrow \text{GetLatentUpdate}(\tilde{h}_\ell, \bar{z}_\ell, \bar{h}_{\ell-1}, \bar{z}_{\ell-1}, y^{z\ell}_\ell, W_\ell, b_\ell, E_\ell)\)

\(\Delta_{h_{\ell-1}} \leftarrow \text{Normalize}(\Delta_{h_{\ell-1}}, c_2)\)

// Recalculate input pre/post activities of subgraph

\(\tilde{h}_{\ell-1} \leftarrow \bar{h}_{\ell-1} - \beta \Delta_{h_{\ell-1}}, \bar{z}_{\ell-1} \leftarrow f_{\ell-1}(\tilde{h}_{\ell-1})\)

// Recalculate output pre/post activities of subgraph

\(h_\ell \leftarrow W_\ell \bar{z}_{\ell-1} + b_\ell, \bar{z}_\ell \leftarrow f_\ell(h_\ell)\)

\(k = k + 1\)  \(\triangleright \) Can also add a stopping criterion here based on local loss at \(\ell\)

\(y^{z_{\ell-1}}_\ell \leftarrow \bar{z}_{\ell-1} \otimes m\)  \(\triangleright \) Update variable holding target for subgraph below

Return \(\nabla_\Theta = \{\nabla_{W_1}, \nabla_{b_1}, \ldots, \nabla_{W_\ell}, \nabla_{b_\ell}, \ldots, \nabla_{W_n}, \nabla_{b_n}\}\)

EndFunction
Algorithm 8 Internal displacement calculation routines to use in the inner loop of LRA.

**Input:** subgraph parameters \( \{W_\ell, b_\ell\} \), feedback parameters \( E_\ell \), representation target \( y_\ell^z \), and the subgraph defined by the input pre/post activities \( \{h_\ell-1, z_\ell-1\} \) and output pre/post activities \( \{h_\ell, z_\ell\} \)

// Calculus-Driven Representation Target Update

**Function** {GetLatentUpdate} \( \{h_\ell, z_\ell, h_\ell-1, z_\ell-1, y_\ell^z, W_\ell, b_\ell, E_\ell\} \)

\[
\frac{\partial L_\ell (z_\ell, y_\ell^z)}{\partial h_{\ell-1}} = (W_\ell)^T \left( \frac{\partial L_\ell (z_\ell, y_\ell^z)}{\partial z_\ell} \otimes \frac{\partial f_\ell (h_\ell)}{\partial h_\ell} \right) \otimes \frac{\partial f_{\ell-1} (h_{\ell-1})}{\partial h_{\ell-1}}, \quad \Delta h_{\ell-1} \leftarrow \frac{\partial L_\ell (z_\ell, y_\ell^z)}{\partial h_{\ell-1}}
\]

Return \( \Delta h_{\ell-1} \)

EndFunction

// Error-Feedback-Driven Representation Target Update

**Function** {GetLatentUpdate} \( \{h_\ell, z_\ell, h_\ell-1, z_\ell-1, y_\ell^z, W_\ell, b_\ell, E_\ell\} \)

\[
\frac{\partial L_\ell (z_\ell, y_\ell^z)}{\partial h_{\ell}} = \left( \frac{\partial L_\ell (z_\ell, y_\ell^z)}{\partial z_\ell} \otimes \frac{\partial f_\ell (h_\ell)}{\partial h_\ell} \right), \quad \Delta h_{\ell-1} \leftarrow E_\ell \left( \frac{\partial L_\ell (z_\ell, y_\ell^z)}{\partial h_{\ell}} \right)
\]

Return \( \Delta h_{\ell-1} \)

EndFunction

at layer \( \ell \). To measure the discrepancy between a hidden layer’s current representation, conditioned on data, and a computed target one may employ many possible loss functions depending on the distributions assumed over hidden units. If we choose to assume a Gaussian distribution (with an identity covariance matrix), the (instantaneous) local loss to measure the agreement between the representation and target would be the L2-norm, defined as

\[
L_\ell (z, y) = \frac{1}{2} \sum_{i=1}^{\left|z\right|} (y_i - z_i)^2. \tag{5.3}
\]

We could also assume the representations should be Laplacian distributed (or the L1-norm), which is defined as:

\[
L_\ell (z, y) = \sum_{i=1}^{\left|z\right|} |(y_i - z_i)|. \tag{5.4}
\]

After preliminary experimentation, we actually found a different loss, the log-penalty, to work much better across the gamut of activation functions, including non-zero mean functions like the logistic sigmoid. The log-penalty function is derived from the log likelihood of the Cauchy distribution. In this chapter, we implemented the log-penalty
loss, for a single vector, as:

\[ \mathcal{L}_\ell(z, y) = \sum_{i=1}^{\left| z \right|} \log\left(1 + (y_i - z_i)^2\right) \]  \hspace{1cm} (5.5)

where the loss is computed over all dimensions \( \left| z \right| \) of the vector \( z \) (where a dimension is indexed/accessed by integer \( i \)). This means we assume a Cauchy distribution, over the representation space at layer \( \ell \), \( P(y_i) \propto \frac{1}{(1 + (y_i - z_i))} \), with a location parameter \( z \) that is learned from data. Future research should develop alternative metrics to the ones we propose in order to learn neural architectures with hidden activation patterns that align better with calculated target representations of input data.

### 5.4.4.1 Handling Non-Differentiable Activations

LRA-fdbk can handle non-differentiable activation functions in one of two general ways. If a nonlinearity such as the signum (or Heaviside step) function, which is discrete-valued, is used at the output of each computation subgraph, the error feedback connections can simply transmit the derivative of the local loss with respect to the post-activation \( z_\ell \) directly to the pre-activation output nodes \( h_\ell \). This creates a small pathway that just directly skips over the non-differentiable activation function.

Another way to circumvent the non-differentiable nonlinearity would be to directly wire derivative of the local loss with respect to the post-activation \( z_\ell \) to the pre-activation input nodes \( h_{\ell-1} \). This option is even more attractive as this entirely short-circuits the normal pathway of the error signal when applying the chain rule of calculus to the operation subgraph. In either case, this effectively “skips” the calculation of the derivative of the signum function altogether.

Building on this second approach, if one puts the problematic nonlinearity “inside” the computation subgraph, such as in the simple architecture investigated in [214], meaning that the post-activation of the output nodes is no longer this non-differentiable function, we can create a short-circuit path from the derivative of the output post-activation directly to the input pre-activities. Surprisingly, this setting worked quite well in this study—and interestingly enough, we found that it was not only robust to initialization but that it permits training from \textit{null}, or zero, initialization.

We call this variation of the LRA procedure \textit{LRA-fdbk}. The full algorithm can be implemented by using the second displacement routine in Algorithm 8 for the
GetLatentUpdate(·) sub-routine in Algorithm 8. LRA-diff refers to using the first displacement function, i.e., reverse-mode differentiation, in Algorithm 8 for GetLatentUpdate(·) in Algorithm 8.

5.4.4.2 Overcoming Poor Initializations

Poor initializations affect networks with various activations differently, as we shall observe in the experiments later in this chapter. LRA, in both forms, can be seen as correcting for poor settings—something back-propagation of errors cannot do. LRA-diff can, if given a large enough local computation budget $K$, “walk away” from poor settings quickly since its goal is to first find better layerwise representations that explain the relationship between the input $x$ and output $y$.

Later, we will investigate how robust LRA is to various settings of the initialization scheme and how other algorithms, especially target propagation and feedback alignment, compare.

5.4.4.3 Overcoming Exploding and Vanishing Gradients

When neural networks are made deeper, error gradients must pass backward through many layers using the global feedback pathway provided by back-propagation. As a result of these extra multiplications, these gradients tend to either explode or vanish [7, 215]. In order to keep the values of the gradients within reasonable magnitudes and prevent zero gradient, we often impose constraints to ensure that layers are sufficiently linear in order to prevent post-activations from reaching their saturation regimes. However, this required linearity can create less than desirable side-effects, e.g., adversarial samples [207, 208].

LRA handles the vanishing gradient problem by tackling the global credit assignment in a local fashion, using the perspective of computation subgraphs as described earlier in Section 5.4.3. In other words, LRA treats the underlying graph of the neural graphical model as a series of subgraphs and then proceeds to optimize each subproblem. This is the essence of local learning in LRA. To overcome exploding gradients, LRA makes use of gradient re-projection, which is often used to introduce stability in recurrent neural networks [7]. Re-projection, embodied in the Normalize(·) function call, is used in two places within the LRA procedure—1) rescale parameter updates $\nabla W_\ell$ to have Frobenius norm and $\nabla b_\ell$ to have $L_2$ norm equal to $c_1$, and 2) rescale the calculated representation displacement to have $L_2$ norm equal to $c_2$. The exact implementation of Normalize(·)
is identical to that of Algorithm 1 of [7].

## 5.5 Experiments: Properties of LRA

For all experiments in this section, we keep the parameter optimization setting the same for all scenarios so that we may tease out the effects of individual learning algorithms instead. Specifically, updates calculated by each algorithm are used in a simple first-order gradient descent with a fixed learning rate of 0.01 and mini-batches of 50 samples. Our goal is not to reach state-of-the-art on any particular task but to instead to investigate the optimization ability and robustness of our algorithm in comparison to back-propagation and alternatives. Since finding out optimal hyperparameters for any neural network is non-trivial task and poor initialization can lead to unstable or poor performance. In our experiments we found that LRA helps neural network in worst settings and overcome limitations of alternative algorithm.

We briefly describe the datasets used to investigate the ability of each learning algorithm in training deep, nonlinear networks. Note that these datasets will remain the same throughout the duration of this chapter.

**MNIST:** The popular MNIST dataset [216] contains 28x28 grey-scale pixel images, each belonging to one of 10 digit categories. There are 60,000 training images, from which we create a validation subset of 10,000 images, and 10,000 test images.

**Fashion MNIST:** Fashion MNIST [217] is a dataset composed of 28x28 grey-scale images of clothing items, meant to serve as a much more difficult drop-in replacement for MNIST itself. The size and structure of the training and testing splits are the same as in MNIST and each image is associated with one of 10 classes. We create a validation set of 10,000 samples from the training split via random sampling without replacement.

### 5.5.1 Effect of Manifold-Walking

We first investigate how LRA-diff’s computation budget \( k \), specifically the number of sub-optimization steps allocated per subgraph, affects its ability to train a highly nonlinear network. Furthermore, we contrast this procedure against the vastly simpler error-feedback variant, LRA-fdbk. To do so, we construct networks of three hidden layers of 64 hyperbolic tangent units with biases initialized from zero and weights according to
Figure 5.2. Drop in test-set error during first few epochs of training tanh networks on the MNIST and Fashion MNIST image datasets.

the following classical heuristic:

\[ W_{ij} \sim U \left[ -\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}} \right], \quad (5.6) \]

noting that \( n_{in} \) is the size of previous/incoming layer of post-activities or the number of columns of the weight matrix \( W \) (if working in column-major form). \( U[-a, a] \) is the
uniform distribution in the interval $(-a, a)$.

For LRA-diff, we varied the computation budget $k = \{5, 10, 30, 50\}$, and for LRA-fdbk, we varied $\sigma_E = \{1.0, 1.25, 1.5, 2.0\}$ (note in the plots this is $sd$), the standard deviation of the error feedback weights. Both variants of LRA employed the Cauchy loss (see Equation 5.5) as the metric for measuring discrepancy between representation and target. Networks were trained over 100 epochs but we only show the first 5 epochs, since roughly after this point, the differences in generalization rates were too similar to warrant visualization.

In Figure 5.2, for LRA-diff, we see that increasing $k$ leads to ultimately better generalization and sooner on both MNIST and Fashion MNIST. However, there is a diminishing return as one dramatically increases the number of steps from $k = 30$ to $k = 50$. More importantly, LRA-fdbk, which is a far faster variation of LRA and only requires that the output post and pre-activation functions of any subgraph are differentiable, reaches the same level of generalization. This means LRA-fdbk is able to use the short-circuit feedback connections to create a useful displacement for the model’s current input representation to help lower its local loss. While, like LRA-diff, multiple steps can also be taken using LRA-fdbk within any subgraph, we found that this was largely unnecessary in preliminary experiments, and taking a single step along the layerwise manifold using simple Gaussian feedback weights worked well across all settings tried in this thesis. We found that the initialization of the error feedback weights more directly affects LRA-fdbk’s performance, though as one raises the standard deviation, the impact is far less severe.
than varying the number of steps in \textit{LRA-diff}.\footnote{Choosing a value for the standard deviation that is too low, especially below one, however, can slow down the learning process. We found that naively using a standard deviation of one worked quite well for the experiments in this thesis and thus did no further tuning after Experiment 5.5.1.}

In Figure 5.3, we show the filters acquired by the feedforward network on Fashion MNIST after a single epoch. To create these filter visualizations, we employ the feature activation maximization approach as discussed in [218]. Furthermore, while this approach generally only applies to the first hidden layer of units, which sit closest to the input pixel nodes, we can apply the same technique to the upper hidden layers of the network, such as the third layer, by simply ignoring the nonlinearity at each level of the model. Thus, we approximately “linearize” the nonlinear network which allows us to collapse successive weight matrices back into a single matrix (taking advantage of this natural property of deep linear networks). This will incur some minor approximation error, since the network is not truly linear, but we found that this approximation gave us a very fast and reasonably good picture of what knowledge might be captured in the synaptic connections that form the memories of the upper layer nodes, i.e., those closest to the output layer. Observe that both versions of LRA (note that for \textit{LRA-diff} we use the network trained with $k = 30$) learn reasonably good and clear filters after just one pass through the data. Back-propagation, however, learns far noisier filters. Again, it is surprising to see that \textit{LRA-fdbk} learn so well with only one pass through the data, given that it is far cheaper computationally than \textit{LRA-diff}. Encouraged by this positive behavior, its low computation requirements, and the fact that \textit{LRA-diff} can also handle a far greater range of activations, such as discrete-valued ones, we focus on \textit{LRA-fdbk} for the rest of the section.

\subsection*{5.5.2 Robustness to Initialization}

It is very difficult to train deep (and thin) networks from simplistic initialization schemes [219]. Furthermore, [220] showed that using the logistic sigmoid as an activation function can slow down learning considerably, largely due to its non-zero mean, which was further investigated in [221]. Given the problems that come with unit saturation and vanishing gradients [215], training a very deep and thin network, especially composed of logistic sigmoid units, with only back-propagation, can be very difficult if not near impossible.

To investigate LRA’s robustness to poor initialization, we use it to train deep nonlinear networks consisting of either logistic sigmoid or hyperbolic tangent post-activation
functions with model parameters initialized from a parametrized, zero-mean Gaussian distribution. The Gaussian distribution is a very simple, common way to initialize the parameters of a neural model, and controlling its standard deviation, $\sigma$, allows us to probe different cases when back-propagation fails. In this experiment, we investigate the settings $\sigma = \{0.025, 0.05, 0.1\}$, and compare LRA and backprop ($BP$) to algorithms
Table 5.1. Training and generalization error (%) of deep sigmoid or tanh networks, trained with various learning algorithms, including backprop and our proposed LRA-fdbk.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MNIST</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Fashion MNIST</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\phi(\cdot) = \text{Sigmoid}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
</tr>
<tr>
<td>Backprop</td>
<td>X</td>
<td>88.66</td>
<td>88.66</td>
<td>88.66</td>
<td>X</td>
<td>88.65</td>
<td>2.67</td>
<td>2.48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FA</td>
<td>88.65</td>
<td>88.65</td>
<td>88.65</td>
<td>88.65</td>
<td>16.1</td>
<td>12.73</td>
<td>11.86</td>
<td>11.81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFA</td>
<td>30.84</td>
<td>31.78</td>
<td>31.77</td>
<td>26.81</td>
<td>5.24</td>
<td>4.11</td>
<td>5.12</td>
<td>4.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTP</td>
<td>X</td>
<td>18.06</td>
<td>16.83</td>
<td>17.06</td>
<td>X</td>
<td>67.51</td>
<td>5.38</td>
<td>3.19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRA-fdbk</td>
<td>2.75</td>
<td>2.85</td>
<td>2.89</td>
<td>2.96</td>
<td>2.69</td>
<td>2.87</td>
<td>2.82</td>
<td>3.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\phi(\cdot) = \text{Sigmoid}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
<td>$\phi(\cdot) = \text{Tanh}$</td>
</tr>
<tr>
<td>Backprop</td>
<td>X</td>
<td>90.00</td>
<td>90.00</td>
<td>90.00</td>
<td>X</td>
<td>29.04</td>
<td>11.48</td>
<td>11.38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FA</td>
<td>90.00</td>
<td>90.00</td>
<td>90.00</td>
<td>89.97</td>
<td>24.82</td>
<td>24.57</td>
<td>21.67</td>
<td>20.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFA</td>
<td>35.91</td>
<td>37.4</td>
<td>41.62</td>
<td>37.7</td>
<td>13.92</td>
<td>12.82</td>
<td>13.95</td>
<td>14.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTP</td>
<td>X</td>
<td>41.1</td>
<td>26.28</td>
<td>24.69</td>
<td>X</td>
<td>79.84</td>
<td>60.19</td>
<td>13.59</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

such as Difference Target Propagation (DTP) [214], Feedback Alignment (FA) [24], and Direct Feedback Alignment (DFA) [25]. Furthermore, we also show the situation where the weights are simply initialized to zero, $\emptyset$. Whenever it was not possible to learn from zero with a given algorithm, such as BP and DTP, we simply marked the appropriate slot with an $X$. To initialize the feedback weights of DFA and FA, we follow the protocol prescribed in [25].

The network architecture each algorithm is responsible for training is the same: a multilayer perceptron containing eight hidden layers of 128 processing elements. We examine the ability of each algorithm to train the same architecture employing logistic sigmoid (a non-zero mean activation function) and the hyperbolic tangent (a zero-mean activation function).

In Table 5.1, we present the best found generalization error rate for the deep architecture learned with each algorithm under each initialization setting. Observe that LRA-fdbk is rather robust to the initializations scheme, and more importantly, is able to train to good generalization regardless of which unit type is used. **LRA-fdbk is able to train deep networks of both logistic sigmoid and hyperbolic tangent networks, even at initializations close to or at zero.** In Figure 5.4 (previous page), we focus on the setting with $\sigma = 0.025$ for the Gaussian distribution used to initialize the networks. Observe that LRA-fdbk is able to reach good classification error ($< 10\%$ on MNIST and $< 20\%$ on Fashion MNIST) within 10 epochs consistently. DFA is competitive with LRA-fdbk.
when using hyperbolic tangent units but trains the same network composed of sigmoid units poorly.

According to our results, for DTP, the inverse mapping used to reconstruct the underlying layerwise targets does not work all that well when weights are initialized from a purely random Gaussian, especially with a low standard deviation. As observed in Table 5.1, DTP struggles to train these deep networks, even when given the advantage and allowed to use an adaptive learning rate unlike the other algorithms. DTP’s struggle might be the result of losing too much of its layerwise target information too soon—the inverse mapping (or decoder of the layerwise auto-associative structure) requires a strong signal at each iteration to learn and if the signal is too weak or lost, the target produced for reconstruction becomes rather useless. However, DTP does far better than FA across the scenarios, although it lacks the ability to train from zero initialization. Our preliminary experimentation with DTP also uncovered that, in addition to requiring a more complex outer optimization procedure (like RMSprop) to even work, the learning procedure is highly dependent on its conditions and internal hyper-parameter settings (and there exist few heuristics on good starting points). To make DTP work well, significant tweaking of its settings would be required on a per-dataset basis in order to produce better targets for the inverse mapping. Since the error of DTP (or target propagation algorithms in general) is represented as the change in activities of the same set of neurons, if any neural activity is unstable, the overall algorithm will fail to train the underlying model effectively.

With respect to DFA, the layers are no longer related through a sequential backward pathway. This means that the lower-level neurons are disconnected from the forward propagation pathway when errors are calculated using the feedback projection weights. In contrast, we find that in FA the error signal is still created by a backward pass as in BP, but this time with the final per-neuron derivatives approximated by the feedback weights that replace the transpose of the forward weights in the BP global feedback pathway. Hence FA fails in cases where we have instability or few gradients are acting on participating neurons. DFA actually works fairly well compared to the other baselines if the activation function is the hyperbolic tangent, and does outperform FA when the logistic sigmoid is utilized. Furthermore, unlike DTP and BP, DFA and FA can train networks from zero, although LRA does a much better job.

Another interesting and important property of LRA-fdbk is that, unlike all of the other approaches investigated, it can automatically decide its depth of credit assignment. Specifically, in the case of the MLP, LRA-fdbk can decide how many subgraphs below
it needs to update. This is possible through the simple per-sample criterion/gate we proposed earlier in Algorithm 7. In Figure 5.5, we observe this dynamic behavior when recording the mean depth (or average number of subgraphs updated over the full training set in one pass), seeing that the network starts, within the first several epochs, by updating all of the subgraphs of the network. However, as learning continues, usually past five epochs, we see the number of subgraphs updated decrease, and, in the case of the tanh networks, approach one or zero. While not presented in the depth plots, we also recorded the average number of updates made at each layer. These logs revealed that, around the same time the average depth approaches one, even updates at the very top subgraph become less frequent. This aligns with our intuition that once latent representations of a lower layer are “good enough”, LRA can quit expending computation on that layer, on a per-sample basis. No other algorithm, including BP, has the property to adapt its computation when calculating parameter displacements (which is also why BP is depicted as horizontal line in Figure 5.5–its cost is the same over each epoch).

5.5.3 Training from Null Initialization

Next, we further experiment with LRA’s ability to train networks from null, or pure zero, initialization. While BP and DTP will fail in this setting, DFA and FA will not. However, in order for DFA and FA to operate in this special setting certain restrictions are applied,
e.g. the activation function must be specific like the hyperbolic tangent [25] and non-zero initialization must be used for certain activations including the linear rectifier. LRA does not impose these restrictions, and furthermore, can easily handle non-differentiable operations, e.g., the signum function, as we shall observe shortly.

To demonstrate LRA’s ability to handle a wide variety of functions, we train models of 3 layers of 800 hidden units with updates estimated over mini-batches of 20 samples. Parameters were updated using the Adam [161] adaptive learning rate. The activation functions we experimented with included the softsign [221], the softplus [222], the linear rectifier [222], local-winner-take-all (LWTA) lateral competition [223], and the signum (or sign).

For the networks that used LWTA and signum units, the architecture for any particular subgraph of the MLP, except the bottommost and topmost subgraphs, is defined as follows:

\[
z_{\ell-1} = f_{\ell-1}(h_{\ell-1}), \quad h_\ell = W_\ell f_d^\ell(z_{\ell-1}) + b_\ell, \quad \text{and} \quad z_\ell = f_\ell(h_\ell)
\]

(5.7)

where \( f_d^\ell \) is a discretization function, e.g., signum or Heaviside step, or lateral competition activation, e.g., LWTA. Note that the signum is specifically defined as: 

\[
\text{sign}(v) = \begin{cases} 1 & \text{if } v \geq 0 \\
0 & \text{if } v < 0 \end{cases}
\]

When using the subgraph above for either signum or LWTA units, we set the other post-activations \( f_\ell \) to be the hyperbolic tangent except in the case of LWTA units, where we experimented with both the hyperbolic tangent (LWTA-1) and the hard hyperbolic tangent [168] (LWTA-2). For the LWTA units, we follow and design lateral competition blocks following the same convention discussed in [223]. Specifically, any processing element \( z_j \), in block \( i \) of \( n \) units (in layer \( \ell \)), is defined by the hard interaction function below:

\[
z_{j}^{i} = \begin{cases} z_{j}^{i} & \text{if } z_{j}^{i} \geq z_{k}^{i} \forall k = 1..n \\
0 & \text{otherwise.} \end{cases}
\]

(5.8)

In this experiment, the LWTA blocks employed grouped four neurons together, with no overlap, yielding 200 blocks of laterally competitive neurons. Index precedence is used to break any ties. This form of structured sparsity through competition has also been observed in a biological neural circuits when modeling brain processes. Specifically, areas of the brain exhibit are structured with neurons providing excitatory feedback to nearby neurons, as evidenced in studies of cortical and sub-cortical regions of the brain.
[224–226]. The concept of inter-neuronal competition also plays a key role in Bayesian theories of the brain, specifically those that build on (sparse) predictive coding [61, 62], which argue that lateral competition allow the underlying system to uncover the few causal factors, out of the many possible, that explain a given input stimulus at any time step. From a practical machine learning perspective, sparsity is highly desirable for a wide variety of reasons [222], and has recently been shown to be useful in even training directed neural generative models of sequential data [52].

Note that with \textit{LRA-fdbk}, one can easily integrate what we will call \textit{soft} lateral competition instead of the hard vector quantization in LWTA. For example, one can use the softmax instead of the argmax operator for each competition block. This would mean the lateral competition block would be defined as:

\[
z^j_i = \frac{\exp(z^i_j)}{\sum_{k=1}^n \exp(z^i_k)}, \tag{5.9}
\]

This soft block could then be treated as the probabilities of a mini categorical distribution and sampled accordingly, if a hard-decision is still required. Since \textit{LRA-fdbk} does not require the derivative of the lateral competition block function, one does not need to compute the expensive Jacobian associated with the pure softmax function.\footnote{When the softmax is used at the output layer in purely differentiable systems, one takes advantage of the analytically simplified derivative of the output with respect to pre-activities when using the categorical log likelihood loss function. When using the softmax \textit{inside} the network, this trick is no longer available to the practitioner.} For this experiment, we refer to the proposed soft LWTA as SLWTA.

In Table 5.2, we see that \textit{LRA-fdbk} is able to successfully train different activation functions. This includes the deep models that contain discrete-valued units. It is interesting to note, that while all networks trained with \textit{LRA-fdbk} generalize reasonably well, a network that performs best on one dataset does not necessarily perform the best on the other. For example, the best-performing network on MNIST was the network that used the signum function while the sparse rectifier network performed better on Fashion MNIST. To dig deeper into the discriminative ability of each layer of a network learned with \textit{LRA-fdbk} versus other algorithms such as BP and DFA, we extracted the hidden representations of the model learned under each when applied to the test-set of Fashion MNIST. Each multidimensional representation vector was then projected to 2D for visualization using t-SNE, Barnes-Hutt approximation [227]. The results of this visualization can be found Figure 5.6. We see, first and foremost, that the model learned
Table 5.2. Generalization error (%) of various networks trained with LRA-fdbk, from null initialization. We also report, next to each “best error”, the end of which epoch the model reached this level of generalization. Note that, for the output of the internal subgraphs, LWTA-1 means tanh was used and LWTA-2 means hard-tanh was used.

<table>
<thead>
<tr>
<th></th>
<th>MNIST</th>
<th>Fashion MNIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi(\cdot)$</td>
<td>Ep. 1</td>
<td>Ep. 50</td>
</tr>
<tr>
<td>LRA-Softsign</td>
<td>5.94</td>
<td>2.13</td>
</tr>
<tr>
<td>LRA-Relu6</td>
<td>8.6</td>
<td>3.14</td>
</tr>
<tr>
<td>LRA-Softplus</td>
<td>14.94</td>
<td>4.1</td>
</tr>
<tr>
<td>LRA-LWTA-1</td>
<td>7.02</td>
<td>2.96</td>
</tr>
<tr>
<td>LRA-LWTA-2</td>
<td>6.96</td>
<td>2.57</td>
</tr>
<tr>
<td>LRA-SLWTA-1</td>
<td>9.15</td>
<td>2.1</td>
</tr>
<tr>
<td>LRA-SLWTA-2</td>
<td>8.99</td>
<td>2.21</td>
</tr>
<tr>
<td>LRA-Signum</td>
<td>6.88</td>
<td>2.09</td>
</tr>
</tbody>
</table>

with \(LRA-fdbk\) has acquired distributed representations that contain information useful for properly separating the data-points by class.

### 5.5.4 Stochastic Networks

We next investigate if LRA, or specifically \(LRA-fdbk\), can successfully handle networks with stochastic units, such as Bernoulli-distributed variables, an important class of non-differentiable activation functions. We compare LRA with DTP and back-propagation of errors. For back-propagation of errors, since a discrete sampling function is non-differentiable, we explore a variety of approximations including the straight-through estimator, variations of the slope-annealing trick, and reinforcement learning approaches to training discrete-valued variables, i.e., REINFORCE.

The stochastic models trained for this experiment each contain two layers of 200 hidden units (which is the setting used in [34]) and parameters are trained over 500 epochs. The specific architecture is as follows:

\[
\begin{align*}
  h^1 &= W_1x, \quad z^1 = \text{sigm}(h^1) \\
  h^2 &= W_2S(z^1), \quad z^2 = \text{sigm}(h^2) \\
  h^3 &= W_3S(z^2), \quad z^3 = \text{softmax}(h^3)
\end{align*}
\]

where \(\text{sigm}(v)\) is the logistic sigmoid, which parametrizes the probability \(p\) of the layer of Bernoulli variables, and \(S(p) \sim B(1, p)\) is a stochastic operator that takes in a probability
Figure 5.6. t-SNE plots of the various latent spaces acquired by the sparse rectifier network trained with backprop (top) or LRA-fdbk (bottom) on Fashion MNIST.

$p$ and returns either a zero or a one, e.g., a binary variable.

In Table 5.3, observe that LRA is able to effectively train networks composed of stochastic binary units, competitive with DTP and outperforming the other estimators used in back-propagation. This is encouraging, since it is well-known that actual neurons communicate via spikes, and modeling this discrete signal as a Bernoulli variable brings us one step closer to incorporating real neuro-biological ideas into artificial neural
Table 5.3. Generalization error (%) of various stochastic binary networks. Errors are calculated using model posterior probabilities averaged over $M = 100$ samples.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>MNIST</th>
<th>Fashion MNIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST-BP, Pass-through STE</td>
<td>85.409</td>
<td>59.040</td>
</tr>
<tr>
<td>BP, Sigmoid-Adjusted STE</td>
<td>96.039</td>
<td>85.17</td>
</tr>
<tr>
<td>BP, Slope-Annealing</td>
<td>97.180</td>
<td>84.790</td>
</tr>
<tr>
<td>BP, REINFORCE</td>
<td>88.811</td>
<td>75.160</td>
</tr>
<tr>
<td>BP, REINFORCE (Variance-Adj.)</td>
<td>94.110</td>
<td>81.360</td>
</tr>
<tr>
<td>DTP</td>
<td>98.460</td>
<td>87.320</td>
</tr>
<tr>
<td>LRA-fdbk</td>
<td>98.280</td>
<td>89.210</td>
</tr>
</tbody>
</table>

architectures. We believe that using spike-like variables in a neural system offers a form of regularization much akin to that of drop-out [143]. The key feature of using spike variables is that at test-time, we do not “shut off” this mechanism as is done in drop-out (where we calculate an expectation over all possible sub-models by multiplying the activities by the drop-out probability used in training). One could easily use a stochastic model such as the one we train to also characterize its uncertainty at the posterior by simply estimating its variance in addition to the mean as we have done in these particular experiments.

5.6 Reformulating LRA using Error-Units

While the algorithm for LRA-fdbk was presented earlier in a somewhat general fashion, allowing one to adapt it to nearly any kind of neural architecture, one lingering issue is that the error feedback weights are not learned. While it appears that is not an issue, since we have shown experimentally that LRA-fdbk can train a wide variety of networks, including those with discrete and stochastic activation functions as well as incorporate neuro-biological mechanisms such as lateral competition, it would seem unlikely that in the brain some of the synaptic connections between neurons are never adjusted. However, in the formulation of LRA we have so far, it is not clear what the learning signals should be for the error weights.

However, we developed one way to evolve the error weights after reformulating LRA-fdbk into an explicit error-unit form. This means that instead of relying exclusively on an automatic differentiation routine to provide first-order derivatives, we explicitly encoded the partial derivatives needed to create targets as error-units, much as is done in predictive coding neuro-computational models [62]. In doing so, we found a surprisingly
effective way to adjust the error weights, the method of which we will describe below in our error-driven variation of the LRA learning algorithm (LRA-E).

Furthermore, we will also propose an improved variation of the DTP learning procedure, mapping it to the notation of LRA since DTP can be considered part of the family of Discrepancy Reduction algorithms. We will compare this new version of DTP to LRA-E as well as a slew of other modern biologically-motivated algorithms on the benchmarks used in the experiments earlier in this chapter.

5.6.1 Error-Driven LRA

To describe this formulation of LRA, we will again use the example of an MLP, but this time we will describe the specific case of a 3-layer feedforward network. Note that our algorithm, like all the others, easily generalizes to models with an arbitrary number of layers.

Algorithm 9 LRA-E: Target computation.

Input: sample \((y, x)\), model parameters \(\Theta = \{W_1, W_2, W_3, E_2, E_3\}\)

// Procedure for computing error units & targets

function COMPUTETARGETS((y, x), \(\Theta\))

// Run feedforward weights to get activities
\(h^1 = W_1z^0, z^1 = \phi_1(h^1)\)
\(h^2 = W_2z^1, z^2 = \phi_2(h^2)\)
\(h^3 = W_3z^2, z^3 = \phi_3(h^3)\)
\(y_3^z \leftarrow y\)
\(e_3 = -\frac{\nabla_3^3}{z^3}, y_2^z \leftarrow \phi_2(h^2 - \beta(E_3e_3))\)
\(e_2 = -2(y_2^z - z^2)\)
\(y_1^z \leftarrow \phi_1(h^1 - \beta(E_2e_2))\)
\(e_1 = -2(y_1^z - z^1)\)
\(\Lambda = (z^3, z^2, z^1, h^3, h^2, h^1, e^3, e^2, e^1)\)

Return \(\Lambda\)
Algorithm 10 LRA-E: Update computation.

Input: sample \((y, x)\), calculations \(\Lambda\)

// Procedure for computing weight updates

function CALCUPDATES\(((y, x), \Theta, \Lambda)\)

\[
\Delta W_3 = (e_3 \otimes \phi_3'(h_3))(z_2)^T \\
\Delta W_2 = (e_2 \otimes \phi_2'(h_2))(z_1)^T \\
\Delta W_1 = (e_1 \otimes \phi_1'(h_1))(x)^T \\
\Delta E_3 = -\gamma(\Delta W_3)^T \\
\Delta E_2 = -\gamma(\Delta W_2)^T \\
\]

Return \((\Delta W_3, \Delta W_2, \Delta W_1, \Delta E_3, \Delta E_2)\)

function CALCUPDATES\(((y, x), \Theta, \Lambda)\)

\[
\Delta W_3 = e_3(z_2)^T \\
\Delta W_2 = e_2(z_1)^T \\
\Delta W_1 = e_1(x)^T \\
\Delta E_3 = \gamma(\Delta W_3)^T \\
\Delta E_2 = \gamma(\Delta W_2)^T \\
\]

Return \((\Delta W_3, \Delta W_2, \Delta W_1, \Delta E_3, \Delta E_2)\)

The pre-activities of the MLP at layer \(\ell\) are denoted as \(h^{\ell}\) while the post-activities, or the values output by the non-linearity \(\phi_\ell(\cdot)\), are denoted as \(z^{L}\). The target variable used to correct the output units \((z^{\ell})\) is denoted as \(y^L\) \((y^L = y\) or \(y^L = x\) if we are learning an auto-associative function). Connecting one layer of neurons \(z^{\ell-1}\), with pre-activities \(h^{\ell-1}\), to another layer \(z^{\ell}\), with pre-activities \(h^{\ell}\), is a set of synaptic weights \(W^{\ell}\). The forward propagation equations for computing pre-activation and post-activation values for a layer \(\ell\) would then simply be:

\[
h^{\ell} = W^{\ell}z^{\ell-1}, \quad z^{\ell} = \phi_\ell(h^{\ell}) \tag{5.13}
\]

Before computing targets or updates, we first must define the set of local losses, one per layer of neurons except for the input neurons, that constitute the measure of total discrepancy inside the MLP, \(\{L^1(y^1_1, z^1), L^2(y^2_2, z^2), L^3(y^3_3, z^3)\}\). With losses defined, we can then explicitly formulate the error units for each layer as well, since any given layer’s error units correspond to the first derivative of that layer’s loss with respect to that layer’s post-activation values. For the MLP’s output layer, we could assume a categorical
distribution, which is appropriate for 1-of-\( k \) classification tasks, and use the following negative log likelihood loss:

\[
\mathcal{L}_\ell(y^\ell, z^\ell) = -\frac{1}{2} \sum_{i=1}^{|z|} y^\ell_i \log z^\ell_i, \quad e^\ell = e^\ell(y^\ell, z^\ell) = \frac{-y^\ell_i}{z^\ell_i}, \quad (5.14)
\]

where the loss is computed over all dimensions \(|z|\) of the vector \(z\) (where a dimension is indexed/accessed by integer \(i\)). Note that for this loss function, we assume that \(z\) is a vector of probabilities computed by using the softmax function as the output nonlinearity, 

\[
z^3 = \frac{\exp(h^3)}{\sum_i \exp(h^3_i)}. \quad \text{For the hidden layers, we can choose between a wider variety of loss functions, and in this chapter, we experimented with assuming either a Gaussian or Cauchy distribution over the hidden units. For the Gaussian distribution (or L2 norm), we have the following loss and error unit pair:

\[
\mathcal{L}_\ell(z, y) = \frac{1}{(2\sigma^2)} \sum_{i=1}^{|z|} (y_i - z_i)^2, \quad e^\ell = e^\ell(y^\ell, z^\ell) = -(2\sigma^2)(y^\ell_i - z^\ell_i) \quad (5.15)
\]

where \(\sigma^2\) is a scalar representing a fixed variance (setting this to \(\sigma^2 = \frac{1}{2}\) gets rid of the multiplicative factor entirely). For the Cauchy distribution (or log-penalty), we obtain:

\[
\mathcal{L}_\ell(z, y) = \sum_{i=1}^{|z|} \log(1 + (y_i - z_i)^2), \quad e^\ell = e^\ell(y^\ell, z^\ell) = \frac{-2(y^\ell_i - z^\ell_i)}{(1 + (y^\ell_i - z^\ell_i)^2)}. \quad (5.16)
\]

For the activation function used in calculating the hidden post-activities, we use the hyperbolic tangent, or \(\phi_\ell(v) = \frac{\exp(2v) - 1}{\exp(2v) + 1}\). Using the Cauchy distribution proved particularly useful in our experiments, most likely because it encourages sparser representations, which aligns nicely with the biological considerations of sparse coding [61] and predictive sparse decomposition [68] as well as lateral competition [62] that naturally occurs in groups of neural processing elements. These are relatively simple local losses that one can use to measure the agreement between the representation and target, and future work should entail developing better metrics.

With local losses specified and error units implemented, all that remains is to define how targets are computed and what the parameter updates will be. At any given layer \(z^\ell\), starting at the output units (in our example, \(z^3\)), we calculate the target for the layer below \(z^{\ell-1}\) by multiplying the error unit values at \(\ell\) by a set of synaptic error weights \(E^\ell\).
This projected displacement, weighted by the modulation factor $\beta^9$, is then subtracted from the initially found pre-activation of the layer below $h^{\ell-1}$. This updated pre-activity is then run through the appropriate nonlinearity to calculate the final target $y^{\ell-1}_z$. This computation amounts to:

$$e^\ell = -2(y^\ell_z - z^\ell), \quad \Delta h^{\ell-1} = E_\ell e^\ell, \quad y^{\ell-1}_z \leftarrow \phi_{\ell-1}(h^{\ell-1} - \beta(\Delta h^{\ell-1})). \quad (5.17)$$

Note this particular way of computing targets does not require the derivative of the activation of layer $\ell$ as was required earlier in this chapter in the first presentation of LRA. This, if combined with the learning rule shown later in Equation 5.19, would mean we no longer require the derivatives of any single function within the network, a highly desirable situation for building more biologically plausible neural systems.

Once the targets for each layer have been found, we can then use the local loss $L^\ell(y^\ell_z, z^\ell)$ to compute updates to the weights $W_\ell$ and its corresponding error weights $E_\ell^{10}$. The update calculation for any given parameter at layer $\ell$ would be:

$$\Delta W_\ell = (e^\ell \odot \phi'(h_\ell))(z_{\ell-1})^T, \text{ and, } \Delta E_\ell = -\gamma(\Delta W_\ell)^T, \text{ or, } \quad (5.18)$$

$$\Delta W_\ell = e^\ell (z_{\ell-1})^T, \text{ and, } \Delta E_\ell = \gamma(\Delta W_\ell)^T \quad (5.19)$$

where $\odot$ indicates the Hadamard product and $\gamma$ is a decay factor (a value that we found should be set to less than 1.0) meant to ensure that the error weights change more slowly than the forward weights. Note that the second variation of the update rule does not require $\phi'(\cdot)$, which makes it particularly attractive in that it does not require the first derivative of the activation function thus permitting the use of discrete and stochastic operations. The update for each set of error weights is simply proportional to the negative transpose of the update computed for its matching forward weights, which is a computationally fast and cheap rule we propose inspired by [64].

In Algorithm 10 and 9, we show how the equations above, which constitute the LRA, are applied to a 3-layer MLP, assuming Gaussian local loss functions and their respective error units. This means $L = 3$ and the model is defined by $\Theta = \{W_1, W_2, W_3, E^2, E^3\}$ (biases $e^\ell$ are omitted for clarity). We will refer to this algorithm as LRA-E. Both

---

9In the experiments of this chapter, a value of $\beta = 0.1$, found with only minor tuning in preliminary experiments on subset of training and validation data proved to be effective in general.

10Except for very bottom set of forward weights, $W_1$, of which there are no error corresponding error weights.
Figure 5.7. In (a), we compare the angle between the updates calculated by LRA-E and backprop. In (b), we show how the total discrepancy, as measured in an LRA-trained MLP, evolves during training, alongside the output loss.

phases of the algorithm, i.e., target calculation and parameter updating, from a graphical perspective, are depicted in Figure 5.8.

With a local loss assigned to each hidden layer, we can measure our neural model’s total internal discrepancy for a given data point, $D(y, x)$, as a simple linear combination of all of the internal local losses. Figure 5.7(b) shows the 3-layer MLP example developed in this section (256 units each), trained by stochastic gradient descent (SGD) and mini-batches of 50 image samples, over the first 20 epochs of learning using a Categorical output loss and two Gaussian local losses. While the output loss continues to decrease, the total discrepancy does not always appear to do so, especially in the earlier part of learning. However, since each layer will try to minimize the mismatch between itself and a target value, any fluxes, or local loss values that actually increases instead of decreases which might raise the total discrepancy, will be taken care of later as the model starts generating better targets. The hope is that so long as the angle of the updates computed from LRA are within 90 degrees of the updates obtained by back-propagation of errors, LRA will move parameters towards the same general direction as back-propagation, which greedily points to the direction of steepest descent, and still find reasonably good local optima. In Figure 5.7(a), this does indeed appear to be the case–for the 3-layer MLP trained for illustrative purposes in this section, we compare the updates calculated by LRA-E with those given by back-propagation after each mini-batch. The angle, fortunately, while certainly non-zero, never deviates too far from the direction pointed by back-propagation (at most 11 degrees) and remains relatively stable throughout the learning process.
Figure 5.8. The two main phases of Local Representation Alignment for a given $(y, x)$.

### 5.6.2 Improving Difference Target Propagation

As mentioned earlier, Difference Target Propagation (DTP) (and also, less directly, recirculation [31, 228]), like LRA-E, also falls under the same family of algorithms concerned with minimizing internal discrepancy. However, DTP takes a very different approach to computing alignment targets than LRA-E does—instead of transmitting messages through error units and error feedback weights as in LRA [52], DTP employs feedback weights to learn the inverse of the mapping created by the feedforward weights. However, [229] showed that DTP struggles to assign good local targets as the network gets deeper and thus more highly nonlinear, facing an initially positive but brief phase in learning where generalization error decreases (within the first epochs) before ultimately collapsing (unless very specific initializations are used). One potential cause of this failure could be the lack of a strong enough mechanism to globally coordinate the local learning problems created by the encoder-decoder pairs that define the system. In particular, we hypothesize this problem might be coming from the noise injection scheme, which is local and fixed, offering no adaptation to each specific layer and making some of the layerwise optimization problems more difficult than necessary. Here, we will aim to remove this potential cause through an adaptive layerwise corruption scheme.

Assuming we have a target calculated from above $y_{\ell}^z$, we consider the forward weights
$W_\ell$ connecting the layer $z^{\ell-1}$ to layer $z^\ell$ and the decoding weights $E_\ell$ that define the inverse mapping between the two. The first forward propagation step is the same as in Equation 5.13. In contrast to LRA-E’s error-driven way of computing targets, we consider each pair of neuronal layers, $(z^\ell, z^{\ell-1})$, as forming a particular type of encoding/decoding cycle that will be used in computing layerwise targets. To calculate the target $y_{\ell-1}$, we update the original post-activation $z^{\ell-1}$ using the linear combination of two applications of the decoding weights as follows:

$$y_{\ell-1} = z^{\ell-1} - \left( \phi_{\ell-1}(V_\ell z^\ell) + \phi_{\ell-1}(V_\ell y_{\ell-1}) \right)$$ (5.20)

where we see that we decode two times, one from the original post-activation calculated from the feedforward pass of the MLP and another from the target value generated from the encoding/decoding process from the layer pair above, e.g. $(z^{\ell+1}, z^\ell)$. This will serve as the target in training the forward weights for the layer below $W_{\ell-1}$. We multiply top layer target with the fixed constant 0.01 as compared to learning rate throughout the experiments for our improved DTP. To train the inverse-mapping weights $V_\ell$, as required by the original proposed version of DTP, zero-mean Gaussian noise $\epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2)$ with fixed standard deviation $\sigma$, is injected into $z^{\ell-1}$ following by re-running the encoder and the decoder on this new corrupted activation vector. Formally, this is defined as:

$$\tilde{y}_{\ell-1} = y_{\ell-1} + \epsilon, \quad \tilde{z}^{\ell-1} = \phi_{\ell-1}(V_\ell \tilde{y}_{\ell-1})$$ (5.21)

This process we will refer to as DTP. In our proposed, improved variation of DTP, or DTP-$\sigma$, we will take an adaptive approach to the noise injection process $\epsilon$. To develop our “adaptive” noise scheme, we have taken some insights from studies of biological neuron systems, which show there are different levels of variability at different neuronal layers [230–233]. It has been argued that this noise variability enhances neurons’ overall ability to detect and transmit signals across a system [233–235] and, furthermore, that the presence of noise yields more robust representation [233, 236, 237]. There is also biological evidence demonstrating an increase in the noise level across successive groups of neurons which is thought to help in local neural computation [233, 238, 239].

The standard deviation $\sigma$ of the noise process should be a function of the noise across layers, and an interesting way in which we implemented this was to make $\sigma_\ell$ (the standard deviation of the noise injection at layer $\ell$) a function of local loss measurements. At the top layer, we can set the standard deviation to be $\sigma_L = \alpha (0.01$ worked well in our
experiments), or, rather, equal to the step-size used to compute the top-most target (when differentiating the output loss with respect to \( z^L \)). The standard deviation for the layers below would be a function of where it is within the network. This means that:

\[
\sigma_\ell = \sigma_{\ell+1} - \mathcal{L}_\ell(y^{\ell-1}_z, z^{\ell-1})
\]

noting that the local loss chosen for DTP is a Gaussian loss (but with the input arguments flipped—the target value is now the corrupted initial encoding and the prediction is the clean, original encoding).

The updates to the weights are calculated by differentiating each local loss with respect to the appropriate encoder weights, or \( \Delta W_{\ell-1} = \frac{\partial \mathcal{L}(x^{\ell-1}, y^{\ell-1}_z)}{\partial W_{\ell-1}} \), or with respect to the decoder synaptic weights \( \Delta V_\ell = \frac{\partial \mathcal{L}(z^\ell, y^\ell_z)}{\partial V_\ell} \). Note that the order of the input arguments to each loss function for these two partial derivatives is important, in keeping aligned with the original paper in which DTP was proposed [34], in order to obtain the correct sign to multiply the gradients by.

As we will see in our experimental results, DTP-\( \sigma \) is a much more stable learning algorithm, especially when training deeper and wider networks. DTP-\( \sigma \) now benefits from a stronger form of global coordination among its internal encoding/decoding sub-problems through the pair-wise comparison of local loss values that drive the hidden layer corruption.

### 5.6.3 A Comment on the Efficiency of LRA-E and DTP

It should be noted that LRA-E is in general faster than DTP in calculating targets. Specifically, if we just focus on matrix multiplications within an MLP, which would take up the bulk of the computation underlying both processes, LRA-E only requires \( 2(L - 1) \) matrix multiplications while DTP (and our proposed DTP-\( \sigma \)) requires \( 4(L - 3) + L \) multiplications. In particular, the bulk of DTP’s primary expense comes from its approach to computing the targets for the hidden layers since it requires 2 applications of the encoder parameters (1 of these comes from the initial feedforward pass through the network) and 3 applications of the decoder parameters in order to properly generate targets to train the forward weights and the inverse-mapping weights.
5.7 Experiments: Evaluation of LRA-E and DTP-\(\sigma\)

In this section, we present experimental results of training MLPs using a variety of learning algorithms including our reformulated LRA and the improved DTP. We make use of, as in the last set of experiments for this chapter, the MNIST and Fashion MNIST datasets to benchmark these algorithms.

For both datasets and all models, over 100 epochs, we calculate updates over mini-batches of 50 samples. Unless internal to the algorithm itself, we do not regularize parameters any further, such as through drop-out [143] or penalties placed on the weights. All feedforward architectures for all experiments were of either of 3, 5, or 8 hidden layers of 256 processing elements. The post-activation function used was simply the hyperbolic tangent and the top layer was chosen to be a maximum-entropy classifier (employing the softmax function). The output layer objective for all algorithms was to minimize the categorical negative log likelihood.

Parameters were initialized using a scheme that gave best performance on the validation split of each dataset on a per-algorithm case. Though we wanted to use very simple initialization schemes for all algorithms, in preliminary experiments, we found that the feedback alignment algorithms as well as difference target propagation (including our improved version of it) worked best when using a uniform fan-in-fan-out scheme [221]. [229] confirms this result, originally showing how these algorithms often are unstable or fail to perform well using a simple initialization based on the uniform or Gaussian distributions. For Local Representation Alignment, however, we simply initialized the parameters using a zero-mean Gaussian distribution, with variance of 0.05.

The choice of update rule was also somewhat dependent on the learning algorithm employed. Again, as shown in [229], it is difficult to get good, stable performance from algorithms, such as the original DTP, when using simple SGD. As done in [214], we used the RMSprop adaptive learning rate with a global step size of \(\lambda = 0.001\). For Backprop, RFA, DFA, and LRA-E, SGD was used with a step-size of \(\lambda = 0.01\).

5.7.1 Classification Performance

In this experiment, we compare all the algorithms discussed earlier. These include back-propagation of errors (Backprop), Random Feedback Alignment (RFA) [26], Direct Feedback Alignment (DFA) [25], Equilibrium Propagation [43] (Equil-Prop) and the
Table 5.4. MNIST supervised classification results.

<table>
<thead>
<tr>
<th>Model</th>
<th>3 Layers</th>
<th></th>
<th>5 Layers</th>
<th></th>
<th>8 Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Test</td>
<td>Train</td>
<td>Test</td>
<td>Train</td>
<td>Test</td>
</tr>
<tr>
<td>Backprop</td>
<td>1.78</td>
<td>3.02</td>
<td>2.4</td>
<td>2.98</td>
<td>2.91</td>
<td>3.02</td>
</tr>
<tr>
<td>Equil-Prop</td>
<td>3.82</td>
<td>4.99</td>
<td>7.59</td>
<td>9.21</td>
<td>89.96</td>
<td>90.91</td>
</tr>
<tr>
<td>RFA</td>
<td>3.01</td>
<td>3.13</td>
<td>2.99</td>
<td>3.4</td>
<td>3.59</td>
<td>3.76</td>
</tr>
<tr>
<td>DFA</td>
<td>4.07</td>
<td>4.17</td>
<td>3.71</td>
<td>3.88</td>
<td>3.81</td>
<td>3.85</td>
</tr>
<tr>
<td>DTP</td>
<td>0.74</td>
<td>2.8</td>
<td>4.408</td>
<td>4.94</td>
<td>10.89</td>
<td>10.1</td>
</tr>
<tr>
<td>DTP-σ (ours)</td>
<td>0.00</td>
<td>2.38</td>
<td>0.00</td>
<td>2.57</td>
<td>0.00</td>
<td>2.56</td>
</tr>
<tr>
<td>LRA-E (ours)</td>
<td>0.86</td>
<td>2.20</td>
<td>0.16</td>
<td>1.97</td>
<td>0.08</td>
<td>2.55</td>
</tr>
</tbody>
</table>

Table 5.5. Fashion MNIST supervised classification results.

<table>
<thead>
<tr>
<th>Model</th>
<th>3 Layers</th>
<th></th>
<th>5 Layers</th>
<th></th>
<th>8 Layers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Test</td>
<td>Train</td>
<td>Test</td>
<td>Train</td>
<td>Test</td>
</tr>
<tr>
<td>Backprop</td>
<td>12.08</td>
<td>14.89</td>
<td>12.1</td>
<td>12.98</td>
<td>11.55</td>
<td>13.21</td>
</tr>
<tr>
<td>Equil-Prop</td>
<td>14.72</td>
<td>14.01</td>
<td>16.56</td>
<td>20.97</td>
<td>90.12</td>
<td>89.78</td>
</tr>
<tr>
<td>RFA</td>
<td>11.99</td>
<td>12.74</td>
<td>12.09</td>
<td>12.89</td>
<td>12.03</td>
<td>12.71</td>
</tr>
<tr>
<td>DFA</td>
<td>13.04</td>
<td>13.41</td>
<td>12.58</td>
<td>13.09</td>
<td>11.59</td>
<td>13.01</td>
</tr>
<tr>
<td>DTP</td>
<td>13.6</td>
<td>15.03</td>
<td>21.078</td>
<td>19.66</td>
<td>21.838</td>
<td>17.58</td>
</tr>
<tr>
<td>DTP-σ (ours)</td>
<td>7.5</td>
<td>13.95</td>
<td>6.34</td>
<td>12.99</td>
<td>6.5</td>
<td>13.01</td>
</tr>
<tr>
<td>LRA-E (ours)</td>
<td>11.25</td>
<td>13.51</td>
<td>9.84</td>
<td>12.31</td>
<td>9.74</td>
<td>12.69</td>
</tr>
</tbody>
</table>

Table 5.6. Effect of update rule on LRA when training a 3-layer MLP on MNIST.

<table>
<thead>
<tr>
<th>Model</th>
<th>SGD Train</th>
<th>Test</th>
<th>Adam Train</th>
<th>Test</th>
<th>RMSprop Train</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRA, MNIST</td>
<td>0.86</td>
<td>2.20</td>
<td>0.00</td>
<td>1.75</td>
<td>0.69</td>
<td>2.02</td>
</tr>
<tr>
<td>LRA, Fashion MNIST</td>
<td>11.25</td>
<td>13.51</td>
<td>5.38</td>
<td>12.42</td>
<td>12.67</td>
<td>14.90</td>
</tr>
</tbody>
</table>

original Difference Target Propagation [34] (DTP). Our algorithms include our proposed, improved version of DTP (DTP-σ) and the proposed error-driven Local Representation Alignment (LRA-E).

The results of our experiments are presented in Tables 5.4 and 5.5. Test and training scores are reported for the set of model parameters that had lowest validation error. Observe that LRA-E is the most stable and consistently well-performing algorithm compared to the other biologically-motivated backprop alternatives, closely followed by our improved variant of DTP. More importantly note algorithms like Equil-Prop and DTP appear to break down when training deeper networks, such as the 8-layer MLP. Note that while DTP was used to successfully train a 7-layer network of 240 units (using RMSprop), we followed the same settings reported for deeper 7 layers network and
in our experiments uncovered that the algorithm begins to struggle as the layers are made wider, starting even as soon as the width of 256 we experimented with in this chapter. However, this problem is rectified using our variant of DTP, leading to much more stable performance and even in cases where the algorithm completely overfits the training set (as in the case of 3 and 5 layers for MNIST). Nonetheless, LRA-E still performs the best with respect to generalization across both datasets, despite using a vastly simpler parameter update rule and naive initialization scheme. Table 5.6 shows the results of using a different update rule other than SGD for LRA-E, such as Adam [161] or RMSprop [59] for 3-layer MLP, with a global step size $0.001$. We see that LRA-E is not only compatible with other learning rate schemes but reaches yet better generalization performance when using them, e.g. Adam.

In Figure 5.9 displays a t-SNE [240] visualization of the top-most hidden layer of a learned 5-layer MLP using either DFA, Equil-Prop, $DTP\sigma$, and LRA-E on the Fashion MNIST test set. Qualitatively, we see that all learning algorithms extract representations that separate out the data points reasonably well, at least in the sense that the points are clustered based on clothing type. However, it appears that $LRA-E$ representations yield more strongly separated clusters as evidenced by the somewhat wider gaps between the clusters, especially the gaps around pink, blue, and black colored clusters.

Finally, DTP, as also mentioned in [229], appears to be quite sensitive to its initialization scheme. For both MNIST and Fashion MNIST, we trained DTP and our improved variant DTP-1 with three different settings, including random orthogonal, fan-in-fan-out, and simple zero-mean Gaussian initializations. Figure 5.10 shows the validation accuracy curves of the DTP and $DTP\sigma$ as a function of epoch for 5 and 8 layer networks (3-layer networks can be found in the appendix) with various weight
initializations such as Gaussian (G), Orthogonal(ortho), and Xavier/Glorot. As shown in the figure, DTP is highly unstable as the network gets deeper while DTP-\(\sigma\) appears to be less dependent on the weight initialization scheme. Thus, our experiments show some promising evidence of DTP-\(\sigma\)'s generalization improvement over the original DTP. Moreso, as the test performance indicates in Tables 5.4 and 5.5, DTP-\(\sigma\) can perform nearly as good as LRA-E.

### 5.8 On Unfolding

An algorithm like LRA can also be applied to recurrent neural networks. In this section, we will describe how LRA-E can be applied to recurrent neural networks, which are trained with back-propagation through time. While it will be the concern of the next chapter to train neural models on temporal/sequential data problems using a Discrepancy Reduction approach, we will describe how one might straightforwardly apply LRA to standard recurrent networks.

Recurrent neural networks (RNNs) are trained with back-propagation through time, as described in Chapter 2. Essentially, we apply standard back-propagation of errors but with one crucial exception—we must unfold the network \(T\) steps back in time\(^{11}\), ideally from the end of the sample sequence back up to its beginning. Note that in practice, we break up our sequences into sub-sequences of length \(K\), where \(K < T\), and process time-varying data in chunks to maintain computational tractability (this is called truncated backprop through time). This creates a very deep feedforward network, with each input

\(^{11}\)Note that this action is the same as unrolling a mathematical recurrence relation, since the hidden state of an RNN is recursively computed.
of each time step being fed into the unfolded graph and the underlying parameters being copied at each time step. This parameter-sharing over time circumvents the need for a number of model parameters that increases with the length of each sequence, which would be highly impractical from a computer memory storage perspective.

Focusing on the case of fully unrolling over the length of the entire sequence, we see that LRA can also be readily applied in much the same way. Assume a simple single-hidden layer Elman RNN with a linear output layer, defined as follows:

\[
\begin{align*}
    z_t &= \phi(Wx_t + Vz_{t-1}), \\
    \hat{y}_t &= Uz_t
\end{align*}
\]  

(5.23)

where the parameters to learn are simply \( \Theta = \{W, V, U\} \) (as usual, biases omitted). The task is next-step prediction, so at each time step, \( y_t = x_{t+1} \). Using LRA-E to train this RNN model requires a set of error units be placed at the output units \( e_t^{out} \), with a local loss \( L(y_t^{out}, \hat{y}_t) \), and the hidden units \( e_t^{hid} \), with a local loss \( L(y_t^{hid}, z_t) \), as well as a single, extra set of recurrent parameters \( E \), or the error weights that will connect the output units to hidden units. Note that we now index each error unit and target with the symbol \( t \), given that they operate at each time step of a sequence the RNN is currently processing.

To train an RNN over \( T \) steps in time, we simply unfold the network as in back-propagation through time but copy the error units and error weights \( K \) times as well. Finding the target for the hidden layers of this unfolded RNN would then amount to:

\[
\begin{align*}
    y_{z,T}^{hid} &= \phi(h_T - \beta(EE_t^{out})), \ldots, y_{z,t}^{hid} &= \phi(h_t - \beta(EE_t^{out})), \ldots, y_{z,1}^{hid} &= \phi(h_1 - \beta(EE_1^{out}))
\end{align*}
\]  

(5.24)

noting that \( e_t^{out} \) can be readily computed since the target for the output units is the data point at the next time step \( t + 1 \), in other words \( y_{z,t}^{out} = x_{t+1} \) where we observe that we effectively apply LRA to each time slice of the RNN model, computing each error unit that measures the difference between model output and target and then generating a target for each hidden state. Once the target, \( y_{z,t}^{hid} \), for any hidden state at time \( t \) has been computed, we can calculate the hidden error units \( e_t^{hid} \) and the appropriate local loss. The updates to the parameters of this model would then be:

\[
\begin{align*}
    \Delta U &= \sum_{t=1}^{T} e_t^{out}(z_t)^T, \quad \Delta V = \sum_{t=1}^{T} (e_t^{hid} \otimes \phi'(h_t))(z_{t-1})^T, \quad \Delta W = \sum_{t=1}^{T} (e_t^{hid} \otimes \phi'(h_t))(x_t)^T
\end{align*}
\]  

(5.25)
where \( z_0 = 0 \) (or the null state) and the output derivative is the identity.

## 5.9 Conclusions

In this chapter, we developed the Discrepancy Reduction family of learning algorithms for training neural architectures without resorting to the global feedback pathway required by back-propagation of errors. Within this family, we developed two simple variations of an algorithm called Local Representation Alignment and an improved, stable version of Difference Target Propagation. All algorithms within this family update parameters of a neural system by first finding target representations to guide the hidden layers towards, and then using these targets within local objective functions to compute updates to synaptic parameters. Furthermore, in developing LRA, we showed how back-propagation can be re-cast in the framework of target propagation and used these insights to develop and motivate the Local Representation Alignment (LRA) algorithm.

Algorithms within the family of Discrepancy Reduction decompose the credit assignment problem of artificial neural networks into smaller, local learning problems. In developing the LRA algorithm, we introduced the notion of the computation subgraph—an object that encompasses two layers of processing elements and the underlying operations and parameter variables that connect them—and how to view a deep network as a linked set of such subgraphs. Motivated by fundamental ideas in representation learning, LRA structures every subgraph within the network to have a target, not just at the output layer, and adjusts the free parameters of the subgraph to move the output closer to the target. In contrast to previous algorithms such as DTP, LRA chooses targets that are in the possible representation of the associated layers and hence the layer’s parameters can be updated more effectively, i.e. layers are not made to match a target that is impossible to achieve. The subgraph approach/decomposition also allowed us to introduce a short-circuit pathway, inspired by the idea of feedback alignment, which allows LRA to handle non-differentiable activation functions.

Unlike previously proposed algorithms, including back-propagation, target propagation, and variants of feedback alignment, LRA is far less sensitive to parameter initialization when training highly nonlinear networks. Furthermore, it is able to avoid the vanishing gradient problem when training deep networks independently of the non-linearity used and can adaptively decide the depth of the credit assignment it needs to conduct on a per-sample basis, which can lead to savings in computation per step. In
addition to being compatible with recent innovations such as batch normalization and drop-out, LRA is architecture-agnostic, so long as the global model can be decomposed into a series of linked subgraphs, where the output each subgraph can be viewed as a hidden representation to which a target can be assigned. For the case of feedforward networks, our experiments on MNIST and Fashion MNIST add strong empirical evidence to support the above claims.

LRA offers a pathway for users of neural networks to design the architecture for the problem at hand rather than for the traits and quirks of back-prop-based algorithms. This means that non-differentiable units and more biologically-plausible ones may be utilized. Furthermore, since algorithms in the Discrepancy Reduction family have connections to the theory of predictive coding, we have established a practical computational framework within which further neuro-cognitive mechanisms might be incorporated into neural systems while still achieving good generalization performance on practical tasks like classification. In the next chapter, we will take the framework of LRA and Discrepancy Reduction and develop temporal neural models to handle multi-task scenarios and sequential data problems, leading up to the Temporal Neural Coding Network, a model that can adapt to time-varying data.
Chapter 6  
Adaptive Learning by Aligning Representations

6.1 Introduction

As mentioned in the first chapter, the overall goal and contribution of this thesis is to provide a pathway towards developing continually-learning machines, which is still an immature and burgeoning field of research. The chapters preceding this one have been used to propose and develop component ideas or mechanisms that we argue are necessary for a computational framework that might govern such machines. This has entailed rethinking how we can restructure and learn the parameters that define modern connectionist architectures. These components and core ideas have included: 1) constructing dual generative-discriminative neural models that require fewer labeled data samples in order to generalize, 2) endowing temporal models with longer-term memory using parsimonious designs, and 3) learning full neural networks using local learning rules that are coordinated by a higher-level goal of minimizing total system discrepancy. These ideas have been shown, in various experiments so far, to allow far more general training of artificial neural networks with far less intervention and expertise required of the external human user.

In this chapter, we will see how these ideas come together as we consider two aspects of lifelong or never-ending learning: 1) incrementally adapting a directed generative model to time-varying data points (potentially in the form of streams) and 2) sequentially learning from a series of tasks and retaining old information in the presence of new information. These are two of the most important problems we argue that a continuously
adaptive neural agent must face.

In order to handle time-varying data points, whether they are discrete, as in language, or continuous, as in video, we will generalize the idea of discrepancy reduction from the last chapter to learning directed neural models of sequential data and propose the Temporal Neural Coding Network [241]. Under the computational framework that governs the Temporal Neural Coding Network, we will see Local Representation Alignment and the Differential State Framework come together to enable a neural system to incrementally and dynamically adapt to data without back-propagation through time, the central algorithm used to train modern recurrent networks on sequences. We will furthermore investigate how our model performs on a streaming, online data problem, where now the issue is to continuously adapt to a potentially infinite stream of examples. While the family of discrepancy reduction algorithms drew inspiration from the theory of predictive coding to inform how learning is conducted in a neural system, the Temporal Neural Coding Network is more directly and strongly connected to actual neuro-computational models of predictive coding, incrementally generating predictions first and then self-correcting its hypotheses given data. It is hoped that such a model, that generalizes nearly as reasonably well as modern connectionist architectures, can serve as a bridge and framework for blending neuro-cognitive mechanisms, currently known and yet-to-be-discovered, in Cognitive Science and Neuroscience with powerful, efficient computational elements of artificial neural networks.

We will how the learning algorithms developed in the last chapter perform in the far more difficult scenario of learning from a sequence of tasks, the very essence of lifelong learning. This means we must confront the catastrophic forgetting problem, arguably one of the largest obstacles that impede modern connectionist systems from reaching their potential when facing sequences of tasks much as humans do. In order to do so, we will propose the Memory-Consolidation algorithm, which will exploit the power of dual generative-discriminative networks developed earlier on this thesis and inspired by some experimental findings about the hippocampus, particularly the rodent hippocampus.
6.2 Continual Adaptation to Time-Varying Data

6.2.1 The Temporal Neural Coding Network

The models trained using our proposed discrepancy reduction algorithms were applied to classification problems. This has also been the case for most other proposed algorithmic alternatives to back-propagation of errors—most of the new approaches to learning artificial neural networks have focused on static problems, such as classification. However, we know that in the human brain, many active processes, including those related to vision and speech, take in sequences of input stimuli and attempt to build a dynamic mental model of the world [64].

Given this view that the brain is dynamic, constructing an implicit, abstract knowledge base that is representative of the structure of the observed environment [242, 243], we will now focus on developing a neural model that can handle stateful problems. From a machine learning perspective, this is important given the success of neural models applied to sequential problems such as language modeling [147, 244]. Typically, recurrent neural networks are used to process such data and are trained using back-propagation through time [245].

Critical to the process of back-propagation through time when training recurrent models is the action of unrolling, a mechanism that is even more clearly neurobiologically implausible than back-propagation of errors itself. In order to implement back-propagation through time, one needs to unroll the underlying computation graph over the length of a given input sequence, creating a longer global feedback path for error information to traverse. As was argued in the last chapter, this global feedback pathway, in addition to not being likely implemented in the brain, is a source of a variety of problems, many of which become far worse when training recurrent networks [7].

The brain, in contrast, relies on an incremental, adaptive process and, without any “unrolling” is able to process time-varying/sequential data points and ultimately learn to make predictions about the future. This is what motivates the development of the key model proposed in this chapter—the Temporal Neural Coding Network, which can learn from sequences without any explicit unrolling. Notably, our proposed approach is based on learning a directed generative model, which is important given the causal structure of the universe, without the need to correct for the imperfections of an approximate

\[ \text{1 Or unfold a recursively defined operation into an explicit chain of events.} \]
inference model, as in variational autoencoding models [100], for example.

### 6.2.2 Computing Targets by Continual Error-Correction

Let us begin by formally defining a TNCN, at time $t$, with three layers of neural variables $(z_0^t, z_1^t, z_2^t)$, where $z_0^t$ refers to the output sensors that directly connect the model with the environment/world. The TNCN architecture distinguishes between two sets of parameters, $\Theta_g = \{G_2, G_1, V_2, V_1, b_1, b_2, b_3\}$ (the generative/predictive parameters) and $\Theta_e = \{E_2, E_1\}$ (the error feedback parameters). We define $(h_0^t, h_1^t, h_2^t)$ to be the pre-activations of each latent variable.

To calculate the necessary statistics for one step of error correction, we first define the model’s generated prediction for any (internal) arbitrary layer to be:

$$ h_\ell^t = V_\ell z_\ell^{\ell-1} + G_{\ell+1} z_{\ell+1}^t + b_\ell, \quad z_\ell^t = \phi_g(h_\ell^t) $$  \hspace{1cm} (6.1)

which assumes that any pre-activation is a linear combination of a filtration and a top-down (expectation) bias. The output units are defined as:

$$ z_0^t = \phi_0(G_1 z_1^t + b_1) $$  \hspace{1cm} (6.2)

where the output nonlinearity $\phi_0(\cdot)$ could be different from the internal activation functions, using the identity function for real-valued data points and the softmax for discrete variables encoded in a 1-of-$k$ format.

The target computations proceed by creating local target representations by using information produced by the error units, which measure the discrepancy between the current layerwise activation and the target. Error messages from any layer $\ell$ are reported back up to layer $\ell-1$ through the appropriate set of error feedback weights. Specifically, an error unit (in this description they are assumed to be Gaussian error units), and the corresponding target, at layer $\ell$, within the model is computed as follows:

$$ e_\ell^t = -2(y_{z,t}^\ell - z_\ell^t), \quad \text{where} \quad y_{z,t}^\ell = \phi_g(h_\ell^t + \beta(E_\ell e_{\ell-1}^t)) $$  \hspace{1cm} (6.3)

where $y_{z,t}^0 = x_t$ and $\phi_g(\cdot)$ is the element-wise post-activation function applied to a layer of neurons. Note that representation target, which makes use of the information coming from the error units, also utilizes the same function, meaning that the representation target
Figure 6.1. Error-driven computation where one pair of cells in layer 1 communicate the discrepancy backwards to an earlier cell in layer 2 to correct its representation. Solid lines represent the initial, feedforward pass of information while dashed lines represent the information flow along the structural feedback connections, which feed into an error unit that computes the corrected state for $z_1^2$.

is a non-linear function of the representation guess and the error message transmitted from below. This equation is reminiscent of the single corrective step found in dynamic predictive coding models formulated as Kalman Filters [64].

The error feedback weights can be fixed or learned, using the update rule proposed for the error-driven version of LRA in the last chapter (but without the negative sign, as preliminary experiments confirmed this worked better for the TNCN model). It is important to note that the error weights are used to compute a displacement to the original pre-activation calculated for any given layer. Thus, in order to obtain the final target for learning, we must re-apply the post-activation function to ensure the units stay within what is currently representable in the model (as was done in the previous chapter). The full process of error-correction, for a single processing element within a layer of the TNCN, is graphically depicted in Figure 6.1.

The above target computation should also be considered as correcting the TNCN’s

---

2Note that we used the hyperbolic tangent, scaled optimally according to [246], as our choice of activation functions for the internal layers of the TNCN in this chapter.
internal states after it has made an initial guess about the state of the world at time $t$. Since the TNCN does not rely on any auto-regressive connections to the immediate data point $x_{t-1}$ as in standard recurrent networks, the model can be viewed as proceeding forward in time by continual error-correction. The newly corrected state at layer $\ell$ will then become the vector summary of the past when moving on to the next time step $t+1$. This is very much in line with the theory of predictive coding, which posits that the brain first generates a prediction and then it self-updates based on information its sensors collect about the environment at that time. This process of continual error-correction is depicted in Figure 6.2.

Furthermore, another important property of the TNCN is that we are permitted to use non-differentiable functions (especially those not amenable to the re-parametrization trick) in two possible cases: 1) applying the non-differentiable function right after calculating the initial representation $z^\ell$ (as was shown to work effectively in the last chapter) and 2) as a possible post-activation function if we use an update rule that does not require the derivative of that activation, which is possible if we use the second variation of LRA-E detailed in the last chapter.
6.2.2.1 A \( \Delta \)-RNN State Model

In the interest of incorporating a simple mechanism for better enabling the TNCN to extract longer-term dependencies from the data it processes, we propose integrating the \( \Delta \)-RNN into the state \( z^\ell \). Doing so will require modifying the target calculation as well, since we must now account for a slowly-moving memory state that is to be updated with new information driven by the error units (instead of projections of the actual data, which is what is done in the \( \Delta \)-RNN of Chapter 4).

Adhering to the Differential State Framework of Chapter 4, the hidden state at a layer \( \ell \) within the TNCN can be designed as a full multiplicative model that integrates the filtration and the top-down bias generated from the layer above. The full state model is calculated as follows:

\[
\begin{align*}
    d^\ell_{rec} &= V^\ell z^{\ell-1}_t, \quad d^\ell_{td} = G^{\ell+1} z^{\ell+1}_t, \\
    d^1_t &= \alpha \otimes d^\ell_{rec} \otimes d^\ell_{dat}, \quad d^2_t = \beta_1 \otimes d^\ell_{rec} + \beta_2 \otimes d^\ell_{dat}, \\
    z^\ell_{i,t} &= \phi_g(d^1_t + d^2_t + b^\ell_t) \\
    h^\ell_t &= (1 - r) \otimes z^\ell_t + r \otimes z^{\ell-1}_t, \quad z^\ell_t = \Phi_g(h^\ell_t)
\end{align*}
\]

where we define the rate-gate to be a somewhat simpler function that is not driven by anything but learnable bias \( r_b \) run through the logistic sigmoid to ensure the rate values stay in the range of \([0, 1]\), i.e., \( r = \text{sigmoid}(r_b) \). Note that \( \Phi_g(\cdot) \) is a nonlinearity considered to be part of the \( \Delta \)-RNN model’s outer mixing function that produces \( z^\ell_t \) while the \( \phi_g(\cdot) \) is part of the inner function that produces the state proposal \( z^\ell_{i,t} \). Normally it is set to the identity, but in the experiments to follow, we also found it more effective to use the hyperbolic tangent as well.

To compute the target representation, the calculation proceeds quite similarly to Equation 6.3. The only real difference in this situation is that we would reapply the outer nonlinearity \( \Phi_g(\cdot) \) to compute the final target instead of the inner nonlinearity \( \phi_g(\cdot) \).

6.2.2.2 Enforcing Sparse Representations:

In predictive coding theories of the brain, it is often assumed that sparsity is a key ingredient. This means we seek representations of data where only a small subset of the latent variables have non-zero values. If the TNCN is to disentangle concepts in its latent representations, the need for sparsity makes sense since it is reasonable to assume that
only a few out of the many possible concepts/variables explain any given datum (useful in tasks such as corrupt image denoising [247]). From a theoretical perspective on feature learning, we desire compact representations of the input in which no information is lost regarding the input [212, 248]. Dense representations, though rich, are highly entangled since small changes in the input can lead to big changes in the representation vector. Sparse representations, in contrast, are robust and mostly conserve the set of non-zero features [222].

Here, we enforce a “weak” form of lateral competition over the neuronal variables through a simple Laplacian prior distribution over the activities. During the step of error-correction, the prior is imposed as a constraint by modifying the error units as follows:

\[ e^\ell_t = \left( -2(y^\ell_{z,t} - z^\ell_t) \right) + \lambda \text{sign}(z^\ell_t). \]  

(6.8)

This means that the lateral competition induced by this prior function is constantly active at each step of the TNCN’s processing, very much inspired by the sparse coding models of [61].\(^3\) To better encourage sparsity, other prior distributions, such as the spike-and-slab distribution [249] (to avoid controlling the activation magnitudes), or architectural modifications [250] might improve generalization and will be the subject of future work. We found in initial preliminary experiments that this even this simple sparsity mechanism was indeed a necessary component in improving performance.

### 6.2.3 Learning by Aligning Representations

To compute the updates to model parameters once the latent representations have been corrected, we adapt the LRA learning procedure of the previous chapter. The cost function that measures the total discrepancy within a TNCN composed of \( L \) layers of latent variables, applied to real-valued input distributions, can be naively formulated as follows:

\[
\mathcal{L}(x_t; \Theta_g, \Theta_e) = \sum_j \frac{-(x^j_t - z^j_0)}{2\sigma^2_0[j]} + \sum_{\ell=1}^L \sum_j \frac{-(y^\ell_{z,t} - z^\ell_t)}{2\sigma^2_\ell[j]} \]  

(6.9)

\(^3\)Another way we experimented with encouraging sparsity was to only have the constraint active during training, meaning we computed updates with the sparsity penalty mixing into the gradients. However, this has the drawback that not all representations would be guaranteed to be sparse. It is quite possible that the model could produce dense representations for data points outside the training sample since only during training is sparsity encouraged.
where $\sigma_0^2 = \sigma_\ell^2 = 1/2$, which effectively cancels out the denominator of the above equation, and maps directly to the error units of the previous section. $j$ is an index that accesses a particular dimension of a vector and we saw that the scalar loss values are realized by summing across dimensions.

If we want to take an information-theoretic view, which aligns even more strongly with the idea of reducing internal discrepancy in the system, we can instead use the Kullback-Leibler Divergence [251] to measure the similarity between the guess and target for each local representation:

$$
\mathcal{L}(x_t; \Theta_g, \Theta_e) = \sum_j \frac{-(x_t[j] - z_0^t[j])^2}{2\sigma_0^2[j]} - \sum_{\ell=1}^L KL(p_{z^\ell_t}||q_{y^\ell,z,t}) 
$$

(6.10)

$$
\mathcal{L}(x_t; \Theta_g, \Theta_e) = \sum_j \frac{-(x_t - z_0^t)^2}{2\sigma_0^2} - \sum_{\ell=1}^L \sum_j \left( \log \frac{\sigma_{z_0^t[j]}^2}{\sigma_{y^\ell,z,t}^2[j]} + \frac{\sigma_{z_0^t[j]}^2 + (z_0^t[j] - y^\ell,z,t[j])^2}{2\sigma_{y^\ell,z,t}^2[j]} - \frac{1}{2} \right)
$$

(6.11)

noting that $\sigma_{z_0^t}^2$ is the variance of $z_0^t$ and $\sigma_{y^\ell,z,t}^2$ is the variance of $z^\ell_{z,t}$ (both of these are diagonal covariance matrices, and fixing these to vectors of ones further simplifies the expression to look quite similar to Equation 6.9). The leftmost term of the right-hand side of the equations for the loss is the partially grounded term in input space, while the rest of the terms are the higher-level terms/objectives. Note that, internally, the TNCN architecture will readily compute the representation targets each time a sequential element $x_t$ is presented.

If we find the first-order partial derivatives of Equation 6.9 with respect to the weights in $\Theta_g$, we get the following updates (assuming we use stochastic gradient descent as the update rule):

$$
\Delta G_\ell \propto \frac{\partial \mathcal{L}(x_t; \Theta_g, \Theta_e)}{\partial G_\ell} \approx (e^\ell \otimes \phi'_g(h^\ell))(z^\ell_{t+1})^T + \eta G_\ell 
$$

(6.12)

$$
\Delta V_\ell \propto \frac{\partial \mathcal{L}(x_t; \Theta_g, \Theta_e)}{\partial V_\ell} \approx (e^\ell \otimes \phi'_g(h^\ell))(z^\ell_{t-1})^T + \eta V_\ell 
$$

(6.13)

$$
\Delta E_\ell \propto \gamma(\Delta G_\ell) + \eta E_\ell 
$$

(6.14)

where $\eta$ is a noise process that is directly applied to the estimated parameter gradient. Such a process can be zero-mean Gaussian noise (with a scalar variance chosen through cross-validation). $\gamma$ is a decay term meant to slow down the growth of the error weights.

\footnote{Note that these could be additional parameters to be learned instead of fixing them as we have for this chapter’s experiments.}
(putting their evolution on a slower time-scale compared to the other synaptic weights).

One favorable property of this learning algorithm for learning TNCNs is the (partial) parallelism one may exploit in calculating parameter gradients, much like the goal of [39]. One simply needs to run the TNCN’s generation and target calculation procedures to get the guesses and targets, but once these statistics have been computed one can treat each layer as a computation subgraph. The parameter updates of each subgraph are independent of the other subgraphs, which means that if we design more intricate subgraphs, i.e, those with internal layers of units and nonlinear operations inside\textsuperscript{5}, we could place the whole process of parameter updating for each subgraph in a separate computing core or graphics processing unit.

### 6.2.4 Putting It All Together

The full algorithm for the TNCN (generation, representation correction, and parameter updating) is depicted in Algorithm 11. The TNCN (or rather its generation/inference mechanisms) and its learning algorithm, are intricately tied together, since the learning procedure will make use of the representation targets created by the architecture’s error-driven correction mechanism. This operates in the spirit of predictive coding which posits that the brain’s generation and inference procedures interact to formulate local learning signals. The mechanism we use for target creation is rather simple, and future work should investigate more sophisticated mechanisms (especially ones with evolving error weights). As is depicted in Algorithm 11, the representation-correction mechanism can be extended to a process where targets can be iteratively refined, in the hopes of shortening the overall learning phase.

With respect to higher-level objectives, we can see that the error units play a crucial role–they are in fact the first-order derivatives of the Gaussian log likelihood (with fixed unit variance). Learning is simple since the error units can be easily re-used to calculate parameter gradients incrementally (when combined with the competition prior) and the only activation function derivative required in this approach is that of the output distribution model (which can be easily worked out for commonly used output distributions, such as the Gaussian, Bernoulli, and Categorical distributions). Note that better error units could be derived if one chose a different tactic for measuring the distance between predicted and corrected representation layers (for example, one could

\textsuperscript{5}This would mean we would consider each actual layer that has a local loss pinned to it as a sort of \textit{meta-representation}. 

163
measure the Manhattan distance instead of the Euclidean distance, as formulated in our framework). However, the general idea is that the TNCN is engaged with ensuring its layer-wise representations are as close to those suggested by the error units—it is optimizing not only on the input space, but also in the latent space giving us some rough measure of the quality of the model’s internal representations. In some sense, this bears a loose resemblance to the bottom-up-top-down algorithm described earlier in this thesis, which proposed a non-greedy way of learning a set of layer-wise experts. Through the feedback mechanism and the top-down generation paths, the local learning rules of the TNCN gain some form of global coordination, which was lacking in the greedy

Algorithm 11 The various sub-routines that define a two-layer TNCN at time step $t$. 

1: **Input**: $x_t$ (mini-batch at time $t$), current parameters $\Theta_{t-1} = \{\Theta^g_0, \Theta^g_t\}$, previous state variables $Z = \{z^0_{t-1}, z^1_{t-1}, z^2_{t-1}\}$, and meta-parameters $\lambda, \beta, \alpha, \gamma$

2: **function** GENERATE($Z, \Theta^g_t$) 
3: \begin{align*}
(z^0_{t-1}, z^1_{t-1}, z^2_{t-1}) &\leftarrow Z, (G_1, V_1, G_2, V_2) \leftarrow \Theta^g_t \quad \triangleright \text{Extract states and parameters} \\
h^0_t &\leftarrow V_2 z^2_{t-1}, z^2_t = \phi_g(h^0_t) \\
h^1_t &\leftarrow V_1 z^1_{t-1} + G_2 z^2_t, z^1_t = \phi^g(h^1_t) \\
h^0_t &\leftarrow G_1 z^1_t, z^0_t = \phi^g(h^0_t) \\
\text{return } h^0_t, h^1_t, h^2_t, z^0_t, z^1_t, z^2_t 
\end{align*}

8: **function** CORRECT($x_t, z^0_t, (h^0_t, h^1_t, h^2_t), \Theta^g_t, \beta$) 
9: \begin{align*}
E_1, E_2 &\leftarrow \Theta^g_t \\
e^0 &\leftarrow -2 \left( y^0_{z,t} - x_t \right) \\
y^1_{z,t} &\leftarrow \phi^g(h^1_t + \beta(E_1 e^0)), e^1 = -2 \left( y^1_{z,t} - z^1_t \right) + \lambda \text{sign}(z^1_t), z^1_{t+1} = y^1_{z,t} \\
y^2_{z,t} &\leftarrow \phi^g(h^2_t + \beta(E_2 e^1)), e^2 = -2 \left( y^2_{z,t} - z^2_t \right) + \lambda \text{sign}(z^2_t), z^2_{t+1} = y^2_{z,t} \\
\text{return } e^0, e^1, e^2, z^0_{t+1}, z^1_{t+1}, z^2_{t+1} 
\end{align*}

14: **function** UPDATEMODEL($x_t, Z, \Theta^g_t, \Theta^g_t, \beta, \alpha, \lambda, \gamma$) 
15: \begin{enumerate}
16: \item \quad // 1) Run generative model (get guesses of each representation) 
17: \quad $h^0_t, h^1_t, h^2_t, z^0_t, z^1_t, z^2_t \leftarrow \text{GENERATE}(Z, \Theta^g_t)$ 
18: \item \quad // 2) Correct states via error-created targets 
19: \quad $e^0, e^1, e^2, z^0_{t+1}, z^1_{t+1}, z^2_{t+1} \leftarrow \text{CORRECT}(x_t, z^0_t, (h^0_t, h^1_t, h^2_t), \Theta^g_t, \beta, \lambda)$ 
20: \item \quad // 3) Adjust parameters via gradient descent & output corrected state variables 
21: \quad $(G_1, V_1, G_2, V_2) \leftarrow \Theta^g_t, (E_1, E_2) \leftarrow \Theta^g_t \quad \triangleright \text{Extract current parameters} 
22: \quad \Delta G_1 = \left( e^0 \otimes \phi^g(h^0_t) \right) (z^1_{t-1})^T, \Delta V_1 = \left( e^1 \otimes \phi^g(h^1_t) \right) (z^1_{t-1})^T 
23: \quad \Delta G_2 = \left( e^0 \otimes \phi^g(h^1_t) \right) (z^2_{t-1})^T, \Delta V_2 = \left( e^2 \otimes \phi^g(h^2_t) \right) (z^2_{t-1})^T 
24: \quad G_1 = G_1 - (\alpha \Delta G_1 + \eta G_1), \quad V_1 = V_1 - (\alpha \Delta V_1 + \eta V_1), \quad \tilde{E}_1 = E_1 - (\gamma \Delta G_1 + \eta E_1) 
25: \quad G_2 = G_2 - (\alpha \Delta G_2 + \eta G_2), \quad V_2 = V_2 - (\alpha \Delta V_2 + \eta V_2), \quad \tilde{E}_2 = E_2 - (\gamma \Delta G_2 + \eta E_2) 
26: \quad \Theta^g_t = \{G_1, \tilde{V}_1, G_2, \tilde{V}_2\}, \Theta^g_t = \{\tilde{E}_1, \tilde{E}_2\}, Z = \{z^0_{t+1}, z^1_{t+1}, z^2_{t+1}\} 
\end{enumerate}

return $\Theta^g_t, \Theta^g_t, Z$
approaches of the past [10, 95] when training deep belief networks and their hybrid variants.

It is important to highlight that learning and inference under this model is ideally intended to be continuous, meaning that the model simultaneously generates expectations and then corrects itself (both representations and parameters) each time a new datum from a sequence is presented. This makes the model directly suited to learning incrementally from data-streams. Furthermore, the TNCN shows how two types of recurrence/feedback are at play when modeling sequences: 1) the model is recurrent across the temporal axis since it is stateful, which means that each processing layer depends on a vector summary of the past, and 2) the model is structurally recurrent, similar to deep Boltzmann machines and Hopfield Networks [252], since error is fed back in order to automatically correct guessed representations.

6.3 Experiments: Adapting to Time-Varying Data

To test our proposed TNCN architecture and its learning algorithm, Discrepancy Reduction, we evaluate model performance on two tasks—the bouncing balls generative modeling problem and next-step character prediction on Penn Treebank (PTB).

**Bouncing Balls:** This task follows the experimental setup used in [253]. This high-dimensional dataset was created by simulating the rudimentary physics of three balls bouncing around in a box. We generate a training set of 4000 training sequences and a test set of 200 sequences (as well as yet another 200 sequences to create a development set). Furthermore, our models are given no prior knowledge of the task, e.g. convolutional weight matrices, much as was done in [254]. On this dataset, Frame t-1 is the simplest possible baseline—a model that predicts the next step as simply the previously seen frame.

**Penn Treebank:** This task follows the exact same problem setup as in Chapter 4 and the data is exactly the same. The task is simply to learn to predict the next character (each encoded using a one-hot representation) in a sequence of a characters.

6.3.1 The Bouncing Balls Problem

We trained TNCNs with two, three, and four layers of latent variables, searching for the size of the layers over the range \( \{1000 - 2000\} \) (with performance measured on the validation subset). In this experiment, the hyperbolic tangent activation function
ultimately proved to be the most useful (we also experimented with nonlinearities, but this yielded the best consistent generalization). Parameters were initialized from centered Gaussian distributions with $\sigma^2 = 0.01$ (except in the 3 and 4-layer TNCNs, setting the top-level recurrent and generative weights using $\sigma^2 = 0.025$ improved performance a bit further). Error feedback parameters were initialized with centered Gaussians of $\sigma^2 = 0.1$ (again, in the case of the 3 and 4-layered models, we used $\sigma^2 = 0.025$ for the top-level parameters). The value of the update noise process standard deviation was set to $\sigma_g = 0.005$ (to control the stochastic approximation of the prior over weights). The sparsity coefficient was $\lambda = 0.001$ and error weight decay parameter was $\gamma = 0.05$ while the correction factor was $\beta = 0.15$. Parameter updates at each time step were estimated using mini-batches of 20 samples (across 20 parallel videos). Parameters were updated using the method of stochastic gradient descent with a step-size of $\lambda = 0.005$. We further apply norm-rescaling to the gradients computed by Discrepancy Reduction (threshold is 5) [7]. Furthermore, we impose a max-norm constraint on the model parameters by projecting the parameter values after each update to the L2 ball, centered at the origin, of radius $l = \{1, 5, 15\}$ (value chosen based on validation).

We report our models’ average squared next-step prediction (20 trials) per frame in Table 6.1 and compare against previously reported errors. The proposed TNCN performs better than all of the baselines. Furthermore, we see that the inclusion of additional hidden layers actually helps the TNCN directed model, pushing it to generalization beyond the deep temporal sigmoid belief networks, which are also directed graphs. Note that all of the other models we compare utilize back-propagation through time as a core mechanism while our approach is incremental and adaptive, requiring no unfolding. To improve the performance of our model even further, we believe using an adaptive iterative inference scheme (where we apply multiple steps of error-correction per time step) combined with learnable variance parameters are key ingredients.

What is most surprising is that our simple way of building non-linear error units was effective in creating useful local representation targets. This is evidenced by the fact that performance vastly improves upon adding additional layers of neurons to the top-down generative model. Since each layer higher up in the network aims to do a better job at explaining the layer representation below, the local targets, created by the error feedback weights, become useful during learning as well as in online inference. The targets, in effect, help keep the model on track as it updates its latent representations given the sequence data it encounters, step by step.
Table 6.1. Test performance on the bouncing ball generative modeling problem. To the left is model test squared error using original dataset originally proposed in [1]. To the right is the performance (also squared error) of models trained on the two-ball version of the problem and evaluated on different test-sets with different number of balls.

<table>
<thead>
<tr>
<th>Ball Model</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame t-1</td>
<td>11.73 ± 0.28</td>
</tr>
<tr>
<td>TSBN-J [255]</td>
<td>9.48 ± 0.39</td>
</tr>
<tr>
<td>RTRBM [253]</td>
<td>3.88 ± 0.33</td>
</tr>
<tr>
<td>SRTRBM [256]</td>
<td>3.31 ± 0.33</td>
</tr>
<tr>
<td>TSBN-4 [255]</td>
<td>3.07 ± 0.40</td>
</tr>
<tr>
<td>DTSBN-D [255]</td>
<td>2.99 ± 0.42</td>
</tr>
<tr>
<td>DTSBN-s [255]</td>
<td>2.79 ± 0.39</td>
</tr>
<tr>
<td>2-TNCN</td>
<td>3.55 ± 0.19</td>
</tr>
<tr>
<td>3-TNCN</td>
<td>2.70 ± 0.15</td>
</tr>
<tr>
<td>4-TNCN</td>
<td>2.55 ± 0.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>RNN</th>
<th>TNCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Balls</td>
<td>2.13</td>
<td>1.68</td>
</tr>
<tr>
<td>3 Balls</td>
<td>5.80</td>
<td>4.54</td>
</tr>
<tr>
<td>4 Balls</td>
<td>16.77</td>
<td>12.33</td>
</tr>
</tbody>
</table>

6.3.2 The Language Modeling Problem

In this task setup, we train a TNCN next-step prediction language model. In this scenario, we are interested in seeing if the TNCN can handle discrete variables, since normally predictive coding models are usually applied to continuous-valued inputs (as in images/frames from videos). Character language modeling provides a good challenge in that a connectionist model would need to learn how to spell and arrange words to compose a sentence.

To train TNCNs on this data, we make use of the same Penn Treebank character-level data and problem setup as in Chapter 4. The TNCN model trained on this problem has three hidden layers and made use of the Δ-RNN state model. With mini-batches of 50 samples, parameters were updated using the method of steepest gradient descent, of which we employed the Adam [161] adaptive learning rate scheme with the step-size fixed to $\lambda = 0.0002$. While most of the other settings such as the sparsity coefficient were the same, we found that actually setting the error weight decay to nearly zero or zero proved more effective in this situation. We used a correction factor of $\beta = 1.0$.

In the rightmost set of results of Table 6.1, we further present an additional experiment investigating how well a TNCN can generalize to test-sets containing n-body systems of balls it has clearly never been exposed to in training. Specifically, we trained a 4-layer TNCN (each layer with 500 units) on a dataset of the same size as the original 3-ball problem but this time with only two balls interacting. After training on video samples of the two-ball system, we evaluate generalization on a 2-ball, 3-ball, and 4-ball test set. The preliminary experimentation revealed using this state model sped up convergence and yielded better generalization on the validation samples.
Table 6.2. Test performance on the Penn Treebank character language modeling problem.

<table>
<thead>
<tr>
<th>PTB Model</th>
<th>BPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>UORO [258]</td>
<td>2.61</td>
</tr>
<tr>
<td>KL-RTRL [258]</td>
<td>1.72</td>
</tr>
<tr>
<td>TBPTT-5 [258]</td>
<td>1.58</td>
</tr>
<tr>
<td>TBPTT-25 [258]</td>
<td>1.56</td>
</tr>
<tr>
<td>LSTM [259]</td>
<td>1.18</td>
</tr>
<tr>
<td>TNCN</td>
<td>1.79</td>
</tr>
</tbody>
</table>

hope is that if the model acquired useful structure from the original 2-ball data that might describe some of the simple physics that underlie the data generating process, the model should be able to generalize somewhat to systems containing more than two balls. We also present, for comparison, a single hidden-layer Elman-RNN model that contains the same number of parameters as the TNCN, trained via truncated back-propagation through time (10 steps). Note that the Elman-RNN sports a set of auto-regressive connections, i.e., its input-to-hidden weight matrix, which means the model gets fed in at each time-step the previous video frame whereas the TNCN does not receive this information, given that it is a directed graph. Interestingly enough, we observe that the TNCN appears to generalize better than the RNN model consistently across different n-body systems. This promising result indicates that the TNCN’s continual error-correction mechanism appears to be useful when adapting to novel scenarios, yielding a better next-step prediction.

In Table 6.2, we measure the TNCN’s bits per character (BPC), defined in Chapter 4, on the test set. Our interest is to see if the TNCN can reach reasonable generalization on this task and compare its performance to other current alternatives to back-propagation through time, such as the Unbiased Online Recurrent Optimization algorithm (UORO) and algorithms including modern incarnations of real-time recurrent learning (RTRL, [257], e.g., Kronecker Factored RTRL (KF-RTRL) [258]. Notably, we observe that the TNCN can outperform UORO estimator and comes close to the performance offered by KF-RTRL. However, these current experimental measurements were the result of only training the TNCN over 30 epochs and with a hidden layer size of 1000 and very minimal tuning. Far better generalization performance is expected with more careful selection of various hyper-parameters. Note that TBPTT is truncated back-propagation through time, which requires unrolling the network back in time over $T = 5$ or $T = 25$ steps.
6.3.3 Learning From Evolving Streams

While it is highly encouraging to see that the TNCN can extract useful structure from sequences in fixed datasets without back-propagation through time, it would be useful to see if the model can handle one long, continuous stream of data-points (which is often the case in time-series/forecasting setups). Being able to do so would be extremely useful for applications in robotics or autonomous vehicles or user applications that require continuous adaptation, where data would naturally come to the learner in one, seemingly infinite temporal stream. Furthermore, building a dynamic, incremental model would yield the benefit that one could access the model that has seen a significant portion of the streaming sample and use it for prediction even before training has finished.

When one considers samples available from a stream, the character of the learning problem changes. First, it is quite likely that once seen, these samples are not to be accessed again by the model in its learning phase, as is common in static, finite data-set settings (i.e., where a model’s progress is measured by epochs, or full passes through the entire data-set). Second, in the case of streams, it is quite likely that the target distribution is changing, meaning that samples drawn from it carry new information about its current state and that information extracted from previous samples may be less useful over time. However, the temporal relationship between old and new samples is not clear, and thus completely forgetting information acquired earlier in the learning process may significantly hinder performance. Given this problem scenario, we are interested in seeing if continual error-correction property of the TNCN might allow it to handle this distributional evolution.

For the final experiment in this chapter related to the TNCN, we reformulate the sequential prediction task above as an online, stream-based problem. This will allow us to investigate how well the TNCN, in comparison to an auto-regressive model, such as the RNN, learns when the data is presented incrementally/online (much as was done in some experiments of Chapter 3). This small experiment represents a first attempt at evaluating neural temporal models in scenarios that require true adaptation, where patterns cannot be explicitly stored for later replay and the information contained within each pattern must be extracted effectively and immediately. In short, this means we will be investigating how well these temporal neural models perform when continually run, i.e., the internal state is constantly carried over and nothing is reset (as is done in static datasets, e.g., in the presence of sentence boundaries in language modeling).
In addition to the original TNCN model used for the original bouncing ball problem, we also present results for a variation we call the *delta-flow TNCN*, or *TNCN-DF*. The TNCN-DF differs from the TNCN in that its hidden states are calculated using a projection of the difference of inputs\(^7\), \((x_{t-1} - x_{t-2})\). This extra information might prove useful to the TNCN and is an appropriate inductive bias given that the data is a sequence of video frames. For the latent layers of the TNCN, delta information is also provided, and in general is calculated for any given layer as \(v^\ell_t = (y^\ell_{z,t-1} - y^\ell_{z,t-2})\) (where \(y^0_{z,t} = x_t\)). For any given layer in the model, the full state, before error correction, is calculated as:

\[
    h^\ell_t = V^\ell_t z^\ell_{t-1} + G^{\ell+1}_{t+1} z^{\ell+1}_t + M v^{\ell-1}_t + b^\ell_t, \quad z^\ell_t = \phi_g(h^\ell_t) \tag{6.15}
\]

where \(M\) is the synaptic weight matrix used to project the delta vector to the hidden state. The above equation simply incorporates the projected delta vector as an additional term in the pre-activity calculation but more sophisticated mechanisms are possible, such as using the projected delta to gate parts of the hidden state vector.

For this streaming setup, we implemented a custom bouncing ball video generator that allows balls to enter and exit the pixel grid at predetermined markers, or specific frames. In this particular simulation, the video frames are generated according to the following process:

1. For the first 4,999 frames, the system contains 2 balls of different radii and different masses bouncing around the pixel grid.

2. At frame 5,000, a new ball is randomly created, with a specified radius and mass, and allowed to enter the frame from outside of the pixel grid (the starting coordinates are randomly generated), creating a 3-body system. Once the new ball is fully within the pixel grid, the wall boundaries are re-activated.

3. At frame 10,000, another ball is randomly created and allowed to enter the grid, creating a 4-body system.

4. At frame 17,500, the boundaries disappear for one of the balls (randomly selected), allowing it to leave the system. Soon after a 3-body system remains.

\(^7\)The intuition behind this delta vector input is loosely inspired by the concept of optical flow. For example, in a visual scene, motion between two consecutive images can be captured by a displacement vector which depicts the movement from the last frame to the current frame.
Figure 6.3. Model performance evaluated on the streaming version of the bouncing ball problem. Depicted, for all of the models in terms of prequential squared error, is (a) the next-step prediction, and (b) the prediction horizon, or 5 steps of closed-loop prediction.

5. At frame 25,000, another ball is randomly dropped out of the system, yielding a 2-body system.

6. At frame 30,000, another ball is generated outside the pixel grid and allowed to enter creating a new 3-body system.

7. Finally, at frame 40,000, a new ball is allowed to enter the system creating a 4-body system for the last 10,000 generated frames.

In Figure 6.3, we can see two online learning curves of the two TNCN variations and the Elman-RNN. The first learning curve corresponds to next-step prediction error and the second curve corresponds to closed-loop prediction\(^8\) over 5 steps into the future, e.g., a forecasting horizon of 5. For the next-step prediction plot, we also track the performance of a simple last-frame prediction baseline, Frame \(t - 1\), which was also used in previous experiments related to the bouncing ball problem. The curves depicted

\(^8\)Open-loop prediction refers to the situation where the model predicts the next data point \(x_{t+1}\) conditioned on the previous ground-truth pattern \(x_t\). Closed-loop prediction refers to the situation where the model is forced to predict \(x_{t+1}\) but fed in its previous prediction \(z_t^{t-1}\) instead of the previous ground-truth pattern.
in Figure 6.3 are created by plotting the current accumulated loss\(^9\), or rather, prequential error (with forgetting factor set to 1.0), as originally defined in Chapter 3.

The current results indicate that the TNCNs generalize much quicker than the RNN model in terms of prequential squared error and consistently maintain better predictive performance, even as it crosses the boundaries when the bouncing ball distribution changes. Towards the very end of the simulation, e.g., closer to frame 50,000, the RNN’s next-step error appears to slowly improve and appears to match the simple TNCN’s generalization error. However, the TNCN-DF model reaches the best overall prequential squared error.

In the case of longer-term prediction, both TNCN models do far better than the RNN model with the simple TNCN performing the best. While the inclusion of the delta vector appears to give TNCN-DF the best performance in next-step prediction, it appears that this information does not prove as useful in longer-term closed loop prediction, although performance is still better than the RNN. The decrease in longer-term performance might be related to the fact that deltas would be computed using model outputs instead of ground-truth video frames (which differs from the training condition where the delta is computed as the difference between two actual frames).

While these initial results are promising, further experimentation is needed to explore the streaming/dynamic generalization of these neural models. Furthermore, the evolving distribution represented by this particular bouncing ball simulator could be changed to make the problem far more difficult and more varied over time. Future work will entail developing more complex simulated streams, perhaps modifying video game emulators (typically used in reinforcement learning experiments) to generate long frame sequences depicting agents attempting to achieve some goal.

### 6.4 Continual Sequential Learning

It is the ultimate goal of machine learning and artificial intelligence research to develop a computational agent that can learn over time and continually accumulate new knowledge while retaining its previously learned experiences [260, 261]. As new tasks arrive (with potentially datasets accompanying them), the agent is expected to process this data and learn the new task but still remember how to complete the old tasks (at least without

---

\(^9\)As is typically done in online learning setups, we first have each model make a prediction and then update parameters, e.g., test-then-train.
significant degradation of performance or loss of generalization). Modern connectionist systems today are trained on modality-specific samples of data (collected in controlled environments) in isolation and random order. This is a far cry from what we really desire from learning machines. Additionally, in the case of developmental learning, where the agent would need to be interactive as well as autonomous, there might not even be a distinction between training and test phases (the current standard setup of modern machine learning systems).

When we look to humans or other animals, we see that they are more than capable of learning in a continual manner, making decisions based on sensorimotor input throughout their lifespans [262]. This ability to incrementally acquire and refine knowledge over long periods of time is driven by neurocognitive processes and functions that come together to create the experience-driven specialization of animal motor and perceptual skills [262, 263]. In order to continually learn and adapt, the brain carries out two important tasks: 1) generalize across experiences and 2) retain specific memories.

Thus, evaluating how neural architectures and the algorithms they use on sequences of tasks, as opposed to single isolated tasks, proves to be a far greater challenge. The earlier part of this chapter attempted to developed an incremental and dynamically adaptive neural system, ultimately suited to processing streams of data, which is indeed a critical component of lifelong learning. However, the data in the earlier experiments were from one modality and belonged to the same prediction problem/task. It will now be the concern of the rest of this chapter to examine the sequential learning problem and experiment with neural systems as they are applied to datasets presented in a task series.

### 6.4.1 Characterizing the Problem of Sequential Learning

A lifelong learning system is defined as an adaptive agent capable of processing continuous streams of information, where the information becomes progressively available over time and where the number of actual tasks to be learned is undefined. When new information is encountered, the agent should be able to integrate it without losing valuable task-relevant knowledge from the past. In this section, we will first attempt to formalize the lifelong sequential learning problem from a very general perspective, and then more narrowly focus on the aspect of it, with respect to connectionist systems, in this chapter.

Starting from an early definition [264] of this form of learning, we assume that a system has performed $N$ tasks. When faced with the $(N + 1)$th task, the system
should use the knowledge it has gained from the previous $N$ tasks to aid in learning and performing the $(N + 1)$th task. As such, a lifelong learning system is defined to be a continuously learning process and at any point, has already performed a sequence of tasks $\{T_1, T_2, \cdots, T_N\}$. Each task within this task history has its own corresponding dataset $\{D_1, D_2, \cdots, D_N\}$. The tasks may be of the same or different types and may come or not come from the same domain\(^{10}\) or modality. When faced with a new $(N + 1)$th task, $T_{N+1}$, along with its corresponding dataset $D_{N+1}$, the learner must utilize its prior knowledge, stored in a construct called a knowledge base (KB) which represents accumulated knowledge from prior experiences, in order to learn $T_{N+1}$. The objective of the system is to optimize its performance on the current task at hand $T_{N+1}$ (though it can optimize on older tasks so long as they are treated as previous tasks) and the KB is updated after learning $T_{N+1}$ is finished.

What the general definition above establishes is that an agent is engaged in continuous learning, accumulates knowledge, maintains it (through a knowledge base), and uses past experiences to ease future learning. The agent could be engaged in a never-ending/infinite series of tasks, and as it learns more and more tasks, the agent should become better at the act of learning as well as more “knowledgeable”. This process of improved learning has also recently been viewed as meta-learning [265], or learning how to learn, which brings along with it additional classes of techniques and methods. Lifelong learning essentially encompasses all other forms of learning, including transfer learning, multi-task learning, and online and semi-supervised learning.

Tasks are given to the learner in a sequence, but the shift from one task to another might happen abruptly or gradually, and the more general view of lifelong learning is that the agent should be able to even find its learning tasks and data by interacting with an environment, i.e., self-motivated learning. However, while certainly more general, looking at truly general lifelong learning is a grand open challenge to the field, and one this thesis, specifically this chapter, will not attempt to tackle.

Past/shared knowledge in lifelong learning systems can be decomposed into two categories: global knowledge and local knowledge. With respect to global knowledge, it is assumed there is a latent global structure shared across all tasks [266]. This latent structure can be learned and is exploited when encountering new tasks. Local knowledge, on the other hand, does not assume global structure that is shared among the tasks [267].

\(^{10}\)Domain usually informally refers to a fixed feature space which may or may not facilitate multiple, different tasks.
In this view, knowledge from previous tasks is broken down into pieces and different pieces from prior tasks might be used in learning $T_{N+1}$. Such bits of knowledge are local to the task they were acquired from, not part of a coherent global structure.

With respect to the tasks themselves, we should distinguish between independence and dependence. A task $T_i$ that is independent of other tasks is one that can be learned independently, although improvements in learning it can still come from shared prior knowledge from prior tasks. A task $T_i$ might be considered dependent if it shares some dependence on other tasks. For example, in cumulative learning, new classes are added over time, which means that a task $T_{i+1}$ might be adding classes in addition to those that have emerged in task $T_i$. A classifier must learn to distinguish data based on prior class categories but also currently added classes.

With respect to the neural networks we will investigate in this part of the chapter, we will assume a sort of global latent structure over tasks as concretely represented by shared hidden layers of a neural network. The synaptic weights that connected these shared hidden layers could be viewed as a sort of implicit knowledge base. We will investigate the scenario of sequential learning in two cases:

1. When the boundaries between tasks are clear/known, which allows us to create a top-level predictor per task. This could be considered as a form of local knowledge that is conditioned on the internal global latent knowledge defined by the hidden representations of the neural system.

2. When the boundaries between tasks are not clear/known, which is the focus of cumulative learning. We will refer to the problem facing the learner in this scenario also as the “growing softmax problem”, since we must expand the prediction layer’s outputs dynamically as new class categories are encountered. Experimenting with this setting is also much closer to how early-stage learning occurs in human infants [268].

In our experiments, we will test on two tasks—one in continual classification and another in a task that we will call continual (or lifelong) autoencoding. The overall flow and an example two-layer MLP used in each type of problem is graphically depicted in Figure 6.4.
6.4.2 The Problem of Catastrophic Interference

It is well-known that when artificial neural networks are trained on more than one task sequentially, the new information in subsequent tasks leads to catastrophic interference, or catastrophic forgetting, with the earlier ones [269]. This happens in connectionist systems when the new data instances to be learned are significantly different from previously observed ones, which causes the new information to overwrite the currently existing knowledge encoded in the system’s synaptic weights, due to the sharing of neural representations over tasks [269, 270] (also known as the representational overlap problem).

Within isolated task-specific learning (offline learning), this type of overwriting does not occur because the patterns are presented to the agent in a pseudo-random fashion multiple times, but in the case of streaming data or sequences of tasks, the process of reshuffling and re-presentation is not possible. The problem of catastrophic interference has been a problem studied widely especially for the case of back-propagation-based networks [271, 272].

Over the decades, there have been many approaches proposed to mitigate or eliminate catastrophic forgetting in neural systems. Some of the earliest attempts proposed memory systems where prior data points were stored and regularly replay these stored patterns to the network, in a process called “rehearsal”, by interleaving these with samples.
drawn from new datasets [273–277]. The main drawback of these approaches, though
effective in combating forgetting, is that they require explicitly storing old data, which
is not only a mechanism known to not exist in the brain, but also leads to exploding
working (hardware) memory requirements. In addition, rehearsal-based approaches do
not tackle the problem of knowledge overwriting itself, not offering any mechanisms to
preserve consolidated knowledge in the face of acquiring new information [263]. Other
approaches attempt to allocate additional neural “resources”, i.e., growing the networks,
when required [278, 279] motivated by earlier work/findings [280], but these lead to
dramatically increasing the computational requirements over time as the networks become
larger and larger. To compound these issues further, systems with growing capacity cannot
know how many resources to allocate at any given time since the number of tasks and
number of samples cannot be known a priori (without imposing strong assumptions on the
input distribution). Other approaches try to block old information from being overwritten
through the development of regularization schemes [281]. From this vast collection of
research, each approach with strengths and weaknesses, three suggested remedies have
emerged: 1) allocate additional neural resources to accommodate new knowledge, 2) use
non-overlapping representations (or semi-distributed representations [282]) if resources
are fixed, and 3) interleave old patterns with new patterns as new information is being
acquired.

One potential source of the forgetting in neural systems might be related to the
very learning algorithm used–back-propagation of errors. Given its many issues, as
discussed in Chapter 5, such as problems related to gradient transmission through the
global feedback pathway, forgetting might also be occurring as a result of the unstable
mechanics of the learning procedure (and further compounded by the various constraints,
such as the need for sufficient linearity, that we impose on the system in order to make
backprop work well for deeper networks). LRA might offer some interesting side-effects
as a result of its independence of a global feedback pathway, and since one possible
pathway to lifelong learning systems is to integrate more neuro-cognitive mechanisms
into the learning and architecture, this might prove promising.

We will first start by operating from the hypothesis that catastrophic forgetting might,
in part, be caused by the very learning procedure utilized in adapting synaptic weights.
Therefore, we will conduct experiments that will backprop to LRA-trained networks on
several task setups that serve as a microcosm of the continually learning problem.
6.4.3 Cumulative Learning: The Growing Softmax Problem

The task of continual learning becomes much more challenging when task boundary information is not available. Without this inductive bias, the model will instead have to learn an output competition layer (such as the soft competition produced by a softmax winner-take-all neural layer) for all of the category units, even as new classes are added with each new subtask. This version of the task is also known as cumulative learning, or cumulative machine learning [283], since it can be considered as a simplification of real human cumulative learning, or the cognitive process by which a human agent accumulates skills and knowledge that will facilitate later cognitive development. In short, a cumulative learning system does not know all of the classes of a labeled dataset a priori and instead must detect when new classes emerge and further update itself (selectively) without requiring the whole system to be entirely retrained.

In this thesis, we will refer to the challenge of cumulative learning as applied to a neural systems as the growing softmax problem, since MLPs typically employ a softmax post-activation to the output layer in order to generate posterior probabilities for class-based prediction. Assuming a left-to-right ordering of class labels (for simplicity), as \( k \) new categories are encountered when the model is tasked with a new problem \( T_i \) and its dataset \( D_i \), we will simply expand the softmax output layer by \( k \) units and concatenate \( k \) new column vectors (randomly initialized) to the right of the (column-major) class output matrix (the output biases are also expanded accordingly). As we will see in the following experiments, this makes the problem of cumulative learning much more challenging, since now the neural model must learn to adapt these new output units without destroying its old predictions.

It is unlikely that using LRA alone can alleviate the catastrophic interference problem in neural models. In this section, we investigate extending the LRA algorithm in two ways in the hopes of improving longer-term knowledge retention in neural systems.

6.4.4 Combining Error-Driven & Anti-Hebbian Learning

As our experiments will reveal, the retention effect of just simply using LRA instead of backprop to train a network quickly goes away in the face of the cumulative learning scenario. One way we will investigate improving an LRA-trained network’s ability to remember past task information is to alter the LRA update rule. Specifically, we will examine combining the error-driven update rule LRA currently utilizes with a weighted
Specifically, the anti-Hebbian update rule we implemented is as follows (within a gradient descent framework, given that Hebbian/anti-Hebbian updates are used in gradient ascent):

$$\Delta_{ah} W^\ell = \xi \frac{z^\ell (z^{\ell-1})^T}{||z^\ell (z^{\ell-1})^T||_2}$$

(6.16)

where we introduce an additional decay term to the update, $\xi$, which we found useful to set at 0.4. Note that this update is normalized by its own L2 norm. This rule is then combined with the original error-driven update as follows:

$$\Delta W^\ell = e^\ell (z^{\ell-1})^T + \xi \frac{z^\ell (z^{\ell-1})^T}{||z^\ell (z^{\ell-1})^T||_2}$$

(6.17)

where, for the left term, we use the error unit activities for the post-synaptic neurons with the input activities, and, for the left term, we use the post-synaptic neuronal activities by the input activities. Again, $(\cdot)^T$ denotes the matrix transpose operator. In this rule, the error units help guide the parameters towards configurations which are useful for the current task while still incorporating an unsupervised learning rule that might encourage weights to extract general statistical structure from the data [285]. Since anti-Hebbian learning minimizes the output variance of the units it is applied to, driving synaptic weights towards zero, this might serve as a useful counter-balance to the ever-growing weights created by the error-driven term (which could be regarded itself, in the case of mean squared error, as the combination of a Hebbian and anti-Hebbian term). This opposite pressure created by anti-Hebbian learning might serve as useful regularization for the network (if nothing else, it might act as a form of gradient noise) as it is trained on a given task in a sequence, perhaps mitigating some of the forgetting of the previous task’s information.

### 6.4.5 The Memory-Consolidation Algorithm

While we have seen that using a learning procedure different than backprop can help in mitigating some information loss, it might be more reasonable to instead separate the task of retaining old information from acquiring new task information. This idea comes from Complementary Learning Systems (CLS) theory [286], where one model actually
works like the hippocampus in the brain and another model, of the same design, serves to store old information and refresh the hippocampus, acting somewhat like the neocortex. To transfer information between the hippocampal network and the neocortical network, pseudo-patterns are used, which are randomly generated vectors (playing the role of $x$) and are propagated through a model to obtain a corresponding $y$. During the training of a specific task, in addition to the samples that are part of the current dataset $D$, pseudo-pattern pairs generated from the neocortex model are used to refresh the hippocampus model. After learning a task, the information from the hippocampus model is then transferred to the neocortex model also through the use of pseudo-patterns. We can think of the hippocampus as the primary discriminative model, which makes the predictions for the desired tasks, and the neocortex as a complementary discriminative model, which is primarily used to approximate function defined by the primary discriminative model and work as long-term pattern storage.

We also will take an approach similar to CLS, however, we will deviate from the typical setup in one key way. Instead of using pseudo-patterns [287], which, as we found in preliminary experimentation, create an issue in the face of using one-hot encoding outputs\textsuperscript{11}, we will seek to generate samples of the original task distribution. This means that we need to learn a complementary generative model instead of a discriminative one as is done in a CLS system.

To develop the complementary generative model, we take an approach similar in spirit to the dual discriminative-generative models, or (deep) hybrid models, developed in Chapter 3, which demonstrated the power of jointly learning a discriminative and generative model for semi-supervised learning problems. However, in this chapter, the dual discriminative-generative model will be directly built on the algorithmic underpinnings of Discrepancy Reduction and makes use of target generation and local losses to guide the generation process. More importantly, the hybrid system will not be using both facets, i.e., the discriminative and generative facet, to make the final predictions, but rather, will use the discriminative facet to serve as the task network and the generative facet to work as a form of active memory [288] that can generate patterns $x$ given a $y$, or a directed generative model that works to roughly “invert” the task network.

The inspiration for this particular dual model comes from studies of the rodent hippocampus itself [289], which examined the relationship between two (complementary)\textsuperscript{11}How do we use the correct encoding since training with real-valued outputs is not applicable in a cross-entropy regime?

\textsuperscript{11}How do we use the correct encoding since training with real-valued outputs is not applicable in a cross-entropy regime?
components of the hippocampal network, the dorsal and the ventral hippocampus, in
the context of two-way active avoidance conditioning. In this study [289], chemical
infusions (of tetrodotoxin and/or muscimol) were used to temporarily, functionally
impair either the ventral or dorsal hippocampus. These infusions were found to ultimately
impair rats’ ability to learn, with impairment of the ventral causing immediate failure
and impairment of the dorsal causing failure later (40-50 minutes after infusion). Our
interpretation of this result is that impairing the ventral component is similar to damaging
or preventing the task network from learning the new task information while impairing
the dorsal component damages or shuts off the active memory needed later when new
information overrides what is currently stored in the task network. In a very loose,
abstract sense, one could consider the task network to be the ventral hippocampus and the
generative network to be the dorsal hippocampus. As such, our complementary neural
system could be considered to crudely and abstractly model two portions of hippocampus
itself (a fast and slow memory component) and not necessarily the hippocampus and
neocortex.

Specifically, we propose the form of the complementary generative model to be a
multi-layer stochastic model that will try to generate the various neural activities of the
discriminative task model. In keeping within the framework of Discrepancy Reduction
developed in this thesis, we will train the generative model with layerwise targets, but
this time, our local targets can come for free from the task model itself and do not require
extra error feedback weights. The generative network will specifically be a deep, directed
Gaussian model, where each layer has a set of weights that generate a mean vector,
or expectation, and another set of weights that generate variance vector, specifically, a
diagonal covariance matrix. Formally, a three-layer generative model, with parameters
$\Theta = \{W^1, W^2, W^3, W^4, W^5, W^6\}$, would take the following form:

$$
\begin{align*}
\mathbf{z}^3 &= \mathbf{y}, \quad \mu_2 = W^3_\mu \mathbf{z}^3, \quad \text{diag}(\Sigma_2) = \text{softplus}(W^3_\Sigma \mathbf{z}^3) \\
\mathbf{z}^2 &= \mu_2 + \sqrt{\text{diag}(\Sigma_2) \otimes \epsilon_2}, \quad \mu_1 = W^2_\mu \mathbf{z}^2, \quad \text{diag}(\Sigma_1) = \text{softplus}(W^2_\Sigma \mathbf{z}^2) \\
\mathbf{z}^1 &= \mu_1 + \sqrt{\text{diag}(\Sigma_1) \otimes \epsilon_1}, \quad \mathbf{z}_0 = \text{sigm}(W^1_\mu \mathbf{z}^1)
\end{align*}
$$

where $\otimes$ is the Hadamard product. Note we employ the reparametrization trick [100]
to draw samples from the learned multivariate neural Gaussian models at each layer.

\footnote{12In this type of conditioning, rats must learn to avoid foot-shocks while crossing from one side of a chamber/room.}
During learning, we substitute the generated samples at each layer with the actual target activities from the task model. This will allow us to track the performance of this generative network as an autoencoder, circumventing drawing multiple samples in order to compute the expectation needed to evaluate it properly as a generative model. The total discrepancy objective for this model uses the full negative Gaussian log likelihood at each layer, defined as follows:

$$\mathcal{L}(x_t; \Theta) = \sum_j \frac{-(x_t[j] - z_0^t[j])^2}{2\sigma_0^2[j]} + \sum_{\ell=1}^L \sum_j \frac{\log(diag(\Sigma_\ell)[j])}{2} - \frac{(y_{\ell,t}[j] - \mu_\ell[j])^2}{2\text{diag}(\Sigma_\ell)[j]}$$

(6.21)

where we assume a diagonal covariance matrix for simplicity and computational ease (since a full covariance matrix would require expensive matrix inversion in a complete, multivariate Gaussian log likelihood objective).

The overall procedure, which we will call Memory-Consolidation, would then train the task network $NET_D$ and generative network $NET_G$ according to the following process when operating on a task $T_i$:

1. Sample a mini-batch of data points, $(y, x)$
2. If $i > 0$, then sample $M$ patterns from $NET_G$, $(\tilde{y}, \tilde{x})$
3. Update $NET_D$ with $(y, x)$ and $(\tilde{y}, \tilde{x})$ (if it exists)
4. Update $NET_G$ with $(y, x)$
5. (Optionally), if at the end of $T_i$, then sample $M$ patterns from $NET_G$ and update $NET_D$ for $K$ iterations.

The training algorithm that uses all but the last step of the procedure sketched above will be referred to as MemConsol-Refresh (or MemCon-R) while the algorithm that also incorporates the optional last step will be referred to as MemConsol-Transfer (or MemCon-T). The second variation of Memory-Consolidation is denoted by the word “transfer” since the extra refresh phase that occurs between tasks works like a rest period where the generative network attempts to fantasize samples that can be used to fully update the task network. Note that a possible extension we will explore of this variation is to induce a form of pre-play, where we can augment input to the generative network with fake, or “future” class input nodes (expanding the input $y$), that can be used to
simulate new classes and perhaps better encourage growing the softmax in the face of true cumulative learning.

The complementary model that underpins Memory-Consolidation is depicted in Figure 6.5.

### 6.5 Experiments: Learning Across Tasks

#### 6.5.1 Continual Classification

In this experiment we train 3 hidden-layer multilayer perceptrons (MLPs) for the scenario of continual classification (similar in spirit to the small setup used in [223]). We re-appropriate the MNIST and Fashion MNIST datasets, last described in Chapter 5, and create a set of four “subtasks”, or rather different classification problems that involve different classes of the original ten in each full dataset. The subtask definitions, which will we reuse in the next experiment as well, are as follows:

1. Task $T_1$: Classify handwritten digit samples according to whether they belong one of the following classes: $C = \{0, 1, 2, 4, 5, 6, 8\}$

*Figure 6.5. The complementary system defined under the memory consolidation—in this setup, a discriminative network, i.e., the network/graph to the left, serve as a task model and temporary memory while another, generative network, i.e., the network to the right, serves as a form of longer-term memory. The generative model actively refreshes the discriminative model while also trying to store statistical structure regarding the data generating process itself. The generative model will, in some sense, serve as the approximate “inversion” of the discriminative model.*
2. Task $T_2$: Classify handwritten digit samples according to whether they belong one of the following classes: $C = \{3, 7, 9\}$

3. Task $T_3$: Classify images of clothing according to whether they belong to one of the following categories: $C = \{0, 1, 2, 3, 4, 6, 8\}$

4. Task $T_4$: Classify images of clothing according to whether they belong to one of the following categories: $C = \{5, 7, 9\}$.

Each hidden layer of the MLP contains 1000 hidden units and the standard hyperbolic tangent is used as the hidden activation function. Weights were initialized using a Gaussian distribution scaled by each layer’s fan-in. Parameters were optimized using stochastic gradient descent with a fixed learning rate of 0.01 and fixed momentum of 0.9. Each mini-batch consisted of 10 samples and the network was only allowed a single pass over each subtask’s dataset, i.e., one epoch. The output layer, one set of parameters allowed per subtask, was a maximum entropy classifier and the objective was to minimize the categorical cross entropy.

To measure model generalization over the sequence of tasks, we utilize the metrics proposed for continual learning in [2]. Backward transfer (BWT) measures the influence that learning a task $t$ has on the performance of task $k < t$. A positive BWT indicates that a learning task $t$ increases performance on a preceding task $k$. As such a higher BWT is better and a strongly negative BWT means there is strong forgetting (measures catastrophic forgetting). Forward transfer (FWT) is used to measure the influence that learning task $t$ has on the performance of a future task $k > t$. In essence, FWT measures the zero-shot learning capability of a given model. Formally, these metrics are defined as follows:

$$ACC = \frac{1}{T} \sum_{i=0}^{T} R_{T,i}$$

$$BWT = \frac{1}{T-1} \sum_{i=1}^{T-1} R_{T,i} - R_{i,i}$$

$$FWT = \frac{1}{T-1} \sum_{i=2}^{T} R_{i-1,i} - \bar{b}_i.$$

Note that ACC is simply average accuracy.

We observe, in Tables 6.3 and 6.4, surprisingly, that LRA-trained MLPs do not appear to catastrophically forget whereas the same cannot be said for backprop-trained networks.
Table 6.3. Generalization accuracy of a network trained on different classification subsets over time. This amounts to the task metric matrix $R$ in [2].

<table>
<thead>
<tr>
<th>Eval Pt.</th>
<th>Back-propagation of Errors</th>
<th>Local Representation Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>After T1</td>
<td>0.9602</td>
<td>0.1943</td>
</tr>
<tr>
<td>After T2</td>
<td>0.7928</td>
<td>0.9718</td>
</tr>
<tr>
<td>After T3</td>
<td>0.6939</td>
<td>0.9186</td>
</tr>
<tr>
<td>After T4</td>
<td>0.4601</td>
<td>0.7883</td>
</tr>
</tbody>
</table>

Table 6.4. Lifelong generalization metrics, based on task performance matrix $R$ [2], for the networks trained on the task set of Table 6.3. For all metrics, higher values are better.

<table>
<thead>
<tr>
<th></th>
<th>Back-propagation of Errors</th>
<th>Local Representation Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>0.6869109</td>
<td>0.8925094</td>
</tr>
<tr>
<td>BWT</td>
<td>-0.29217702</td>
<td>-0.004020234</td>
</tr>
<tr>
<td>FWT</td>
<td>0.2010528</td>
<td>0.20712167</td>
</tr>
</tbody>
</table>

This is an interesting result, and provides some encouraging positive evidence that our earlier hypothesis that part of the issue of forgetting might be tied to the use of back-propagation of errors for adapting synaptic weights. One aspect of LRA that might be facilitating the storage of information across tasks is the error weights themselves—since these adapt quite slowly (or not at all if we use LRA-fdbk), the error parameters might be implicitly serving as a form of longer-term memory. The LRA learning process might be forcing the MLP to anchor itself to the error patterns produced by the interaction between the error units and error weights, providing it with some ability to distinguish between pattern vectors from different datasets.

### 6.5.2 Continual Autoencoding

In this experiment, instead of tackling each subtask above as a classification problem, we make the problem much harder by instead ignoring the labels and focusing on the task of image reconstruction, or the task of auto-association. The splits to create the subtasks above are the same, but this time, we evaluate each model’s ability to reconstruct by reporting the mean squared error (MSE) over the test sets. This is certainly a much more challenging problem as now the autoencoding architecture must deal with extracting useful features about each dataset sequentially. The feature information acquired in the model’s shared hidden representations might prove useful across tasks, although once the boundary between digits and clothing is crossed (between tasks $T_2$ and $T_3$), the system
Table 6.5. Test reconstruction error of a network trained on different classification subsets over time. This amounts to a reconstruction error matrix (each slot is the squared error).

<table>
<thead>
<tr>
<th>Eval Pt.</th>
<th>Back-propagation of Errors</th>
<th>Local Representation Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>Init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>After T1</td>
<td>5.439</td>
<td>190.413</td>
</tr>
<tr>
<td>After T2</td>
<td>11.7686</td>
<td>4.5142</td>
</tr>
<tr>
<td>After T3</td>
<td>44.646</td>
<td>36.604</td>
</tr>
</tbody>
</table>

(a) Backprop-trained reconstructions.  
(b) LRA-trained reconstructions.

Figure 6.6. Reconstructions, on a randomly sampled test example, tracked across each new learned task. At the very top of each column is the original sample. Each row shows the reconstruction of the selected example after learning a new task.

should essentially just try to retain and keep separate the information it finds useful for reconstructing digits as the information it might need in reconstructing clothing images would be different.

The training settings remained the same as above. The architecture of the auto-associative network used in this experiment took the form of a bottleneck architecture where 600 units were in the first hidden layer, 400 were in the next, and 200 were in the penultimate hidden layer. The objective used was mean squared error and the output units used the logistic sigmoid (since the values on both datasets were preprocessed to lie in the range of \([0, 1]\)). The difference in this architecture compared to the one in the
last section, made clearer in Figure 6.4(b), is that it utilized a separate decoder for each
dataset but had to share the encoder across the series of tasks.

In Table 6.5, we observe again, the favorable property that LRA-based training ap-
ppears to mitigate the catastrophic interference effect (when compared to using backprop). Figure 6.6 shows, qualitatively, that the reconstruction ability of the auto-associative
systems begins to degrade with subsequent auto-association tasks when using back-
propagation. However, only minor degradation appears to happen when using LRA to
train the networks.

6.5.3 Cumulative Learning

As observed in Figure 6.7 and Table 6.6, our proposed approaches to neural learning do
appear to mitigate forgetting in the face of full cumulative learning, but to varying degrees.
We compare our proposed algorithms to standard back-propagation of errors (backprop),
Elastic Weight Consolidation (EWC) [281], and activation drop-out (Dropout), as in [290],
which argued drop-out can mitigate forgetting. Remembering that the goal of remedies
to catastrophic forgetting are ultimately trying to maximizing ACC while maximizing
BWT (or at least reach zero BWT). As we can see in our results, LRA with anti-
Hebbian learning actually appears to help, raising the ACC from Backprop’s 0.35 to
0.5, indicating that simply changing the way learning is conducted might improve
information retention in connectionist networks when faced with sequential learning.
Furthermore, this already improves other previously proposed modern remedies to
catastrophic interference including EWC and Dropout.

More importantly, we observe that we can actually reach much better performance
and memory retention when using the Memory-Consolidation algorithm. Notably, the
simplest version of Memory-Consolidation, MemCon-R appears to work the best, yielding
an impressive ACC of 0.77 and BWT −0.17 while MemCon-T does improve over the
simple anti-Hebbian LRA but appears to have damaged the task network’s performance
on each task in general (as indicated by lower accuracies along the diagonal of the
appropriate task matrix in Figure 6.7). Figuring out why the transfer phase between tasks
will be the subject of future work, but one reason for this damage might be due to the
deficiencies of the generative network which might become more pronounced when used
in a pure one-way transfer process.

Overall, these results present some promising directions to take in developing models
and algorithms for mitigating forgetting in neural systems. More importantly, there might be value in attempting to model other aspects of human/animal memory systems, specifically focusing on modules within a neural system such as the hippocampus instead of designing systems that broadly cover two or more different systems such as the hippocampus and the neocortex. This opens the door to alternative complementary systems and furthermore, this system shows that pseudopatterns are not necessarily needed to improve memory retention.

Figure 6.7. Test retention results on the four-task cumulative learning problem.

Table 6.6. Cumulative learning generalization metrics, based on the task performance matrices in Figure 6.4. For all metrics, higher values are better.
6.6 Conclusions

In this chapter, we investigated two key dimensions of the lifelong learning problem—
1) learning adaptively and incrementally from sequences and streams of inputs and 2) 
learning across a series of tasks. In doing so, we developed the Temporal Neural Coding 
Network, a model that heavily draws its inspiration from theories of predictive coding, 
which is a directed generative model learned with a form of Local Representation Align-
ment that can adapt to time-varying datapoints without back-propagation through time. 
We also investigated how neural models trained with Local Representation Alignment 
generalized in the more challenging domain of sequential task learning, quantitatively 
and qualitatively characterizing the catastrophic forgetting that occurs.

It is worth noting that the Temporal Neural Coding Network (TNCN) unites several 
strands of work that seek biologically-plausible alternative learning algorithms that 
generalize better to out-of-sample data. In this chapter, we connected these various 
research paths with the aim of tackling the difficult problem of learning representations 
of data in an unsupervised manner. With this target in mind, we argued for the use 
of higher-level objectives, a concept first described in Chapter 5 and even first crudely 
encountered in Chapter 3 which can be interpreted as local learning rules that still result 
in a globally coherent model.

Breaking free from the global feedback path required in back-propagation, as we do in 
this thesis, brings us closer to building models that are better suited for true unsupervised 
learning [291]. Unsupervised learning requires the computer to capture all possible 
dependencies between all observed variables, since inputs are no longer distinguished 
from outputs, as they are in supervised learning. It is this latter form of learning that is 
more closely related to how humans learn, where much of the incoming data does not 
come with labels (or, at best, comes with very few labels, as was the case in Chapter 
3). A successful unsupervised learning system is one that can discover all of the useful 
concepts and underlying causes to explain what it perceives [5] just as an infant must do, 
by observation alone.

More importantly, learning generic representations in an unsupervised system would 
the rather inflexible models created from the task-specific nature of 
supervised learning. Since downstream supervised/reinforcement learning approaches 
focus on task-specific measurements, the objectives used in unsupervised learning must 
then attempt to measure the quality of the generic representations acquired by the
generative models we train. Defining what is a good-quality general representation is itself an open problem and an active area of theoretical research [212, 213]. However, in lieu of the ideal metric, we took a small step towards higher-level objectives by literally interpreting this concept as a set of simple reconstruction loss terms that measure how well our neural architecture can predict a set of representation targets. Far better performance can be reached, we hypothesize, if better representation measurements and metrics can be developed.
Chapter 7  |  Conclusions

7.1 Summary of Contributions

The central goal of this thesis was to develop a family of algorithms, inspired and motivated by current neuro-cognitive insights and ideas on how the human brain works, that would prove useful in training connectionist systems, especially in the lifelong learning setting. Through developing new learning procedures, we were able to attack the problem of catastrophic forgetting, one primary reason why current artificial neural systems fail to generalize and retain previous task information in the face of sequences of tasks, by challenging the very algorithm that has become ubiquitous in training them—back-propagation of errors. Beyond its biological implausibility, back-propagation is plagued with constraints and problems that prohibit the types of architectures we can design that might be able to combat forgetting. Some of the things back-propagation-based models cannot use include highly nonlinear operations such as lateral competition and discrete-valued activations. Furthermore, training through back-propagation is highly sensitive to initialization conditions. Our coordinated local learning algorithms do not suffer such limitations, creating new possible directions to explore. These possible directions to take offer the promise of bringing us closer to better integrating concepts and mechanisms known in Cognitive and Neuroscience with modern, powerful artificial neural systems.

The first part of this thesis, Chapters 2-4, introduced the ideas of hybrid, or dual discriminative-generative learning, improved longer-term memory through fast and slow states, and globally-coordinated local learning. This early work laid the foundation necessary for the algorithms and architectures developed in the second part, Chapters
5 and 6, namely, those that embody the principle reducing mismatch between not only output predictions and data, but also within its own internally generated representations and targets that better describe the data from higher levels of abstraction, e.g., reducing discrepancy.

The central idea behind the second part is that we can create learning algorithms that do not depend on a global feedback pathway, which is central to back-propagation of errors. Motivated by the theory of predictive coding, we developed the Discrepancy Reduction family of learning algorithms, which, unlike back-propagation-based methods, does not limit the kinds of neural architectures we design. In short, we want to emulate the brain’s ability to do local learning, which circumvents many of the serious problems inherent to back-propagation of errors. The unifying idea of all algorithms that fall within this family is that a procedure must specify how to do the following two things:

1. Search for representations that better explain the environment, also known as target representations. This leads to the concept of higher-level objectives.

2. Directly minimize the mismatch between its current representations and these found target representations.

These two specifications facilitate the development of coordinated local learning rules. Several variants of an algorithm we call Local Representation Alignment were derived and a new, improved variation of target propagation was designed after viewing it as another case of Discrepancy Reduction.

When faced with time-varying data, even in the form of streams, we have shown how Local Representation Alignment can be adapted to learning temporal neural models without back-propagation through time. Specifically, the Temporal Neural Coding Network was developed, presenting how directed neural graphical model, strongly grounded in the structure of classical predictive coding models, can learn based on the additional principle of error correction. Though the results found so far in this dissertation are encouraging, we remark significant further research is to be conducted to fully explore the strengths and weaknesses of the Temporal Neural Coding Network and Local Representation Alignment.

Finally, when training neural systems across tasks, either in simple sequential learning or cumulative learning scenarios, this thesis showed that we can mitigate catastrophic forgetting by: 1) changing the learning procedure to a coordinated local approach (in tandem with more biologically-plausible update rules such as Hebbian learning), and 2)
by taking on a hybrid approach to learning–by simultaneously learning discriminative and generative models of the data, we form a simple variation of Complementary Learning Systems where a loosely-coupled generative model helps refresh the task network by learning to model statistical structure in not only the data that came before but also the task model itself. This leads to the Memory-Consolidation algorithm, a novel procedure that combines nearly all foundational concepts developed throughout this thesis to significantly minimize the forgetting of previously acquired task knowledge when learning something new.

7.2 Future Directions

There are many possible directions to take given the ideas and algorithms we have presented and developed.

7.2.1 Incorporation of Additional Neuro-Cognitive Mechanisms

One key way to proceed forward is to further synthesize key aspects of models of cognition (such as ACT-R [292] and SOAR [293]) and biological neuro-circuitry. By better incorporating theories of mind and brain functionality, predictive coding serving as the focus in this thesis, we might be able to create even better learning algorithms and model architectures that generalize better to unseen data and adapt to novel situations. This process works in both directions–in developing better intelligent systems by merging statistical learning and neuro-cognitive principles, we might further gain further insight into the workings of natural intelligence itself. Some immediately important concepts we intend to next focus on include designing better forms of lateral competition and exploitation of the properties of more biologically sound learning rules, such as Hebbian learning or Spike-Timing-Dependent-Plasticity, as used in artificial spiking neural networks (SNNs) [35]. Strong development along this direction might uncover even more efficient learning procedures for neural architectures, both on static and temporal problems/tasks. Another important space to explore is alternative activation functions, especially ones that are non-differentiable, since the algorithms in this thesis permit working with a broader class of operations that might offer powerful new ways for neural systems to generalize and even open up the pathway to discrete systems that better facilitate symbolic reasoning.
7.2.2 Large Scale Evaluation of Discrepancy Reduction

While the proposed algorithms that fall under Discrepancy Reduction have proven to be competitive on the benchmarks and simulations used in this thesis, far greater validation of their performance is needed, especially on datasets and tasks, both static and temporal, where the sample size is very large. Examination of the performance of the Temporal Neural Coding Network on a wider variety of discrete and continuous datasets is also very important in order to properly characterize its strengths and weaknesses and if it can truly serve as a proper replacement to networks trained with back-propagation through time. We hypothesize that models like the Temporal Neural Coding Network and their corresponding learning algorithms, e.g., Local Representation Alignment, will certainly not outperform RNNs trained with back-propagation through time across all problem instances. However, these will become particularly important in the realm of online learning, where data will never be available in the form of fixed datasets and the task will require quick, continual adaptation. Furthermore, evaluating adaptive temporal neural models over longer forecasting horizons, definitely beyond the next time step, will become increasingly important. One critical application of an adaptive generative model of temporal data would be in planning—where an agent, such as robotic agent, must first internally simulate possible futures in order consider possible consequences of its actions before making decisions to achieve its goals.

7.2.3 Memory & Large Scale Continual Learning

Though this thesis did develop various promising ways for improving information retention in neural systems, there is much room for improvement. Following in the spirit of the first proposed research direction, we believe that the key to building connectionist models with long-term memory is again to look to models of human cognition. The human memory system is far more complex and modular than the best-performing artificial neural models used in practice. By studying and computationally modeling what these various modules specialize in and how they interact with each other, the secret to capturing longer-term dependencies in sequence data and conducting truly general cumulative learning might be unlocked. One particularly important neuro-cognitive concept to integrate would be implicit, e.g. procedural and perceptual (as well as priming effects), versus explicit memory, e.g., episodic and semantic memory. Given the encouraging reduction of forgetting with the simple, proposed Memory-Consolidation algorithm, which
decomposes neural memory into a simple two-module system, e.g., complementary discriminative and complementary generative memory, it would be fruitful to investigate the benefits of building practical computational models of other known aspects of the human memory systems. Another interesting use of the Memory-Consolidation algorithm is to implement a phase of preplay, where the system “fantasizes” future, or even potentially nonsensical, class targets that could help the neural system learn how to not forget even before new tasks arrive. This mechanism would connect research in connectionist-based cumulative learning with what is currently understood on how sleep/dreaming affects human cognition and memory.

We have outlined several potential research directions based on what has been developed so far in this dissertation. In pursuing the above lines of inquiry, as well other currently unknown yet complementary directions, and solving the problems found in each, we believe that we will be able to build more intelligent machines that can aggregate knowledge and experiences across tasks much as humans do. Looking forward, if we can develop the algorithms and systems capable of conducting lifelong learning, key application domains such as robotics, where statistical learning struggles greatly, will benefit the most. Even greater challenges still lie ahead and yet more difficult questions remain to be answered. However, it is our hope that this dissertation can serve as a stepping stone to those who seek to develop general-purpose, adaptive agents in humanity’s quest for Artificial General Intelligence.
Bibliography


URL http://dl.acm.org/citation.cfm?id=104279.104286


URL http://doi.acm.org/10.1145/12130.12132


URL http://dl.acm.org/citation.cfm?id=104134.104148


208


URL http://dx.doi.org/10.1162/neco.1997.9.8.1735


Vita
Alexander G. Ororbia II

Education
The Pennsylvania State University, University Park, PA, USA
Ph.D., Informatics (Graduate minor in Social Data Analytics), 2018
Bucknell University, Lewisburg, PA, USA
B.S., Computer Science & Engineering (Minors: Mathematics, Philosophy), 2013

Awards and Honors
Circle of Ph.D. Research Distinction, IST Graduate Award, 2018
IST Graduate Travel Award, 2015
The Alfred P. Sloan Scholarship, 2014
NSF IGERT Fellowship, 2014
Bunton-Waller Graduate Fellowship, 2013
Jordan Rednor Fellowship, 2013

Selected Publications
• Iulian Serban*, Alexander G. Ororbia II*, Joelle Pineau, and Aaron Courville. “Piecewise Latent Variables for Neural Variational Text Processing”. In: Proceedings of Empirical Methods in Natural Language Processing, Copenhagen, Denmark (pp. 422-432). (* First two authors contributed equally.)