The Pennsylvania State University
The Graduate School
College of Engineering

AN EXPERIMENTAL ANALYSIS OF PREDICTIVE CODING BASED ON ARTIFICIAL NEURAL NETWORKS FOR IMAGE DECODING

A Thesis in
Computer Science and Engineering
by
Yanbo Sun

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2018

The thesis of Yanbo Sun was reviewed and approved* by the following:

C. Lee Giles
Professor of Computer Science and Engineering
Thesis Adviser

Daniel Kifer
Professor of Computer Science and Engineering

Chita Das
Professor of Computer Science and Engineering
Department Head

*Signatures are on file in the Graduate School.

Abstract

Prediction techniques have been extensively studied for images and video compression over the last few decades. Currently, most existing images and video compression solutions contain some predictive encoding in their algorithms. For example, it is well known that the 25-year history of the joint image group (JPEG) image coding standard predictive encodes quantified DC transform coefficients. There some other sorts of predictive coding techniques which implement redundancies inside images and videos. Similarly, in some particular cases such as image sequences, the compression system usually uses the motion compensation prediction to utilize the time redundancy. We discuss some of the most essential predictive coding techniques used in state-of-the-art image and video encoders. Then we provide some basic concepts about video color spaces representation. Various technologies including linear and non-linear predictions methods are presented in the following part.

In this thesis, we have implemented and presented an Artificial Neural Network Approach for the encoding of JPEG images. We have shown that JPEG compression can be significantly improved with our decoding method. In these experiments, we have implemented MLP and RNN models on tiny-imagenet data sets. We performed stateless and stateful prediction without using spatial information as well as predictions using partial spatial information and full spatial information. We use the partial and full spatial redundancy in the image to implement a nonlinear mapping function. Compared with the original image, the proposed method outperforms the JPEG image in generating predictive images with relatively small distortion.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my deepest gratitude to Dr. Giles and Dr. Kifer who provided helpful comments and suggestions. I am also indebt to Alexander Ororbia, Ankur Mali, whose comments made enormous contribution to my work. I would also like to express my gratitude to my family for their moral support and warm encouragements.

Chapter 1 | Introduction

1.1 Introduction

Compression is used to generate a compact signal compared to original data. When the signal is defined as an image, video or audio signal, the problem of compression is to minimize its bit rate. Obviously, most applications are not feasible without compression.

In many previous research work in image compression, they tried to design a whole system for image compression including encoding and decoding part, which make it challenging for neural network based system to achieve. Besides, it is also significant in most image compression to work on the prediction with spatial dependencies and make use of causal and non-causal statistical redundancies. However, in the state-of-the-art compression standards, prediction or predictive coding involve linear prediction. Deep Neural Networks, on the other hand, are good at providing non-linear mappings, which make it perform better in prediction. In this thesis, we are aiming at providing non-linear decoders for JPEG encoded data and they are able to reduce the information loss from JPEG.

In order to measure how well our methods are working, we use four common metrics in image processing for evaluation purpose. They are MSE(Mean Squared Error), PSNR(Peak Signal to Noise Ratio), SSIM(Structural Similarity) and MSSSIM(Multi-scale Structural Similarity). We measure each reconstructed images with these four metrics and compare them with the JPEG baselines.

1.2 Lossless and Lossy Compression

Lossless compression and lossy compression are features that explain the difference between the two file compression formats. Lossless allows all raw data to be recovered when the file is uncompressed. Lossy way of working is different because it eliminates all unnecessary information in the original file, making it even smaller when compressed. The question that some people often ask is, How can I make a choice between the two? The answer to this question depends very much on several factors. One is the size and type of file to be compressed or converted. Second, the amount of space available and the amount of space you want to save. Lossless compression and lossy compression are terms that describe whether or not all the original data can be recovered when the file is uncompressed when the file is compressed. With lossless compression, all the first data in the file is preserved after the file has not been compressed. All the information is completely restored. This is usually a text or spreadsheet file selection technique, in which loss of words or financial data can cause problems. Graphics Interchange File (GIF) is an image format used on the Web that provides lossless compression. [1]

Although lossy file compression is often associated with image files, it can also be used for audio files. Files such as mp3, mp4, AAC files, or images like JPEG format are usually compressed into a lossy format. Whenever you choose to use a lossy format zip file, many of the redundant information in the file is eliminated. Therefore, whenever people reduce the size of bitmap images or other files, lossy compression is used. This does not mean that lossy file compression does not work well when reducing and saving files from their original state. In fact, the original image has been compressed using a lossy format that will retain about 80% of its original mass. On the other hand, lossy compression reduces files by permanently eliminating certain information, especially redundant information. When the file is uncompressed, only a portion of the original message still exists although the user may not have noticed. Lossy compression is often used for video and sound, in which case most users do not detect a certain amount of information loss. JPEG image files, commonly used for photos and other complex still images, are a lossy image. With JPEG compression, the creator can decide how much to introduce and how to trade-off between file size and image quality. [2]

Unlike lossy file compression, using a lossless format can eventually reduce the size of the file without losing its original quality. Now with the new compression algorithm, lossless compressed files are saved better than before. Compressing files using lossless compression will rewrite the data in the same way as the original file. However, audio and image files compressed using lossless compression tend to be larger in size than lossy. [3]

1.3 Image Compression Standards

The purpose of image compression is to represent images with less data in order to save storage costs or transmission time. Without compression, file size is significantly larger, usually several megabytes, but with compression it is possible to reduce file size to 10 percent from the original without noticeable loss in quality [4]. As we discussed in the previous part, image compression can also be divided into two main parts: lossless and lossy.

Lossless compression is preferred for archival purposes and is often used in medical imaging, technical drawings, clipboards or comics. Lossy compression methods, especially when used at low bit rates, introduce compression distortion. Lossy compression produces compression artifacts on the image, especially when the sharp edges are blurred by intense compression. For natural images such as landscape photos, lossy compression is a good option because a slight loss of sharpness is acceptable, resulting in smaller files. Lossy methods are especially useful for natural images, where applications may experience small, sometimes undetectable loss of fidelity, resulting in a significant reduction in bit rates. Lossy compression with negligible difference can be visually lossless. [5]

The method of lossless image compression includes: 1. Run length encoding 2. Area image compression 3. DPCM and predictive coding 4. Entropy coding 5. Adaptive dictionary algorithms such as LZW 6. DEFLATE for PNG, MNG and TIFF 7. Chain code

The research work of this thesis will focus on the predictive coding part. RLE represents only a long sequence of the same data in a shorter form. Entropy coding allocates symbols to symbols so that the most frequently occurring symbols use the shortest code. The most common entropy coding technique is Huffman coding. The dictionary encoder sets up a string table and replaces them with shorter code. The LZW (Lempel- ziv-welch) algorithm is probably the most commonly used dictionary encoder for use in formats such as GIF and ZIP files. [5–7]

The lossy compression method is as follows: [5]

1. Reduce the color space to the most common color in the image. The selected color is specified in the palette of compressed image titles. Each pixel references only the color index in the palette. This method can be combined with dithering to avoid posterization problems.

2. Chroma sampling. This takes advantage of the fact that the human eye perceives the spatial change in luminance faster than the spatial change in color by averaging or discarding some color information in the image.

3. Transform coding. Fourier correlation transforms such as discrete cosine transform (DCT) are widely used. In the case of a series of discrete cosine transforms, DCT is sometimes called "DCT-II". Recently developed wavelet transform is also widely used, followed by quantization and entropy coding.

Lossy compression is usually based on the removal of details that the human eye would not normally notice. When a pixel is only slightly different from its neighbor, its value can be replaced. This can lose some information, but if the algorithm is good enough, it usually attracts little attention from the human eye. Then it usually uses RLE or Huffman encoding to compress the data. The two main parts of lossy compression method is based on the fact of the human eye more sensitive to brightness than color, so can store more than the color details of luminance details to optimize the file size. Fractal compression is also used, but not universally. [8]

Next we are going to introduce some popular image formats. There are various image formats to implement these compression methods. Some use lossless compression only, some use lossless and lossy methods or do not compress at all. The most common image formats are BMP, JPEG, GIF, PNG and TIFF. [9]

BMP is an uncompressed image format used in Windows and takes up a lot of space. JPEG is the most widely used image format, using lossy compression methods such as DCT and chroma sampling. It also uses lossless methods such as RLE and Huffman coding, but it does not actually allow for lossless storage. JPEG is especially good for natural images. [10]

GIF USES lossless compression algorithm LZW. Therefore, it can reduce the file size without reducing the visual quality. But GIFs only allow 256 colors, so they are not suitable for natural photos made up of millions of colors. PNG also uses lossless compression called DEFLATE, which implements a combination of the

LZ77 dictionary encoder and Huffman encoding. PNG provides better compression and more functionalities than GIF, but it does not support GIF animation. [11, 12]

TIFF format is a flexible and adaptable file format. It can store multiple images in a single file. TIFF can be used as a container for JPEG, or can be chosen to use lossless compression, such as RLE or LZW. Because TIFF supports multiple images in a single file, multi-page documents can be saved as a single TIFF file rather than a series of files per scanned page. [13]

A new version of JPEG, called JPEG2000, also supports lossless compression. This is lossy wavelet, which produces better image quality than JPEG, but is not widely used due to patent issues. Microsoft has also claimed a "next-generation" image format called HD Photo, formerly known as Windows Media Photo. [14]

There are also SVG graphics for creating vector graphics. Unlike raster graphics (JPEG, GIF, PNG), which specify the color of each pixel, SVG uses mathematical expressions to specify the coordinates of the shape. [15]

Another important aspect we introduce is the image compression by neural network. In recent years, there is an increasing interest in the use of neural networks on image compression. Up to now, some work of image compression in terms of neural are accomplished. Theoretically, neural networks provides a complex nonlinear mappings between input and output. Neural networks enable better performance of different compression techniques such as object-based, model based and fractal coding techniques.

Chapter 2 Related Work

2.1 Predictive Coding Technologies

Predictive Coding is the approach that achieves good compression without significant overload. It is based on eliminating inter pixel redundancies by extracting and coding only the new information in pixels of closely spaced pixels. It is a compression technique based on the difference between the original and predicted values.

Predictive Coding techniques are widely investigated in the last decades for image and video compression. Most of the existing image and video compression involve some kinds of predictive coding in their algorithms. JPEG, specifically, predictively encodes the quantized DC transform coefficients. Other kinds of video compression incorporate motion compensated prediction to exploit temporal redundancies.

2.1.1 Methods of Prediction in Image and Video Coding

The JPEG committee of the International Standards Organization (ISO) issued a call in 1994 for compression of non-destructive and near-lossless compression of sustained tone pictures. This requirement has led to an increased interest in non-destructive image compression schemes, especially the construction of a prediction block from the neighbors of the current block, which is defined by the pattern. The "LIP" scheme takes advantage of the spatial correlation that may exist between the predicted and actual pixels in the target block. Prediction is also widely used in several video codecs. Much work has been done to improve the efficiency of H.264 / AVC internal prediction. Conklin [16] proposed expansion of encoded pixel lines to include more pixel samples for prediction.

Next we are going to discuss the whole process of state-of-the-art compression. Traditional Lossy Image and Video Compression like JPEG usually includes these typical algorithms: quantization, Discrete Cosine Transform(DCT), entropy coding. The detailed steps of JPEG are as the following [17]:

1. An RGB to YCbCr color space conversion 2. Original image is divided into blocks of 8 x 8. 3. The pixel values within each block range from[-128 to 127] but pixel values of a black and white image range from [0-255] so, each block is shifted from[0-255] to [-128 to 127]. 4. The DCT works from left to right, top to bottom thereby it is applied to each block. 5. Each block is compressed through quantization. 6. Quantized matrix is entropy encoded. 7. Compressed image is reconstructed through reverse process. This process uses the inverse Discrete Cosine Transform (IDCT).

As for the video compression, in general, videos consist of a sequence of frames. There are mainly two types of predictions-one is intra-prediction, another is inter-prediction. Inter-prediction is usually called motion compensation, which is an essential technique for reducing temporal redundancies.

Yu and Chrysafis [18] posted an idea which is further extended by introducing a concept similar to motion compensation in which a displacement vector measures the distance between a reference and a target block in the same frame. This method is called shifting the internal prediction (DIP). As revealed by the experimental results [18], DIP is scene/complexity related. Due to the spatial correlation allowing for better internal prediction results, Shiodera et al. [19] proposed changing the coding order of sub-blocks, ie. , The sub-block is first encoded to the right bottom position. In [20, 21] introduced a template matching technology to improve the coding efficiency of the texture surface.

Due to the hierarchical structure of H.264/SVC, its internal prediction is different for the base and enhancement layer prediction. The "LIP" in H.264/AVC is still the base layer prediction of H.264/SVC. For enhancement layers, additional modes are used to improve the coding efficiency through the layer association. I BL mode, copying 7 co-located blocks in the base layer of upsample (using an effective upsampling filter [22]) as one of the prediction of the target block in the enhancement layer, referred to as H.264 / SVC Inter-layer prediction. Park et al. [23] observed the global/local phase shift between the substrate and the enhancement layer and claimed that the up-sampling filter could produce output samples located at different positions compared to the input sample. Therefore, frame-based (or block-based) shift parameters can be used to improve coding efficiency. Wong et al. [24] proposed to add the correct and downward pixel lines in the grassroots to improve the prediction performance. Multi-Hypothesized Motion Compensated Prediction (MH-MCP) was proposed in [25], further utilizing long-term prediction and multi-parameter reference frame structure. Theoretical studies carried out by MH-MCP in [26] have shown the potential to further improve the prediction. MH-MCP uses the inter-frame temporal correlation to predict. However, the coefficient is usually fixed in each prediction due to the high complexity of the optimization coefficient and the prediction signal at the same time.

In standards like JPEG 2000, each frame is coded individually, which allows for random access and complexity reduction. While in other standards like MPEG, inter-frame coding is introduced to improve compression efficiency. Motion compensation represents a picture related to the transformation of a reference frame to the current frame. The reference picture may be previous in time or even from the future. If images can be precisely synthesized from previously transmitted or stored images, the compression efficiency can be enhanced. [27]

As for the MPEG, there are three kinds of coded frames: I frames, P frames and B frames. "I" frames, which represent for intra frames are key frames; "P" frames, which represent for predicted frames, are predicted from the most recently reconstructed I or P frame. B or bidirectional frames are predicted from the neighbor I or P frames. Each macroblock in a P frame can either come with a vector and difference DCT coefficients for a close match in the last I or P, or it can just be intra coded if there was no good match. During the process, the encoder looks for the matching blocks among frames, and tries three different things to determine the best: the forward vector, the backward vector, the average of the two blocks from the past and future frames and subtraction of the result from the block being coded.

A typical sequence of decoded frames might be: "IBBPBBPBBPBBI", which means 2 "B" frames separating each "P" frames, and "I" frames with 12 frames of distance. The video decoder recovers the video by decoding the data stream frame by frame. Decoding starts with an I-frame, which is decoded independently. However, P and B frames should be decoded together with current reference image.

An I-frame, or intra frame, is a self-contained frame that can be independently decoded without any reference to other images.

The first image in a video sequence is always an I-frame. I-frames are needed as starting points for new viewers or resynchronization points if the transmitted bit stream is damaged. I-frames can be used to implement fast-forward, rewind and other random access functions. An encoder will automatically insert I-frames at regular intervals or on demand if new clients are expected to join in viewing a stream. The drawback of I-frames is that they consume much more bits, but on the other hand, they do not generate many artifacts, which are caused by missing data.

A P-frame, which stands for predictive inter frame, makes references to parts of earlier I and/or P frame(s) to code the frame. P-frames usually require fewer bits than I-frames, but a drawback is that they are very sensitive to transmission errors because of the complex dependency on earlier P and/or I frames. A B-frame, or bi-predictive interframe, is a frame that makes references to both an earlier reference frame and a future frame. Using B-frames increases the latency.

The generally accepted method for temporal prediction is motion compensation using block matching. Though block matching has a good overall performance, it also has several drawbacks. Block matching can only predict purely translational motion, and fails, at least to some extent, when motion has rotational components, or when the scale or the shape of objects changes (due to, e.g., zooming). As a result, the block structure is clearly visible in the prediction image. Several attempts have been made to overcome these problems, but the complexity of the proposed improvements has often prevented their wide-spread use. The conventional block based motion compensation also has the advantage that it is well suited to be used together with block based transform coding methods, such as the discrete cosine transform (DCT). The motion compensation block borders generally coincide with the borders of the DCT blocks, and, therefore, they do not result in additional high frequency components to be transmitted. The transmission of the motion vectors can also be effectively done in connection with the transmission of the DCT blocks. If a new method for describing the motion is used, the amount of overhead information can easily increase. [28]

Another important method for video compression is the Discrete Cosine Transform (DCT), which is a widely used transformation in image compression. The JPEG still image compression standard, the H.261 (p*64) video-conferencing standard, the H.263 video-conferencing standard and the MPEG (MPEG-1, MPEG-2, and MPEG-4) digital video standards use the DCT. In these standards, a two-dimensional (2D) DCT is applied to 8 by 8 blocks of pixels in the image that is compressed. DCT enables images to be represented in frequency domain rather than time domain given the fact that images can be represented in the frequency domain using less information than in the time domain. The DCT takes each block consisting of a 64-point discrete signal and breaks it into 64 basis signals. The output of the operation is a set of 64 basis-signal amplitudes, which is called DCT coefficients. These coefficients are unique for each input signal. The DCT provides a basis for compression because most of the coefficients for a block will be zero (or close to zero) and do not need to be encoded. There are various versions of DCT, usually known as DCT-I to DCT-IV. The most popular is the DCT-II, also known as even symmetric DCT, or as âĂIJthe DCTâĂİ.

2.1.2 Prediction in Image Coding

In image compression scenarios, the prediction process includes statistical estimates of future random variables of past and present observable random variables and the prediction of image pixels or a group of pixels may come from previously transmitted pixels. The predicted pixel is subtracted from the original pixel to obtain a residual signal. In general, if the energy of the remaining signal is lower than the energy of the original signal, the prediction is successful. Therefore, the remaining pixels can be encoded more efficiently by the entropy encoding algorithm, and the number of transmitted bits is smaller [28].

Predictive coding has proven to be an effective technique and is widely used in lossless and lossy image coding. A compression method commonly used to describe predictive coding is called Differential Pulse Code Modulation (DPCM) [29]. This method does not pass pixel values directly, but instead passes the difference between pixel and prediction. The main motivation for DPCM is that most source signals, such as image and video signals, exhibit high correlation between adjacent pixels.

The basic principle of DPCM is to predict the current pixel from the previous pixel or multiple pixels and to encode the prediction error or resulting from the difference between the original and predicted pixels. The compression performance of DPCM depends on the accuracy of the used prediction technique. In DPCM, the prediction is based on the linear combination of the previously decoded pixels, which is known as linear prediction. Since the coding procedure is done in raster scan order, both encoder and decoder can use the pixels X'(n-1), X'(n-2), X'(n-3) and X'(n-4) which are available in current and previous pixel rows, in order to generate the prediction for the current pixel, X'(n) [28]. The decoded (or reconstructed) pixels should be used instead of the original ones, in both encoder and decoder sides, because the decoded pixels may differ from the original pixels due to the quantisation step. Since the original pixels are not available in the decoder side, the use of decoded pixels guarantees that the prediction signal is identical in both encoder and decoder sides. Otherwise, if the encoder used the original pixels for prediction, a cumulative mismatching error could be observed between the decoded pixels at the encoder and decoder, leading to an increased distortion in the decoder reconstructed image [30].

Linear Predictive Coding (LPC) techniques have been extensively studied in literature and successfully used in image, speech and audio processing applications. For the case of image compression, several algorithms based on DPCM, using an adaptive approach of LPC have been presented since the early days of digital image compression.

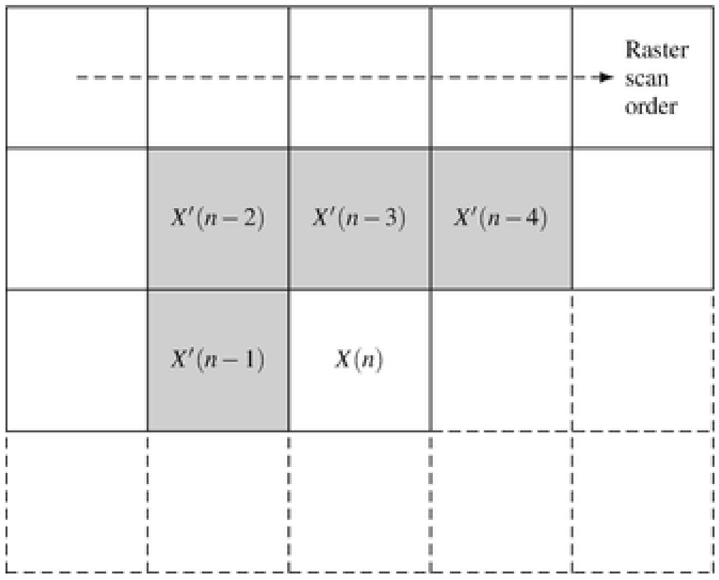

Figure 2.1. Spatial prediction in a raster scan order encoder [30]

The above diagram from [30] shows the principle of causal blocks, which is similarly used in the following neural network based algorithm for image decoding.

Although DPCM compression is commonly associated to spatial or intra-frame coding, it can be designed to exploit inter-frame similarities, such as temporal redundancy in the case of video signals. While intra-frame coding usually uses the causal neighbouring pixels belonging to the same frame, the inter-frame coding can be conducted using the causal pixels belonging to another frame [30].

2.1.3 State-of-the-Art Prediction Methods

Pixel-based compression schemes using adaptive linear predictive coding techniques have also been successfully applied in lossless image compression. For example, adaptive prediction can be achieved by testing a set of different predictors in the encoder and signaling the chosen predictor to the decoder. The prediction methods play an important role in image and video coding standards, due to their ability to reduce the signal redundancy based on the previously encoded samples. With the evolution of compression standards, more complex and sophisticated predictive coding techniques have been investigated and adopted [28]. These standards includes H.264/AVC and H.265/HEVC [31] [32]. Although these algorithms have been primarily conceived for video compression applications, they also present efficient results for still image coding. In the image coding scenario, only intra-prediction methods can be used, which is the primarily method of our work.

2.1.4 Methodology of The Non-linear Decoding in Predictive Coding

In state-of-the-art compression standards, most of the prediction algorithms use linear prediction as we mentioned before. However, it is not enough to only provide linear representations for image blocks. Machine Learning and Artificial Neural Network can better perform non-linear mapping and regression of image blocks. There have been several Artificial Neural Network methods that try to construct the whole system of image compression similar to JPEG. In this thesis, instead of building the whole system, we propose the implementation of a decoding approach that fully exploit statistical dependencies within images. The proposed research exploits naive predictive coding of one-to-one block as well as causal and non-casual blocks which take advantages of spatially statistical dependencies. We are able to achieve a higher decoding accuracy with the encoded JPEG images.

Designing a full neural network compression systems requires both encoding and decoding as well as quantization mapping functions which are efficient and accurate. Since the quantization requires discrete functions, it is hard to train the network with back-propagation based methods.

Therefore, instead of design a whole system, we focus on the decoding part, aiming to construct a optimized non-linear decoder without approximating quantization and make full use of the existing encoders and quantization steps included in JPEG. This can also help us reduce distortion between compressed and original images. The goal of this thesis is to present and discuss the predictive coding functions developed in the research and its implementation in JPEG images.

2.1.5 Implementation of Artificial Neural Networks in Predictive Coding

Traditional image compression algorithms have been widely used but there can be some improvements. In traditional image compression, predictive coding is usually conducted by linear representation. In contrast, machine learning algorithms can better performance non-linear combinations of features and to automatically capture latent statistical structure. The last decade has witnessed a proliferation of image compression methods using neural networks [33] [34] [35].

Using the Outdoor MIT Places dataset, JPEG 2000 and JPEG are able to acheieve 30 and 29 times compression respectively. Toderici et al. [36] proposed an RNN-based technique which is able to achieve better PSNR compared to JPEG and JPEG 2000 with just a quarter the bitrate. Van den Oord et al. [37] gives a deep neural network that predicts image pixels which are achieved by two-dimensional recurrent layers. While autoencoders are used to reduced the dimensionality of images, another representation learning framework of autoencoder is also proposed for compression. Toderici et al. gives the an appealing architecture of convolutional and deconvolutional LSTM models [38]. They also proposed several RNN-based encoder and decoder and an ANN for encoding binary representations. It can be seen as the first ANN architecture that performs better than JPEG in image compression. Theis et al. [39] proposed autoencoder compression which uses a discrete rounding function and smooth approximation of the upper bound of the discrete entropy loss to achieve continuous relaxation. A CNN-based compression framework is proposed by Jiang et al. to achieve high quality image compression at low bit rates [40].

In addition, Liang et al. added duplicate connections at each layer of the feed-forward CNN so that the activities of each unit can be adjusted by the activities of neighboring units in the same layer [41]. Stollenga et al. added feedback links to the trained CNN in order to implement the attention selection filter for the model for better object classification [42]. Balle et al. used generalized split normalization (GDN) to achieve joint non-linearity and implemented continuous relaxation using additive uniform noise instead of rounding quantization [43]. Canziani et al. used the feedforward discriminant subnetwork and the feedback generation subnet to establish a bidirectional model and connected the two subnetworks through a horizontal connection; training the video predictive model allowed the model to obtain more stable objects for a given video input identification [44]. These studies highlight the role of feedback and recurrent processes in computing or learning better representations, rather than just models with feedforward processes. Li et al. proposed a weighted content compression method based on image importance maps [45]. For lossless compression, the method proposed by Theis et al. And van den Oord and others made gratifying results.

2.2 Introduction of Image Quality Metrics

The simplest and most widely used quality metric is the mean squared error (MSE). The mean squared error (MSE) is calculated from the average distortion and the squared intensity difference of the reference image pixels, along with the associated peak signal to noise ratio (PSNR). These factors are attractive because they are simple to calculate and have a clear physical meaning, and are mathematically convenient in an optimized environment. But they don't quite match the perceived visual quality. Blow are definition and equation of MSE and PSNR.

Mean Square Error (MSE), MSE is computed by averaging the squared intensity of the original image and the resultant image pixels as (1.4):

$$MSE = \frac{1}{NM} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} e(m,n)^2 \tag{2.1}$$

Where e(m, n) is the error difference between the original and the distorted images.

Peak Signal-to-Noise Ratio (PSNR), SignalâĂŞto-noise ratio (SNR) is a mathematical measure of image quality based on the pixel difference between two images [46]. The SNR measure is an estimate of quality of reconstructed image compared with original image. PSNR is defined as (1.5):

$$PSNR = 10 * log \frac{s^2}{MSE} \tag{2.2}$$

The quality-estimated image signal can be considered as the sum of the undistorted reference signal and the error signal. One widely used assumption is that the loss of perceived quality is directly related to the visibility of the error signal. The simplest implementation of this concept is MSE, which objectively quantifies the strength of the error signal. But two distorted images with the same MSE may have very different types of errors, some of which are more pronounced than others.

The Structural Similarity Index (SSIM) is a perceptual metric that quantifies image quality degradation* caused by processing such as data compression or by losses in data transmission. It is a full reference metric that requires two images from the same image captureâĂŤ a reference image and a processed image. The processed image is typically compressed. It may be obtained by saving a reference image as a JPEG then reading it back in. SSIM is best known in the video industry, but has strong applications for still photography [47]. SSIM actually measures perceptual differences between two similar images. It cannot determine which of the two images is better, which must be inferred by knowing which is raw and which has undergone additional processing, such as data compression.

The calculation of SSIM is given by the following equation. The SSIM index is calculated on various windows of an image. The measure between two windows x and y of common size N x N [48]:

$$\mathrm{SSIM}(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{2.3}$$

Where μ_x is the average of x, μ_y is the average of y, σ_x^2 is the variance of y, σ_{xy}

is the covariance of x and y. $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$ are two variables to stabilize the division with weak denominator. L is the dynamic range of pixel values, which is 255 in our case. $k_1 = 0.01$ and $k_2 = 0.03$ are default.

In practice, one usually requires a single overall quality measure of the entire image. We use a mean SSIM (MSSIM) index to evaluate the overall image quality. The MS-SSIM is given by:

$$MSSIM = \sum_{M}^{j=1} W_j \cdot SSIM(x_j, y_j) \tag{2.4}$$

Where M is the number of samples in the quality map, $\sum_{M}^{j=1} W_j = 1$; x_j and y_j are the reference and the distorted images, respectively; and are the image contents at the th local window; and M is the number of local windows of the image.

Many image quality assessment algorithms have been shown to behave consistently when applied to distorted images created from the same original image, using the same type of distortions (e.g., JPEG compression). However, the effectiveness of these models degrades significantly when applied to a set of images originating from different reference images, and/or including a variety of different types of distortions [49].

Chapter 3 Introduction and Discussion of Artificial Neural Network

3.1 Multiple Layer Perceptron

3.1.1 Gradient Descent

Gradient descent is a first order iterative optimization algorithm for minimizing functions. To find a local minimum of a function using gradient descent, it is necessary for us to take the step of proportional to the negative gradient of a function at the current point. If we take steps that are proportional to the positive direction of the gradient, we are closing to the local maximum of the function, which is called gradient rise.

Gradient descent requires a cost function. We need this cost function because we want to minimize it. Minimizing any function means finding the deepest valley in the function. Remember that the cost function is used to monitor errors in the ML model predictions. So minimizing this basically means getting as low an error as possible or increasing the accuracy of the model. In short, we improve the accuracy by iterating the training data set while adjusting the model parameters (weight and deviation).

The core of the algorithm is to achieve the minimum error value. We need to adjust the parameters of the model in order to find the smallest error (the deepest valley) in the cost function (about a weight). Using calculus, we know that the slope of a function is the derivative of a function with respect to a value. The slope always points to the nearest valley.

Here are some fundamental concepts for Gradient Descent:

1. Learning rate: step length determines the length of each step in the negative direction of the gradient in the process of gradient descent iteration. In the example above, the step length is the length of the step along the steepest and easiest descent at the current position of the step.

2. Feature: refers to the input part of the sample, such as sample (x_0,y_0), (x_1,y_1), then sample feature is x and sample output is y.

3. Hypothesis function: In supervised learning, the hypothesis function used to fit the input sample is denoted as $h_\theta(x)$. For example, for the sample (x_i,y_i) (I =1,2... N), the fitting function can be adopted as follows:

4. Loss function: In order to evaluate the goodness of model fitting, loss function is usually used to measure the degree of fitting. The minimization of loss function means that the fitting degree is the best, and the corresponding model parameter is the optimal parameter. In linear regression, the loss function is usually squared between the sample output and the assumed function. For example, for the sample (x_i,y_i) (I =1,2... N), using linear regression, the loss function is:

$$J(\theta_0, \theta_1) = \sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 \tag{3.1}$$

Where x_i represents the i th element of sample feature x, y_i represents the i th element of sample output y, and $h_\theta(x_i)$ represents the hypothesis function.

As for more specific scenario, the following formula shows the gradient computation for linear regression:

$$\frac{\partial}{\partial_j} MSE(\theta) = \frac{2}{m}\sum_{i=1}^{m}(\theta^T \cdot x^{(i)} - y^{(i)})x_{(j)}^{i} \tag{3.2}$$

Here from the Figure 4.1, we know the relationship between the graph of the cost function and the weight. Now, if we calculate the slope of the cost function of this weight, we get the direction we need to get to the local minimum (the nearest deepest valley). This model has only one weight.

Detailed descriptions for Gradient Descent algorithm are:

1. Prerequisites: To confirm the hypothesis function and loss function of the optimization model.

For example, for linear regression, let's assume that the function is written as $h_\theta(x_1, x_2, ...x_n) = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n$, and that the θ_i(i=0,1,2...,n) is the model

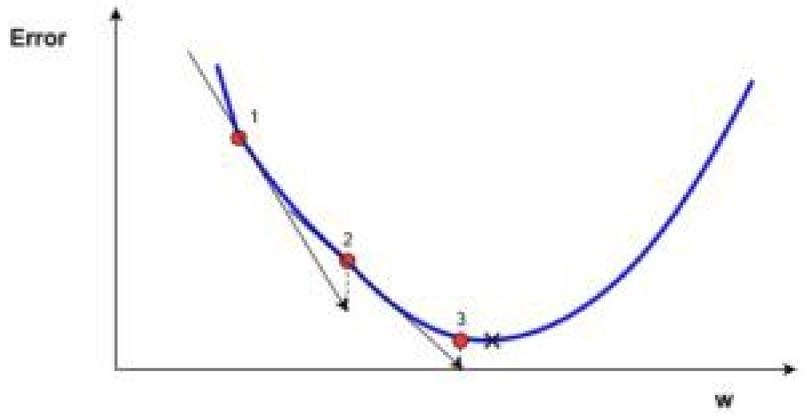

Figure 3.1. Cost function against one weight

parameter, x_i (i = 0,1,2...,n) is n eigenvalues of each sample. This representation can be simplified, so let's add a feature $x_0 = 1$, such that: $h_\theta(x_0, x_1, ...x_n) = \sum\limits_{i=0}^{n} \theta_i x_i$

Also for linear regression, corresponding to the hypothesis function above, the loss function is:

$$J(\theta_0, \theta_1 ..., \theta_n) = \frac{1}{2m} \sum_{j=0}^{m} (h_\theta(x_0^{(j)}, x_1^{(j)}, ...x_n^{(j)}) - y_j)^2 \tag{3.3}$$

2. Initializing the relevant parameters of the algorithm: it is mainly initializing $\theta_0, \theta_1 ..., \theta_n$, the termination distance ε as well as the step length α. Let us initialize the θ value to be 0 and α to be 1.

3. Algorithm process:

1) Determine the gradient of the loss function at the current position, for θ, the expression of the gradient is as follows: $\frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1 ..., \theta_n)$

2) Multiply the step length by the gradient of the loss function, and get the drop distance of the current position, i.e. $\alpha \frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1 ..., \theta_n)$ corresponding to a step in the previous example of mountaineering.

3) Determine whether all of the input θ_i and the gradient descent distance are less than ϵ, and if it is less than ϵ, the algorithm will be terminated. θ_i(i=0,1,2,...,n) is the final result. Otherwise, step 4.

4) To update all of them, and the updated expression of θ_i is as follows. Move on to step 1 after the update is complete. $\frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1 ..., \theta_n)$

The following is an example of linear regression to describe gradient descent. Let's say our sample is $(x_1^{(0)}, x_2^{(0)}, ...x_n^{(0)}, y_0), (x_1^{(1)}, x_2^{(1)}, ...x_n^{(1)}, y_1), ...(x_1^{(m)}, x_2^{(m)}, ...x_n^{(m)}, y_m)$, loss function as mentioned in the previous prerequisite is :

$$(x_1^{(0)}, x_2^{(0)}, ...x_n^{(0)}, y_0), (x_1^{(1)}, x_2^{(1)}, ...x_n^{(1)}, y_1), ...(x_1^{(m)}, x_2^{(m)}, ...x_n^{(m)}, y_m) \tag{3.4}$$

In step 1 of the algorithm process, the partial derivative of θ_i is calculated as follows: $\frac{\partial}{\partial\theta_i} J(\theta_0, \theta_1..., \theta_n) = \frac{1}{m}\sum\limits_{j=0}^{m}(h_\theta(x_0^{(j)}, x_1^{(j)}, ...x_n^{(j)}) - y_j)x_i^{(j)}$

Because there is no x_0 in the sample we let all x_0^j be 1.

The updated expression of θ_i in step 4 is as follows: $\theta_i = \theta_i - \alpha\frac{1}{m}\sum\limits_{j=0}^{m}(h_\theta(x_0^{(j)}, x_1^{(j)}, ...x_n^{j}) - y_j)x_i^{(j)}$

When gradient descent is used, tuning is required. There are aspects for appropriate tunings:

1. Algorithm step size selection. In real scenarios, step values depends on the data sample, we should take some more value, from big to small, running algorithm respectively, and see the iterative effect, if the loss function in small, effective value, or to increase the step length. If the step size is too large, the iteration will be too fast, and even the optimal solution may be missed. If the step length is too small, the iteration speed is too slow, and the algorithm cannot be finished for a long time. Therefore, the step length of the algorithm needs to be run many times before it can get a better value.

2. Initial value selection of algorithm parameters. The initial values differ and the minimum values obtained are also likely to be different, so the gradient descent might be obtained with only local minimum values. Of course, if the loss function is convex, it must be the optimal solution. Due to the risk of local optimal solution, the algorithm needs to be run with different initial values several times. The minimum value of the key loss function is selected, and the initial value of the minimum loss function is selected.

3. Normalization. Due to the different characteristics of the sample is not the same as the scope, it may result in the slow iterations. In order to reduce the influence of characteristic value. We should normalize the characteristic data. For each feature x, we are able to get the expectation $\overline{x}$ and the standard deviation std(x), then is transformed into: $\frac{x-\overline{x}}{std(x)}$. The new expectation of this feature is 0, the new variance is 1, and the time of iterations can be greatly accelerated.

In practical, there are three types of gradient descent algorithms, including batch gradient descent, stochastic gradient descent and Mini-batch gradient descent. Batch gradient descent computes the gradient of the cost function to θ for the entire training dataset. Since we need to calculate the gradient of the entire dataset to perform an update, the batch gradient descent can be very slow and tricky for datasets that are not suitable for memory. We have the parameter θ as the following:

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \tag{3.5}$$

Batch gradient descent can converge to a global minimum for a convex function and to a local minimum for a non-convex function.

Gradient update rules for Stochastic Gradient Descent is different. Compared to all gradients calculated by BGD using all data, SGD updates each sample every time it is updated.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}) \tag{3.6}$$

For large datasets, there may be similar samples, so BGD calculates gradients. There will be redundancy when SGD updates once at a time, there is no redundancy, and it is faster and new samples can be added. However, because SGD is updated more frequently, it will cause serious fluctuations in cost function.

Mini-batch gradient descent gradient update rule takes the best of and performs an update for every mini-batch of n training examples. MBGD uses a small number of samples, i.e. n samples, for each calculation. In this way, it can reduce the variance when the parameters are updated, and the convergence is more stable. On the other hand, it is possible to make full use of highly optimized matrix operations in the deep learning library to perform more efficient gradient calculations. The difference with SGD is that each cycle does not affect each sample, but a batch with n samples.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3.7}$$

However, Mini-batch gradient descent does not guarantee good convergence. If the selection of learning rate is too small, the convergence speed will be very slow. If it is too large, the loss function will fluctuate or even deviate at the minimum value. In addition, this method applies the same learning rate when all parameters are updated. If our data is sparse, we prefer to make a minor update to features that have a low frequency of occurrence. For non-convex functions, we should avoid trapped in local minimum, or saddle point, because the error around the saddle point is the same, the gradient of all dimensions is close to 0. Therefore SGD is easily trapped here.

3.1.2 Learning Rate

In gradient descent, there is a ratio where the weight's gradient is subtracted from the weight that influences the speed and quality of learning. This ratio is called the learning rate. The greater the ratio, the faster the neuron trains, but the lower the ratio, the more accurate the training is. With a higher learning rate, we can cover more in a function at each step, but we may exceed the lowest point because the slope of the hillside is constantly changing. At a very low learning rate, we can move toward the negative gradient because we recalculate it so frequently. The low learning rate is more accurate, but calculating the gradient is time-consuming, so it takes a long time to reach the bottom.

3.1.3 Cost Function

The cost function is a wrapper for our model function, which tells us how good our model is in predicting a given parameter. In machine learning problems, we use the cost function to update the parameters. Our cost function can take many forms because there are many different cost functions available. Commonly used cost functions include: mean squared error, root mean squared error, and log loss. The cost function can be estimated by iterating the operating model to compare the estimated prediction with the actual situation. Therefore, the goal of our model is to find parameters, weights, or structures that minimize the cost function.

For our scenario, we use the Mean Squared Error(MSE) as our cost function. We have the formula of Mean Squared Error as the following:

$$MSE = \frac{1}{N} \sum_{i=1}^{n} (y_i - (mx_i + b))^2 \tag{3.8}$$

Where N is the total number of observations (data points)
$\frac{1}{N} \sum_{i=1}^{n}$ is the mean

y_i is the actual value of an observation and (mx_i + b) is our prediction.

3.1.4 Activation Functions

Different activation functions actually have different properties. The only purpose of the activation function is to be non-linear in most practical scenarios. If we don't put the activation function between the two layers, the two layers won't be better together than one, because they're still just a linear transformation. For a long time, people used the sigmoid function and tanh, and they chose it very casually, but sigmoid was more popular. Until recently, ReLU became the mainstream non-leniency treatment. ReLU is used between layers because it is unsaturated and computes faster. Think about the graph of the s function. If the absolute value of x is large, then the derivative of the sigmoid function is small, which means that when we propagate the error backwards, when we return to the various layers, the error gradient will disappear very quickly. For all positive inputs, its derivative is 1, so the gradient of those activated neurons will not be changed by the activation unit, nor will it slow down the gradient descent.

For regression, it is usually activated with sigmoid or tanh because you want results between 0 and 1. For classification, we only need one output of 1, all the others are zero, but there is no differentiable way to implement it accurately, so we need to approximate it with a softmax.

3.1.4.1 Types of Activation Functions

3.1.4.1.1 Linear Activation Function It is a simple linear function of the form f(x) = x. Basically, the input passes to the output without any modification.

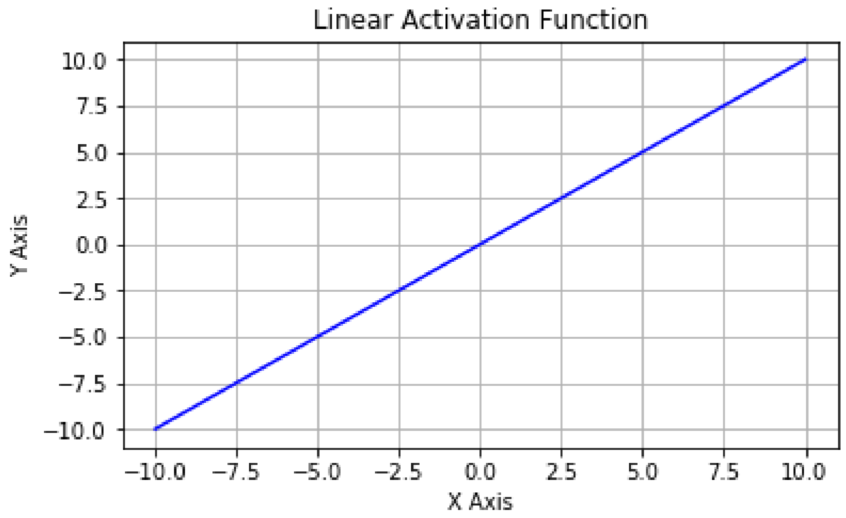

Figure 3.2. Linear Activation Function

3.1.4.1.2 Non-Linear Activation Functions These functions are used to separate the data that is not linearly separable and are the most used activation functions.A non-linear equation governs the mapping from inputs to outputs. Few examples of different types of non-linear activation functions are sigmoid, tanh, relu, lrelu, prelu, swish, etc. We will be discussing all these activation functions in detail.

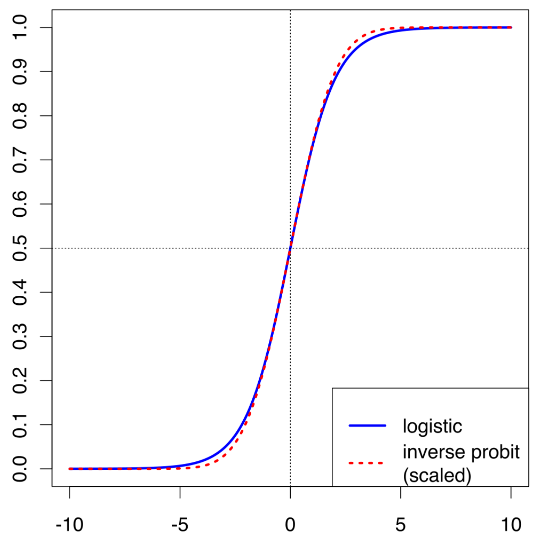

Figure 3.3. Non-Linear Activation Function

3.1.4.2 Types of Nonlinear Activation Function

3.1.4.2.1 Sigmoid It is also called logical reasoning activation. It takes a real number and compresses it between 0 and 1. It's also used in the output layer, and the ultimate goal is to predict probability. It converts big negative Numbers to 0, big positive Numbers to 1. It's expressed mathematically as: $\sigma(x) = \frac{1}{1+e^{-x}}$

The following figure shows the sigmoid function and its derivative graphically

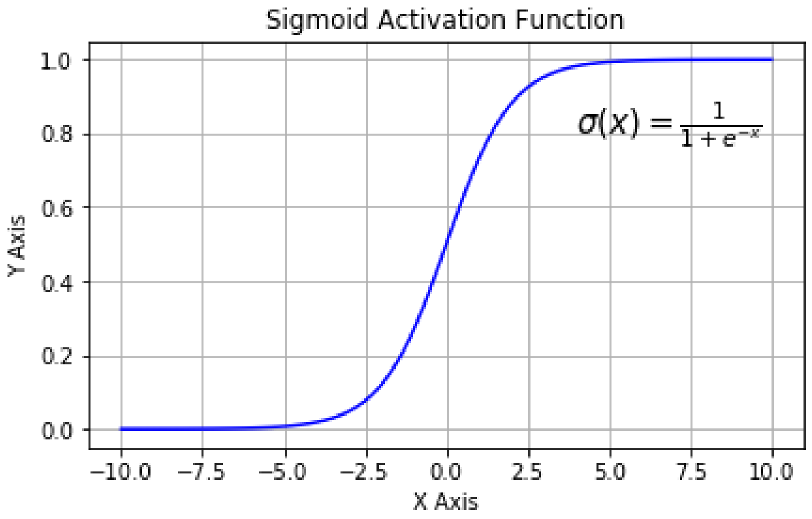

Figure 3.4. Sigmoid Activation Function

The three main disadvantages of sigmoid are:

1. Vanishing gradient: note that the sigmoid function is flat around 0 and 1. In other words, the gradient of sigmoid is around 0 and 1. In the process of sigmoid activated network reverse propagation, the gradient of the neuron output approaching 0 or 1 is close to 0. These neurons are called saturated neurons. Therefore, the weights in these neurons do not renew. Not only that, but the weight of the neurons attached to these neurons is slowly being renewed. This problem is also known as the vanishing gradient.

2. Zero center :Sigmoid output is not zero center.

3. Computational overhead: exp() functions are more computationally expensive than other nonlinear activation functions.

3.1.4.2.2 Tanh

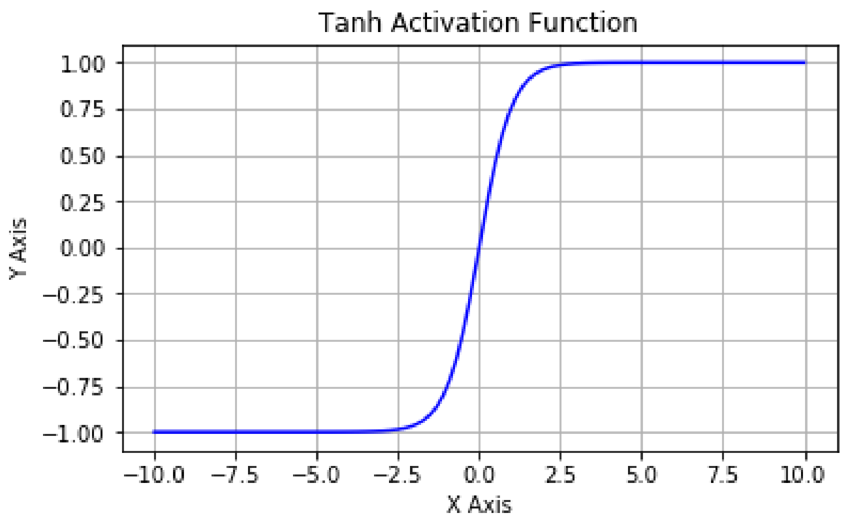

Figure 3.5. Tanh Activation Function

Tanh function is able to address the zero-centered problem in sigmoid. It is also called a hyperbolic tangent activation function. Like sigmoid, tanh accepts a real number, but compresses it between -1 and 1. Unlike sigmoid, the tanh output is zero center, because the range is between -1 and 1. You can think of the tanh function as two sigmoids put together. Actually, tanh is better than sigmoid. The negative input is considered strong and negative, the zero input value is mapped to the vicinity of zero, and the positive input is considered positive. The only drawback of tanh is that:

The tanh function is also affected by the asymptotic gradient problem, so when saturated, it kills the gradient.

In order to solve the problem of disappearing gradient, another nonlinear activation function, namely fixed linear unit (ReLU), is introduced. It is far better than previous two activation function, and is widely used now.

3.1.4.2.3 Rectified Linear Unit(ReLU)

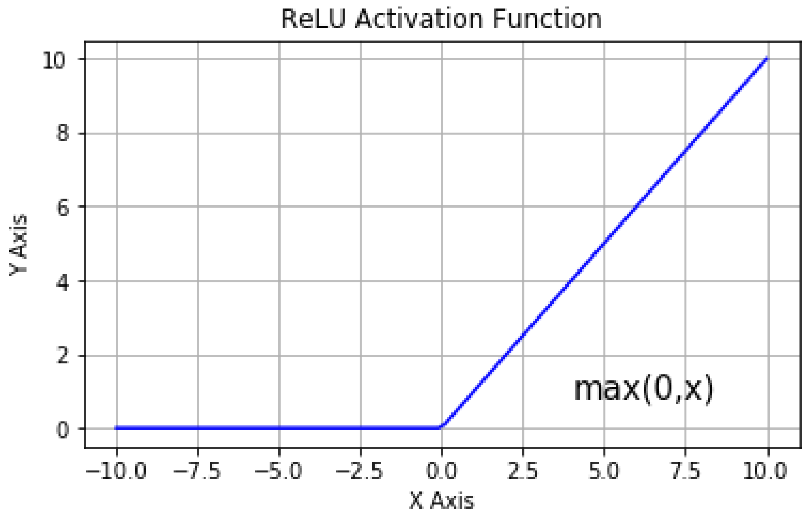

Figure 3.6. Relu Activation Function

As can be seen from the figure above, ReLU is semi-integral from the bottom. Mathematically, it's given by this simple expression: $f(x) = max(0, x)$

This means that when x is less than 0, the output is 0, and if x > is 0, the output is x. It's unsaturated, which means it's at least resistant to the vanishing gradient in the positive region (when x >), so the neuron doesn't propagate all the zeroes backwards in at least half of its regions. ReLU is computationally efficient because it is implemented using simple thresholds. But ReLU neurons have several disadvantages:

1. Not centered at zero: output is not centered at zero, similar to the sigmoid activation function.

2. Another problem with ReLU is that if x < 0 is kept inactive during the forward transmission, the neurons will kill the gradient during the backward transmission. Therefore, the weight will not be updated and the network will not learn. When x is equal to 0, the slope is not defined at this point, but the problem is solved by selecting the left gradient and the right gradient.

To solve the gradient problem that disappears in the ReLU activation function when x < 0, we have something called Leaky ReLU, which is trying to fix the dead ReLU problem. Let's take a closer look at the leaked data.

3.1.4.2.4 Leaky ReLU

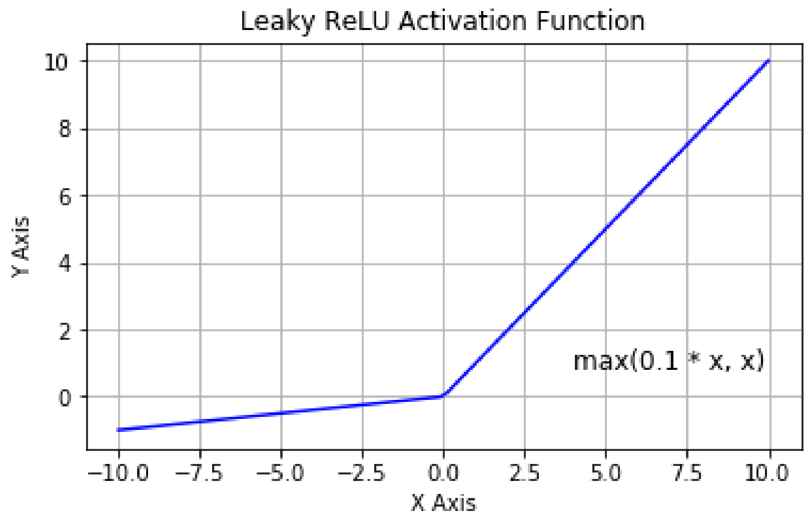

Figure 3.7. Leaky Relu Activation Function

This is an attempt to alleviate the dying ReLU problem. The function to calculate is $f(x) = max(0.1x, x)$.

The concept of leakage ReLU is that when x is less than 0, its slope is 0.1. This function eliminates the dead ReLU problem to some extent, but the results are not consistent. Although it has all the characteristics of the ReLU activation function, that is. , high computational efficiency, fast convergence speed, unsaturated.

The concept of leakage relays can be further extended. In real scenarios, we don't have to multiply the constant term by x we can multiply it by a super parameter which looks more like ReLU for leakage. This extension to the leakage relay is called the parameter relay.

3.1.4.2.5 Parameter ReLU The PRelu is given by: $f(x) = max(\alpha x, x)$, where α is a superparameter. The idea of PRelu is to introduce an arbitrary super parameter that this α can be learned because you can propagate it backwards. This gives neurons the ability to choose the optimal slope in the negative region, and with this ability, they can become a ReLU or a leaky ReLU.

3.2 Recurrent Network Network

The basic idea of RNNs is to use the sequential information. In traditional neural networks, all inputs and outputs are independent of each other. But under other scenarios, for example, if we want to predict the next word in a sentence, we are more like to know which words are before it. RNNs are recurrent because there are feedbacks in the sequence, with the output related to the previous computations. That is to say, RNNs have a memory, which memorize information about the calculation it has been made so far. Therefore, RNNs are able to make use of information in arbitrary long sequences. However, they can only see back a few steps before in practice. Recurrent neural network creates an internal state and gives held to interesting dynamic behaviors. Dissimilar to feedforward neural network, recurrent one can employ their internal memory to process arbitrary input sequences. This memory can be exploited for the multi-step prediction by storing previously predicted values to generate new values through time.

Compared to the tradition neural network, in each step of RNN, there are the same parameters like U, V, and W. Therefore, we get to know the fact that we implement the same task at each step, with only differences in input. In this way, we are able to reduce the number of factors that we need to learn.

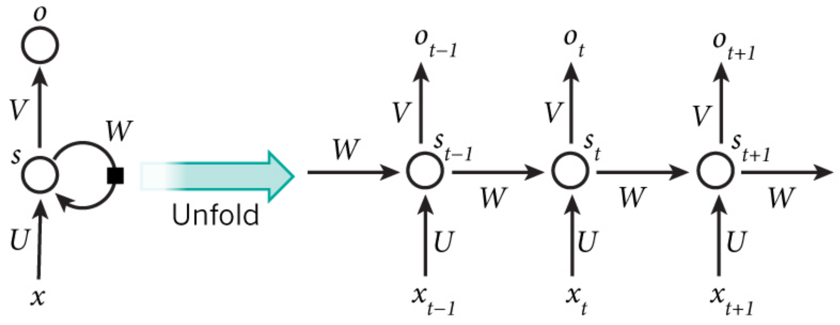

Figure 3.8. Structure of Recurrent Neural Network
[50]

The diagram [50] above shows how RNNs being unfolded into a full neural network. Unfolding means that the network need to be written out for the complete sequence. One common type of RNN is the standard Multilayer Perceptron and added loops. Generally, time t has to be discretized, with the activations updated at each time step. For the structure, the W is described as connection weight. The whole structure is described as follows: x_t is the input at time t and o_t is the output at time t. s_t is the hidden state at time t. s_t is calculated based on the past hidden state and the input of the current step:

$$s_t = f(Ux_t + Ws_{t-1}) \tag{3.9}$$

Where f is usually a nonlinear function. s_-1 is usually initialized to all zeros. s_t is the hidden state which is the memory of the network. It captures information about what happened in all past time. O_t is calculated based on the memory at time t. As we mentioned before, s_t cannot store information that comes from too many steps before.

Compared to the tradition neural network, in each step of RNN, there are same parameters like U, V, and W. Therefore, we get to know the fact that we implement the same task at each step, with only differences in input. In this way, we are able to reduce the number of factors that we need to learn.

Another important issue we concern is that how we train the RNN. We use the back-propagation algorithm to train RNN. As the parameters are the same in all time steps, the gradient at each output depends on both the present time and the previous time. In order to calculate the gradient at t = 3, we need to back propagate 2 steps and sum up the gradients. That is called Back Propagation Through Time (BPTT). It is a natural extension of standard back-propagation that performs gradient descent on a complete unfolded network.

BPTT is an adaptation of the well-known back-propagation training method from feed-forward networks. It is the most commonly used training method for feed-forward networks. BPTT is to unfold the recurrent network in time by stacking identical copies of the RNN, redirecting connections within the network to obtain connections between subsequent copies. The training problems lie on the determination of weights on each connection, the choice of initial activities of all hidden units.

As mentioned before, in RNNs we have cyclical dependencies. Unlike the feed forward nets, there is no layering as in the feed forward networks because neurons do not form a direct acyclic graph. Therefore, our way to use back-propagation for RNNs is converting RNNs into a new structure. This lies on the unrolling of the RNNs. By unrolling the RNNs, we are able to take its inputs, outputs and hidden parts and repeat it in every time step. For example, if the original RNN has a connection from neuron i to neuron j with a weight W, then in the unfolded neural network, we can draw a connection of weight W in every layer t_k from i and connect it to every layer t_{k+1} of neuron j.

Long Short Term Memory networks are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter, Schmidhuber [51], and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used. It is now one of the most commonly used gated neural models when modeling sequence data. The long-term memory can be explicitly represented by a single unit state c_t.

The state function of LSTM are:

$$\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned} \tag{3.10}$$

Where $x_t \in \mathbb{R}^d$ is the input vector to the LSTM unit. $f_t \in \mathbb{R}^h$ is forget gate's activation vector. $i_t \in \mathbb{R}^h$ is the input gate's activation vector. $o_t \in \mathbb{R}^h$ the output gate's activation vector. $h_t \in \mathbb{R}^h$ is the output vector of the LSTM unit. $c_t \in \mathbb{R}^h$ is the cell state vector $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$ are weight matrices and bias vector parameters which need to be learned during training where the superscripts d and h refer to the number of input features and number of hidden units, respectively. We can see from the below diagram of the LSTM cell.

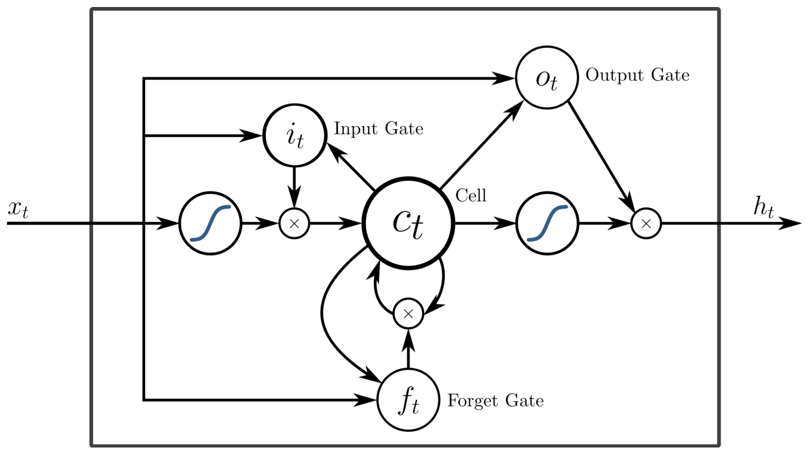

Figure 3.9. Structure of The LSTM Cell
[52]

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.

In our scenario, we conducted RNN in predictive coding for single image block prediction and find better performance with the same layer size.

Chapter 4 Implementations and Applications of Neural Network on Predictive Coding

In image compression, the prediction process includes statistical estimates of future random variables of past and present observable random variables. In an image compression scenario, the prediction of an image pixel or a group of pixels can be obtained from previously transmitted pixels. The predicted pixel is subtracted from the original pixel to obtain a residual signal. In general, if the energy of the remaining signal is lower than the energy of the original signal, the prediction is successful. Therefore, the remaining pixels can be encoded more efficiently by the entropy encoding algorithm and transmitted in fewer bits. In the first series of experiments, our work is based on just one image block prediction from JPEG image, while the second series is based on partial spatial information prediction and the third is based on full spatial information prediction.

4.1 Stateless and Stateless Neural Network without Using Spacial Dependencies

The key to this paradigm is the structure of information processing systems. Neural networks are used to estimate functions that depend on a large number of inputs, usually unknown. It was designed as a brain based computational model to solve certain problems. It is often described as a system of interconnected neurons that transmit information between them. Connected neurons with digital weights can be adjusted to experience to make neural networks possible inputs. Through the learning process, ANN can be configured for specific applications, such as pattern recognition or data classification. Learning in the biological system involves adjusting synaptic connections between neurons, similar to neural networks. In the first series of experiments, we are going to apply two kinds of neural networks on the decoding to make predictions of image blocks without using the surrounding blocks information in contrast to the following two sets of experiments. It means that we only use one block of 8x8 blocks with one image to make prediction of the next block of decoded image.

The basic idea and diagram of this decoding part is as follows:

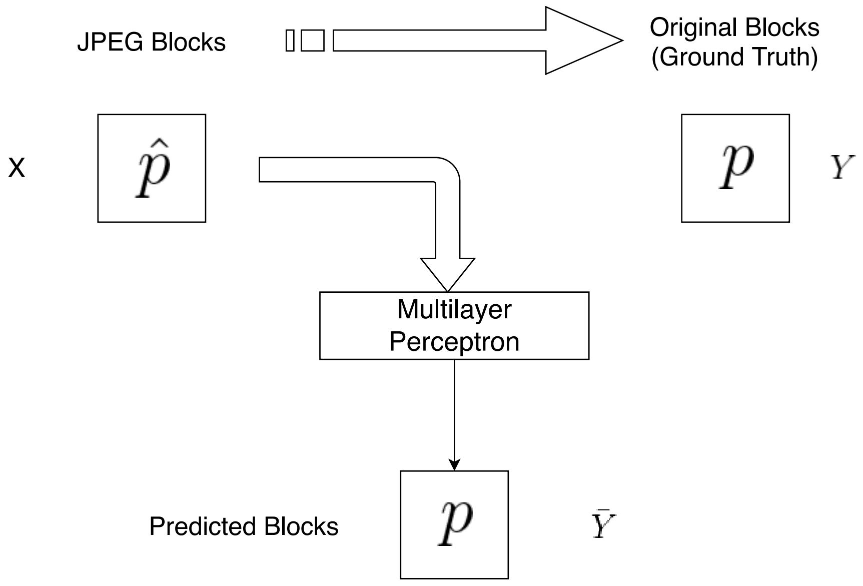

Figure 4.1. Diagram of mak

When we get the prediction of the single block, we will compare it with the original block and get the MSE cost function between Y and $\hat{Y}$. Then we use the MSE cost function in the training procedure.

The neural network is usually a nonlinear network system. It is composed of a large number of neurons, so that it has the ability to approach nonlinear function and fast learning speed. Feedforward networks usually consist of three layers: input layer, hidden layer and output layer. In the hidden layer, there may be several layers in which layer neurons receive the output of previous layer neurons. By controlling the weighted value and structure, we can realize the nonlinear mapping between input and output neurons.

At present, there are many types of neural networks, such as multi-layer BP algorithm, Hopfield neural network and adaptive neural network.

We will also implement and train recursive neural networks. Recurrent neural network (RNN) is a kind of neural network that connects neurons to form a directed cycle. There are many different types of recurrent neural networks. A recursive neural network consists of at least one feedback connection.

In our research, we also need normalization. In some applications, due to the use of distances or feature variances, it is required that the input vector must be normalized, because if not normalized, the result would be seriously affected by features with large variance and different scales. Normalized input can help numerically optimal methods such as gradient descent to converge faster and more accurately. In our scenario, we divide pixel values by 255 to achieve this normalization.

In the first series of experiments, to measure the performance of different activation functions, we conduct several of them in Tensorflow and compare their prediction results. We use non-linear functions in our scenario because it is proven that it is impossible to approximate even simple function without using non-linear functions. It is also proven that a two-layer neural network can be a universal function approximator. [53] When multiple layers uses linear activation functions(identity function), the whole network is equal to a single-layer model. For most activations functions, they are continuously differentiable for enabling gradient-based optimization methods while relu is not. But relu still works with gradient descent optimization. As we can see from the table, we conduct different activation functions with fixed number of hidden layers and size of hidden layers. As shown in the table, we get the best performance measured by MSE using the elu function. The baseline of the MSE value for the test dataset is 281.376, which means that the average value of Squared Error between original images and JPEG images is 281.376. The baseline of PSNR, SSIM and MSSSIM are 24.551, 0.836 and 0.981, respectively. The MSE, PSNR, SSIM, MSSSIM for validation dataset are 302.034, 24.083, 0.838 and 0.981, respectively. Then we implement our predictive models on the JPEG images and get our prediction from JPEG images. In this series of experiments, we measure and evaluate MSE values between predicted images and JPEG images both compared with original test images. The dataset we use is the Tiny Imagenet Dataset, which has 100000 images for training, 10000 for validation and 10000 for testing. For each series of experiments, we run 10 trials and get the average of their MSE, PSNR, SSIM and MSSSIM. For each 10 trials, we also have its standard derivation to measure the dispersion of results.

4.1.1 Cross Validation and Overfitting

When using training sets to train a model, it is widely implemented that an entire training set is usually divided into three parts. It is generally divided into three parts: training set, validation set and test set. In fact, it is data that does not participate in training at all and is only used to observe the test effect.

Because in actual training, the training result is usually good for the training set because the initial conditions are sensitive. But the degree of fitting to the data outside the training set is usually not satisfactory. Therefore, we usually do not use all the data sets for training, but instead part (this part does not participate in training) to test the parameters generated by the training set to relatively objectively determine the parameters of these data outside the training set The degree of compliance.

In the dataset we use, it is already divided into three part as we mentioned before. To keep track of overfitting, we have the following graphs for training, validation and testing loss with the default model structure(1024 hidden units and 1 hidden layer). For each result shown below, we have the average of 10 trials and its standard derivation.

To decide which activation function performs the best, we train the a model with 1024 hidden units in 1 hidden layer and get the following results of MSEs. For each result listed, we have the average of 10 trials and its standard derivation.

From Figure 4.2 we can see that when the training epochs increases, the training loss decrease rapidly and final remain in the same level. After 100 epochs, the validation MSE does non decrease. Therefore we can conclude that with more training epochs we cannot get better performance or could possibly even overfit in the validation set. Then in the following experiments, we all set the default training epochs to be 100.

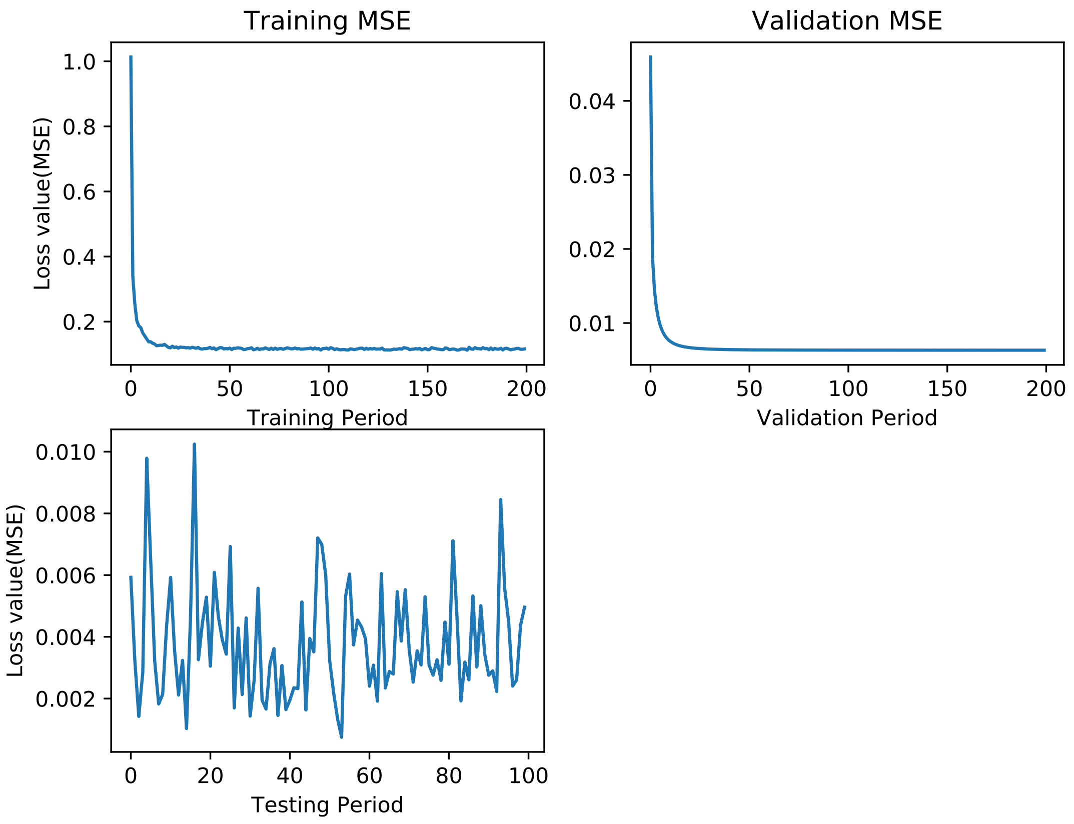

Figure 4.2. Training, Validation and Test MSEs

4.1.2 Results of Different Activation Functions

From the cross validation results we are going to see that the most appropriate training epoch is in 100. Furthermore, to decide which activation function performs the best, we train the a model with the default model size(1024 hidden units, 1 hidden layer) with 100 epochs and get the following results of MSEs. For each result listed, we also have the average of 10 trials and its standard derivation.

Table 4.1. Performance of Different Activation Functions of Test Dataset

Activation Functions	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
Tanh	273.727±2.921	24.423±0.063	0.827±0.004	0.980±0.000
Relu	243.065±1.070	24.973±0.027	0.841±0.001	0.982±0.000
Relu6	242.151±0.987	25.016±0.038	0.841±0.000	0.980±0.000
Elu	245.403±0.943	24.910±0.036	0.832±0.002	0.981±0.000
Sigmoid	281.973±2.594	24.096±0.056	0.822±0.000	0.974±0.000
Selu	245.075±3.927	19.418±8.994	0.585±0.397	0.704±0.447

Table 4.2. Performance of Different Activation Functions in Validation Dataset

Activation Functions	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
Tanh	246.039±2.844	24.930±0.069	0.830±0.004	0.981±0.000
Relu	218.968±1.038	25.487±0.03	0.845±0.001	0.982±0.000
Relu6	212.458±1.133	25.982±0.078	0.845±0.000	0.982±0.000
Elu	552.052±179.811	22.679±0.784	0.801±0.012	0.968±0.006
Sigmoid	279.782±2.269	24.126±0.051	0.822±0.000	0.975±0.000
Selu	880.720±422.846	17.291±7.556	0.557±0.378	0.689±0.437

From results of the first series of experiments, we are able to conclude that relu6 function has the best performance among the widely used activation functions. Therefore we use relu6 function as the activation function in the following research work.

We also discuss about the efficiency of Relu6 activation function. As we know, the output of a ReLU unit is y = max(x, 0) and ReLU6 activation function is y = min(max(x, 0), 6), which will cap the function at 6. In the series of tests, this encourages the model to learn sparse features earlier. In the formulation of [54], this is equivalent to imagining that each ReLU unit consists of only 6 replicated bias-shifted Bernoulli units, rather than an infinite amount.

As we discussed before about the learning rate, we need to tune learning rate to find an optimized learning rate for gradient descent. In the first series of experiments, we get several performance values from different learning rate and choose the learning rate to be 0.01.

Table 4.3. Prediction Quality over Learning Rate on Test Dataset

Learning Rate	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
0.01	237.072±0.194	26.003±0.042	0.842±0.003	0.983±0.001
0.001	272.867±0.731	24.62±0.016	0.828±0.001	0.98±0.000
0.0001	241.356±0.363	25.071±0.007	0.842±0.000	0.982±0.000
1.00E-05	247.933±0.203	24.943±0.005	0.839±0.000	0.982±0.000

Table 4.4. Prediction Quality over Learning Rate in Validation Dataset

Learning Rate	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
0.01	302.064±0.147	25.018±0.054	0.836±0.002	0.982±0.001
0.001	315.729±35.825	24.207±0.241	0.824±0.004	0.978±0.002
0.0001	322.388±59.330	24.207±0.241	0.824±0.004	0.98±0.002
0.00001	309.981±6.563	24.241±0.052	0.834±0.001	0.98±0.000

Moreover, there are multiple widely used optimizer in tensorflow including gradient descent optimizer, Adam optimizer and Adagrad optimizer. We also implement these three optimizer with the same model structure and have results of their performance.

In the following we will discuss algorithms that are widely used by the Deep Learning community to deal with the optimization challenges and the experimental results we get in the first series of experiments.

4.1.3 Momentum

Momentum is a method that helps speed up the development of SGD in the relevant direction and suppress fluctuations. It is implemented by adding the fraction Îş of the update vector of the past time step to the current update vector.

$$v_t = \gamma v_{t-1} + \eta \bigtriangledown_\theta J(\theta) \tag{4.1}$$

$$\theta = \theta - v_t \tag{4.2}$$

4.1.4 Adagrad and Adadelta

Adagrad is a gradient based optimization algorithm that can do a larger update parameters of low frequency and less work on the high frequency of updates. Also, therefore, its performance is very good for sparse data to improve the robustness of SGD. For this reason, it is ideal for handling sparse data. Adagrad has the advantage of reducing the manual adjustment of the learning rate. Its disadvantage is that the denominator will continue to accumulate, so that the learning rate will shrink and eventually become very small [55].

As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we use gt to denote the gradient at time step t. $g_{t,i}$ is then the partial derivative of the objective function w.r.t. to the parameter θ_i at time step t:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}) \tag{4.3}$$

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \tag{4.4}$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \tag{4.5}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i,i is the sum of the squares of the gradients w.r.t. θ_i up to time step t, while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e^{-8}$). Interestingly, without the square root operation, the algorithm performs much worse.

As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters Îÿ along its diagonal, we can now vectorize our implementation by performing a matrix-vector product $\odot$ between G_t and g_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{4.6}$$

One of the main advantages of Adagrad is that there is no need to manually adjust the learning rate. Most implementations use the default value of 0.01 and keep it.

Adadelta is an extension of Adagrad that aims to reduce its positive, monotonous learning speed [56]. Instead of accumulating all the past square gradients, Adadelta restricts the window of the accumulated past gradient to a fixed size w.

Instead of storing w previous squared gradients inefficiently, the sum of the gradients is recursively defined as the average of the decay of all past squared gradients. The running average $E[g^2]_t$ at time step t depends only on the previous average and the current gradient as a fraction γ similarly to the momentum term:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2 \tag{4.7}$$

In our scenario, γ is usually set to a similar value as the momentum term, which is around 0.9. For clarity, the vanilla SGD update in terms of the parameter update vector $\Delta\theta_t$ can be rewritten as:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{4.8}$$

The Adagrad parameter update vector that we derived earlier is the form:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{4.9}$$

We can change the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{4.10}$$

Since the denominator is simply the Root Mean Square (RMS) error criterion for the gradient, we can replace it with a short-term criterion:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \tag{4.11}$$

The authors of Adadelta note that the units in this update do not match, i.e.

the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \tag{4.12}$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{4.13}$$

Since $RMS[\Delta\theta]_t$ is unknown, it can be approximated with the parameterized RMS until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ eventually produces the Adadelta update rule:

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{4.14}$$

With Adadelta, we do not need to set the default learning rate because it has been eliminated from the update rule.

4.1.5 RMSprop

RMSprop is an adaptive learning rate method proposed by Geoff Hinton. Both RMSprop and Adadelta are designed to solve the problem of sharp decline in learning rate of Adagrad.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{4.15}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{4.16}$$

RMSprop also divides the learning rate by the exponential decay average of the squared gradient. Hinton recommends setting γ to 0.9, and a good learning rate defaults to 0.001.

4.1.6 Adam

Adam is another algorithm that computes adaptive learning rates for each parameter. In addition to algorithms like Adadelta and RMSprop which stores the exponentially decaying average of past squared gradients v_t, it is also like momentum to keep the exponentially decaying average of past gradients of m_t.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{4.17}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{4.18}$$

Among them, m_t and v_t are the estimated values of the first time (average) and second time (non-statistical variance) of the gradient, respectively, and therefore are the names of the methods. Since m_t and v_t were initialized to vectors of 0, the authors of Adam observed that they were biased toward zero, especially in the initial time step when the decay rate was small (i.e. β_1 and β_2 are close to 1).

From various projects we can see that popular deep learning libraries usually use the recommended default parameters. For the implementation in TensorFlow: learning rate=0.001, β_1=0.9, β_2=0.999, ϵ=$1e^{-8}$.

4.1.7 The Choice of Optimizers

If our input data is sparse, then we would likely achieve the best results using one of the adaptive learning-rate methods. In our scenario, the image data is not sparse. An additional benefit is that we won't need to tune the learning rate but likely achieve the best results with the default value.

In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Kingma et al. show that its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser [55]. Insofar, Adam might be the best overall choice.

Interestingly, many recent papers use vanilla SGD without momentum and a

Table 4.5. Prediction Qualities Using Different Optimizer in Test Dataset

Optimizer	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
Adam	833.851±45.749	19.561±0.305	0.532±0.027	0.899±0.007
ProximalAdagrad	239.181±0.184	23.242±5.923	0.759±0.263	0.89±0.293
Adagrad	239.075±0.197	25.117±0.004	0.843±0.000	0.983±0.000
Momentum	237.072±0.194	26.003±0.042	0.842±0.003	0.983±0.001
Adadelta	244.893±0.111	25.006±0.002	0.841±0.000	0.982±0.000
RMSProp	906.880±122.852	19.029±0.685	0.483±0.059	0.857±0.034
Ftrl	255.934±2.922	24.881±0.037	0.837±0.001	0.982±0.000
ProximalGradientDescent	242.055±1.831	25.083±0.018	0.84±0.001	0.981±0.000
GradientDescent	241.987±0.831	25.092±0.216	0.841±0.000	0.981±0.000

Table 4.6. Prediction Qualities Using Different Optimizer in Validation Dataset

Optimizer	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
Adam	855.203±47.364	19.436±0.308	0.534±0.026	0.899±0.007
ProximalAdagrad	316.122±49.071	22.698±5.734	0.754±0.262	0.887±0.292
Adagrad	305.121±28.823	24.532±0.136	0.836±0.002	0.979±0.001
Momentum	299.065±10.124	25.032±0.114	0.836±0.001	0.981±0.002
Adadelta	304.895±10.518	24.342±0.061	0.834±0.002	0.979±0.001
RMSProp	1272.346±709.717	18.399±1.274	0.477±0.053	0.855±0.035
Ftrl	365.215±63.445	24.276±0.152	0.831±0.001	0.978±0.001
ProximalGradientDescent	307.814±15.412	24.103±0.081	0.832±0.001	0.979±0.002
GradientDescent	309.124±22.815	23.173±0.092	0.831±0.002	0.976±0.001

simple learning rate annealing schedule. As has been shown, SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima.

Therefore, in our experiments, we conducted mainly Gradient Descent Optimizer, Adam Optimizer, Adagrad Optimizer and Momentum Optimizer and hence find essential results that Momentum Optimizer performs the best with the Multiple Layer Perceptron in the first series of experiments and Adam Optimizer works best for Recurrent Neural Network. The following table shows the MSE and PSNR results vs. different optimizer. Each of them is implemented with the same model which has 1 hidden layer with 1024 hidden units.

To minimize the effect of overfitting, we add dropout to the neural network and we can see clearly that it is efficient in reducing overfitting and getting better prediction results.

Table 4.7. Dropout Ratio vs. Prediction Quality in Test Dataset

Dropout Keep Probability	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
0.5	300.219±30.602	24.078±0.518	0.809±0.009	0.976±0.004
0.6	251.356±3.278	24.876±0.072	0.829±0.002	0.980±0.000
0.7	276.604±5.454	24.397±0.092	0.822±0.003	0.979±0.001
0.8	242.151"±"0.987	25.016±0.038	0.841±0.000	0.980±0.000
0.9	248.451±3.142	24.880±0.075	0.839±0.001	0.982±0.000

Table 4.8. Dropout Ratio vs. Prediction Quality in Validation Dataset

Dropout Keep Probability	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
0.5	346.359±82.402	23.760±0.825	0.805±0.013	0.974±0.006
0.6	257.949±3.645	24.779±0.072	0.828±0.002	0.980±0.000
0.7	249.278±5.514	24.899±0.108	0.829±0.003	0.979±0.001
0.8	232.408±5.765	25.214±0.135	0.837±0.003	0.981±0.001
0.9	253.686±3.242	24.398±0.089	0.834±0.001	0.980±0.000

From experimental results we choose the dropout ratio to be 0.8, which is used in the following experiments. Then we get the following results using Momentum optimizer with the learning rate of 0.01. To find out the relationship between the size of layers, hidden units and the model performance, we conduct the following experiments of changing hidden units and hidden layer sizes.

First we fix the number of hidden units of 1024 and change the hidden layer sizes correspondingly and get the following results:

Table 4.9. Prediction Quality vs. Number of Hidden Layers in Test Dataset

Number of Hidden Layers	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
1	237.072±0.194	26.003±0.042	0.842±0.003	0.983±0.001
2	1295.148±791.609	18.109±2.613	0.553±0.108	0.906±0.043
3	10537.238±706.487	8.157±0.274	0.010±0.001	0.10±0.001
4	10622.043±1599.61	8.122±0.600	0.010±0.001	0.000±0.000
5	10729.213±638.181	8.093±0.245	0.010±0.001	0.000±0.000

Table 4.10. Prediction Quality vs. Number of Hidden Layers in Validation Dataset

Number of Hidden Layers	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
1	232.408±5.765	25.214±0.135	0.837±0.003	0.981±0.001
2	1455.507±773.334	17.801±2.446	0.554±0.104	0.904±0.041
3	10815.643±700.113	8.120±0.262	0.010±0.001	0.001±0.000
4	10908.044±1587.231	8.071±0.564	0.009±0.001	0.000±0.000
5	11008.154±627.145	8.062±0.233	0.010±0.000	0.000±0.000

Therefore we can see from the above results of MSE and PSNR that one hidden layer performs the best. As we know before, one hidden layer model is sufficient enough to perform as a universal regression model to make predictions. In our model, if the number of hidden layer is too many, there is possibility that it highly overfit in test dataset. So we would choose one hidden layer as our default model structure.

The following shows the plot of MSE vs. number of hidden units:

We can see from the above table that when we increase the number of hidden layer units, the prediction will gradually go better. Furthermore, when the hidden units are extremely high, it is possible that the neural network will have higher change to overfit, which will lead to a worse prediction result. From the following

Table 4.11. Number of Hidden Units vs. Prediction Quality in Test Dataset

Number of Hidden Units	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
256	300.477±19.376	23.668±1.257	0.813±0.018	0.973±0.018
512	291.754±6.761	24.211±0.120	0.815±0.004	0.979±0.001
1024	257.588±5.158	24.707±0.110	0.831±0.003	0.981±0.001
2048	247.296±1.759	24.899±0.044	0.837±0.002	0.982±0.000
4096	245.776±1.758	24.926±0.045	0.84±0.001	0.982±0.000

Table 4.12. Number of Hidden Units vs. Prediction Quality in Validation Dataset

Number of Hidden Units	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
256	338.151±181.089	24.119±1.411	0.820±0.020	0.973±0.020
512	261.58±7.276	24.211±0.12	0.823±0.004	0.979±0.001
1024	232.408±5.765	25.214±0.135	0.837±0.003	0.981±0.001
2048	222.585±1.837	25.42±0.052	0.843±0.002	0.982±0.000
4096	221.306±1.793	25.445±0.052	0.845±0.001	0.982±0.000

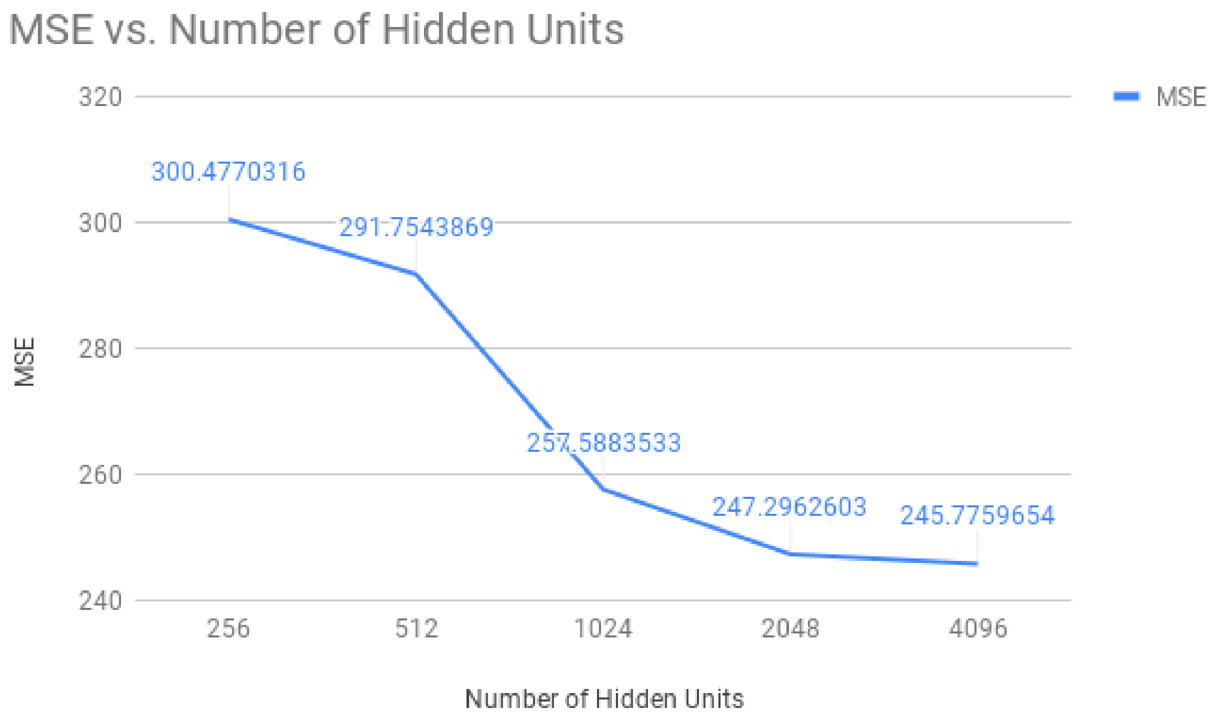

Figure 4.3. MSE vs. Number of Hidden Units using Stateless Network

we can also see images of original image, JPEG image and reconstructed image from left to right.

Then we are going to discuss results we get from the LSTM. As we discuss before, LSTM is based on RNN. Based on the first frame, the RNN is able to predict future frames and continue the process until the last frame. Therefore, we only need to store the first frame and the weights of the neural network. With the small amount of data, we are able to reconstruct the whole video.

The key advantage of using an LSTM unit over a traditional neuron in an RNN

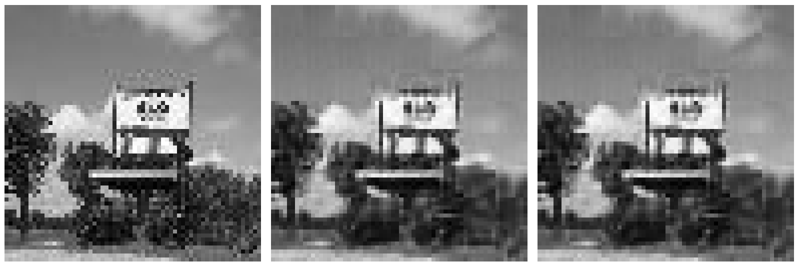

Figure 4.4. Original Image, JPEG image and Reconstructed Images with 1 Hidden Layer and 1024 Hidden units

is that the cell state in an LSTM unit sums activities over time. Since derivatives distribute over sums, the error derivatives do not vanish quickly as they get sent back into time. This makes it easy to do credit assignment over long sequences and discover long range features [52].

There are several key parameters for the RNN training. Firstly, the seed is the determinism of program. We can change this to affect the random number generation. Secondly, the number of epochs determines how many times model should see the entire video to compress. Then we have a couple of parameters: the number of hidden units and the number of hidden units2. Moreover, we need to set the step size, which is the learning rate of this RNN.

The number of previous frames that we use is indicated by the window size. Also we choose the learning rate of 0.001, which is the step size of the training program. While the main parameters are number of compression time, the number of hidden units 1 and the number of hidden units 2. We made a series of experiments to see the effects of these parameters. In these experiments, the number of compression time has the most significant impact on the loss and video quality. While the window size does not influence much, the number of hidden units is the second most important parameter.

In our scenario, we take the 8 pixels within one row as the input size and 8 columns in one image as 8 time steps. In the implementation of Tensorflow, the situation is slightly special. We use the For BasicLSTMCell as the LSTM cell. Because LSTM can be seen as having two hidden states h and c, the corresponding hidden layer is a Tuple, each is the shape of $(batch_size, state_size)$. The model we implement can be seen as an autoencoder. The structure of the autoencoder is shown in the following diagram [52]:

The input to the model is a series of vectors, which are the 8 pixels in one row.

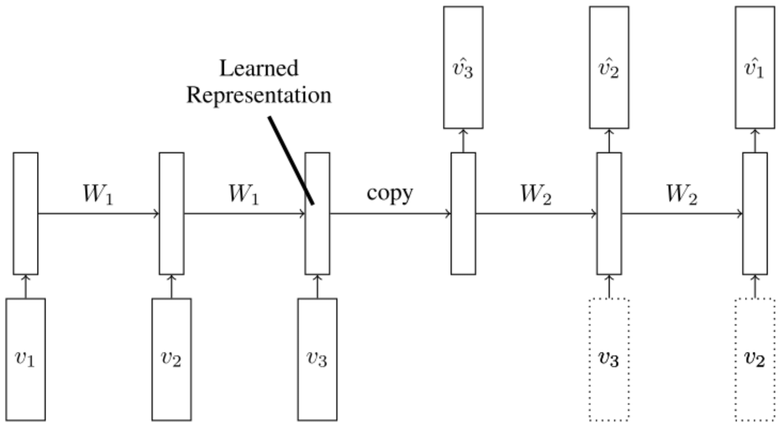

Figure 4.5. LSTM Autoencoder Model

The encoder LSTM reads in this order. After reading the last input, the decoder LSTM takes over and outputs the prediction of the target sequence. The target sequence is the same as the input sequence but in the opposite order. Reversing the target sequence makes the optimization easier because the model can be implemented by looking at the low-range correlation. The encoder can be seen as creating a list by applying the cons function on the previously constructed list and new input. The decoder essentially unrolls this list, extracting the top element of the list (car function) with hidden output weights, and the hidden weight extracting the remainder of the list (cdr function). Therefore, the first element out is the last element in. [52]

The basic diagram of how the lstm works in our procedure is shown below:

For the predictor, we set different number of hidden LSTM units and have the following results of prediction qualities.

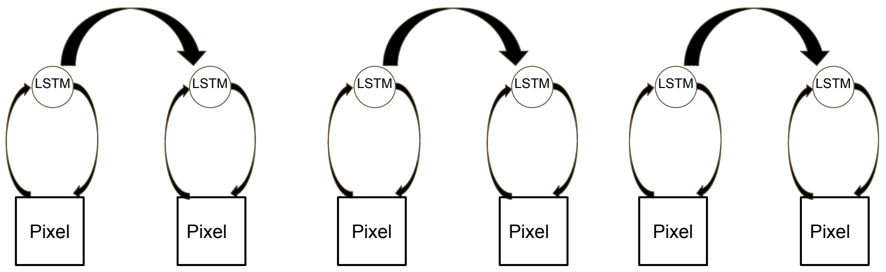

Figure 4.6. The Diagram of LSTM Model in Image Predictions

Table 4.13. Predicting Qualities vs. Number of Hidden Units of LSTM

Number of Hidden Units	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
256	229.938±0.373	25.318±0.011	0.843±0.000	0.982±0.000
512	252.876±2.328	25.015±0.037	0.839±0.002	0.982±0.000
1024	255.035±0.715	24.918±0.087	0.841±0.002	0.981±0.001
2048	249.628±0.894	25.042±0.075	0.839±0.000	0.982±0.000
4096	248.045±1.013	25.115±0.032	0.840±0.001	0.982±0.000

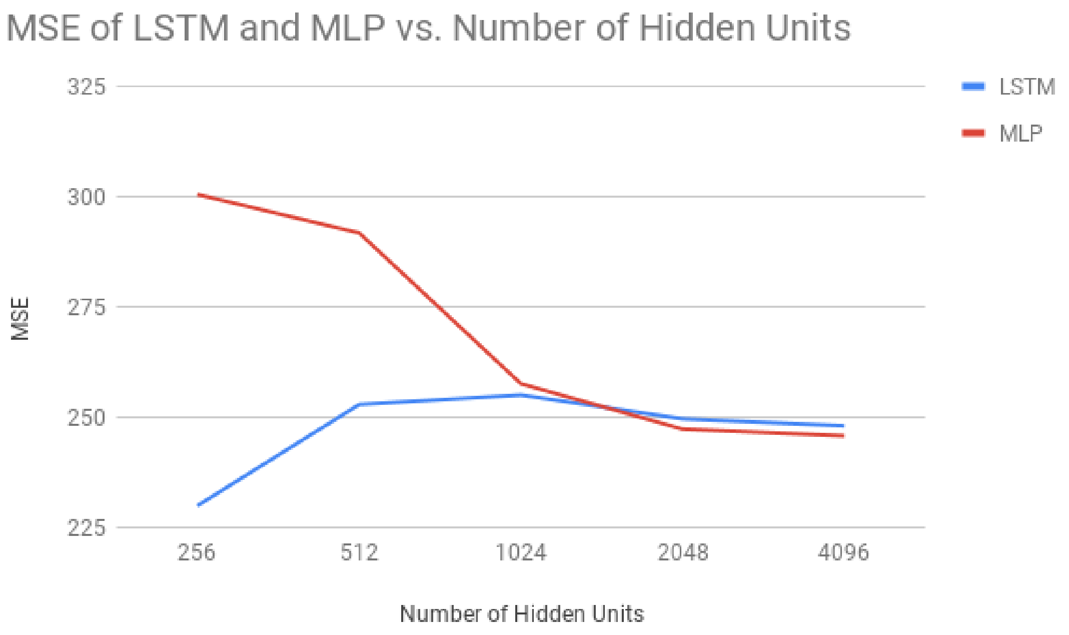

Figure 4.7. Performance of LSTM and MLP vs. Number of Hidden Units

In summary, in the first series of experiments, the best results we are able to achieve of the MSE, PSNR, SSIM and MSSSIM are 229.938, 25.318, 0.843 and 0.982 respectively. The baseline for JPEG images are . Compared with results of JPEG, we are able to obtain as much as a 0.9 dB gain over the standard JPEG algorithm in PSNR and 51.44 less in MSE.

4.2 Stateless Neural Network Using Partial Spacial Dependencies

In images, there are lots of information redundancies among blocks. As natural signals are highly correlated, the difference between neighboring samples is usually small. The value of a pixel x can be therefore predicted by its neighbors a, b, c, and d with a small error:

$$e_1 = x - a, \quad e_2 = x - b, \quad e_3 = x - \frac{a+b+c+d}{4} \tag{4.19}$$

In the Lossless JPEG algorithm, the predicted value $\hat{x}_0$ for a pixel x_0 is a linear combination of all available neighbors a_i, $(i = 1, \cdots, n)$:

$$\hat{x}_0 = \sum_{j=1}^{n} a_j x_j \tag{4.20}$$

Therefore, to achieve higher performance than JPEG, we implement Artificial Neural Network in predictive coding and use the spacial contextual information to make better predictions in the second series of experiments. In this experiments set, we only use dependency of the 3 blocks that are before the block which is to be predicted. For comparing the coding efficiency of each codec with neural network, we performed coding experiments for different scenarios. $\hat{p}_1$ to $\hat{p}_4$ are given from the JPEG compression and p_4 is the prediction result, which forms the predicted image and compare with the ground true images.

The pattern of selecting patches for training is similar with partial and full spatial information predictive coding. For the partial spatial prediction part, we divide the image into a group of 8x8 non-overlapping small blocks, 64 small blocks for a size of 64x64 images. Then, we create a training patch consisting of 4 adjacent blocks that are fed as input to our compression model to predict a 8x8 target block size .

For example, suppose the input image is divided into 8x8 non overlapping patches and each patch is numbered sequentially from the upper left which starts from the lower right corner of the image. We index the blocks within one image using a 8x8 matrix. The procedure for extracting pixel blocks is that we use the block (0,0),(0,1),(1,0),(1,1) to make the prediction patch (1,1). Similarly, the next

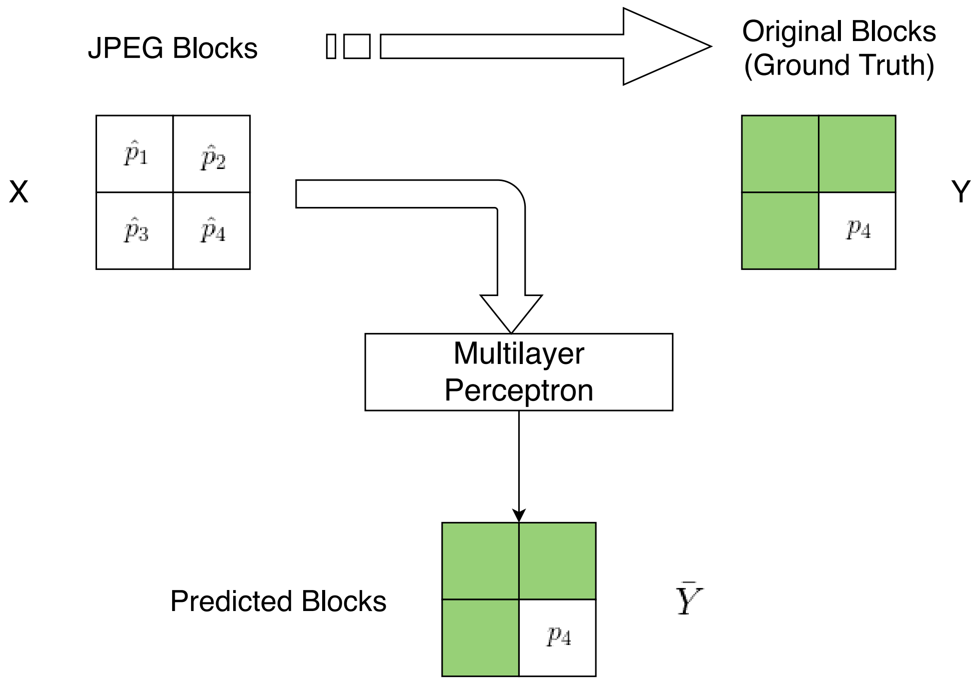

Figure 4.8. The Diagram of Partial Spatial Blocks Method

block collection contains (0,1),(0,2),(1,1),(1,2), which is used to predict (1,2).

To predict a bounding block such as (0,0), we consider the same nine blocks as before: (0,0),(0,1),(1,0),(1,1). But the model will reconstruct (0,0) this time instead of (1,1). We use the 4 input patches to reconstruct the image patch at the edge of a image without using zero padding. In the prediction process, the output of the neural network is the prediction for p_4, which is also know as the $\bar{Y}$. $\bar{Y}$ is compared with the ground truth of p_4 and the training part deals with the MSE loss of predicted and original blocks Y.

For partial spatial information prediction, first we conduct different optimizer and find the best one. Next we choose the most appropriate learning rate and conduct dropout. We use the same model structure, which has 1 hidden layer with 1024 hidden units.

Concluding from following results, in this situation, we get the best performance using gradient descent optimization. Similar to previous experiments, we also conduct the model with a learning rate of 0.01 as well as 0.8 dropout.

With a fixed number of hidden units we change number of hidden layers to find the optimized hidden layer size. Then we implement increasing hidden units and see the pattern of changing layer units. Finally similar to before, we conduct

Table 4.14. Prediction Qualities of Different Optimizers in Test Dataset

Optimizer	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
Adam	943.851±75.459	19.032±0.415	0.493±0.014	0.872±0.005
ProximalAdagrad	278.231±0.051	23.982±1043	0.814±0.004	0.979±0.002
Adagrad	270.249±0.169	24.529±0.004	0.834±0.000	0.981±0.000
Momentum	269.255±5.400	24.58±0.111	0.829±0.002	0.98±0.001
Adadelta	296.834±0.927	24.275±0.020	0.823±0.000	0.978±0.000
RMSProp	1127.761±95.452	19.013±0.485	0.512±0.034	0.844±0.013
Ftrl	315.473±2.431	24.231±0.045	0.821±0.001	0.969±0.001
ProximalGradientDescent	272.051±0.987	24.418±0.121	0.83±0.000	0.981±0.000
GradientDescent	273.078±1.045	24.325±0.031	0.827±0.001	0.98±0.000

Table 4.15. Prediction Qualities of Different Optimizers in Validation Dataset

Optimizer	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
Adam	958.991±42.075	18.132±0.921	0.445±0.021	0.799±0.052
ProximalAdagrad	378.412±0.017	23.078±1043	0.819±0.004	0.965±0.001
Adagrad	377.336±87.181	24.096±0.178	0.828±0.002	0.978±0.001
Momentum	352.443±4.576	24.281±0.113	0.829±0.001	0.979±0.001
Adadelta	366.749±24.062	23.796±0.084	0.819±0.003	0.976±0.002
RMSProp	1043.583±64.892	19.132±0.551	0.533±0.025	0.847±0.009
Ftrl	396.514±3.815	23.141±0.092	0.816±0.002	0.957±0.002
ProximalGradientDescent	381.198±0.521	23.049±0.009	0.799±0.000	0.961±0.001
GradientDescent	383.096±0.982	23.015±0.031	0.798±0.003	0.962±0.002

LSTM and see the performances of RNN in the partial spatial information block prediction.

In the next set of experiments, we fix the number of hidden units of 1024 and apply different hidden layer sizes correspondingly using the method of partial spatial information blocks and get the following results:

Table 4.16. Prediction Quality vs. Number of Hidden Layers Using Partial Spatial Information in Test Dataset

Number of Hidden Layers	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
1	269.255±5.4	24.58±0.111	0.829±0.002	0.98±0.001
2	258.965±1.153	24.77±0.022	0.834±0.001	0.981±0
3	265.058±6.556	23.589±1.877	0.834±0.003	0.972±0.015
4	266.101±1.863	24.69±0.038	0.833±0.002	0.981±0
5	271.045±1.304	24.633±0.028	0.833±0	0.98±0

Table 4.17. Prediction Quality vs. Number of Hidden Layers Using Partial Spatial Information in Validation Dataset

Number of Hidden Layers	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
1	406.675±75.394	23.925±0.238	0.822±0.003	0.976±0.002
2	354.411±49.497	24.235±0.178	0.828±0.002	0.978±0.001
3	414.378±57.298	23.953±0.152	0.825±0	0.976±0
4	421.875±41.925	23.782±0.001	0.823±0.002	0.975±0.001
5	451.995±59.257	23.601±0.043	0.821±0.004	0.976±0.002

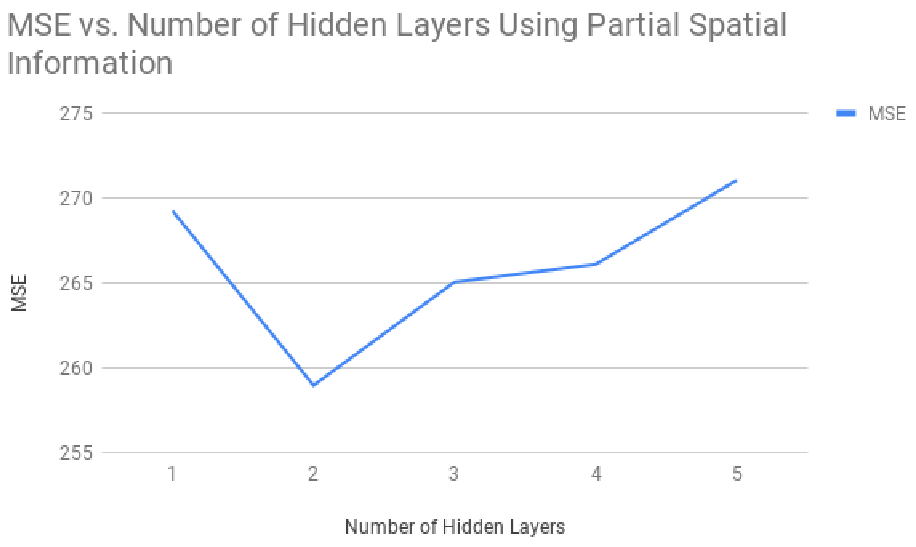

Figure 4.9. The Diagram of MSE vs. Hidden Layers

Similarly we can see from results of MSE and PSNR as well as SSIM and MSSSIM that 2 hidden layers performs the best. As we know before, the model of 2 hidden layers is sufficient enough to perform as a universal regression model to make predictions. Increasing number of hidden layers will also increase the possibility of overfitting in this scenario.

Then we apply different number of hidden units with the fixed 2 hidden layer using the partial spatial block method. When we increase the number of hidden units, it is likely that the neural network is able to learn and predict more efficiently.

Table 4.18. Prediction Qualities vs. Number of Hidden Units Using Partial Spatial Blocks in Test Dataset

Number of Hidden Units	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
256	289.502±15.473	23.898±1.038	0.820±0.009	0.975±0.0013
512	277.584±6.774	24.319±0.458	0.823±0.005	0.979±0.002
1024	258.965±1.153	24.77±0.022	0.834±0.001	0.981±0.000
2048	257.062±1.431	25.522±0.096	0.841±0.001	0.982±0.000
4096	255.113±1.072	25.734±0.092	0.842±0.003	0.982±0.001

Table 4.19. Prediction Qualities vs. Number of Hidden Units Using Partial Spatial Blocks in Validation Dataset

Number of Hidden Units	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
256	431.501±75.678	22.045±1.782	0.814±0.009	0.954±0.003
512	386.553±66.992	23.624±0.630	0.823±0.005	0.977±0.001
1024	354.411±49.497	24.235±0.178	0.828±0.002	0.978±0.001
2048	352.078±28.771	24.375±0.710	0.825±0.001	0.979±0.002
4096	350.745±15.032	25.013±0.554	0.827±0.000	0.980±0.000

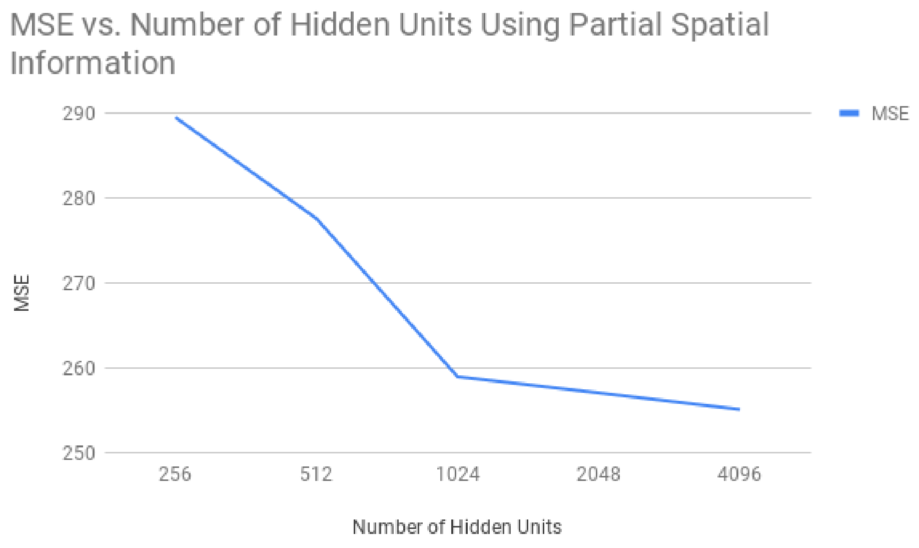

Figure 4.10. The Diagram of MSE vs. Hidden Units

We can see from the above results that when the number of hidden units increases, the qualities of predicted images gradually get better but not so significant in improvements. The best performance is located in the hidden units of 4096 for the neural network.

As we can conclude from the previous results, the partial spatial blocks' method does now work better than the previous series of methods. This phenomenon is probably because our dataset consists of small size images, which contains much less blocks and information compared to the experimental results from others. Therefore, there are not much redundancy within one images, which makes it harder for partial spatial blocks to predict better images.

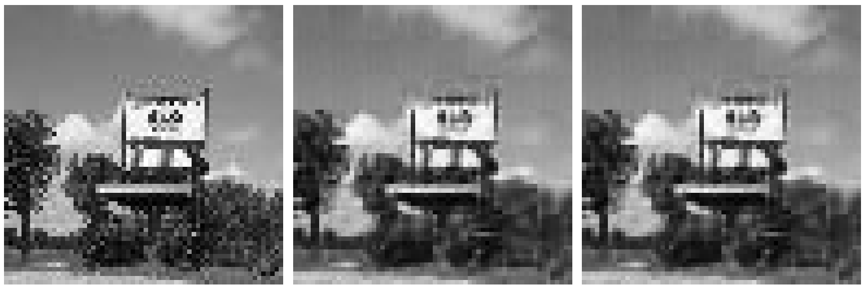

Figure 4.11. Original, JPEG and Reconstructed Images Using Partial Spatial Information Blocks Prediction with 1 Hidden Layer and 1024 Hidden Units

4.3 Stateless Neural Network Using Full Spatial Dependencies

The third section of experiments is another set of experiments using spacial information of image blocks. In this set, we use the contextual spacial information of all surrounding blocks of the one that is going to be predicted. The input image is also divided into 8x8 non overlapping patches and each patch is numbered sequentially from the upper left which starts from the lower right corner of the image. We index the blocks within one image using a 8x8 matrix. We can see from the following diagram that, the artificial neural network takes nine blocks as the input and gives one block as the output for the prediction. $\hat{p}_1$ to $\hat{p}_9$ are given from the JPEG compression and p_5 is the prediction result, which forms the predicted image and compare with the ground true images.

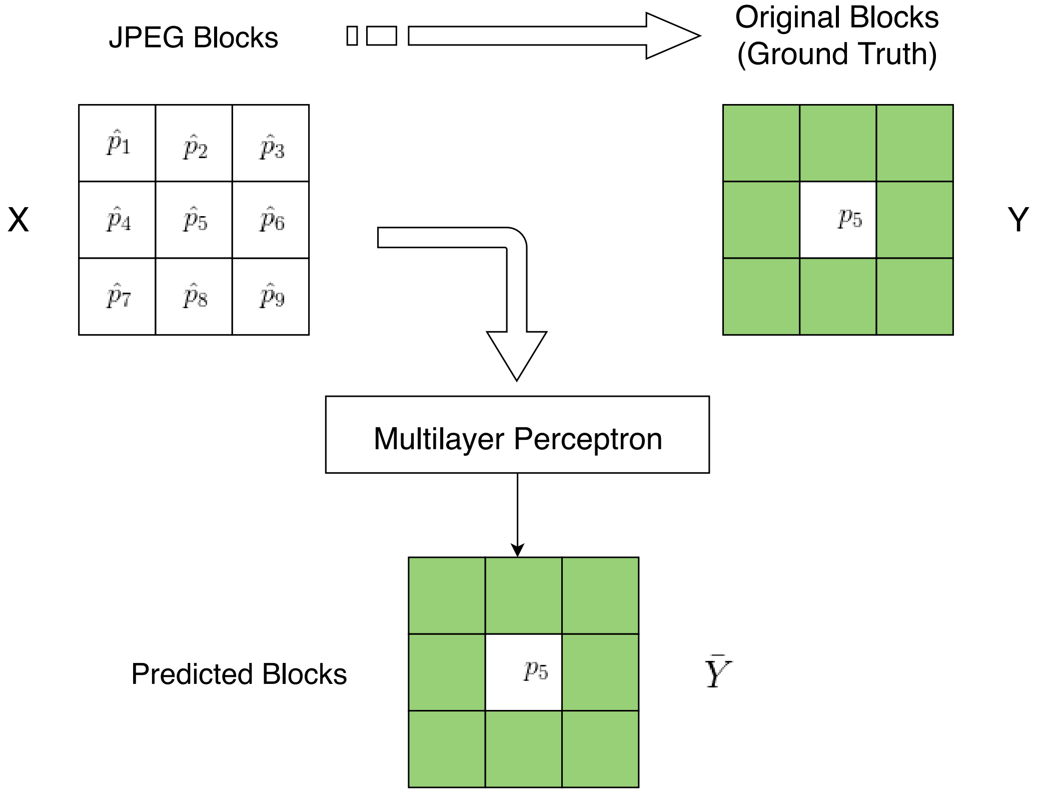
JPEG Blocks
Original Blocks (Ground Truth)
X
$\hat{p}_1$ $\hat{p}_2$ $\hat{p}_3$
$\hat{p}_4$ $\hat{p}_5$ $\hat{p}_6$
$\hat{p}_7$ $\hat{p}_8$ $\hat{p}_9$
p_5
Y
Multilayer Perceptron
Predicted Blocks
p_5
$\bar{Y}$

Figure 4.12. The Diagram of Full Spatial Information Blocks Method

The procedure for extracting pixel blocks is that we use the block (0,0),(0,1),(0,2),(1,0),(1,1),(1, (2,2) to make the prediction patch (1,1). Similarly, the next block collection contains (0,1),(0,2),(0,3),(1,1),(1,2),(1,3),(2,1),(2,2),(2,1) (2,3), which is used to predict (1,2).

To predict a bounding block such as (0,0), we consider the same nine blocks as before: (2,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2 ,1) (2,2). But the model will reconstruct (0,0) this time instead of (1,2). We also use the 9 input patches to reconstruct the image patch at the edge of a image without using zero padding.

Similarly, in the series of experiment of full spatial information prediction, we conduct different optimizer and find the best one. Next we choose the most appropriate learning rate and conduct dropout.

Table 4.20. Full Spatial Prediction Qualities of Different Optimizers in Test Dataset

Optimizer	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
Adam	952.192±75.478	19.123±0.389	0.482±0.009	0.855±0.003
ProximalAdagrad	318.121±1.234	23.511±1.411	0.819±0.003	0.979±0.001
Adagrad	315.013±1.326	24.051±0.020	0.824±0.000	0.979±0.000
Momentum	310.656±33.774	24.836±0.503	0.817±0.011	0.978±0.003
Adadelta	369.222±0.381	23.464±0.016	0.800±0.000	0.974±0.000
RMSProp	1095.387±72.322	18.329±0.327	0.551±0.098	0.812±0.023
Ftrl	376.287±10.012	23.059±0.098	0.801±0.003	0.954±0.002
ProximalGradientDescent	337.521±0.987	23.218±0.121	0.812±0.000	0.979±0.000
GradientDescent	339.802±25.714	23.647±0.843	0.817±0.009	0.977±0.001

Table 4.21. Full Spatial Prediction Qualities of Different Optimizers in Validation Dataset

Optimizer	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
Adam	988.075±70.003	18.965±0.458	0.455±0.010	0.841±0.002
ProximalAdagrad	339.121±1.234	23.289±1.076	0.821±0.002	0.979±0.001
Adagrad	336.457±0.998	23.412±0.018	0.824±0.001	0.979±0.000
Momentum	324.125±25.374	23.675±0.498	0.827±0.011	0.978±0.003
Adadelta	383.666±0.543	23.464±0.016	0.800±0.000	0.969±0.000
RMSProp	1171.912±72.212	18.145±0.289	0.504±0.081	0.811±0.009
Ftrl	388.283±10.103	22.814±0.099	0.799±0.002	0.941±0.001
ProximalGradientDescent	344.145±0.831	23.155±0.098	0.814±0.000	0.980±0.000
GradientDescent	346.586±17.891	23.017±0.732	0.815±0.003	0.977±0.001

From previous results we choose the same optimizer momentum and 0.8 as the dropout ratio. With a fixed number of hidden units we change number of hidden layers to find the optimized hidden layer size. Then we implement increasing hidden units and see the pattern of changing layer units.

First we conduct different hidden layers with a fixed hidden layer size of 1024. We can see that the best performance one is in the 3 hidden layers. With the increasing of hidden layers, we can also see that it gets worse prediction qualities which is more likely to overfit.

Table 4.22. Prediction Qualities vs. Number of Hidden Layers Using Full Spatial Information Blocks Method in Test Dataset

Number of Hidden Layers	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
1	310.656±33.774	24.836±0.503	0.817±0.011	0.978±0.003
2	287.397±16.784	24.324±0.582	0.830±0.011	0.980±0.001
3	286.781±20.505	24.371±0.826	0.832±0.005	0.981±0.001
4	287.641±15.575	24.356±0.745	0.831±0.003	0.981±0.002
5	304.691±1.777	24.174±0.043	0.825±0.002	0.979±0.001

Table 4.23. Prediction Qualities vs. Number of Hidden Layers Using Full Spatial Information Blocks Method in Validation Dataset

Number of Hidden Layers	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
1	468.031±86.030	23.023±0.821	0.804±0.013	0.972±0.003
2	463.783±15.375	23.544±0.921	0.821±0.003	0.973±0.001
3	440.869±25.193	23.665±0.841	0.822±0.002	0.975±0.001
4	450.024±13.798	23.556±0.981	0.821±0.004	0.975±0.000
5	495.500±119.073	23.463±0.251	0.816±0.004	0.973±0.002

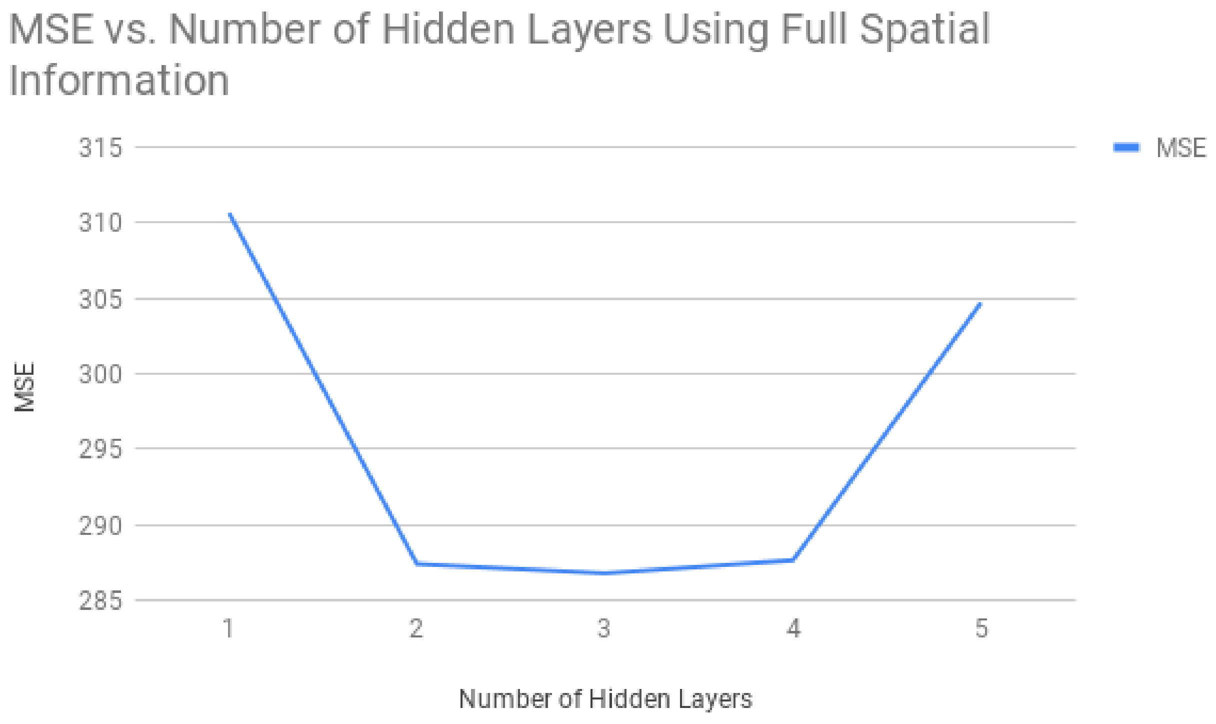

Figure 4.13. The Diagram of MSE vs. Hidden Layers Using Full Spatial Information

Table 4.24. Prediction Quality vs. Number of Hidden Units in Test Dataset

Number of Hidden Units	MSE	PSNR	SSIM	MSSSIM
JPEG Images	281.376	24.440	0.836	0.981
256	804.018±25.805	19.309±1.215	0.573±0.044	0.901±0.034
512	543.801±48.941	21.277±0.965	0.683±0.065	0.945±0.023
1024	286.781±20.505	24.371±0.826	0.832±0.005	0.981±0.001
2048	285.072±15.489	25.011±0.044	0.839±0.001	0.980±0.001
4096	284.155±11.192	24.998±0.009	0.840±0.000	0.982±0.001

Table 4.25. Prediction Quality vs. Number of Hidden Units in Validation Dataset

Number of Hidden Units	MSE	PSNR	SSIM	MSSSIM
JPEG Images	302.034	24.083	0.838	0.981
256	830.544±60.330	19.157±1.217	0.573±0.043	0.901±0.032
512	579.309±57.971	20.966±0.957	0.679±0.065	0.943±0.022
1024	440.869±25.193	23.665±0.841	0.822±0.002	0.975±0.001
2048	437.960±25.784	21.475±0.458	0.834±0.042	0.954±0.065
4096	435.854±17.715	22.037±0.583	0.834±0.009	0.978±0.006

Then we apply different number of hidden units with the fixed one hidden layer. Similarly, when we increase the number of hidden units, it is likely that the neural network is able to learn and predict more efficiently.

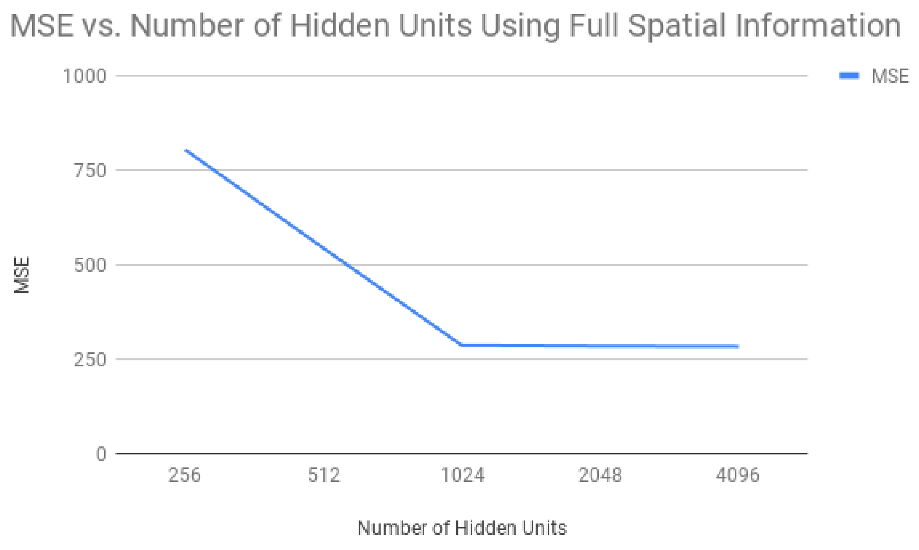

Figure 4.14. The Diagram of MSE vs. Hidden Units Using Full Spatial Information

Similarly, we can conclude that for small images, full spatial information blocks also not work well for providing good prediction qualities. Full spatial information blocks contains more information redundancies than partial spatial blocks and it can perform well with larger resolution image datasets.

In [57], results of that paper indicate that the predictive coding of iterative refinement yields lower distortion compared to JPEG and state-of-the-art GOOG. It conducts the $\Delta - RNN$ in [58] for the predictive coding. In terms of PSNR, it is able to acheive a nearly 0.9 dB gain over JPEG and a 0.6 db gain over GOOG with Kodak image dataset. Results of both MLP and RNN show that they perform than JPEG. It realizes a nonlinear estimator of an iterative decoder via a recurrent neural network that exploits both partial and full spatial statistical dependencies in images. In that scenario, the image dataset is of size 512x512, which makes it convincing that with larger resolution datasets, the artificial neural network is able to generate better reconstruction images using full spatial and partial spatial methods.

Chapter 5 Conclusion and Future Work

5.1 Conclusion

In this thesis, we have implemented and presented an Artificial Neural Network Approach for the encoding of JPEG images. We have shown that JPEG compression can be improved significantly by our approaches of decoding. In those experiments, we have implemented MLP as well as RNN models on the tiny-imagenet dataset. In the first series of experiments, we conduct one-to-one prediction by those two neural network-based methods. Experimental results show the availability and efficiency of the proposed approach in terms of generating predicted images with a relatively less distortion compared with the original images than JPEG images. We have also proposed the causal and non-causal methods aiming for better prediction results. However, these approaches are not very efficient for small images.

We compared our approach to standard JPEG compression and our algorithm is able to perform better at the given bit rate from JPEG. We make use of causal and non-causal statistical redundancies within the image to achieve the non-linear mapping function. In these series of experiments, predictions results are not more effective than the normal one block prediction, which remains as the future work.

5.2 Future Work

In this thesis, we only conduct our methods on small resolution(64x64) image datasets. This remains the future work on larger scale dataset and higher resolution images. Moreover, we focus on one part in the image compression of the decoding part from JPEG images. There are more research work in the whole process in the image compression include lossy and lossless parts. As some techniques in image compression can also be implemented in video compression, future work may also concentrate on neural network based video compression.

Appendix A
Overview of Tensorflow

A.1 Introduction

TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and also used for machine learning applications such as neural networks. [48] is used for both research and production at Google.

A.2 Getting Started

An Estimator is TensorFlow's high level representation of a complete model. It handles the details of initialization, logging, saving and restoring, and many other features so you can concentrate on your model. For more details see Estimators.

An "Estimator" is any class derived from tf.estimator.Estimator. TensorFlow provides a collection of pre-made Estimators (for example, LinearRegressor) to implement common ML algorithms. Beyond those, you may write your own custom Estimators. We recommend using pre-made Estimators when just getting started with TensorFlow. After gaining expertise with the pre-made Estimators, we recommend optimizing your model by creating your own custom Estimators.

To write a TensorFlow program based on pre-made Estimators, we must perform the following tasks:

Create one or more input functions. Define the model's feature columns. Instantiate an Estimator, specifying the feature columns and various hyperparameters. Call one or more methods on the Estimator object, passing the appropriate input function as the source of the data.

Bibliography

[1] Ponlatha, S. and R. Sabeenian (2013) "Comparison of video compression standards," *International Journal of Computer and Electrical Engineering*, **5**(6), p. 549.

[2] Tekalp, A. M. (2015) *Digital video processing*, Prentice Hall Press.

[3] Richardson, I. E. (2004) *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*, John Wiley & Sons.

[4] Taubman, D. and M. Marcellin (2012) *JPEG2000 image compression fundamentals, standards and practice: image compression fundamentals, standards and practice*, vol. 642, Springer Science & Business Media.

[5] Wikipedia contributors (2018), "Image compression — Wikipedia, The Free Encyclopedia," `https://en.wikipedia.org/w/index.php?title=Image_compression&oldid=842049840`, [Online; accessed 19-June-2018].

[6] ——— (2018), "Lempel-Ziv-Welch — Wikipedia, The Free Encyclopedia," `https://en.wikipedia.org/w/index.php?title=Lempel%E2%80%93Ziv%E2%80%93Welch&oldid=837053360`, [Online; accessed 19-June-2018].

[7] ——— (2017), "Entropy encoding — Wikipedia, The Free Encyclopedia," `https://en.wikipedia.org/w/index.php?title=Entropy_encoding&oldid=805743669`, [Online; accessed 19-June-2018].

[8] ——— (2018), "Color quantization — Wikipedia, The Free Encyclopedia," `https://en.wikipedia.org/w/index.php?title=Color_quantization&oldid=843192099`, [Online; accessed 19-June-2018].

[9] ——— (2018), "Comparison of graphics file formats — Wikipedia, The Free Encyclopedia," `https://en.wikipedia.org/w/index.php?title=Comparison_of_graphics_file_formats&oldid=829973127`, [Online; accessed 19-June-2018].

[10] ——— (2018), “JPEG — Wikipedia, The Free Encyclopedia,” `https://en.wikipedia.org/w/index.php?title=JPEG&oldid=846328956`, [Online; accessed 19-June-2018].

[11] ——— (2018), “GIF — Wikipedia, The Free Encyclopedia,” `https://en.wikipedia.org/w/index.php?title=GIF&oldid=846328929`, [Online; accessed 19-June-2018].

[12] ——— (2018), “Portable Network Graphics — Wikipedia, The Free Encyclopedia,” `https://en.wikipedia.org/w/index.php?title=Portable_Network_Graphics&oldid=846328984`, [Online; accessed 19-June-2018].

[13] ——— (2018), “TIFF — Wikipedia, The Free Encyclopedia,” `https://en.wikipedia.org/w/index.php?title=TIFF&oldid=846328544`, [Online; accessed 19-June-2018].

[14] ——— (2018), “JPEG 2000 — Wikipedia, The Free Encyclopedia,” `https://en.wikipedia.org/w/index.php?title=JPEG_2000&oldid=845406400`, [Online; accessed 19-June-2018].

[15] ——— (2018), “Scalable Vector Graphics — Wikipedia, The Free Encyclopedia,” `https://en.wikipedia.org/w/index.php?title=Scalable_Vector_Graphics&oldid=846449315`, [Online; accessed 19-June-2018].

[16] Conklin, G. (2001) “New intra prediction modes,” *document VCEG N*, **54**, pp. 24–28.

[17] Raid, A., W. Khedr, M. El-Dosuky, and W. Ahmed (2014) “JPEG image compression using discrete cosine transform-a survey,” *arXiv preprint arXiv:1405.6147*.

[18] Yu, S.-L. (2002) “New intra prediction using intra-macroblock motion compensation,” in *Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG Meeting, May 2002*.

[19] Shiodera, T., A. Tanizawa, and T. Chujoh (2007) “Bidirectional intra prediction,” *ITU-T SG16/Q*, **6**.

[20] Balle, J. and M. Wien (2007) “Extended texture prediction for H. 264 intra coding,” *ITU-Telecommunications Standardization Sector, Study Group*, **16**, pp. 15–16.

[21] Tan, T. K., C. S. Boon, and Y. Suzuki (2006) “Intra prediction by template matching,” in *Image Processing, 2006 IEEE International Conference on*, IEEE, pp. 1693–1696.

[22] Segall, A. and S. Lei (2005) “Adaptive upsampling for spatially scalable coding,” *Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG JVT-o010. doc.*

[23] Park, S.-W., D. H. Yoon, J.-H. Park, and B.-M. Jeon (2005) “Intra BL prediction considering phase shift,” *Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG*, pp. 1–16.

[24] Wang, Z., J. Liu, Y. Tan, and J. Tian (2006) “Inter layer intra prediction using lower layer information for spatial scalability,” in *Intelligent Computing in Signal Processing and Pattern Recognition*, Springer, pp. 303–311.

[25] Sullivan, G. (1993) “Multi-hypothesis motion compensation for low bit-rate video coding,” in *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, vol. 5, IEEE, pp. 437–440.

[26] Girod, B. (2000) “Efficiency analysis of multihypothesis motion-compensated prediction for video coding,” *IEEE Transactions on Image Processing*, **9**(2), pp. 173–183.

[27] Shaikh, M. A. and S. B. Sagar (2014) “Video Compression Algorithm Using Motion Compensation Technique,” *International Journal of Advanced Research in Electronics and Communication Engineering*, **3**(6), pp. 625–628.

[28] Lucas, L. F. R., E. A. B. da Silva, S. M. M. de Faria, N. M. M. Rodrigues, and C. L. Pagliari (2017) “Prediction Techniques for Image and Video Coding,” in *Efficient Predictive Algorithms for Image Compression*, Springer, pp. 7–33.

[29] MITRA, A. D. and P. K. SRIMANI (1979) “Differential pulse-code modulation,” *International Journal of Electronics Theoretical and Experimental*, **46**(6), pp. 633–637.

[30] Rosário Lucas, L. F., E. A. Barros da Silva, S. M. Maciel de Faria, N. M. Morais Rodrigues, and C. Liberal Pagliari (2017) *Prediction Techniques for Image and Video Coding*, Springer International Publishing, Cham, pp. 7–33.
URL `https://doi.org/10.1007/978-3-319-51180-1_2`

[31] Grois, D., D. Marpe, A. Mulayoff, B. Itzhaky, and O. Hadar (2013) “Performance comparison of h. 265/mpeg-hevc, vp9, and h. 264/mpeg-avc encoders,” in *Picture Coding Symposium (PCS), 2013*, IEEE, pp. 394–397.

[32] Kalva, H. (2006) “The H. 264 video coding standard,” *IEEE multimedia*, **13**(4), pp. 86–90.

[33] Aiazzi, B., L. Alparone, and S. Baronti (2002) "Context modeling for near-lossless image coding," *IEEE Signal Processing Letters*, **9**(3), pp. 77–80.

[34] Baig, M. H., V. Koltun, and L. Torresani (2017) "Learning to Inpaint for Image Compression," in *Advances in Neural Information Processing Systems*, pp. 1246–1255.

[35] Santurkar, S., D. Budden, and N. Shavit (2017) "Generative compression," *arXiv preprint arXiv:1703.01467*.

[36] Toderici, G., D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell (2017) "Full resolution image compression with recurrent neural networks," in *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, IEEE, pp. 5435–5443.

[37] Oord, A. v. d., N. Kalchbrenner, and K. Kavukcuoglu (2016) "Pixel recurrent neural networks," *arXiv preprint arXiv:1601.06759*.

[38] Toderici, G., S. M. O'Malley, S. J. Hwang, D. Vincent, D. Minnen, S. Baluja, M. Covell, and R. Sukthankar (2015) "Variable rate image compression with recurrent neural networks," *arXiv preprint arXiv:1511.06085*.

[39] Theis, L., W. Shi, A. Cunningham, and F. Huszár (2017) "Lossy image compression with compressive autoencoders," *arXiv preprint arXiv:1703.00395*.

[40] Jiang, F., W. Tao, S. Liu, J. Ren, X. Guo, and D. Zhao (2017) "An end-to-end compression framework based on convolutional neural networks," *IEEE Transactions on Circuits and Systems for Video Technology*.

[41] Liang, M. and X. Hu (2015) "Recurrent convolutional neural network for object recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3367–3375.

[42] Stollenga, M. F., J. Masci, F. Gomez, and J. Schmidhuber (2014) "Deep networks with internal selective attention through feedback connections," in *Advances in neural information processing systems*, pp. 3545–3553.

[43] Ballé, J., V. Laparra, and E. P. Simoncelli (2016) "End-to-end optimized image compression," *arXiv preprint arXiv:1611.01704*.

[44] Canziani, A. and E. Culurciello (2017) "Cortexnet: a generic network family for robust visual temporal representations," *arXiv preprint arXiv:1706.02735*.

[45] Li, M., W. Zuo, S. Gu, D. Zhao, and D. Zhang (2017) "Learning convolutional networks for content-weighted image compression," *arXiv preprint arXiv:1703.10553*.

[46] Martens, J.-B. and L. Meesters (1998) “Image dissimilarity,” *Signal processing*, **70**(3), pp. 155–176.

[47] Wang, Z., A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli (2004) “Image quality assessment: from error visibility to structural similarity,” *IEEE transactions on image processing*, **13**(4), pp. 600–612.

[48] contributors, W. (2018), “TensorFlow — Wikipedia, The Free Encyclopedia,” [Online; accessed 10-February-2018].
URL `\url{https://en.wikipedia.org/w/index.php?title=TensorFlow&oldid=823263959}`

[49] Rao, K. and H. Wu (2005) “Structural similarity based image quality assessment,” in *Digital Video image quality and perceptual coding*, CRC Press, pp. 261–278.

[50] Britz, D. (2015), “Recurrent Neural Networks Tutorial, Part 1–Introduction to RNNs,” .

[51] Hochreiter, S. and J. Schmidhuber (1997) “Long short-term memory,” *Neural computation*, **9**(8), pp. 1735–1780.

[52] Srivastava, N., E. Mansimov, and R. Salakhudinov (2015) “Unsupervised learning of video representations using lstms,” in *International conference on machine learning*, pp. 843–852.

[53] (2006) *Mathematics of Control, Signals, and Systems: MCSS.*, v. 18, Springer International.
URL `https://books.google.com/books?id=4RtVAAAAMAAJ`

[54] Nair, V. and G. E. Hinton (2010) “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814.

[55] Ruder, S. (2016) “An overview of gradient descent optimization algorithms,” *arXiv preprint arXiv:1609.04747.*

[56] Zeiler, M. D. (2012) “ADADELTA: an adaptive learning rate method,” *arXiv preprint arXiv:1212.5701.*

[57] Ororbia, A. G., A. Mali, J. Wu, S. O’Connell, D. Miller, and C. L. Giles (2018) “Learned Iterative Decoding for Lossy Image Compression Systems,” *arXiv preprint arXiv:1803.05863.*

[58] Ororbia II, A. G., T. Mikolov, and D. Reitter (2017) “Learning simpler language models with the differential state framework,” *Neural computation*, **29**(12), pp. 3327–3352.