ON-DEMAND VIDEO PROCESSING IN WIRELESS NETWORKS

AN APPLICATION: ACTION RECOGNITION

A Thesis in
Computer Science and Engineering
by
Noor Felemban

© 2016 Noor Felemban

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2016
The thesis of Noor Felemban was reviewed and approved* by the following:

Thomas La Porta  
Evan Pugh Professor and William E. Leonhard Chair Professor, Computer Science and Engineering  
Director, School of Electrical Engineering and Computer Science  
Thesis Advisor

Guohong Cao  
Professor, Computer Science and Engineering  
Director, Mobile Computing and Networking Lab

Chitaranjan Das  
Distinguished Professor, Computer Science and Engineering  
Interim Department Head, Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

With the widespread use of mobile devices with built-in cameras, the number of captured videos has increased. Videos are a rich source of information, and they may be collected and processed for various reasons. Even though mobile devices are becoming more powerful, video processing still remains a resource-hungry process that drains a mobile device’s power and takes a relatively long time to perform locally. Given a network that consists of mobile devices and more powerful video clouds, mobile devices can offload videos to the clouds for faster processing. We aim to minimize the query response time of performing action detection on videos stored across the network, which is an NP-hard problem. We first study the effect of video parameters on QoI. Then we develop two heuristics to solve the problem. The first algorithm is a distributed algorithm, where each mobile device has to make a decision whether to offload a video or process it locally based on its limited knowledge of the network. The other solution is a centralized algorithm where each video cloud appropriately schedules mobile device to offload videos to it. Based on our simulation results both algorithms perform better than the baseline algorithm, and the centralized algorithm is close to the optimal solution.
# Table of Contents

## List of Figures
vi

## List of Tables
vii

## Acknowledgments
viii

## Chapter 1

### Introduction
1

#### 1.1 Motivation
1

#### 1.2 Quality of Information
2

## Chapter 2

### Background and Related Work
4

#### 2.1 Deep Learning and Computer Vision
4

##### 2.1.1 Convolutional Neural Networks
6

##### 2.1.2 Applications Using CNNs
8

#### 2.2 Vision Applications on Mobile Devices
9

## Chapter 3

### Problem Formulation
12

#### 3.1 Overview
12

##### 3.1.1 Caffe
12

##### 3.1.2 Convolutional 3D (C3D)
13

##### 3.1.3 Action Recognition from Videos
14

#### 3.2 System Design and Workflow
16

#### 3.3 Experiments and Parameters
17

##### 3.3.1 Terminology
18

##### 3.3.2 Completeness and Video Resolution
18

##### 3.3.2.1 Setup
19

##### 3.3.2.2 Observations
19
Chapter 4
Algorithms and Simulations

4.1 Algorithms ................................................................. 28
  4.1.1 Distributed Algorithm ............................... 28
    4.1.1.1 Initialization .................................. 31
    4.1.1.2 Offloading Decision ......................... 31
    4.1.1.3 Dealing With Empty Queues .......... 32
  4.1.2 Centralized Algorithm ............................ 32

4.2 Simulations .............................................................. 34
  4.2.1 Parameters ............................................. 34
  4.2.2 Results .................................................. 36

Chapter 5
Conclusion and future work

Bibliography
List of Figures

2.1 LeNet-5 architecture ........................................ 7
2.2 Convolution features visualization ............................ 8

3.1 Vision applications on videos ................................. 14
3.2 C3D architecture .............................................. 14
3.3 C3D feature extraction from videos ........................... 15
3.4 System design .................................................. 16
3.5 Completeness percentage for videos of different scales (1) .... 20
3.6 Completeness percentage for videos of different scales(2) ........ 21
3.7 Scaling 1080p video to different scales ....................... 22
3.8 Processing time on CPU for videos of different sizes ........... 23
3.9 Example of completion time of video clouds .................. 26

4.1 Distributed algorithm: mobile device queues .................. 29
4.2 Centralized algorithm: mobile device queues .................. 33
4.3 Scaling different 1080p videos to 320x180 ..................... 35
4.4 Video sizes before and after scaling ......................... 36
4.5 Varying $|V|$ .................................................... 39
4.6 Varying $|U_d|$ ................................................... 40
4.7 Varying $|U_c|$ ................................................... 41
4.8 Varying $r$ ...................................................... 42
List of Tables

3.1 Variable Definitions ............................................. 17
3.2 Ground Truth vs Detection ...................................... 18
3.3 CPU and GPU processing times ................................. 23
I would like to thank my adviser Dr. Thomas La Porta for his endless support, continues encouragement, valuable advice and insightful criticism. Thank you for making me part of your team and giving me the chance to work and get to know these great people. I would like to thank Dr. Zongqing Lu for helping me throughout this process and always being available when I had any questions or doubts. Thank you Dr. La Porta and Dr. Lu for guiding me throughout my work and for steering me in the right direction whenever I needed it. I would also like to thank Dr. Guohong Cao for serving as a member in my thesis committee, and for reviewing and evaluating my research. Finally, I would like to thank my family for being the best support system I could ask for. Thank you for your endless love and support, words can never express how grateful I am for you.
CHAPTER 1

Introduction

1.1 Motivation

Surveillance cameras, smartphones, hand-held cameras, and tablets are all used to record videos. U.S. News shows that in 2015, 68% of American adults owned smartphones, and 45% owned tablet computers [1]. In 2011, there were more than 30 million surveillance cameras in the U.S alone [2]. With the large spread and easy access to cameras, a large amount of videos are being recorded daily.

Videos are a rich source of information, which can be gathered for various purposes and uses. In many incidents videos were used to identify suspects. For example, analysts used surveillance videos to trace a student’s movement at the University of Virginia when she was reported missing [3]. In the 2013 Boston Marathon Bombing, crowdsourcing was used to collect videos and photos from the public [4].

Crowdsourcing, or requesting information from the public may help escalate the investigation process. Since videos are a good information source, and give a great amount of insight on different situations, collecting, and analyzing them
are common acts. Using human resources to manually go through the videos is inefficient, wasting both time and valuable resources.

Deep learning algorithms based on computer vision have been developed for these tasks. Videos are recorded using mobile devices that have constrained resources, and limited computational power. Meanwhile, video processing is costly and computationally demanding. One might think that using a powerful server running a deep learning algorithm would be a good solution. All the videos could be sent to the centralized server and processed there. However, this method may result in overloading the server causing the aggregate server performance to degrade. Therefore, we propose a system that exploits both the mobile devices and the powerful servers that are GPU-enabled (Graphics Processing Units), known as video clouds, to minimize the completion time of processing after a query is issued.

1.2 Quality of Information

Traditionally, matrices such as throughput, delay and loss are used to evaluate network performance. A recent area of research focuses on evaluating and building networks based on quality of information (QoI) matrices such as correctness, freshness, precision, accuracy, completeness and timeliness [5]. Bahjat et al. [6] define timelessness as the time the information arrives relative to the request time, and completeness as a measure of relevance to the ground truth. For example, if 90 out of 100 videos where classified correctly the correctness would be 90%.

QoI measures the usefulness of the data and its usage for decision making. QoI is affected by both the network and the different processing tasks that take place in the network nodes. For instance, data transmission causes a delay, which makes the data lose freshness and can delay timeliness, and that lowers the QoI. On the
other hand, as mentioned in [5], a node that runs a deep learning algorithm for feature extraction may increase the QoI at the receiver.

Different information queries might be issued such as: "Was there a person riding a bike at 1 am to 2 am downtown?". All videos, related to that query, taken by mobile devices should be processed to recognize the action of "biking". By related to the query, we mean taken in a geographical location close to the region "downtown" during the time frame "1 am to 2 am". Timeliness, would be the time since the query is issued until all related videos are processed. Completeness and accuracy are related to the number of processed videos and their predicted accuracy compared to the truth values.

Depending on the request, QoI requirements, original data quality and resolution, and type of data pre-processing, different methodologies and algorithms can be applied. In this thesis, we design a system that allows crowdsourcing for videos. In our system action recognition is performed either locally on mobile devices or videos are offloaded to video clouds to be processed there. We perform experiments to determine the general cost of performing different parts of the detection process. Based on those results, we form an optimization to minimize the completion time. Since the problem is NP-hard we propose and analyze two heuristics for solving the problem.

The remaining of the thesis is structured as follows: Chapter 2 discusses related work and background on technologies used such as deep learning and cloud offloading. Chapter 3 presents the problem definition that includes the system structure, experimentation results and the optimization problem. Chapter 4 explains the heuristics proposed to solve the optimization problem and the evaluation and analysis of the algorithms. Finally, Chapter 5 concludes the thesis and talks about future work.
CHAPTER 2

Background and Related Work

In this chapter we define deep learning and see how it is used in the computer vision field. We briefly describe convolutional neural networks (CNN), a type of deep learning neural networks. We then discuss some vision applications using CNNs, specifically applications that run on mobile devices. Finally, we discuss a previous research that, similar to our study, uses crowdsourcing and cloud offloading to distribute the video processing application and minimize the completion time of a system.

2.1 Deep Learning and Computer Vision

Deep learning as defined in [7] is ‘a class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification’. There are many active research fields in deep learning such as speech analysis, text processing, vision applications (image and video), and bioinformatics. Deep
learning is an end-to-end process that helps learn features. As the definition states, deep learning algorithms are a chain of layers, each layer or stage passes features to a higher level that learns more complicated features.

Computer vision is a field of study that aims to understand and infer the state of the real world based on the visual data (e.g. images, videos). Three main components are needed to solve a vision problem: a mathematical model that relates the input data and the world state, a learning algorithm, and an inference algorithm that draws an inference from the data to the real world, such as inferring a person identity from an observed image [8].

Learning algorithms are mappings between inputs and outputs. Different classes of learning algorithms exist. Two main types are supervised, and unsupervised learning. With **Supervised learning**, the set of training data is labeled. This method is the most common in classification problems. With **unsupervised learning**, the training data is not labeled, making it more difficult to evaluate the performance of an algorithm [9].

Early stages of computer vision algorithms use linear filters, convolution and gradients on single images to extract local features like edges, corners, blobs and bars. Using more than one camera, or having multiple views of a scene, enables the inference of 3-D scenes. In middle stages, segmentation and grouping take place. Finally, applications such as classification, object detection and recognition occur in higher levels [10].

To build a vision system, data (images or video) must be captured, and an algorithm must interpret the captured data. With the new hardware technologies and their wide spread use, it is easy to capture high resolution images and videos. The more critical part is the design and implementation of the learning algorithms. Despite the fact that processing speeds and storage capabilities on computers are
improving, vision problems that include processing high resolution clips or images still remain a challenge. This field of active research became more attractive with the recent improvements and implementations enabling the use of GPUs.

2.1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of multilayer neural network, trained by back-propagation. CNN for vision applications aim to extract image patterns from raw pixels with minimum preprocessing while avoiding handcrafted algorithms. A CNN consists of multiple convolutional, RELU (Rectified Linear Units), pooling and fully connected layers. There are two different processes, a forward pass and back-propagation. The forward pass simply takes an input and produces an output based on the functionality and computation of the layers. Back-propagation is basically the chain rule, computing the derivatives, with the goal of minimizing the loss/error function by updating the weights of the network.

Applying machine learning techniques in the vision field is not a new technique. LeCun [11] in 1998, proposed a multilayer neural network, LeNet-5, for character recognition. He mentioned that the traditional pattern recognition model that is divided into a feature extraction module and a classifier is problematic, because the feature extraction module is specific to the task and does not scale well. During feature extraction the input is transformed into a vector of features, usually numerical, that represents the input and can be processed and compared easily. A classifier takes the feature vector as input and produces class scores. The development of powerful machine learning techniques, the improvement of arithmetic units and the availability of large labeled datasets enabled researchers to move from this old handcrafted feature extraction module to using machine learning. Figure 2.1
from [11] is the architecture of LeNet-5, showing the traditional architecture of a convolutional neural network.

Figure 2.1: LeNet-5 architecture

Stanford’s Class on ‘Convolutional Neural Networks for Visual Recognition’ [12] is a great source for more details on CNNs. Briefly describing the CNN layers, we have the convolutional layer which is the main building block of a CNN and where most of the computations take place. The idea is to use convolution filters (kernels) of size $w \times h \times d$, where $w$ is the width, $h$ is the height and $d$ is the depth, or the number of channels. For example, in a RGB image $d = 3$. In the forward pass, the result of convolving (dot product) the filter with the input would give us a 2D activation, sometimes called a feature map. For each convolution layer in the network, there is a set of learnable convolution filters, activated based on specific features, such as edges, colors, contours and objects depending on the level and depth of the layer as shown in Figure 2.2 from [13]. The output of the convolution layer would be a $w \times h \times f$ volume, where $f$ is the number of filters.

RELU is simply a threshold function at zero, more known as the activation function $f(x) = \max(0, x)$. Pooling layers are used for down-sampling, where methods such as max-pooling and mean-pooling are used. Finally, fully connected layers, such as softmax, are the decision makers. For example, in a classification problem, the last layer (softmax) is used to compute class scores.
A neural network takes a large set of labeled data called the training set, runs them through the network, and the network will learn how to recognize specific patterns based on the training set. The larger the training set the more the network will be able to learn. Therefore, large labeled data sets are essential to improve accuracy. One of the most known large datasets for images is ImageNet [14] that, as of 2014, includes 14,197,122 annotated images.

2.1.2 Applications Using CNNs

A digital image is an array of pixels, or numbers representing intensity levels. A video is a series of digital images called frames. Frames can be extracted from videos for processing. Applications such as image classification, object detection and action detection have been studied, and several algorithms and methods using CNNs have been proposed and developed.

Krizhevsky et al. [15] optimized GPUs for running 2D convolutions and developed a network that yields state of the art performance on classification tasks. Other problems such as object detection and localization are solved by using CNNs.
State of the art results have been shown using CNN for these problems in [16].

Many datasets have been collected for action recognition. Some are image sets such as PASCAL VOC 2012, while others are video sets such as UCF-101 [17] and HMDB-51 [18]. Action recognition relies on recognizing specific poses, and interactions with objects to identify an action. In [19], the authors used transfer learning from one data set to another, reusing layers trained on ImageNet, and tested it on smaller datasets. For the action task, they tested on PASCAL VOC 2012, and achieved higher accuracy on the dataset.

2.2 Vision Applications on Mobile Devices

With the spread of mobile devices, people are actively taking videos and capturing photos. Several applications have emerged and researchers have been actively investigating this area of study. Many computer vision libraries have been developed for mobile devices, such as Starfish [20] and Poster [21], to speed the vision processing computation and decrease power consumption. Many applications using image processing features have been developed on smartphones, such as TagSense [22] for automatic tagging, and CrowdSearch [23], an application that uses human validation for image search.

Mobile crowdsensing defined by [24] as a “sensing paradigm that empowers ordinary citizens to contribute data sensed or generated from their mobile devices” helps in collecting more content relevant to a query. MoVi [25] is a social application for video highlights; it filters events according to social relevance. Based on that, it enables the extraction of the best video view of an event. CrowdSense@Place [26], uses captured images from smartphones to create an application that classifies locations.
As mentioned previously, image and video processing are resource-intensive services. Attempting to process videos and images on mobile devices, even if possible, would require a long time, compared to more powerful servers, and consume a great deal of power. Therefore, cloud offloading was introduced as a solution to this problem in an area known as Mobile Cloud Computing (MCC) [27], that aims to solve mobile devices’ storage, speed and power limitations by utilizing cloud resources in an on-demand fashion.

Many research efforts on cloud offloading have been undertaken such as Odessa [28], an application for offloading decisions for interactive perception applications. eTime [29], uses knowledge of network connectivity to prefetch frequently used data when the connection is good to balance the energy consumption and transmission delay. In [30] the authors propose a framework for mobile stream applications, with the goal of maximizing throughput. Other systems such as MAUI [31], Thinkair [32], Clonecloud [33] and Cuckoo [34] are designed to offload code or computation from mobile devices to enhance the performance.

Lu et al. [35] take another approach, and consider optimizing on-demand video processing latency for a complete system that consists of mobile devices and video clouds. Two algorithms are proposed to minimize the maximum completion time of the nodes in the system. After a query is made, the decision of offloading is based on the number of videos, the processing speeds of each node, and the transmission delays. The first algorithm makes the greedy choice of reallocating a video from the mobile device that has the maximum completion time to a cloud in a manner that minimizes the increasing time in the selected cloud. The second algorithm is an online algorithm that considers the dynamic changes of transmission rates in the network. This paper considers QoI metrics for performance evaluation rather than quality of service (QoS).
We look at a similar optimization problem, but rather than considering object detection we consider action detection. From our experimental results we realize that decreasing the resolution of videos resolution to a certain level decreases the transmission time of videos while only incurring a small loss in QoI.
Problem Formulation

In this chapter we give an overview of Caffe, the deep learning framework we use for action recognition. We then provide a brief overview of the workflow of video processing and perform experiments to show how different video parameters effect QoI. We also examine the time required to preform different parts of the processing task. Based on these results, we define our system and formulate the optimization problem.

3.1 Overview

3.1.1 Caffe

BVLC caffe [36] is an open source deep learning framework developed at Berkeley Vision and Learning Center. The authors aimed to design a toolkit for state of the art deep learning algorithms. Caffe supports both CPU and GPU computations, and is suitable for use on different platforms and machines.

The components of the framework are blobs, layers and nets. Blobs are N-
dimensional arrays acting as wrappers over data. Layers are the building blocks of
the neural network. They take a blob as an input (bottom blob) and produce an
output (top blob). Each layer defines the forward and backward pass computations.
Caffe has a catalog of layers that includes, but is not limited to, convolution,
pooling, local response normalization (LRN) and softmax. Finally, the Net is the
set of layers and their connections. Layers are connected in a directed acyclic
graph (DAG), and are defined in plain-text modeling language. The Caffe model
architecture is defined in a protocol buffer definition file. Different examples and
predefined network architectures can be found on Caffe’s website.

For training, Caffe uses the stochastic gradient descent algorithm. For a
classification problem the network usually starts with a data layer that takes
input images and labels, and ends with the loss and gradients produced by the
classification loss layer.

Many researchers use Caffe for different applications such as object detection
and segmentation in [37] and [38], text recognition in [39] and action recognition
in [40], [41] and [42].

### 3.1.2 Convolutional 3D (C3D)

As mentioned in the previous chapter some action recognition approaches rely on
processing single images, or frames extracted from video clips, and use them as
inputs to the CNN. This process leads to the loss of the temporal dimension. 3D
CNNs [43] aim to leverage the motion information in consecutive frames for better
action recognition. Instead of applying a \( w \times h \times d \) kernel on a 2D frame, they
apply the kernel to a volume of stacked frames, as shown in figure 3.1.

Tran et al. [42], go a step further and use video clips as inputs, without any
preprocessing or frame extraction. They note that 2D convolutions, whether applied to single images or a stack of images (treated as channels as in [44]) will result in a 2D output (feature map). However, 3D convolution will result in an output volume. The same idea applies to other layers such as 3D pooling. The architecture of C3D is shown in Figure 3.2. The numbers underneath each layer indicates the number of filters, and both convolution and pooling layers are 3D. The CNN is trained on the Sports-1M [45] dataset. Sports-1M is a large video dataset that consists of 1 million YouTube videos from 487 classes. The classes are all sports related activities.

![Figure 3.1: Vision applications on videos](image)

We will use the C3D code (a modification of caffe) for feature extraction from videos for action recognition. Both the code and pre-trained model are available on their website.

### 3.1.3 Action Recognition from Videos

C3D’s default settings extract features from 16-frame video clips. For example, if the input video is 10 seconds and has a frame rate of 30 fps, the number of extracted features would be 18, according to (3.1).
\[ \text{num\_of\_features} = \left\lfloor \frac{\text{frame\_rate}[fps] \times \text{vid\_duration}[sec]}{16[frames]} \right\rfloor \] (3.1)

Figure 3.3 shows a flow chart for feature extraction from videos using C3D. In the paper, the authors studied the type of patterns that are distinguished in each level of the network. They found that early convolution layers learn low level moving patterns such as color changes and moving edges, mid level layers learn features such as textures and corners, while in higher level layers, motion patterns such as moving objects are learned. C3D is reported to achieve a 61.1% accuracy on top-1 and 85.2% on top-5 for the classification task on Spots-1M video clips. Top-1 only considers the class with the highest probability of matches, and top-5 allows the correct label to be within the top five probabilities.

![Flow chart for feature extraction from videos using C3D](image)

Figure 3.3: C3D feature extraction from videos

The CNN is as described in Figure 3.2.
3.2 System Design and Workflow

Our system consists of a set of mobile devices and video clouds, as shown in figure 3.4. Our goal is to identify actions in the videos in response to a query, in the minimum amount of time, given a target completeness requirement. We define completeness to be the number of videos with the target action correctly identified out of all the videos with the target action.

After a query is initiated, videos that are in mobile devices are scaled to the minimum resolution that meets the completeness requirement. Then a choice of local processing or offloading is made.

To understand the solution space, we first study the effect of video parameters on QoI. We realize a trade-off between video resolution and QoI. As we reduce the resolution of a video by scaling, the faster it is to transmit it to the cloud, but the
less completeness we achieve. The following experiments quantify these relations.

Given that the system aims to minimize the completion time, we closely study the different delays in the system.

In the remaining of this chapter we precisely define our terminology, present experimental results to understand how the characteristics of a video impact the completeness and timeliness. Based on the experimentation results, we define our problem and model the system.

Table 3.1 defines the variables used in the remaining of the thesis:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>Set of videos stored in the network</td>
</tr>
<tr>
<td>$V_i$</td>
<td>$V_i \subset V$ set of videos stored in node $i$</td>
</tr>
<tr>
<td>$U$</td>
<td>Set of nodes in the network $U = U_d \cup U_c$</td>
</tr>
<tr>
<td>$U_d$</td>
<td>Set of mobile devices in the network</td>
</tr>
<tr>
<td>$U_c$</td>
<td>Set of video clouds in the network</td>
</tr>
<tr>
<td>$L$</td>
<td>Set of links in the network</td>
</tr>
<tr>
<td>$L_i$</td>
<td>$L_i \subset L$ Set of links connecting mobile device $i$ to video clouds</td>
</tr>
<tr>
<td>$T_{comp}$</td>
<td>The time the last node in the system completes processing</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Completion time of node $i$</td>
</tr>
<tr>
<td>$p_{a,i}$</td>
<td>Processing time of video $a$ at node $i$</td>
</tr>
<tr>
<td>$c_{a,i}$</td>
<td>Time from the when the query is issued until video $a$ is received at cloud $i$</td>
</tr>
<tr>
<td>$d_{a,i}$</td>
<td>Transmission delay video $a$ incurs when being transmitted to node $i$</td>
</tr>
<tr>
<td>$t_{scale,i}$</td>
<td>Scaling time for all videos initially in node $i$</td>
</tr>
<tr>
<td>$q_{a,i}$</td>
<td>Queuing delay of video $a$ in node $i$</td>
</tr>
</tbody>
</table>

Table 3.1: Variable Definitions

### 3.3 Experiments and Parameters

For the experimentation, a Dell Precision T7910 with NVIDIA GTX TITAN X 12 GB GPU was used as a video cloud. To model a mobile device we use the CPU in the Dell by activating the CPU flag in the C3D feature extraction tool.
<table>
<thead>
<tr>
<th>Detector</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Positive</td>
</tr>
<tr>
<td>Positive</td>
<td>True Positive (TP)</td>
</tr>
<tr>
<td>Negative</td>
<td>False Positive (FP)</td>
</tr>
</tbody>
</table>

Table 3.2: Ground Truth vs Detection

3.3.1 Terminology

- **Processing time**: The time it takes for the video descriptors to be extracted using C3D.

- **Communication delay**: The time from when the query is initiated until a video is received at a video cloud.

- **Transmission delay**: The time it takes for a video to be transmitted from a mobile device to a cloud.

- **Scaling time**: The time it takes for a video to be scaled from its original resolution (1920×1080 on smartphones) to the desired resolution that fulfills the completion requirement.

- **Completeness**: The percentage of true positives (TP), as shown in table 3.2.

- **Threshold**: The minimum probability score for a class, or action, for it to be marked as present.

3.3.2 Completeness and Video Resolution

First we study the effect of video resolution on the completeness QoI metric. Our small experimentation dataset consists of 155 ten-second video clips from the testing
set of Sports-1M. From that set, 63 videos contain biking-related activities, and 92 are random videos that do not contain any biking-related activities. Sports-1M contains 13 variations of biking-related activities: cycling, unicycle, mountain unicycling, bicycle, bmx, freestyle bmx, cyclo-cross, cross-country cycling, road bicycle racing, track cycling, downhill mountain biking, freeride and dirt jumping.

### 3.3.2.1 Setup

To prepare the data set, we do the following:

1. Download the videos using youtube-dl [46], a free command line tool for downloading YouTube videos.

2. Crop each video to a 10 second clip using ffmpeg [47]

3. Produce clips with different resolutions using the scale filter in ffmpeg. We start with the following resolutions: 640×320, 544×306, 480×270, 320×180, 256×144, 160×90, 128×72, 96×54 and 64×36.

4. Pass the 1,395 (155 videos × 9 scales) clips through the C3D network and extract ‘.prob’ video descriptor

5. Classify video clips based on top-1 score. C3D is used with its default settings, i.e. sampling step size is 16-frames, Sports-1M pre-trained model, and mini batch size of 50.

### 3.3.2.2 Observations

Figure 3.5 shows the completeness percentage after processing the 1,395 clips, based on the top 1 probability. The plot displays the percentages for five different thresholds. Given the way softmax is designed, even if none of the actions in the
pre-defined classes take place in a video, a class will still be given as a result but with a very small probability. Therefore, choosing a very low threshold such as 0 or 0.1 would not be a good choice. The threshold is as defined in section 3.3.1.

Figure 3.5: Completeness percentage for videos of different scales (1)
x-axis labels are as follows: 1-64x36 2-96x54 3-160x90 4-128x72 5-256x144 6-320x180 7-480x270 8-544x306 9-640x320

From the plot we realize that scaling to larger scales (i.e. 640x320, 544x306, 480x270) would not give us an additive advantage regarding QoI. Thus, we can choose to scale to smaller resolutions such as 320x180 saving on processing time and transmission time while still obtaining approximately the same QoI. On the other hand, a very small resolution such as 64x36 gives a very low completeness percentage. Therefore, for our problem we will focus more closely on the smaller resolution range from [320x180 - 96x54].

Figure 3.6 shows the completeness percentage for threshold =0.3 with the new range of resolutions. As shown in the figure, the data can be fit to a non-linear curve (rectangular hyperbola) with the following equation:
3.3.3 Time Delays

3.3.3.1 Scaling Delay

Second, we study the scaling time of videos taken by smartphones at a default resolution of 1080p to the different scales that we consider in our problem. In Figure

\[ \text{completeness} \% = \frac{a \times \text{resolution}}{b + \text{resolution}} \]

Where \( a \) and \( b \) are constants, in this specific case \( a = 89 \) and \( b = 1495 \). The constants change based on the threshold. We note that the completeness equation varies depending on the action that is detected, the vision algorithm used, and the dataset. For this case, this plot demonstrates our experimental results for the dataset specified above.
3.7, the numbers on the scale-axis (x-axis) represent the following resolutions: 1-64x36 2-96x654 3-128x72 4-160x90 5-192x108 6-224x126 7-256x144 8-288x162 9-320x180. We realize that resizing to a smaller resolution (e.g. 64x36) costs less time than scaling to a larger resolution (e.g. 320x180). However, smaller resolutions as shown in the previous section give us lower completeness percentages. We can conclude from this that smaller resolution videos have lower transmission delays, and are faster to scale to, but as a trade-off give us lower completeness percentages. Once we reach a very low resolution the completeness percentages are very bad they are not useful.

![Figure 3.7: Scaling 1080p video to different scales](image)

### 3.3.3.2 Processing Delay

Third, the processing time using C3D is studied. C3D defines a parameter called mini-batch-size, which is the number of 16-frame clips, as shown in figure 3.3. Using the default setting of C3D with a mini-batch-size of 50, we note that the number-of-mini-batches is related to the number of extracted features from the input video.
For example, if 50 or less features are extracted then the number-of-mini-batches equals one, if 100 features are extracted the number-of-mini-batches equals two, however if 101 features are extracted the number-of-mini-batches equals three. This is mentioned because we will see how this parameter effects the time for feature extraction.

Using a sample (46 sec) video captured by an iphone, 11 videos of duration 10, 16, 21, 26, 31, 36, 41 and 46 seconds were segmented. Figure 3.8 shows the processing (feature extraction) time for these videos on the CPU. Table 3.3 shows the comparison between the processing times of the CPU and GPU.

![Figure 3.8: Processing time on CPU for videos of different sizes](image)

<table>
<thead>
<tr>
<th>Video size [kB]</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length [sec]</td>
<td>10</td>
<td>16</td>
<td>21</td>
<td>26</td>
<td>31</td>
<td>36</td>
<td>41</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU [sec]</td>
<td>51.718</td>
<td>52.800</td>
<td>51.646</td>
<td>51.799</td>
<td>98.065</td>
<td>97.955</td>
<td>97.866</td>
<td>97.294</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: CPU and GPU processing times
From Table 3.3 we can find a relation between the processing time using CPU and GPU. The average processing time for videos with mini-batch of size 1 is 52 seconds on the CPU and 8 seconds on the GPU. For videos with mini-batch of size 2, the average processing delay is 92 seconds on the CPU and 14 seconds on the GPU. On average, the GPU is approximately 7x faster than the CPU.

### 3.4 Problem Definition

The system is a wireless network that consists of a number of mobile devices and video clouds, with different processing rates. Nodes are able to communicate via wireless links. Initially, videos are stored on any node in the network. Slower nodes, mobile devices, have low processing rates, hence the processing delay in those devices is large. Slower nodes are able to offload videos to more powerful nodes, video clouds, for faster processing. However, a communication delay will occur.

Video processing is computationally intensive and cannot be processed in parallel even on GPU, thus each node processes videos in a serialized manner. When a video is being processed, other videos will be buffered for later processing until resources, CPU or GPU, are available. We assume videos are captured by mobile devices and have the same initial resolution and frame rate. Videos can be scaled to decrease their size, and therefore decrease both communication and processing time. Preprocessing such as video fragmentation (clipping), filtering based on timestamps, and geo-filtering can take place but are not considered in this model. In this problem offloading between mobile devices is not considered. Only devices with lower processing rates offload to machines with higher processing rates. Scaling is only done on mobile devices, since the main goal of scaling is to decrease the
communication time.

Many recent video processing algorithms require the availability of the entire video clip before processing can occur; therefore, videos can only be processed after being fully received by the node. If node $i$ is transmitting more than one video to node $j$, the videos should be sent sequentially to enable the receiving node to start processing received videos faster, and node $j$ should receive videos in a serialized manner to minimize the completion time of a single video, i.e. one video should not incur additional delay sharing the transmission channel with another video.

### 3.4.1 Model and Notations

The objective is to minimize the query response time given a completeness requirement. *Completeness* is, as defined previously, the percentage of true positives. To minimize the time, we need to process each video exactly once on one node in the network.

Let $V$ be the set of the videos stored in the network that are related to the query, and $U = U_d \cup U_c$ is the set of nodes in the network. $U_d$ is the set of mobile devices, and $U_c$ is the set of video clouds or servers that have GPU support. The completion time $T_{\text{comp}}$ is the time the last node in the system takes to complete processing all its assigned videos. The assigned videos for a video cloud are the videos initially stored there, in addition to the videos that are offloaded to it from mobile devices. The assigned videos for the mobile devices are the remaining videos after offloading to the clouds. If $T_i$, $i \in U$ is the completion time of node $i$ then $T_{\text{comp}} = \max_{i \in U} T_i$. 
3.4.1.1 Completion Time

Let $p_{a,i}$ be the processing time of video $a$ at node $i$. Note that the processing delay varies depending on the processing algorithm and the type of features or outputs that are extracted. For a specific application, machines with the same processing rates should have approximately equivalent processing times. Let $V_i$ be the set of videos in node $i$, $c_{a,i}$ is the communication delay, and $t_{scale,i}$ is the scaling time for all videos initially in $i$. Similar to [35], in order to capture the sequential transmission of videos, and to account for the scaling time in mobile devices, the completion time can be modeled as shown in figure 3.9.

![Figure 3.9: Example of completion time of video clouds](image)

for cloud $i$, $T_i = c_{b,i} + p_{b,i} + p_{c,i}$

The completion time for node $i$ can be modeled as:

If $i$ is a mobile device:

$$T_i = t_{scale,i} + \sum_{a \in V_i} p_{a,i}$$

If $i$ is a video cloud:

$$T_i = \max_{a \in V_i} (c_{a,i} + \sum_{b \in V_i} \alpha_{a,b,i} \cdot p_{b,i}),$$

where $\alpha_{a,b,i} = 1$, if $c_{b,i} \geq c_{a,i}$ and 0 otherwise.
For the example in figure 3.9:

\[ T_i = \max(c_{a,i} + p_{a,i} + p_{b,i} + p_{c,i}, c_{b,i} + p_{b,i} + p_{c,i}, c_{c,i} + p_{c,i}) \]

\[ = c_{b,i} + p_{b,i} + p_{c,i} \]

The completion time can be generalized as:

\[ T_i = \max_{a \in V_i}(c_{a,i} + t_{scale,i} + \sum_{b \in V_i} \alpha_{a,b,i} * p_{b,i}) \] (3.2)

where \( c_{a,i} = 0 \) on mobile devices, and \( t_{scale} = 0 \) in video clouds.

### 3.4.1.2 Optimization Problem

Suppose \( x \) is a solution from the solution space for the scheduling problem to minimize \( T_{comp} \). The problem can be formulated as:

\[
\min_{i \in U} \max_{a \in V_i(x)} (c_{a,i}(x) + t_{scale} + \sum_{b \in V_i(x)} \alpha_{a,b,i} * p_{b,i}) \\
\text{s.t. } \alpha_{a,b,i} = 1 \text{ if } c_{b,i} \geq c_{a,i}, \text{ otherwise } 0, \\
\forall i, j \in U, \forall a, b \in V_i(x)
\] (3.3)

The problem formulation is close to [35], and is a processing scheduling problem. The problem is NP-hard by reduction to machine scheduling. Given the complexity of finding an optimal solution for the problem, we propose two heuristics that are detailed and evaluated in the next chapter.
To solve the optimization problem we propose two heuristic methods, a distributed and a centralized algorithm. The details of the heuristics are discussed below, followed by the results of the simulation to compare the performance of these methods with the optimum.

4.1 Algorithms

4.1.1 Distributed Algorithm

In the distributed method each mobile device has no knowledge of the channel conditions or the state of other nodes. Mobile devices use probing to estimate the transmission rates and the queue backlog of video clouds. After scaling, each mobile device makes an independent decision of whether to offload a video or process it locally.

Let $V_i \subset V$ be the set of videos in node $i \in U_d$. Each mobile device in the system is connected to a number of video clouds via different links. Each video in
the mobile device is associated with two tasks: scaling and processing.

Each mobile device \( i \in U_d \) has three queues as shown in Figure 4.1. The scaling queue \( (Q_{scale}) \) originally holds the local videos that are related to the query for scaling locally. A second queue \( (Q_{loc}) \) has the scaled videos to be processed locally on the device. Finally, the offload queue \( Q_{offload} \), that holds the videos waiting to be offloaded to the cloud. \( Q_{offload} \) is connected to a number of channels/links, one channel for each video cloud to which it is connected. Each video cloud has a single queue \( Q_{GPU} \) that holds videos that are waiting to access the GPU.

![Figure 4.1: Distributed algorithm: mobile device queues](image)

After a query is initiated, all the videos related to the query are enqueued into \( Q_{scale} \), and scheduled to be scaled to the minimum resolution that satisfies the completeness requirement. After each video is scaled, the decision of local processing (enqueuing to \( Q_{loc} \)) or offloading to a server (enqueuing to \( Q_{offload} \)) is made.

Since both scaling and local processing take place in the CPU, enqueuing to \( Q_{loc} \) is a function of the size of the queue and the time required for \( Q_{scale} \) to be empty (i.e. all the videos have been scaled). Enqueuing to \( Q_{offload} \) is a function of the size the queue, the size of the video that we want to make the decision for, and the processing round trip time \( (prt) \). \( prt \) is the total time from the first video byte to be transmitted from \( Q_{offload} \) until an acknowledgment \( (ack) \) is received, which
includes the queuing time at the server and the processing time in the GPU.

If \( d_{a,j} \) is the transmission delay for video \( a \) to reach node \( j \), \( q_{a,j} \) is the queuing delay at video cloud \( j \) until video \( a \) gets access to the GPU and \( GPUp_{a,j} \) is the GPU processing time for video \( a \) in node \( j \), then:

\[
prt_{a,j} = d_{a,j} + q_{a,j} + GPUp_{a,j}
\]  

(4.1)

let \( L \) be the set of links in the system, \( L_i \subset L \) is the subset of links connecting mobile device \( i \) to the video clouds, and \( l \in L_i \) is the link connecting node \( i \) to a specific cloud. \( |L_i| \) is the number of video clouds that mobile device \( i \) is connected to. Since videos vary in size, we can normalize the value of the processing round trip time by dividing it by the size of the video.

\[
prt_{i,l} \ [\text{msec/byte}] = \frac{prt_{a,j} \ [\text{msec}]}{\text{size}(a) \ [\text{bytes}]},
\]

(4.2)

where \( prt_{i,l} \) is the \( prt \) value of link \( l \) in mobile device \( i \), the source node (mobile device) is \( i \) and the destination node (video cloud) is \( j \). Let \( prt_{i \in U_c} \) be the average \( prt \) value for all the links, as defined in 4.3, which is updated after an \( ack \) is received. We use the average because over time individual \( prt_{i,l} \ in L_i \) values change, but we assume that the average remains almost constant. Later on, when the video is at the head of the queue we make the choice of to which cloud to offload. From now on, \( prt_i \) for a mobile device will indicate the average processing round trip time for all the video clouds to which a mobile device is connected.

\[
prt_i = \frac{\sum_{l \in L_i} prt_{i,l}}{|L_i|}
\]

(4.3)

Each node can calculate an estimation of a video’s completion time if offloaded
according to this equation:

\[ ctime_{a \in V_i} = prt_i \text{[msec/byte]} \times \text{size(a)}, \]  

(4.4)

where \( ctime_{a \in V_i} \) is the approximate completion time for a video \( a \) if offloaded, including the communication, queuing and processing times.

### 4.1.1.1 Initialization

Initially the \( prt_{i \in U_d} \) for all mobile devices is infinity. Assuming a mobile device is connected to \( n \) video clouds, \( n \) videos are scaled by the CPU and offloaded to each cloud to determine initial values for \( prt \).

### 4.1.1.2 Offloading Decision

After a video is scaled, a decision of whether to offload the video or process it locally is made. Recall that \( i \in U_d \) is a mobile device, \( p_{a,i} \) is the processing time of video \( a \) locally in node \( i \), and \( t_{scale,i} \) is the time until all videos are scaled. Let \( ctime_a \) be the estimated completion time of video \( a \) if offloaded to a video cloud.

For a currently scaled video \( a \in V_i \), the following is calculated:

\[
\begin{align*}
\text{cost}_{\text{loc}} &= t_{\text{scale},i} + p_{a,i} + \sum_{b \in Q_{\text{loc}}} p_{b,i} \\
\text{cost}_{\text{off}} &= ctime_a + \sum_{b \in Q_{\text{offload}}} ctime_b
\end{align*}
\]  

(4.5)

(4.6)
Algorithm 1 Offloading decision

if \( \text{cost}_{\text{loc}} < \text{cost}_{\text{off}} \) then

add the video to \( Q_{\text{loc}} \)

else

add the video to \( Q_{\text{offload}} \)

end if

For a video \( a \) in \( Q_{\text{offload}} \), when it reaches the head of the queue, it is offloaded to the cloud that currently has the lowest \( \text{prt}_{i,t} \).

4.1.1.3 Dealing With Empty Queues

If either \( Q_{\text{loc}} \) or \( Q_{\text{offload}} \) become empty before the other, we test and relocate videos to optimize for the total query response time as following, assuming \( a \in V_i \) is the video at the end of the non-empty queue.

Algorithm 2 Relocation

if \( Q_{\text{off}} \) is empty and \( c\text{time}_a < \sum_{b \in Q_{\text{loc}}} \text{p}_b \) then

move \( a \) to \( Q_{\text{offload}} \)

end if

if \( Q_{\text{loc}} \) is empty and \( \sum_{b \in Q_{\text{offload}}} \text{prt}_b < \text{p}_{a,i} \) then

move \( a \) to \( Q_{\text{loc}} \)

end if

4.1.2 Centralized Algorithm

For a video cloud \( j \in U_c \), let \( U_{d \rightarrow j} \subset U_d \) be the subset of mobile devices that are connected to \( j \). Each mobile device \( i \in U_{d \rightarrow j} \) has two queues, \( Q_s \) and \( Q_p \) as shown in figure 4.2. \( Q_s \) is the scaling queue, and initially holds all the videos related to
the query. The processing queue ($Q_p$) is a doubly-ended queue where videos are enqueued after being scaled. Each video cloud has a single queue $Q_{GPU}$ that holds videos waiting to access the GPU.

![Figure 4.2: Centralized algorithm: mobile device queues](image)

In the centralized method each video cloud has knowledge about the channel conditions, the length of $Q_p$ in the mobile nodes, and the information about the video at the head of the queue. This information is updated each time a cloud receives a video and its channel is free and ready to receive a new video. After a query is initiated, all the videos related to the query are enqueued into $Q_s$ and scheduled to be re-sized to the specific resolution that satisfies the completeness requirement. After a video is scaled, it is added to $Q_p$. When all the videos have been scaled, the CPU starts processing videos locally from the end of $Q_p$.

Recall $p_{a,j}$ is the processing time for video $a$ on video cloud $j$, and let $d_{a,i,j}$ be the transmission time of video $a$ from mobile device $i$, to video cloud $j$. Using an opportunistic scheduling algorithm, each video cloud will make a greedy choice based on the sizes of $Q_p$ in mobile devices. Since clouds have higher processing rates, we aim to optimize the GPUs and decrease their idle time. Simultaneously, we aim to decrease the amount of videos processed locally, hence give mobile nodes...
with larger \( Q_p \) a higher chance to offload their videos.

**Algorithm 3 Centralized Algorithm**

```plaintext
for \( Q_i \) in \( Q(1), Q(2), ..., Q(min) \) do
    let \( a \) be the video at the head of \( Q_i \)
    if \( d_{a,i,j} \leq \sum_{b \in Q_{GPU}} p_{b,j} \) then
        \( j \) will schedule mobile device \( i \)
        break;
    end if
end for
if no device was scheduled then
    schedule mobile device with the best channel
end if
```

If two mobile devices have the same queue length, the mobile device with the best channel will be chosen for this slot. If two mobile devices have the same channel rate (best channel), one of them is randomly chosen.

### 4.2 Simulations

In order to evaluate the performance of the proposed heuristics, a simulation environment was set up. In this section we show the different parameters used for the simulation, and discuss the results.

#### 4.2.1 Parameters

Based on the experimental results in Chapter 3, we see the effect of the number of mini-batches on the processing time. We only work in the first region (mini-batch
Based on that, to model the video sizes, we use a normal distribution $N(\mu, \sigma)$, where $\mu = 10,000$ [kB] and $\sigma = 5,000$ [kB].

For the simulation, we assume that the query completeness requirement is satisfied by scaling the videos to resolution 320x180. Figure 4.3 plots the scaling times for scaling from the original video resolution (1080p) to 320x180 for videos of different sizes. From that we use the following equation to calculate the scaling time [sec]:

$$t_s = 0.00012 \times \text{vidsize} + 0.75$$  \hspace{1cm} (4.7)

![Scaling time from 1080p to 320x180](image)

Figure 4.3: Scaling different 1080p videos to 320x180

To estimate the size of a video [kB] after being scaled, Figure 4.4 has been plotted. From it we find the following relation:

$$\text{newsize} = \alpha \times \text{oldsize} + \beta$$  \hspace{1cm} (4.8)

In the simulation we model $\alpha$ by a normal distribution $N(0.04, 0.02)$ and $\beta$ by a normal distribution $N(120, 20)$, based on the equation in figure 4.4.
Finally, the processing rate for the mobile devices and video clouds are set using a uniform distribution between $[\gamma s_d, s_d]$ and $[\gamma s_c, s_c]$ respectively, where $\gamma = 0.35$, the processing rate of mobile devices ($s_d$) is set to 15 [kB/sec], and the processing rate of video clouds ($s_c$) is set to 130 [kB/sec]. The transmission rates are assigned to each channel randomly based on a normal distribution $N(r, 500)$, where $r$ is the transmission rate specified as an input parameter. The parameters are chosen based on the experimentation results, and modified slightly by trial and error.

### 4.2.2 Results

The default settings for the parameters are as follows:

$$|V| = 300, |U_c| = 3, |U_d| = 30, r = 16[MB/sec]$$

Two algorithms are used for comparison:

- **No Offloading**: Each node processes the videos it has locally, and no
offloading takes place.

- **Baseline:** Mobile devices offload all the videos to the video cloud with the best channel.

To estimate the minimum time we used the following algorithm:

\begin{algorithm}
\caption{Minimum time estimation}
\label{alg:time_estimation}
\begin{algorithmic}
\State $T_{opt} = 0$
\While {true}
\State $weight = 0$
\State $temp = 0$
\For {i in $U_d$}
\If {$T_i \geq T_{opt}$}
\State $temp += T_i \times \frac{s_i}{r_{max}}$
\State $weight += \frac{s_i}{r_{max}}$
\EndIf
\EndFor
\For {i in $U_c$}
\State $temp += T_i$
\EndFor
\For {i in $U_d$}
\State $temp += t_{scale}$
\EndFor
\State $temp = \frac{temp}{weight + num\_clouds}$
\If {$temp == T_{opt}$}
\State break;
\Else
\State $T_{opt} = temp$
\EndIf
\EndWhile
\end{algorithmic}
\end{algorithm}

where $T_i$ is the time it takes for a node to process the videos in its queue, $s_i$ is a node’s processing speed, $r_{max}$ is the channel with the maximum transmission rate, and $t_{scale}$ is the time a mobile device takes to scale the videos initially located there.

The main idea of this estimation is to calculate the average of the completion time at nodes, considering scaling time, processing time, and communication delay,
but ignoring the communication constraints.

Scaling times and processing times at the clouds are delays that have to occur in the system. The processing times in mobile nodes, when accounted for, are multiplied by a weight of the node’s processing speed divided by the rate of the fastest channel, to account for either processing locally or transmitting. The sum of these delays is average over the number of video clouds and the sum of weights of mobile devices. Since $T_i$ is scaled by the weight $\frac{\alpha_i}{r_{max}}$ for mobile devices, the sum of these weights are taken into consideration when calculate the average.

The following figures show the simulation results when varying different system parameters. The completion time is when the last node finishes processing, denoted by $T_{max}$. For each set of results we will vary one parameter, to study its effect on the system. Two plots are shown, one is the absolute time ($T_{max}[sec]$) and the other is the comparison to the estimated minimum time ($T_{max}/T_{opt}[\%]$). It is shown that both the distributed and centralized algorithms outperform the baseline and no-offloading benchmarks. However, the distributed algorithm performs poorly compared to the centralized algorithm. Some insights in regards to this issue are discussed at the end of this section.

First, we study the effect of varying the number of videos in the system. In all the four algorithms we can see that as the number of videos in the system increase so does the maximum completion time of the system. We also realize that both the distributed and centralized algorithms have steady performance compared to the optimal solution.

Second, the effect of varying the number of mobile devices is shown. For the baseline, since the videos are always offloaded, the number of devices has almost no effect on the completion time. For the other three algorithms, as the number of devices increases the completion time decreases, as expected. Given that the more
Figure 4.5: Varying $|V|$

(a) Varying $|V|$: Absolute time

(b) Varying $|V|$: $T_{max}/T_{opt}$
(a) Varying $|U_d|$: Absolute time

(b) Varying $|U_d|$: $T_{max}/T_{opt}$

Figure 4.6: Varying $|U_d|$
Figure 4.7: Varying $|U_c|$

(a) Varying $|U_c|$: Absolute time

(b) Varying $|U_c|$: $T_{\text{max}}/T_{\text{opt}}$
Figure 4.8: Varying $|r|$: Absolute time

Figure 4.8: Varying $|r|$: $T_{\text{max}}/T_{\text{opt}}$
devices there are in the system, the more the load (videos) is distributed. For the centralized algorithm, we see that as the number of devices increase the solution is closer to optimal.

Third, the simulation is run using different numbers of clouds. As the number of clouds decrease, the solutions are closer to optimal, given that less choices for offloading are available. For example, if the system has 7 video clouds, the completion time will drastically decrease given that more processing resources are available. However, compared to the optimal, the performance gets worse. The reason is that the optimal solution does a complete search and considers all the possible combinations of offloading videos. However, in our heuristics an instantaneous choice has to be done, whether by the mobile devices as in the distributed algorithm, or by the mobile clouds as in the centralized algorithm. For the centralized algorithm, the more video clouds there are, the higher the probability that the scheduling will vary from the optimal solution.

Finally, we measure the effect of varying the transmission rate. We can see that the system is steady regardless of the transmission rate. The reason is that the main delay when offloading the videos from mobile devices to the cloud is the queuing delay at the GPU not the transmission delay. However, when we go to a lower transmission rate such as 125 kB/sec we see that the completion time of the distributed algorithm increases, meaning that at this rate the transmission delay is getting higher than the queuing delay at the GPUs.

From the results we note that the distributed algorithm preforms poorly compared to the centralized algorithm. To understand the reasons behind this low performance we take a case where the difference is large, such as the case in Figure 4.6a with 10 mobile devices. It is found that in the distributed algorithm 57.5% of the videos in mobile devices are processed locally, while in the centralized algorithm
only 32.4% are. Due to the decision making process in the distributed algorithm, a high percentage of the videos are assigned to be locally processed. That leads to a longer processing time, given the slow processing rates in mobile devices compared to video clouds.
Conclusion and future work

In this thesis, we propose two heuristics for solving the processing scheduling problem in wireless networks given a completeness constraint. The problem aims to minimize the maximum query response time in a network that consists of mobile devices and video clouds. We start by studying the effect of video parameters on two QoI metrics: completeness and timeliness.

After a query is issued, videos related to the query are scaled to the minimum resolution that meets the completeness requirement, then the videos are either processed locally in the devices or offloaded to video clouds.

The first algorithm is a distributed algorithm, where each mobile device makes a decision whether to offload the video or process it locally based on a locally calculated variable called the processing round trip time (prt). The second algorithm is a centralized algorithm. Each cloud has knowledge of the channel conditions and the mobile devices’ local queues. Based on that, each video cloud schedules mobile devices to offload their videos according to an opportunistic algorithm.

Following the problem formulation and the algorithm designs we run simulations.
We see that both the algorithms perform better than the baseline algorithm and no-offloading. We also notice that the distributed algorithm performs worse than the centralized algorithm given that it processes a higher percentage of the videos locally rather than offloading them. The performance of the centralized algorithm is close to the optimal solution.

In future research, other QoI metrics such as accuracy can be investigated. Another interesting study would be to examine the effect of other video parameters, such as frame and bit rate, on QoI. Improvements on the distributed algorithm can be done by finding a method to offload a higher percentage of videos. With the improvement of deep learning vision algorithms, faster and more accurate action detection algorithms can be developed, leading to different problem definitions and system requirements.
Bibliography


