

# VIDEO BASED AUTOMATIC SPEECH RECOGNITION USING NEURAL NETWORKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Alvin Lin

December 2020

© 2020

Alvin Lin

ALL RIGHTS RESERVED

## COMMITTEE MEMBERSHIP

TITLE:      Video Based Automatic Speech Recognition  
                 Using Neural Networks

AUTHOR:     Alvin Lin

DATE SUBMITTED:      December 2020

COMMITTEE CHAIR:     Jane Zhang, Ph.D.  
                         Professor of Electrical Engineering

COMMITTEE MEMBER:   Xiao-Hua (Helen) Yu, Ph.D.  
                         Professor of Electrical Engineering

COMMITTEE MEMBER:   Wayne Pilkington, Ph.D.  
                         Associate Professor of Electrical Engineering

## ABSTRACT

### Video Based Automatic Speech Recognition Using Neural Networks

Alvin Lin

Neural network approaches have become popular in the field of automatic speech recognition (ASR). Most ASR methods use audio data to classify words. Lip reading ASR techniques utilize only video data, which compensates for noisy environments where audio may be compromised. A comprehensive approach, including the vetting of datasets and development of a preprocessing chain, to video-based ASR is developed. This approach will be based on neural networks, namely 3D convolutional neural networks (3D-CNN) and Long short-term memory (LSTM). These types of neural networks are designed to take in temporal data such as videos. Various combinations of different neural network architecture and preprocessing techniques are explored. The best performing neural network architecture, a CNN with bidirectional LSTM, compares favorably against recent works on video-based ASR.

Keywords: Convolutional Neural Network, Long Short-Term Memory, Lip reading, Automatic Speech Recognition

## ACKNOWLEDGMENTS

Thanks to

- My brother for helping me with many coding issues
- My friends for making my experience at Cal Poly one of a kind
- My professors for giving me a great education at Cal Poly
- Professor Jane Zhang for being a great advisor

## TABLE OF CONTENTS

	Page
LIST OF TABLES .....	viii
LIST OF FIGURES .....	ix
CHAPTER	
1. Motivation/Approach .....	1
2. Scope of Thesis .....	3
3. Literature Review (Related Work).....	4
4. Data considerations .....	8
5. Software and Hardware.....	11
5.1 Docker .....	12
6. Preprocessing of Data .....	13
6.1 Digital Images and Video Representation.....	13
6.2 Dlib overview .....	14
6.3 Histogram of Oriented Gradients (HOGs) .....	14
6.4 Support Vector Machine (SVM) .....	16
6.5 Dlib Library .....	18
6.6 Lip Frame Differences .....	20
6.7 File Manipulation .....	22
7. Neural Networks .....	24
7.1 CNN .....	24
7.2 LSTM .....	28
7.3 LSTM-CNN .....	30
7.4 Practical Considerations .....	31
7.4.1 Batch Size .....	31
7.4.2 Learning Algorithm .....	33
7.4.3 Activation Function .....	35
7.4.4 ARCHITECTURE THEORY .....	37
7.4.5 OVERFITTING .....	38
7.5 Architectures .....	39
7.5.1 Architecture #1: 3D-CNN.....	39
7.5.2 Architecture #2: 3D CNN with 5 Fully Connected Layers.....	40

7.5.3	Architecture #3: VGG-16 based architecture .....	41
7.5.4	Architecture #4: Multiple Towers.....	43
7.5.5	Architecture #5: LSTM-CNN .....	44
7.5.6	Architecture #6: CNN and Bidirectional LSTM.....	46
8.	Results.....	48
8.1	Preprocessing result.....	48
8.2	Neural Network results.....	48
8.2.1	Architecture #1: 3D-CNN.....	48
8.2.2	Architecture #2: 3D CNN with 5 Fully Connected Layers.....	49
8.2.3	Architecture #3: VGG-16 Based Architecture.....	50
8.2.4	Architecture #4: Multiple Towers.....	52
8.2.5	Architecture #5: LSTM-CNN .....	53
8.2.6	Architecture #6: CNN and Bidirectional LSTM.....	54
8.2.7	Comparison of Different Architectures .....	55
8.3	Using Difference of Frames .....	56
8.4	Confusion Matrix .....	57
8.5	Failed approaches not included in results .....	58
9.	Conclusion .....	61
9.1	Summary .....	61
9.2	Challenges .....	63
9.3	Future Work .....	64
	BIBLIOGRAPHY .....	67
	APPENDICES	
A.	LIP DETECTION MODULE .....	72
B.	LIP FRAMES TO NUMPY .....	74
C.	BI-DIRECTIONAL LSTM-CNN .....	78
D.	LSTM-CNN .....	82
E.	Multiple Towers.....	83
F.	VGG-16 Based Architecture.....	84
G.	Extended 3D-CNN.....	84
H.	3D-CNN.....	85

## LIST OF TABLES

Table	
Page	
1. Hardware .....	11
2. Architecture #1.....	40
3. Architecture #2.....	41
4. Architecture #3.....	43
5. Architecture #4.....	44
6. Architecture #5.....	45
7. Architecture #6.....	47
8. Preprocessing results.....	48
9. Architecture #1 results .....	49
10. Architecture #2 results .....	50
11. Architecture #3 results .....	51
12. Architecture #4 results .....	52
13. Architecture #5 results .....	53
14. Architecture #6 results .....	55
15. Overall test accuracy and parameters .....	55
16. Comparing two different dataset types .....	56
17. Most confused word pairs .....	58



## LIST OF FIGURES

Figure

Page

1. High level flowchart of thesis .....	3
2. EF and MT architectures as shown in Chung and Zisserman's paper [17] .....	5
3. CNN and LSTM as detailed by Lip reading using CNN and LSTM [26] .....	7
4. Example video data from the AVICAR dataset [29] .....	8
5. Example dataset from the MODALITY Corpus dataset [16] .....	9
6. Example from BBC's Lip Reading in the Wild's dataset .....	10
7. RGB image.....	13
8. Grayscale image.....	14
9. A quick description of the HOG algorithm [31] .....	16
10. SVM decision boundary [32] .....	17
11. Facial landmark points for the Dlib library [14] .....	19
12. Extracted lip data for the word "Emergency" .....	20
13. Extracted differences between frames for the word "EMERGENCY" .....	21
14. File Structure.....	22
15. The fully connected layer [11] .....	26
16. Visualization of fitting in a neural network [12] .....	27
17. LSTM cells can connect with one another [2] .....	28
18. The input block [2].....	28
19. The forget block [2] .....	29
20. The forget block [2] .....	30

21. A practical implementation of a LSTM-CNN model .....	31
22. Training speed and efficiency compared to batch sizes [24] .....	32
23. Adam outperforms other learning rate algorithms [18] .....	33
24. Effect of learning rate on converging to the optimum rate [5] .....	34
25. Visualization of the optimal learning rate [5] .....	35
26. Sigmoid function (left) and tanh function (right) .....	36
27. Rectified Linear Unit (ReLU) activation function.....	37
28. Neural network depth compared to test error [21].....	38
29. A standard neural network compared to a neural network with dropout applied [22]	39
30. The original VGG neural network architecture configurations [20] .....	42
31. Bidirectional flow of information of a bidirectional LSTM [15] .....	46
32. Accuracy changes per epoch for architecture #1 .....	49
33. Accuracy changes per epoch for architecture #2 .....	50
34. Accuracy changes per epoch for architecture #3 .....	51
35. Accuracy changes per epoch for architecture #4 .....	53
36. Accuracy changes per epoch for architecture #5 .....	54
37 Accuracy changes per epoch for architecture #6 .....	55
38. Confusion matrix for 500 classes.....	57
39. Inception module with dimension reduction [35] .....	59

## 1. Motivation/Approach

Automatic speech recognition (ASR) systems use algorithms to translate spoken words to text. Companies, such as Google Cloud, Microsoft Azure, IBM Watson, and YouTube, employ ASR systems for the purposes of education, transcription, and assisting the disabled. These are all able to provide high quality transcripts. A study showed that compared to manual transcription, which has a word error rate of 17.4% [19], YouTube was able to transcribe with a word error rate of 28% [19]. The study however only uses high quality FLAC files with little background noise. With lower quality recordings and more background noise, word error rate would increase. Video data can be used for speech recognition as it is resistant to audio noise.

Lip reading is the skill of recognizing speech with only visual information such as movement of the lips and face. Lip reading has many practical use cases such as transcription of silent films or verbal exchanges on closed circuit television (CCTV) cameras. This skill is useful for recognizing speech where there is no audio information and is commonly used by the deaf to perceive speech. For people without hearing impediment, this skill can be useful in very noisy environments such as restaurants. A study done by the University of Oklahoma shows that the lip-reading accuracy for humans is 45% [28].

The objective of this thesis is to develop a speaker-independent video-based ASR algorithm to match or beat lip reading accuracy for humans.

A great advantage of video-based ASR compared to audio-based ASR is its ability to differentiate between words that sound the same but look different when lip reading.

Examples are the words “fair” and “pear”. The words may sound similar at first listen, but the lip movements for “f” and “p” in those words are very different.

On the other hand, a disadvantage for video-based ASR is when words sound distinct but have similar lip movements. For example, the words “pat” and “bat” sound very different but look almost the same to a lip reader. Because visemes are ambiguous compared to speech sounds, it has been the centerpiece of comedic videos where actual speech is dubbed with speech that look identical in lip reading (i.e. YouTube channel “Bad Lip Reading”). The ambiguity of visemes are the main challenge that contributes to a reduction in accuracy for video-based ASR systems.

The traditional approach for time series data would be to use Hidden Markov Models (HMM). In an HMM, there are states and transition probabilities. For ASR, a state is a phenome, and the transition probabilities are the chances that one phenome leads to another phenome. Neural networks are a type of supervised learning algorithm. Its flexibility allows it to be adapted to many different applications, including ASR. Recent studies show that deep neural networks outperform the traditional HMM approach by up to 24% [27]. Neural networks are indeed a new paradigm when it comes to ASR.

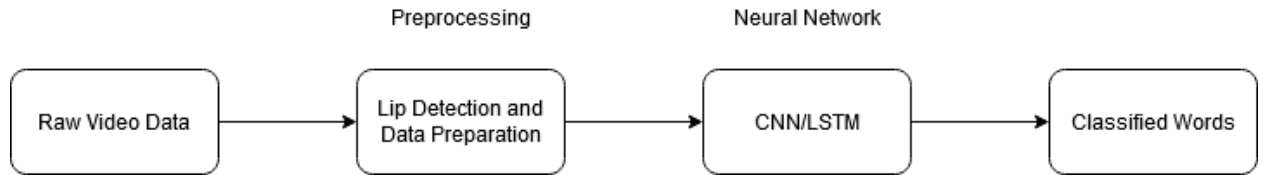
Convolutional neural networks (CNNs) will be used along with recurrent neural networks (RNNs). CNNs adjusts a set of filters for every convolutional layer to determine what the best filters are for determining key aspects in images. While CNNs can only handle static data shapes, for example images, RNNs also consider temporal data. In this thesis, cutting edge RNNs such as Long Short-Term Memory (LSTM) will be utilized along with CNNs to enhance accuracy of lip reading.

## 2. Scope of Thesis

The overarching goal of this thesis is to experiment with different neural networks and signal processing methods to achieve maximum video-based ASR accuracy.

There are many things to consider for the thesis. First, a quality and user-friendly data source must be obtained. The data must come from a real-world environment to ensure that the system will perform well in a practical real-life setting. Second, the obtained data is preprocessed so that our neural network can use the data. Lastly, the structure of the neural network needs to be modified and experimented with to maximize accuracy.

Neural networks require time-consuming tuning processes so that it will run optimally. A good starting point will be well-researched architectures. Then, trial and error are needed to achieve better results.



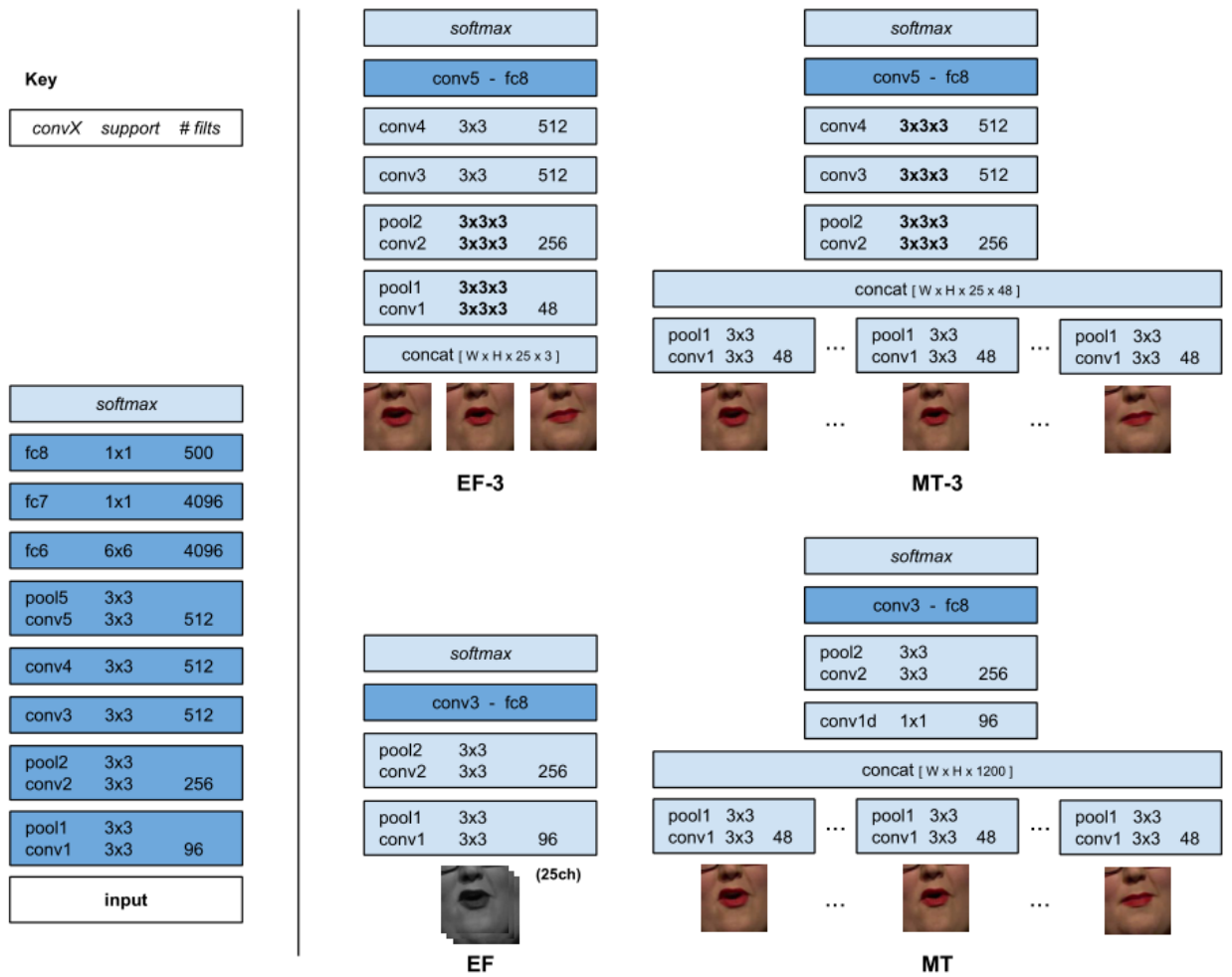
**FIGURE 1. HIGH LEVEL FLOWCHART OF THESIS**

This thesis is roughly organized into the flowchart shown above. In chapter 4, dataset considerations are discussed. In chapter 6, principles and concepts such as Histogram of Oriented Gradients and Support Vector Machines are discussed. A practical approach will be discussed and implemented. In chapter 7 neural networks and the practical considerations, such as batch size, learning rate, and architectures, will be discussed. In chapter 8, the outcomes and analysis of the outcomes will be discussed. Lastly, chapter 9 will give a summary of results, challenges faced, along with new discoveries.

### 3. Literature Review (Related Work)

*Lip Reading in the Wild (2016)* [17], authored by Joon Son Chung and Andrew Zisserman from the University of Oxford, starts off by approaching the dataset problem. Different datasets have their issues that make using it as training data less than ideal. In this paper, they point out that most datasets are recorded in an ideal lab environment. The one dataset, AVICAR, which is in a car environment, has both low number of classes and low number of test subjects. Chung and Zisserman introduce the Lip Reading in the Wild dataset, which solves all the problems. It is recorded in a realistic and uncontrolled environment and has many classes and test subjects.

In the same paper, they present several 3D CNN architectures that they use for video-based ASR. The architectures that they introduce are early fusion (EF) and multiple towers (MT).



**FIGURE 2. EF AND MT ARCHITECTURES AS SHOWN IN CHUNG AND ZISSERMAN'S PAPER**

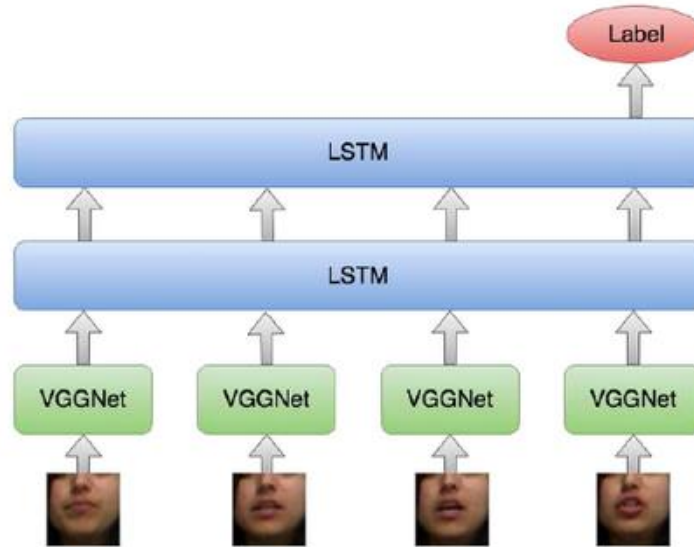
[17]

EF is similar to a classic CNN architecture where the data flows in a linear fashion. MT, unlike EF, starts by running a CNN for every frame, then concatenating the data so that it can be run on a normal CNN. It is found that the MT architecture, using grayscale representation for the videos instead of color, had the best testing results at 61.1% for a dataset of 500 words. This will be my goal to improve upon.

*Visual Speech Recognition Using a 3D Convolutional Neural Network (2019)* [25], authored by Matt Rochford, is a recent paper on video-based ASR. In this paper, he uses Dlib to extract the lips from the faces. Instead of using MATLAB to run the neural network, Matt uses TensorFlow and python to run his code. In total, there are six architectures tried. For each architecture, certain parameters such as number of filters, convolutional layers, and fully connected layers are tested. It is found that 2x2x2 convolutional filters performed the best instead of the more traditional 3x3x3 convolutional filter. However, the architectures used here did not include any zero padding. This means that for every convolutional layer, data is lost.

*Lipreading with Long Short-Term Memory (2016)* [23], a paper written by Michael Wand, Jan Koutník, Jürgen Schmidhuber, discusses using LSTMs along with other non-neural network methods. The dataset used is the GRID corpus, which is coincidentally discussed in Chung and Zisserman's *Lip Reading in the Wild* paper. This dataset is relatively limited in classes and samples compared to LRW. There is only 52 total words in the dataset used. The non-neural network approach that this team uses involves feature extraction and classification such as Histogram of Oriented Gradients (HOG), Eigenlips, and Support Vector Machines (SVM). This paper also uses LSTMs but does not combine with CNNs. The accuracy of LSTM method is 79.6%, compared to the HOG and SVM method, which has an accuracy of 71.3%. The LSTM performs 8.3% better than the HOG and SVM method. [23]





**FIGURE 3. CNN AND LSTM AS DETAILED BY LIP READING USING CNN AND LSTM [26]**

*Lip reading using CNN and LSTM (2016)* [26], a paper by Amit Garg, Jonathan Noyola, and Sameep Bagaida from Stanford university, discusses specifically about the CNN and LSTM approach to lip reading. While this paper does not discuss specifically about word lip reading, the LSTM and CNN architecture used is of great inspiration to me. The CNN used is VGGNet. VGGNet is a CNN architecture introduced in 2014 with the paper *Very Deep Convolutional Networks for Large-Scale Image Recognition* by Karen Simoyan and Andrew Zisserman at the University of Oxford. While heavy in memory usage, this neural network compares well against other CNNs such as GoogLeNet [17]. In the *Lip reading using CNN and LSTM*, VGGnet is run for every frame, then passed through a stacked LSTM layer. In this thesis, I will be using a similar neural network.

#### 4. Data considerations

The AVICAR corpus [29] was collected and transcribed by the University of Illinois at Urbana-Champaign with funding from Motorola in 2003-2004. This dataset was recorded using multiple camera angles and microphones within a car. The video is recorded in various circumstances, such as when moving and when sitting still. The words spoken consists of isolated numbers and letter, phone numbers, and TIMIT sentences (a speech corpus developed in 1993). This model introduces noise conditions that are realistic and real-world. However, the dataset is relatively small with low quality recording. The recordings are at an 360x240 pixel resolution at 30 frames per second (fps).



**FIGURE 4. EXAMPLE VIDEO DATA FROM THE AVICAR DATASET [29]**

MODALITY corpus [16] is a much more recent database recorded in 2015. The focus of this dataset was to provide high quality recordings in a studio environment. Videos are recorded at full HD resolution (1920x1080p resolution) and at 100 fps. Speakers consists of both native and non-native English speakers. Words spoken are 168 different commands including single words and sentences. This database is 2.1 terabytes, which is

significantly larger than AVICAR. However, the greatest issue is that the video data is not organized. The videos are organized in a way that, for a single word, all the speakers speak in the same video file. It would take a significant amount of time to manually clip the video data so that they are the same size for training. The greatest issue of this dataset is that the recordings are done in an ideal studio environment. This means that while a neural network may perform well on this dataset, it may perform poorly when put to the test in a real-world environment.



**FIGURE 5. EXAMPLE DATASET FROM THE MODALITY CORPUS DATASET [16]**

Oxford-BBC Lip Reading in the Wild (LRW) [17] dataset consists of 500 different words spoken with over 1000 samples each. The videos are taken from BBC 1 HD channel news segments and aligned with subtitles. The LRW dataset consists mainly of regular news segments. There is a combination of indoor and outdoor recording environments for regular news. BBC HD channels consists of high quality 1080p recording. LRW consists of thousands of hours of spoken text with over one million word instances and over one thousand different speakers [17].

The LRW dataset is processed by both automatic and manual means. First, a linear SVM classifier is trained to detect the speaker is on screen, then cropped out to create the video data. The data is then manually checked for errors such as misalignment in audio and video.



**FIGURE 6. EXAMPLE FROM BBC’S LIP READING IN THE WILD’S DATASET**

This dataset is chosen mainly because all of the difficult and time-consuming manual cropping work is done already. The dataset consists of high-quality recording from BBC’s HD channels, and there is a large pool of words to train on. This dataset strikes the best balance between convenience and quality.

## 5. Software and Hardware

At a glance, hardware and software used:

CPU	2x AMD EPYC 7742 64 Core Processor
GPU	2x Tesla V100, 32GB HBM2
RAM	755GB

**TABLE 1. HARDWARE**

The computer has an AMD 64 core EPYC computer built on TSMC’s new 7 nm process. The NVidia Tesla V100 is a current generation graphics card that utilizes 32 gigabytes of fast HBM2 memory. The server also has a large pool of memory. These server specifications would be closest to Amazon’s p3.16xlarge instance, which would cost 24.48 dollars an hour.

Many thanks to the Computer Science department for providing this machine to me!

Software: Linux (CentOS/Ubuntu), Python, Docker, Dlib, OpenCV

The server is running on CentOS while the docker container is run on Ubuntu. Docker is similar to Conda, where virtual environments are created where libraries can be installed. This allows code to be run in those environments without directly running in the main system. For docker, these virtual environments are known as “containers”. Luckily, docker containers can be easily built by “pulling” prebuilt operating systems (Ubuntu is used). A TensorFlow environment can be easily set up using the same “pull” command [10]. Details of setting up the docker environment will be discussed later.

Dependencies used are image processing libraries such as “opencv2”, “dlib”, and “imutils”. These are responsible for extracting facial features for the neural network. The

neural network is built using Google's TensorFlow with Keras as a front end. Keras is a user-friendly front end to TensorFlow that helps in implementing neural networks.

## 5.1 Docker

Docker is a tool that allows users to set up a virtual environment that has very little hardware overhead. Unlike a virtual machine (VM), a docker container can fully utilize system resources. Users can install dependencies and libraries in a Docker container without adjusting anything on the base system.

The first step in setting up a docker environment is to pull from a source [10]. Ubuntu is an easy to use and beginner friendly operating system. Another environment pulled is a prebuilt TensorFlow environment.

After pulling the environments using docker, dependencies can be then installed in these containers. Dependencies such as "imutils", "dlib" and "opencv" are extracted to perform preprocessing. The container can have storage from the host system mounted to it. In the interest of loading speeds, data files should be kept out of the container.

The containers are run with admin privileges to install dependencies. Otherwise, when running programs, the tag "-u \$(id -u)" is needed to access the mounted files. Special tags are used to enable GPU usage within the container [8].

## 6. Preprocessing of Data

### 6.1 Digital Images and Video Representation

A digital image is made of pixels. The pixel has different formats to represent different colors or intensities. The most common color representation for pixels is RGB. In this color space, there is three dimensions: red, green, and blue [3]. These values for red, green, and blue have values from 0 to 255 where 0 is the least intense and 255 is the most intense. The combination of red, green, and blue can make up millions of colors. An example image with 100 width and 60 height in RGB will have the following dimension values:

$$I = (60,100,3)$$

The “3” in the last dimension represents the RGB color space where red, green, and blue have separate values.



**FIGURE 7. RGB IMAGE**

For this thesis, data will be converted to grayscale for reasons explained later. Grayscale is a one-dimensional representation of intensity. The single dimension is the intensity of an image where low intensity corresponds to a darker gray color while high intensity corresponds to a brighter gray. An image with a height of 60 and width of 100 in grayscale format would have the following dimension values:

$$I = (60,100,1)$$

Where the last dimension has a value of “1” due to its one-dimensional grayscale representation.



**FIGURE 8. GRAYSCALE IMAGE**

A video is the concatenation of frames to create an illusion of moving pixels. The measure of video data is in frames per second (fps). The fps of the video data used in the Lip Reading in the Wild dataset is recorded at 29 fps. Therefore, the video data dimension with 29 frames, 60 pixel height, and 100 pixel width in grayscale format looks like:

$$\text{Video} = (\text{frames}, \text{height}, \text{width}, \text{intensity}) = (29, 60, 100, 1)$$

## 6.2 Dlib overview

Dlib is a library written in C++ that contains a pre-trained algorithm to detect facial features. It detects facial features such as eyes, eyebrows, nose, lips and jawline. The pre-trained algorithm is a histogram of oriented gradients (HOG) and linear SVM object detector. It achieves a 99.38% accuracy in the standard LFW (Labeled Faces in the Wild) face recognition benchmark, which is equivalent to other state-of-the-art face recognition algorithms [1].

## 6.3 Histogram of Oriented Gradients (HOGs)

HOGs have been crucial to many feature detection algorithms. The basis of HOGs is the gradient. A gradient is the measure of rate of change. The mathematical expression of a gradient for a 2D vector is shown below.



$$\Delta f(p) = \begin{bmatrix} \frac{\delta f}{\delta x_1}(p) \\ \frac{\delta f}{\delta x_2}(p) \end{bmatrix}$$

The direction of the gradient will always be perpendicular to the direction of greatest rate of change. If the change is larger, the gradient will be larger in magnitude.

The first step of applying a HOG to an image would be to convert the image to greyscale, if originally in color. Then, the gradient is calculated at every pixel. Places that are likely to contain features are likely to be the one where there are large changes in image intensity. For an image, the smallest unit where change occurs is one pixel. Therefore, the gradient is calculated by simply take the difference between its neighbors in the x and y direction. The resultant vector would be the difference in the x direction as the x vector, and difference in the y direction as the y vector. Below is a simple example for a greyscale image.

	16	
99		62
	50	

The gradient for the middle cell would then be

$$[x, y] = [62 - 99, 16 - 50]$$

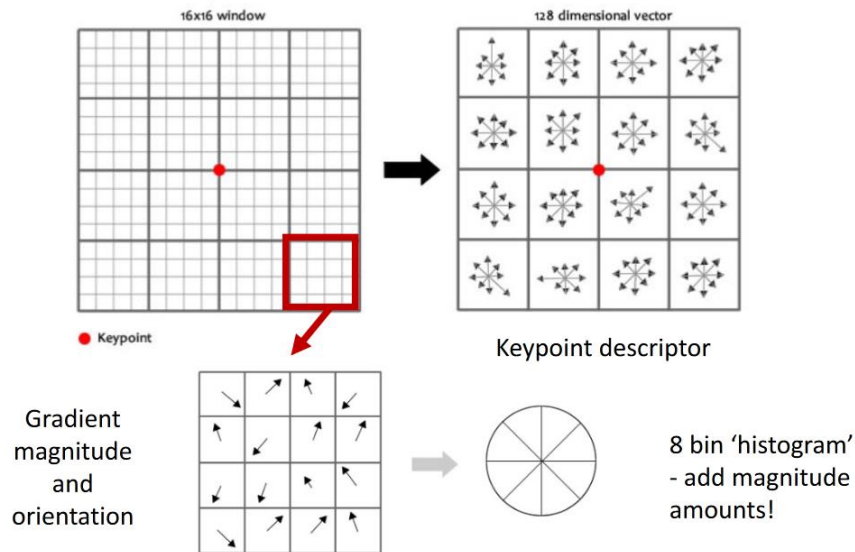
$$[x, y] = [-37, -34]$$

The angle would then be

$$angle = 222.58^\circ$$

After gradients are calculated, a frame usually of size 8x8 is run through the image. For each 8x8 frame, bins are created, divided evenly over 360 degrees. The gradients are then

sorted in each bin. The magnitude of each bin is the sum of magnitudes of the gradients inside the bin. This frame is then shifted by a pixel and the process of binning is repeated. For every pixel, there is a bin containing the angles and magnitudes of the gradients.

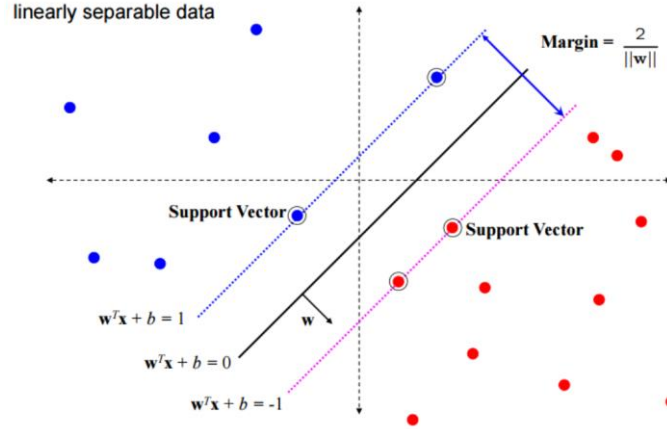


**FIGURE 9. A QUICK DESCRIPTION OF THE HOG ALGORITHM [31]**

The HOG outputs a feature vector containing the bins and magnitude of the bins [30]. This feature vector can then be fed to a classification algorithm such as Support Vector Machines (SVM).

#### 6.4 Support Vector Machine (SVM)

A support vector is a linear classifier introduced in the 1990's by Vladimir Vapnik and quickly became popular due to its performance. The objective of an SVM is to find a hyperplane in an N dimensional space that can correctly classify the data points within.



**FIGURE 10. SVM DECISION BOUNDARY [32]**

The goal of an SVM is to choose the best decision boundary between the data points. This decision boundary should be as far as possible from the different data classes as possible. The picture above shows the margin of the decision boundary. The goal is to maximize this margin [32]. The support vectors in the image are the data points that lie the closest to the decision boundary and have a direct effect on the optimum decision boundary location.

For a two-dimensional case, as shown in the above picture, a simple linear classifier can classify all of the data. The equation  $g(x) = \mathbf{w}^T \mathbf{x} + b = 0$  is the decision boundary where any point above it is classified as the first class and any point below is the second class.  $\mathbf{w}$  represents the “slope” or weight of the decision boundary.  $\mathbf{x}$  is the input data and  $b$  are the bias.

The margin is given by the following equation

$$\frac{\mathbf{w}^T}{\|\mathbf{w}\|} (x_+ - x_-) = \frac{2}{\|\mathbf{w}\|}$$

Where  $x_+$  and  $x_-$  are data points closest to the decision boundary.

The next step is to maximize the margin, which then becomes a constrained optimization problem. Since the SVM is not a key topic of this thesis, I will not go over the derivation of maximizing the margin. The optimized  $g(x)$  simplifies to: [32]

$$\min(\frac{1}{2} ||w||^2) + C \sum_{i=1}^n \xi_i$$

Subject to

$$y_i(w^T x_i + b) \geq 1 - \xi_i$$

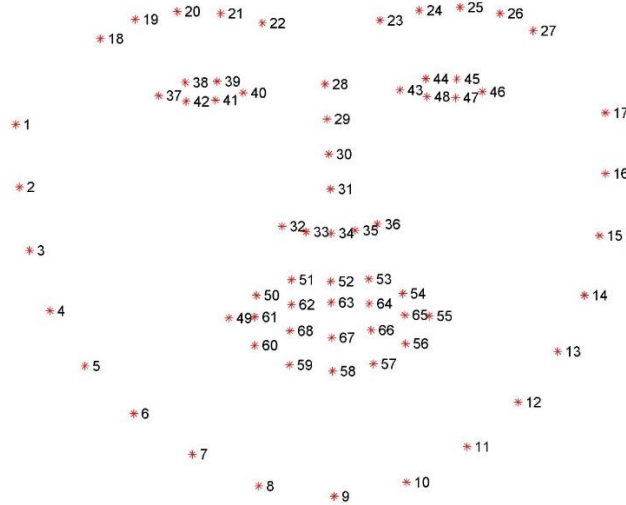
Where C controls the “hardness” of the margin. For example, a large C value would act as a strict margin that does not allow any misclassifications (but may have small margins). A small value of C would have softer margins where there are some allowed errors.

Combining the histogram of oriented gradients and SVM will allow us to classify certain features. The HOG is responsible for extracting facial features while the SVM is responsible for classifying the extracted features. Dlib uses HOGs and SVMs to classify facial features but is of much higher complexity than the example described above. Dlib’s face recognition algorithm maps the human face to a 128-dimensional vector space for the SVM to classify.

## 6.5 Dlib Library

The Dlib shape predictor function uses before mentioned methods to extract facial landmarks. It is trained on the iBUG 300-W dataset using an SVM [14]. It can label the following facial features: mouth, right and left eyebrows, right and left eyes, nose, and

jaw. For the purpose of this thesis, Dlib is used to extract the mouth feature from the face of the video.



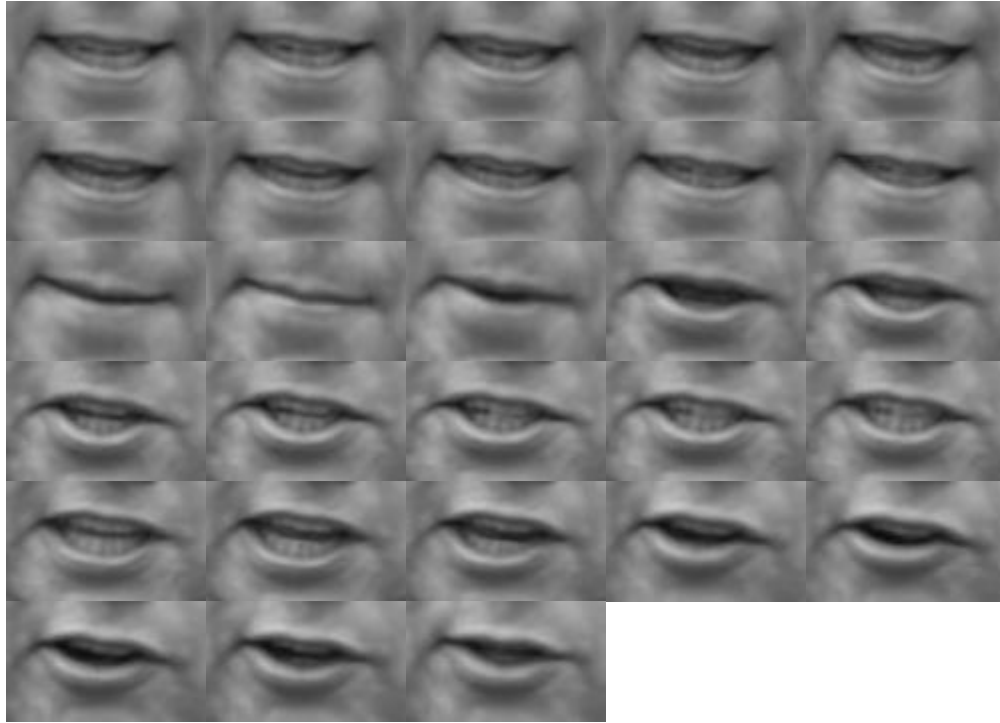
**FIGURE 11. FACIAL LANDMARK POINTS FOR THE DLIB LIBRARY [14]**

A pre-trained SVM is downloaded from the Dlib website to detect and isolate these facial landmark points. The facial features 49-68 correspond to the mouth facial feature desired.

After extracting the mouth feature, OpenCV is used to crop the mouth region for processing. The crop size is 100x60 pixels. In the preprocessing phase, the data will be converted to grayscale. It may be tempting to think that providing color data to a neural network may increase its performance as more data is being provided to it, performance degrades [17]. The extra color data theoretically should improve performance, but instead causes overfitting of data which ultimately degrades performance.

Lip Reading in the Wild's dataset is conveniently clipped to 29 frames per file. Since there is no system memory concerns, as the server has 755 gigabytes of memory, all 29 frames will be used. The final size of the pre-processed file is 29x100x60x1.

The lip detection program is responsible for extracting the lip landmarks using Dlib, cropping the lips in a 60x100 frame, and stacking the frames using numpy. This lip detection program has checks built in so that videos with multiple or no faces detected will not be used.



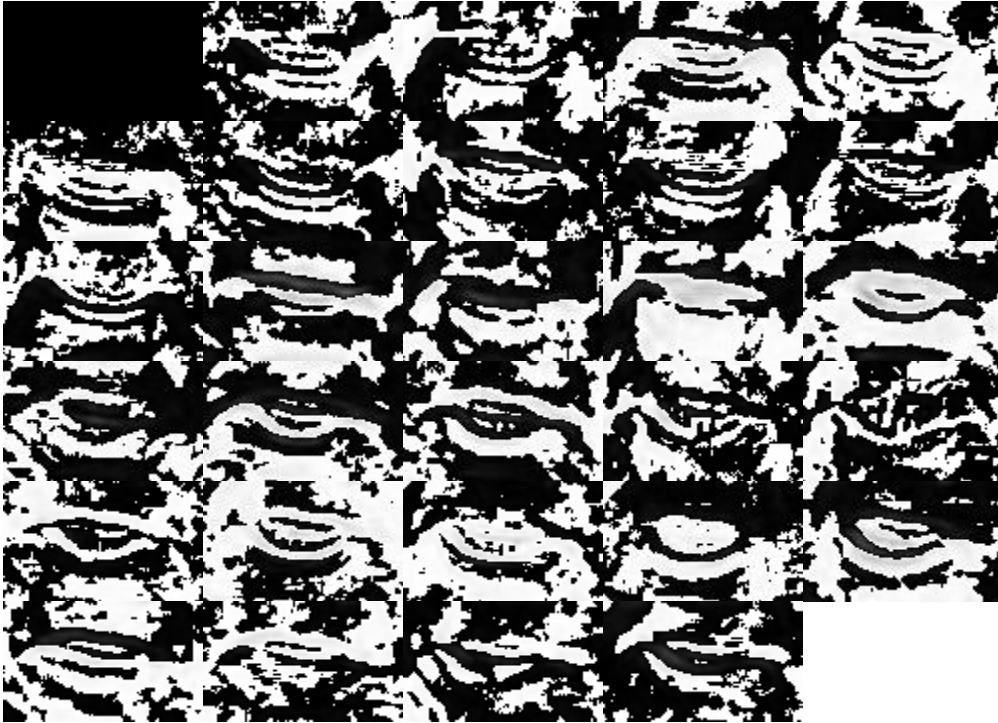
**FIGURE 12. EXTRACTED LIP DATA FOR THE WORD “EMERGENCY”**

### 6.6 Lip Frame Differences

Another dataset is created where, instead of the raw processed data frames, the differences between the frames are used as inputs for the neural network. For many frames, the lip appears to be the same. The idea is to capture the lip motions instead of the lip data itself. The differences between frames should yield a video size of

$$\text{Video} = (28, 60, 100, 1)$$

There should be 28 frames after the difference is taken. However, for convenience of input, a blank frame with zero intensity is padded at the beginning to bring the video back to 29 frames. Take note in the image below that more intense (brighter) areas corresponds to change in between frames.



**FIGURE 13. EXTRACTED DIFFERENCES BETWEEN FRAMES FOR THE WORD  
“EMERGENCY”**

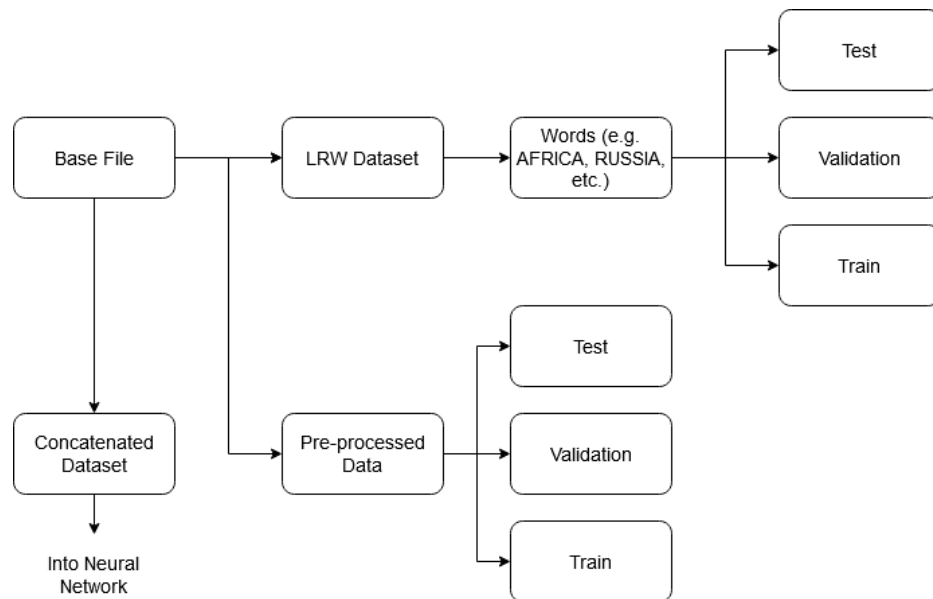
Ideally, this approach is supposed to capture moving lip contours of the image, where high intensity parts represent movement compared to the last frame. Even with good framing, a speaker usually nods and tilts his or her head when speaking. This means that the lips will shift relative to the last frame and that the whole image will have high intensity (change). Since this head motion when speaking differs from person to person, results can be inconsistent.

## 6.7 File Manipulation

After isolation of the lip frames, the data needs to be written into a numpy format so that it can be input into the neural networks. Multiprocessing is used to speed up the process. The entire list of words is loaded, then assigned to a CPU thread (which there are 128 of). Once the task is finished, the next unprocessed word is loaded into a thread to be converted. This significantly cuts down the time needed to convert and scales almost linearly with the number of threads that is available.

The raw extracted data will be categorized into four different folders. “Labels” will contain labels for the words. This is one hot encoded, meaning that every word is encoded as a “1” bit while everything else is zero. The labels are 500 bits long.

Every single word in the database contains “train”, “val” and “test” files. After processing, the output is sorted into the three folders “train”, “val” and “test”. Below is the file structure of the dataset.



**FIGURE 14. FILE STRUCTURE**



Before loading into the neural network, the video data needs to be normalized between 0 and 1. Grayscale images have a range of 0 to 255, so the video data is divided by 255.

TensorFlow also requires five dimensions for its input, which is reserved for the output of the filters. Therefore, an additional dimension is created using numpy.

```
input = (filters, frames, height, width, color channels)
```

The video data is then converted to FP16 to save on memory usage. The data is originally in FP64, or double precision. This isn't needed and consumes a large amount of memory.

Using FP16, or half precision (FP32 being full precision), allows for deployment for larger neural networks without losing much precision. Half precision floating point format uses four times less memory than double precision. NVidia's Turing architectures can take full advantage of FP16 compute while their older architectures cannot [13].

## 7. Neural Networks

### 7.1 CNN

Neural networks in its most basic form rely on backpropagation of data to update weights in neurons. CNNs, or convolutional neural networks, is a neural network based on convolutions. In every neuron or layer, the data is passed through a convolutional filter. A quick explanation behind two dimensional discrete convolutions are explained below.

The image data fed into CNNs is two dimensional and discrete. Convolutions are performed on such images by using a kernel. The principle behind the convolution, namely the “flip and shift” approach of one-dimensional convolutions, are similar. For two dimensional discrete convolutions, kernels are shifted across the image. The multiplied result is then summed to form the result of the convolution. The output will be at the center of the convolutional kernel. For a 3x3 kernel, the center where the output will be is at (2,2). A simple 3x3 convolution on a 3x3 pixel image will be demonstrated

$$image = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$convolutional\ kernel = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The convolutional kernel is then multiplied by the flipped image

$$\begin{aligned} & \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \\ &= 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 + 5 * 5 + 6 * 6 + 7 * 7 + 8 * 8 + 9 * 9 \\ &= 285 \end{aligned}$$

Note that for this example image, the output is only one pixel. To ensure that the image stays the same size after convolution, some form of padding must be used. Padding adds some value to the edge of the image. An example of zero padding of the same image is shown below.

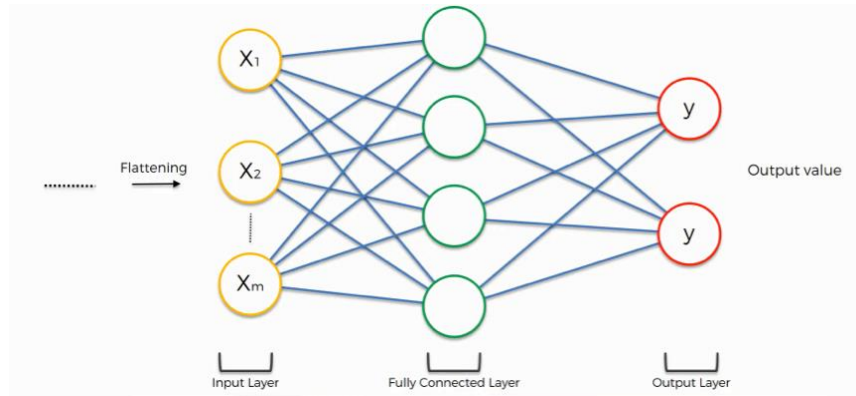
$$\text{zero padded} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The convolutional kernel is also known as a convolutional filter. These convolutional filters are used to extract certain features from the image. An example is the Sobel filter, which is based on gradient calculations.

$$\text{Sobel Detector} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

In a CNN, the convolutional filters are adjusted after every training epoch, or cycle. With enough training, the goal is to find the filter that extracts the correct features to identify the data.

After convolving, a pooling layer usually follows. The pooling layer serves to condense the data so that the convolutional layers can extract higher level data from the images. A CNN usually contains many layers of convolutional and pooling layers. After convolutional and pooling layers come the flattening layer, where the multi-dimensional data is flattened into a single dimension to be processed by the fully connected layers. The fully connected layers act similarly to a multi-layer perceptron. The inputs, which are one dimensional at this stage, are mixed then output to an output layer.



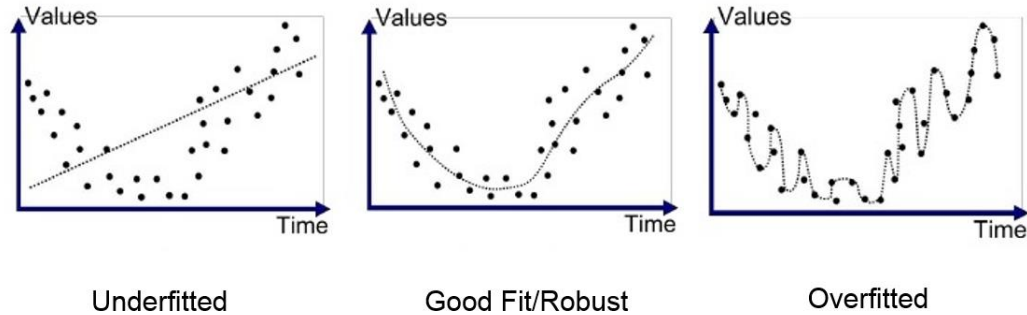
**FIGURE 15. THE FULLY CONNECTED LAYER [11]**

After convolution there is usually a pooling layer that condenses the data. Same as a basic neural work, weights are updated in CNNs. The weights updated in a CNN would be the convolutional filters. After multiple convolutional and pooling layers, CNNs feed that data through fully connected layers. In this layer, data is flattened into a stream of data, then compressed to the number of outputs desired. For example, if there are ten different classes, the fully connected layer should compress to ten outputs.

The goal of a CNN, just like a basic neural network, is to continuously update the convolutional filters until a certain desired performance level is reached. This process can be adjusted through learning rates. Learning rates could be something as simple as a fixed number or more sophisticated such as using Nesterov momentum to have a varying learning rate.

As powerful as CNNs are, there are certain limitations. Overfitting occurs when a neural network is overly trained for a particular dataset. The performance of an overfit neural network is excellent when using old data. When the neural network is tested with new and completely different data, the performance suffers. This can be mitigated by

including “dropout” rates in the neural network. At random, the neural network will drop neurons to prevent the neural network from over-adapting to the training data.



**FIGURE 16. VISUALIZATION OF FITTING IN A NEURAL NETWORK [12]**

Research has shown that, in general, a neural network with more layers will give better results. A research paper by Lawrence, Giles, Tsoi from the University of Queensland shows that as the number of hidden nodes in a neural network increases, the mean squared error of test data (misidentified test data) goes down [21]. However, in some cases a larger neural network is more susceptible to overfitting. Not only that, larger neural networks need to deal with vanishing gradient problem.

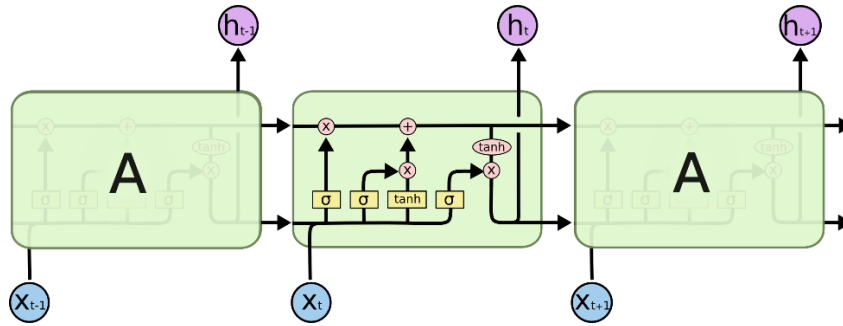
The vanishing gradient problem is due to the back propagation of a neural network. Back propagation occurs when a neural network is so deep that back propagation results in almost-zero adjustments of gradients. In CNNs, since datasets tend to be very large, data is often converted to half precision, or FP16. Since the adjustments to the convolutional filters can be near zero, there may not be enough precision to represent any adjustments to the filters.

The vanishing gradient problem can be solved by recurrent neural networks. This is a relatively recent breakthrough in neural networks. A recurrent neural network (RNN) can

process temporal data. RNNs can “remember” inputs and feed data backwards in the neural network. This reduces the vanishing gradients effect on deep neural networks.

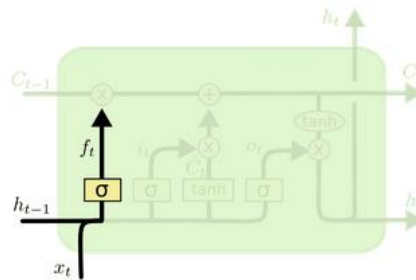
## 7.2 LSTM

LSTMs, or Long Short-Term Memory, is a type of RNN. The most important aspect of LSTMs is that the LSTM cells can pass information to another LSTM cell [2].



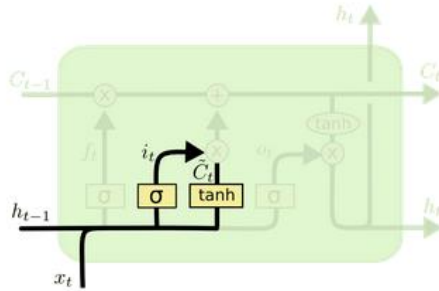
**FIGURE 17. LSTM CELLS CAN CONNECT WITH ONE ANOTHER [2]**

LSTM cells take in temporal data, represented as  $x_{t-1}, x_t, x_{t+1}$ , and pass it through the LSTM module. The top row that passes through each cell is what conveys information between the LSTM cells. The LSTM cell itself manipulates the information passed through from the previous cell. The outputs of the LSTM are the cell state and output for a fully connected layer.



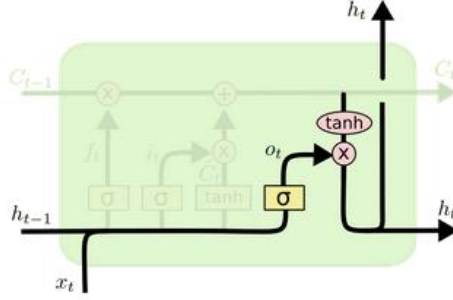
**FIGURE 18. THE INPUT BLOCK [2]**

The first part of the LSTM receives the information from the previous LSTM block, represented as  $h_{t-1}$ , and the input from the current time,  $x_t$ . The information  $h_{t-1}$  will first pass through a sigmoid gate, where it will decide to forget or keep the information. The output for this part,  $f_t$ , will be a number between 0 and 1. A “0” will mean to forget the information while a “1” will mean to keep this information. This number is then multiplied by the previous cell state  $C_{t-1}$ . This part of the LSTM is known as the “forget” gate.



**FIGURE 19. THE FORGET BLOCK [2]**

The next part of the LSTM block decides what information to store.  $h_{t-1}$ , the information passed from the last block, is passed through a sigmoid gate and through a hyperbolic tangent layer. The outputs of these operations are multiplied, then added to the output of the previous part. The resulting output is the new cell state,  $C_t$ . The part with the sigmoid function is known as the “input gate”.



**FIGURE 20. THE FORGET BLOCK [2]**

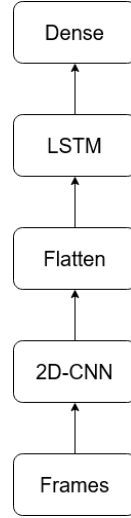
The last part of the LSTM updates the information passed to the next block and the output of the LSTM. The information output,  $h_t$ , is the cell state  $C_t$  passed through a hyperbolic tangent layer then multiplied by the last information input passed through a sigmoid gate. The output  $h_t$  is output to both a fully connected layer and to the neighboring LSTM cell. The cell state  $C_t$  is passed only to the next LSTM cell. This stage of the LSTM cell is known as the “output” gate.

With the combination of forget, input, and output gates, these LSTM cells can selectively remember and forget information passed from both the previous cell and current information input. LSTMs should perform well when there is temporal data involved, such as videos. In this thesis, since the data is temporal in nature, an LSTM architecture will be utilized.

### 7.3 LSTM-CNN

A specific type of neural network, the LSTM-CNN model, is designed to interface with video data [26]. A rough description of the model is shown below.





**FIGURE 21. A PRACTICAL IMPLEMENTATION OF A LSTM-CNN MODEL**

A 2D CNN is run for every frame of the video in parallel. The 2D CNNs run independently of each other, meaning that the filters within are adjusted separately. The output of the 2D CNNs are flattened then concatenated. This concatenated data is passed to the LSTM to evaluate. Then, a dense layer combines the data for output. The dense layer should have the number of classes that we are evaluating.

In this thesis, traditional CNNs will be explored along with LSTM/CNN architecture for performance.

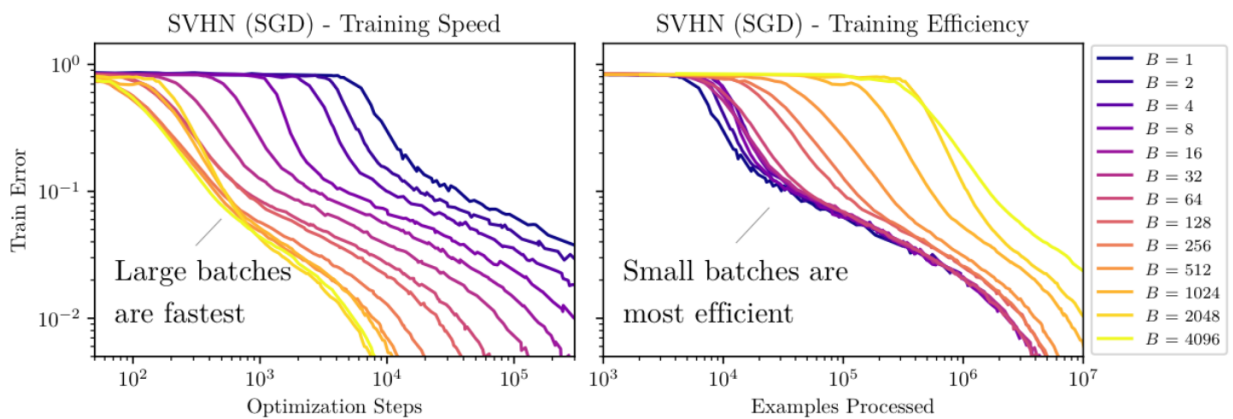
## 7.4 Practical Considerations

### 7.4.1 Batch Size

Batch size is defined as how many samples are processed at once. How large the batch of samples that can be trained is limited to memory. With more powerful hardware, larger batches could be assigned to the graphics card. This will speed up training per epoch (one training or testing cycle through the entire dataset) because more data is processed per batch. For example, for a dataset with 500 samples, a batch size of 100 will take 5 cycles

to complete an epoch. This would be faster compared to if the batch size was 5. It would take 20 cycles to complete an epoch compared to 5 using batch size of 100.

While it may be tempting to maximize the amount of graphics card memory used by using large epoch sizes, doing so can negatively affect convergence. OpenAI, a team that created an AI that can defeat professional players in a video game *DotA 2*, has done extensive study on this topic.



**FIGURE 22. TRAINING SPEED AND EFFICIENCY COMPARED TO BATCH SIZES [24]**

What is shown here is that with a large batch, training speed is the fastest. However, training efficiency is the best with small batch sizes. With smaller batches, it takes less epochs to reach maximum accuracy [24]. There is a tradeoff to be made with training speed and efficiency here.

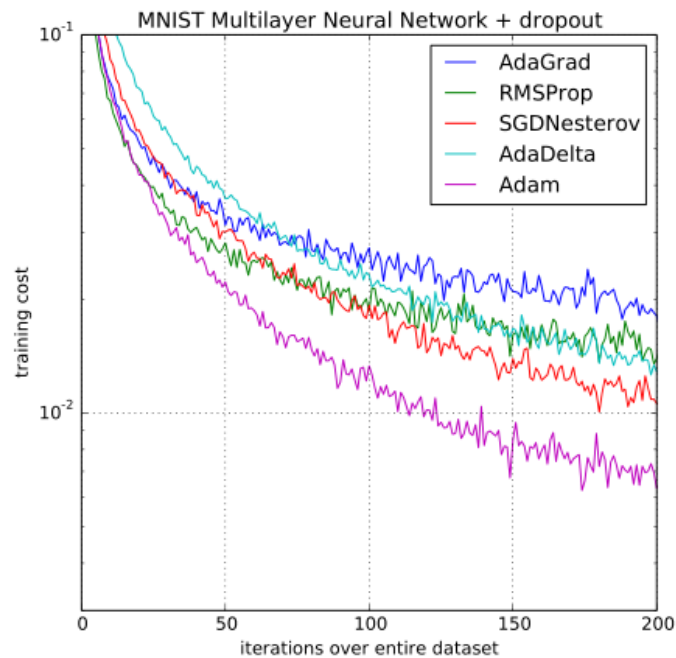
The default batch size used by TensorFlow is 32 samples [30]. Depending on the size of my neural network, I could potentially use batch sizes of 1024 to fill up all the memory on the graphics card. Training speed per epoch may be fast, but it would be inefficient and take many epochs to converge to an optimum result. The optimal batch size that was found, ironically, is the default TensorFlow batch size of 32.

### 7.4.2 Learning Algorithm

Learning rate is a scale factor that is applied to the back propagation during training.

Usually, learning rate is below 1 to avoid backpropagating increasing values. The classic fixed learning rate used is known as the stochastic gradient descent (SGD). In

TensorFlow, this implements a fixed learning rate with no adjustments made to it as training progresses. However, the fixed learning rate will take longer to reach a global optimum (in finding decision boundaries) compared to a more modern adaptive learning rate.



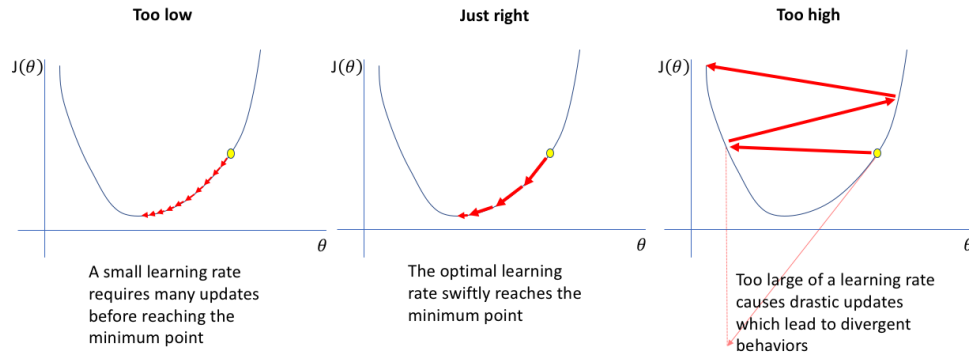
**FIGURE 23. ADAM OUTPERFORMS OTHER LEARNING RATE ALGORITHMS [18]**

More modern adaptive learning rate algorithms are AdaGrad, RMSProp, and Adam (Adaptive Momentum). Adam is the most recent one with it coming out in 2015.

Theoretically, Adam has the best performance; the number of epochs to reach peak

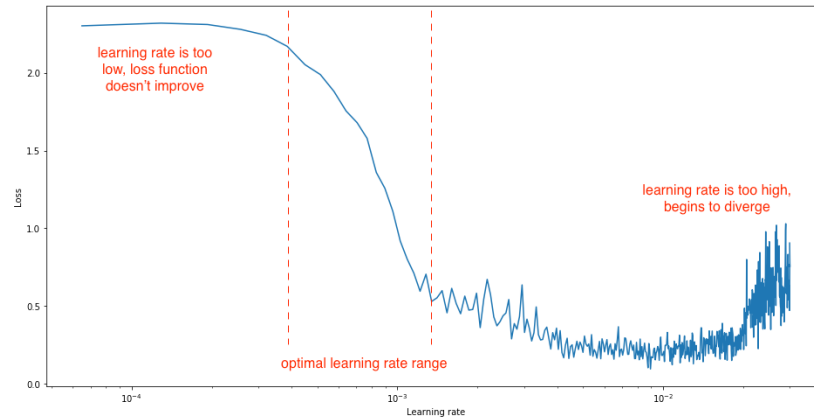
validation accuracy is the smallest out of all the other adaptive learning rate algorithms [6]. However, in practice Adam may not always be the best learning rate algorithm to use.

Finding the proper learning rate (the constant) is visualized below.



**FIGURE 24. EFFECT OF LEARNING RATE ON CONVERGING TO THE OPTIMUM RATE [5]**

A small learning rate will eventually approach the optimum, but it will also take longer than desired. Smaller learning rates could also get “stuck” on local optimums and not reach the global optimum solution. A large learning rate may learn quickly, but it can cause drastic changes that can lead to divergent behavior. Divergent behavior, for example, would be a neural network “unlearning” as it is run for more epochs. The “correct” learning rate is one that finds the optimum quickly without diverging but also not get stuck on local optimums.



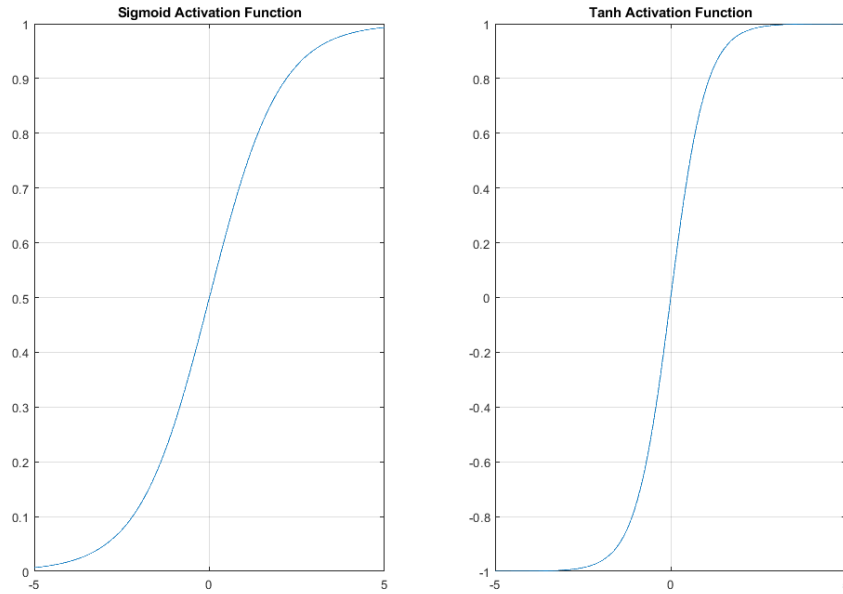
**FIGURE 25. VISUALIZATION OF THE OPTIMAL LEARNING RATE [5]**

Typically, learning rates are between 0.1 to 0.0001. This depends on the type of learning rate algorithm used. For example, stochastic gradient descent uses larger gradients such as 0.01 while Adam has its default set to 0.001. Adam also has other parameters to adjust, namely epsilon. The epsilon value is there to prevent any divide by zero in the implementation. The default is set at  $1e-8$ . However, the TensorFlow documentation suggests using 0.1 or 1 for image neural networks [33].

For the architectures detailed in a later section, the learning rate algorithms used will be SGD and Adam. The exact values of parameters will be different for every architecture.

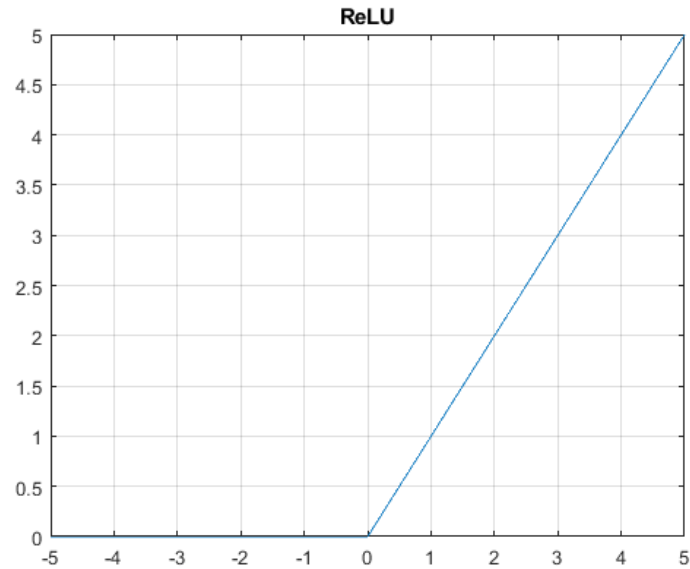
### 7.4.3 Activation Function

For each neuron in a neural network, the input of the data is multiplied by the weight of the neuron to form the output. To avoid the output ballooning to a large size, the output is normalized by activation functions. The activation function is needed to normalize the output data of a neuron to be between one and zero.



**FIGURE 26. SIGMOID FUNCTION (LEFT) AND TANH FUNCTION (RIGHT)**

Traditionally, activation functions such as hyperbolic tangent (tanh) and sigmoid are used. However, tanh and sigmoid is susceptible to the vanishing gradients problem, where for very high values of  $x$  there is no change in the output, causing the neural network to stop learning effectively. The solution to this is the rectified linear unit (ReLU) activation function. The function is linear above  $x$  equal to zero and equal to zero when  $x$  is negative.



**FIGURE 27. RECTIFIED LINEAR UNIT (ReLU) ACTIVATION FUNCTION**

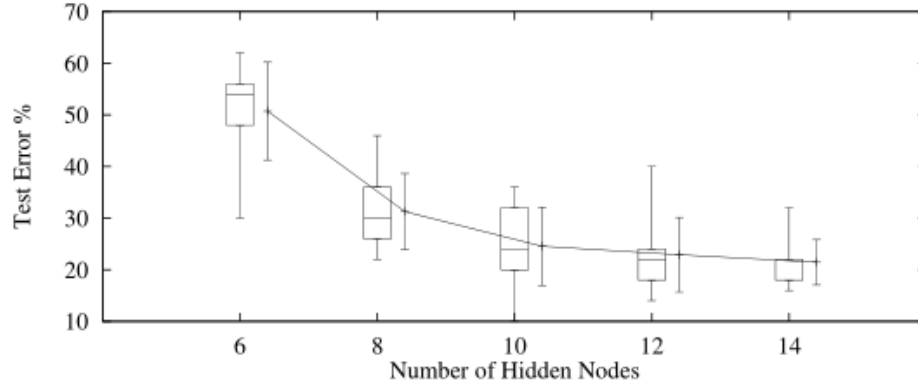
This solves the issue of vanishing gradients as the output scales linearly with input while also allowing for backpropagation. This is the activation function that will be used for the convolutional layers.

Since there are many classes (500 total), an activation function that accommodates multi-class output is needed. Therefore, the only solution is to use the softmax activation function as it can handle many classes at once. The softmax activation function returns a list of possibilities for the results. This activation function should only be used in the final fully connected output layer of the neural network.

#### 7.4.4 ARCHITECTURE THEORY

In theory, a larger and more complicated neural network should perform better than a shallower and simpler neural network. In a paper by Steve Lawrence, C. Lee Giles, Ah Chung Tsoi, *What Size Neural Network Gives Optimal Generalization? Convergence*

*Properties of Backpropagation* [21], it is found that deeper neural networks with more hidden layers generally produce lower testing error.



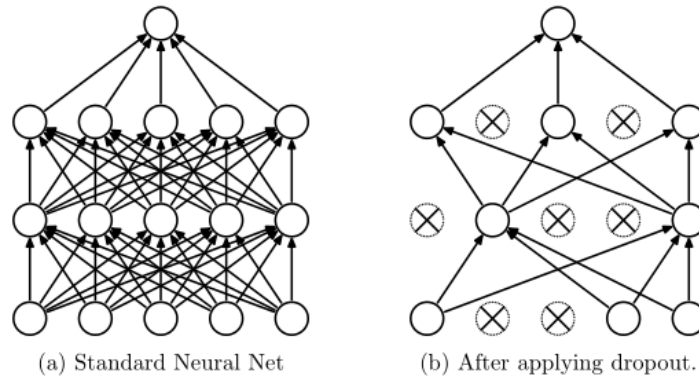
**FIGURE 28. NEURAL NETWORK DEPTH COMPARED TO TEST ERROR [21]**

The paper concludes that a simple explanation for improved performance for deeper neural networks is that “the extra degrees of freedom can aid divergence, i.e. the addition of extra parameters can decrease the chance of becoming stuck in local minima or on ‘plateaus’”. A CNN with more filters can form more complex decision boundaries for classification. In this thesis, deeper architectures will be compared to shallower ones to explore this theory.

#### 7.4.5 OVERFITTING

Overfitting is caused by the neural network overly adapting to one set of test data. Since neural networks are often run for more than 30 epochs, the filters in the neural network often overfit. A simple way to combat overfitting is to reduce the number of fully connected layers in a neural network in the dense layers towards the end.





**FIGURE 29. A STANDARD NEURAL NETWORK COMPARED TO A NEURAL NETWORK WITH DROPOUT APPLIED [22]**

Dropout prevents overfitting by randomly dropping connections in the fully connected layers at the end of a neural network. The downside in using dropout is that the neural network will train slower. A balance between training speed and overfitting will need to be made for the neural network architectures in this thesis [22].

## 7.5 Architectures

### 7.5.1 Architecture #1: 3D-CNN

This architecture is the same as Matt Rochford's thesis that had the highest testing accuracy in his results. This architecture consists of five convolutional and four pooling layers that are passed through a flatten layer, then to three fully connected layers, each with decreasing amounts of outputs. The architecture is detailed below.

Layer	Filters	Strides	Kernel	Output
Convolution 3D	16	1	2x2x2	28x59x99x16
Max Pooling 3D	-	1x2x2	1x2x2	28x29x49x16
Convolution 3D	32	1	2x2x2	27x28x48x32
Max Pooling 3D	-	1x2x2	1x2x2	27x14x24x32
Convolution 3D	64	1	2x2x2	26x13x23x64
Max Pooling 3D	-	1x2x2	1x2x2	26x6x11x64
Convolution 3D	128	1	2x2x2	25x5x10x128

Max Pooling 3D	-	1x2x2	1x2x2	25x2x5x128
Convolution 3D	256	1	2x2x2	24x1x4x256
Flatten	-	-	-	24576
FC1 (dropout=.5)	-	-	-	1024
FC2 (dropout=.5)	-	-	-	512
FC3	-	-	-	500

**TABLE 2. ARCHITECTURE #1**

This architecture is unique in that it uses 2x2x2 convolutional blocks. Most 3D CNNs use 3x3x3 convolutional blocks as there is a clear center to this block. Since this architecture does not include zero padding at the edges when convolving, the image gets smaller by one pixel in the frame, pixel width and height dimension every time when convolving. For a dataset of 100 words, the neural network has a 64.86% accuracy. For the full dataset, this figure should be significantly lower. This architecture will serve as the baseline result to compare to for my other architectures.

The optimal learning algorithm is found to be the stochastic gradient descent (SGD) with a learning rate of 0.02 and Nesterov momentum applied. The Nesterov momentum adjusts the learning rate by changing its value based on a momentum term. This architecture is run for 30 epochs.

### 7.5.2 Architecture #2: 3D CNN with 5 Fully Connected Layers

The idea of this architecture is to add more fully connected layers at the end of this neural network. The idea is to test the idea that a deeper neural network with more hidden nodes will produce less error [21]. The data, after being passed through the convolutional and flatten layers, are passed through more fully connected layers. Perhaps the extra fully connected layers will fine-tune the output of the convolutional layers more. Three extra fully connected layers with 1024 outputs are added in this architecture.

Layer	Filters	Strides	Kernel	Output
Convolution 3D	16	1	2x2x2	28x59x99x16
Max Pooling 3D	-	1x2x2	1x2x2	28x29x49x16
Convolution 3D	32	1	2x2x2	27x28x48x32
Max Pooling 3D	-	1x2x2	1x2x2	27x14x24x32
Convolution 3D	64	1	2x2x2	26x13x23x64
Max Pooling 3D	-	1x2x2	1x2x2	26x6x11x64
Convolution 3D	128	1	2x2x2	25x5x10x128
Max Pooling 3D	-	1x2x2	1x2x2	25x2x5x128
Convolution 3D	256	1	2x2x2	24x1x4x256
Flatten	-	-	-	24576
FC1	-	-	-	1024
FC2	-	-	-	1024
FC3	-	-	-	1024
FC4 (dropout=.5)	-	-	-	1024
FC5 (dropout=.5)	-	-	-	512
FC6	-	-	-	500

**TABLE 3. ARCHITECTURE #2**

The learning algorithm used in this architecture is Adam with default learning rate and an epsilon of 0.1. Only the last two dense layers will have dropout included. This architecture is run for 30 epochs.

### 7.5.3 Architecture #3: VGG-16 based architecture

VGG-16 is a classic image classification CNN. This architecture needed to be adapted to 3D to run for video data. The detail of the original architectures and its variations are shown below.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 <b>LRN</b>	conv3-64 <b>conv3-64</b>	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 <b>conv3-128</b>	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 <b>conv1-256</b>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 conv3-256 <b>conv3-256</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

**FIGURE 30. THE ORIGINAL VGG NEURAL NETWORK ARCHITECTURE CONFIGURATIONS**

[20]

Shown above are the different configurations of VGGNet, with configuration A having 11 layers and configuration E having 19 layers. VGG-16 is known to be a very large and cumbersome neural network with many filters. The number of filters and convolutional layers means that the VGG neural network has many parameters. Since video data is much larger than image data per batch, adjustments to the number of filters were made to avoid going over the graphics card memory limit. The adjustments are shown below.

Layer	Filters	Strides	Kernel	Output
Convolution 3D	16	1	3x3x3	29x60x100x16
Max Pooling 3D	-	1x2x2	1x2x2	29x30x50x16
Convolution 3D	32	1	3x3x3	29x30x50x32
Max Pooling 3D	-	1x2x2	1x2x2	29x15x25x32
Convolution 3D	64	1	3x3x3	29x15x25x64

Convolution 3D	64	1	3x3x3	29x15x25x64
Max Pooling 3D	-	1x2x2	1x2x2	29x7x12x64
Convolution 3D	128	1	3x3x3	29x7x12x128
Convolution 3D	128	1	3x3x3	29x7x12x128
Max Pooling 3D	-	1x2x2	1x2x2	29x3x6x128
Convolution 3D	128	1	3x3x3	29x3x6x128
Convolution 3D	128	1	3x3x3	29x3x6x128
Max Pooling 3D	-	1x2x2	1x2x2	29x1x3x128
FC1	-	-	-	2048
FC2	-	-	-	1024
FC3	-	-	-	500

**TABLE 4. ARCHITECTURE #3**

The learning algorithm used is Adam with a learning rate of 0.001 (default learning rate) and epsilon of 0.1, as suggested from the TensorFlow documentation. For this neural network, no dropout is applied as per the original VGG algorithm. This architecture is run for 30 epochs.

#### 7.5.4 Architecture #4: Multiple Towers

Multiple towers (MT) is an architecture introduced in Chung and Zisserman's *Lip Reading in the Wild*. The frames are processed individually through 2D CNNs before concatenating them. After concatenation, the data is passed through a 3D CNN similar to the previous architectures.

This architecture delays the time-domain operations until after the first convolutional and pooling layers. This gives some tolerance for errors in lip tracking in between frames. For example, the lip detection algorithm could frame the lips differently for every frame. The first frame could have the lips centered properly while the next frame could have the lips offset by a significant margin. This architecture is meant to be resilient to these failures. Theoretically, if the lip detection and framing process is done perfectly, this architecture should not affect accuracy results much.

Layer	Filters	Strides	Kernel	Output
Convolution 2D	16	1	3x3	60x100x16
Max Pooling 2D	-	1x2x2	1x2	30x50x16
Concatenate	-	-	-	29x30x50x16
Convolution 3D	32	1	3x3x3	29x30x50x32
Max Pooling 3D	-	1x2x2	1x2x2	29x15x25x32
Convolution 3D	64	1	3x3x3	29x15x25x64
Convolution 3D	64	1	3x3x3	29x15x25x64
Max Pooling 3D	-	1x2x2	1x2x2	29x7x12x64
Convolution 3D	128	1	3x3x3	29x7x12x128
Convolution 3D	128	1	3x3x3	29x7x12x128
Max Pooling 3D	-	1x2x2	1x2x2	29x3x6x128
Convolution 3D	128	1	3x3x3	29x3x6x128
Convolution 3D	128	1	3x3x3	29x3x6x128
Max Pooling 3D	-	1x2x2	1x2x2	29x1x3x128
FC1	-	-	-	2048
FC2	-	-	-	1024
FC3	-	-	-	500

**TABLE 5. ARCHITECTURE #4**

Keras makes running 2 dimensional CNNs for each frame trivial. The function

*TimeDistributed()* allows a 2D-CNN to be run for every frame with just using one line of code [34].

The learning algorithm used is Adam with the default learning rate and an epsilon of 0.1.

Since MT is based off architecture #3 (VGG based), no dropout will be included. This architecture is run for 30 epochs.

#### 7.5.5 Architecture #5: LSTM-CNN

This architecture is known as the CNN-LSTM architecture. A 2-dimensional CNN is run for every frame. Afterwards, the data from the CNNs are concatenated, then fed through the LSTM modules explained in the above sections. Keras function *TimeDistributed()* is once again used to run the 2D-CNNs in parallel. The outputs are flattened then concatenated to be input to the LSTM modules. The LSTM modules can have many

outputs to connect to a fully connected layer [4]. With trial and error, the best results were found to be using LSTM output dimension of 1024.

In this architecture, a stacked LSTM layer is used instead of just a single layer. The output of the LSTM layers is then fed to a fully connected layer with 500 outputs corresponding to the number of classes.

Layer	Filters	Strides	Kernel	Output
Convolution 2D	16	1	3x3	60x100x16
Max Pooling 2D	-	1x2	1x2	30x50x16
Convolution 2D	32	1	3x3	30x50x32
Max Pooling 2D	-	1x2	1x2	15x25x32
Convolution 2D	64	1	3x3	15x25x64
Convolution 2D	64	1	3x3	15x25x64
Max Pooling 2D	-	1x2	1x2	7x12x64
Convolution 2D	128	1	3x3	7x12x128
Convolution 2D	128	1	3x3	7x12x128
Max Pooling 2D	-	1x2	1x2	3x6x128
Convolution 2D	128	1	3x3	3x6x128
Convolution 2D	128	1	3x3	3x6x128
Max Pooling 2D	-	1x2	1x2	1x3x128
Flatten	-	-	-	384
Concatenate	-	-	-	29x384
LSTM	-	-	-	29x1024
LSTM				1024
FC				500

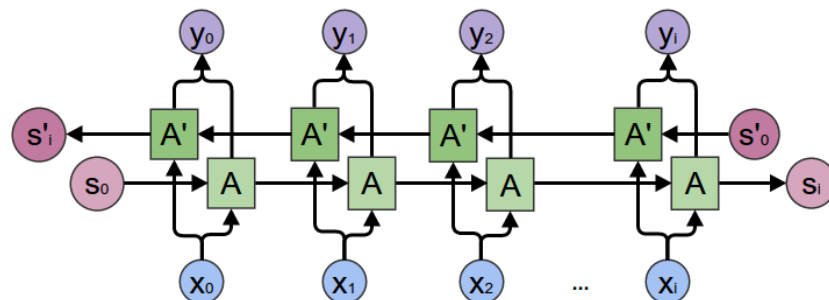
**TABLE 6. ARCHITECTURE #5**

Theoretically, because LSTMs were designed specifically for temporal data, it should outperform 3D-CNNs and the MT architecture. LSTMs are more suited for prediction of future data using past data. This can include lyric writing and orchestral composition, where past data and context are important in predicting future data. Therefore, the gains in accuracy is expected to improve over 3D-CNNs and MT, but not by a significant margin.

The learning algorithm used is Adam with the default learning rate and an epsilon of 0.1. Since this architecture is based off architecture 4 (VGG based), no dropout will be included. This architecture is run for 100 epochs.

#### 7.5.6 Architecture #6: CNN and Bidirectional LSTM

This architecture is very similar to architecture #5, but with bidirectional LSTM layers. For a unidirectional LSTM, information only flows in one direction. Therefore, unidirectional LSTMs only preserves information from the past. Bidirectional LSTMs will take inputs and run it in both directions, meaning that it can take into account past information and future information. Since information flows both ways in a bidirectional LSTM, the amount of LSTM cells needed are doubled. One can create a bidirectional LSTM by using two unidirectional LSTMs. One unidirectional LSTM will have information flow one way and the other LSTM the opposite way. The two LSTM layers are then concatenated to form a bidirectional LSTM. Luckily, TensorFlow has a built-in function that performs this task.



**FIGURE 31. BIDIRECTIONAL FLOW OF INFORMATION OF A BIDIRECTIONAL LSTM [15]**

It is difficult to say for sure if results will improve using a bidirectional LSTM compared to a unidirectional LSTM. Because bidirectional LSTMs take into account both past and future information, it should in theory be able to understand context better than



unidirectional LSTMs. Perhaps by feeding both future and past lip frames into the LSTM, accuracy may improve. However, one can argue that accuracy will not improve because there is little need for context to predict words based on lip frame data.

The CNN used for this architecture is simply architecture #3, which is the VGG based architecture. The results will be discussed in the next section.

Layer	Filters	Strides	Kernel	Output
Convolution 2D	16	1	3x3	60x100x16
Max Pooling 2D	-	1x2	1x2	30x50x16
Convolution 2D	32	1	3x3	30x50x32
Max Pooling 2D	-	1x2	1x2	15x25x32
Convolution 2D	64	1	3x3	15x25x64
Convolution 2D	64	1	3x3	15x25x64
Max Pooling 2D	-	1x2	1x2	7x12x64
Convolution 2D	128	1	3x3	7x12x128
Convolution 2D	128	1	3x3	7x12x128
Max Pooling 2D	-	1x2	1x2	3x6x128
Convolution 2D	128	1	3x3	3x6x128
Convolution 2D	128	1	3x3	3x6x128
Max Pooling 2D	-	1x2	1x2	1x3x128
Flatten	-	-	-	384
Concatenate	-	-	-	29x384
Bi-LSTM	-	-	-	29x2048
Bi-LSTM				2048
FC				500

**TABLE 7. ARCHITECTURE #6**

## 8. Results

### 8.1 Preprocessing result

Most of the videos output from our lip detection algorithm is valid. As a reminder, any video with more than one face or no face are not used. A chart of the results is shown below.

File	Valid Videos	Total Videos	Valid Percentage
train	473,847	500,000	95%
val	24,241	25,000	97%
test	24,194	25,000	97%

**TABLE 8. PREPROCESSING RESULTS**

Since the number of invalid videos are relatively small (less than 5% for all data types), I believe that there is no need to investigate the small portion of invalid results.

### 8.2 Neural Network results

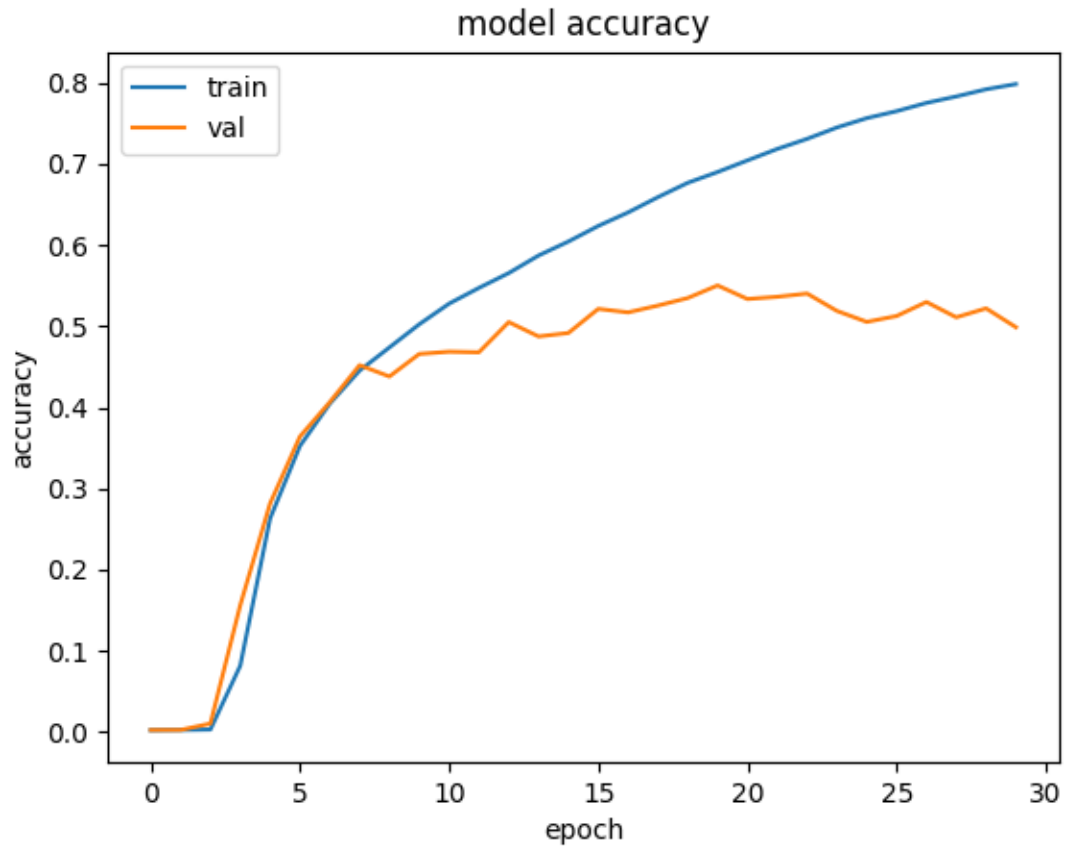
For a classification task with 500 different words, a correct random guess would have a probability of 0.2%. With such a large number of classes, a 50% correct identification would be considered good (This is compared to other similar words, such as Chung and Zisserman’s *Lip Reading in the Wild*). My target is to surpass human lip-reading accuracy (45%) and Chung and Zisserman’s work (61.1%).

#### 8.2.1 Architecture #1: 3D-CNN

Matt Rochford’s best architecture ends up with a final testing accuracy of 53.02%. This surpasses human lip-reading accuracy but does not surpass Chung and Zisserman’s work. This architecture will be my baseline to improve upon.

Peak Training Accuracy	Peak Validation Accuracy	Testing Accuracy	Learning Algorithm
79.86%	55.03% at epoch 19	53.02%	SGD, LR=0.02, Nesterov Momentum

**TABLE 9. ARCHITECTURE #1 RESULTS**



**FIGURE 32. ACCURACY CHANGES PER EPOCH FOR ARCHITECTURE #1**

### 8.2.2 Architecture #2: 3D CNN with 5 Fully Connected Layers

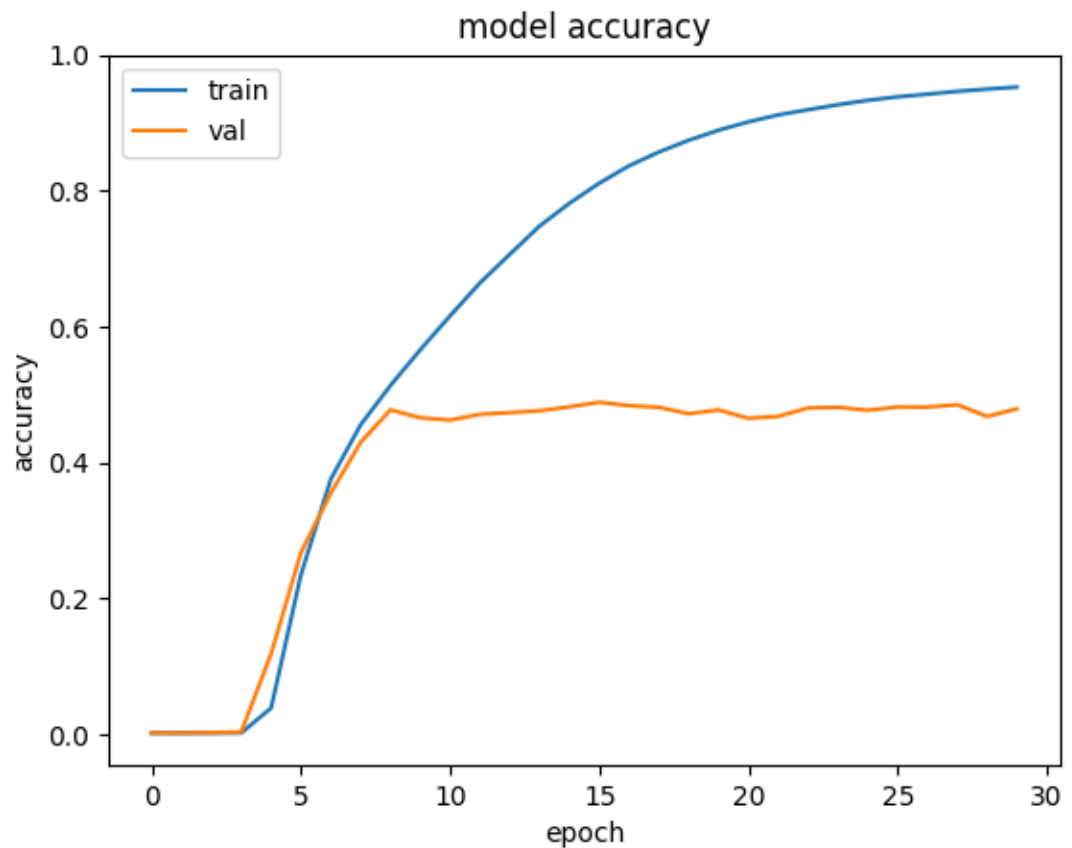
More fully connected layers should give more parameters to adjust so that a more fine-tuned decision boundary can be made. The results of this architecture compared to baseline is worse by roughly 5%.

An explanation for the worse performance is that the increased number of dense layers at the end does nothing for the neural network itself. The dense layers at the end merely “mixes” the output of the convolutional layers and seems to have a negative effect if there are too many layers. A large amount of work is done by the convolutional filters for

the video data. Increasing the number of convolutional layers instead of dense layers may increase performance.

Peak Training Accuracy	Peak Validation Accuracy	Testing Accuracy	Learning Algorithm
95.27%	48.87% at epoch 16	47.77%	SGD, LR=0.02, Nesterov Momentum

**TABLE 10. ARCHITECTURE #2 RESULTS**



**FIGURE 33. ACCURACY CHANGES PER EPOCH FOR ARCHITECTURE #2**

### 8.2.3 Architecture #3: VGG-16 Based Architecture

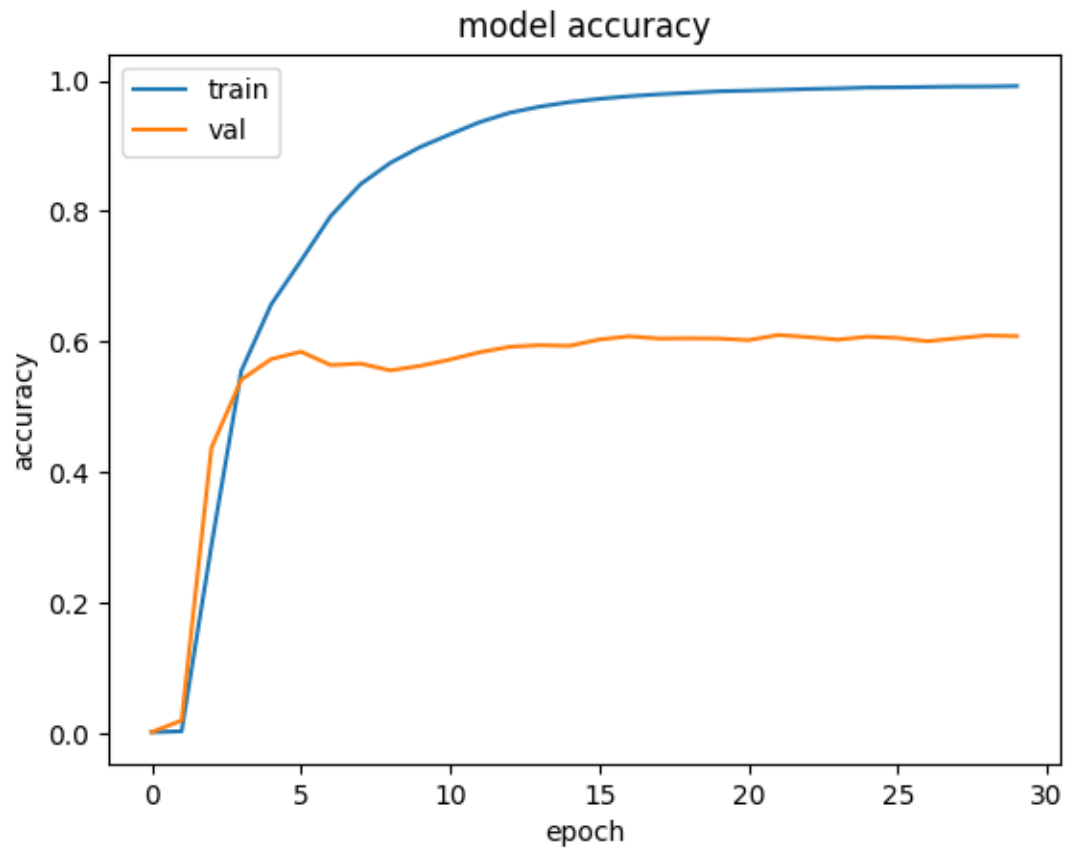
The VGG-16 based architecture comes very close to Chung and Zisserman's multiple towers architecture accuracy of 61.1%.

This somewhat supports the theory discussed earlier about deeper neural network architectures yielding better accuracy. There are simply more convolutional filters to adjust, which lets the neural network have a more fine-tuned decision boundary.

Compared to the baseline architecture, there is an improvement of around 7%.

Peak Training Accuracy	Peak Validation Accuracy	Testing Accuracy	Learning Algorithm
99.15%	61.02% at epoch 22	59.95%	Adam, LR=0.001, Epsilon = 0.1

**TABLE 11. ARCHITECTURE #3 RESULTS**



**FIGURE 34. ACCURACY CHANGES PER EPOCH FOR ARCHITECTURE #3**

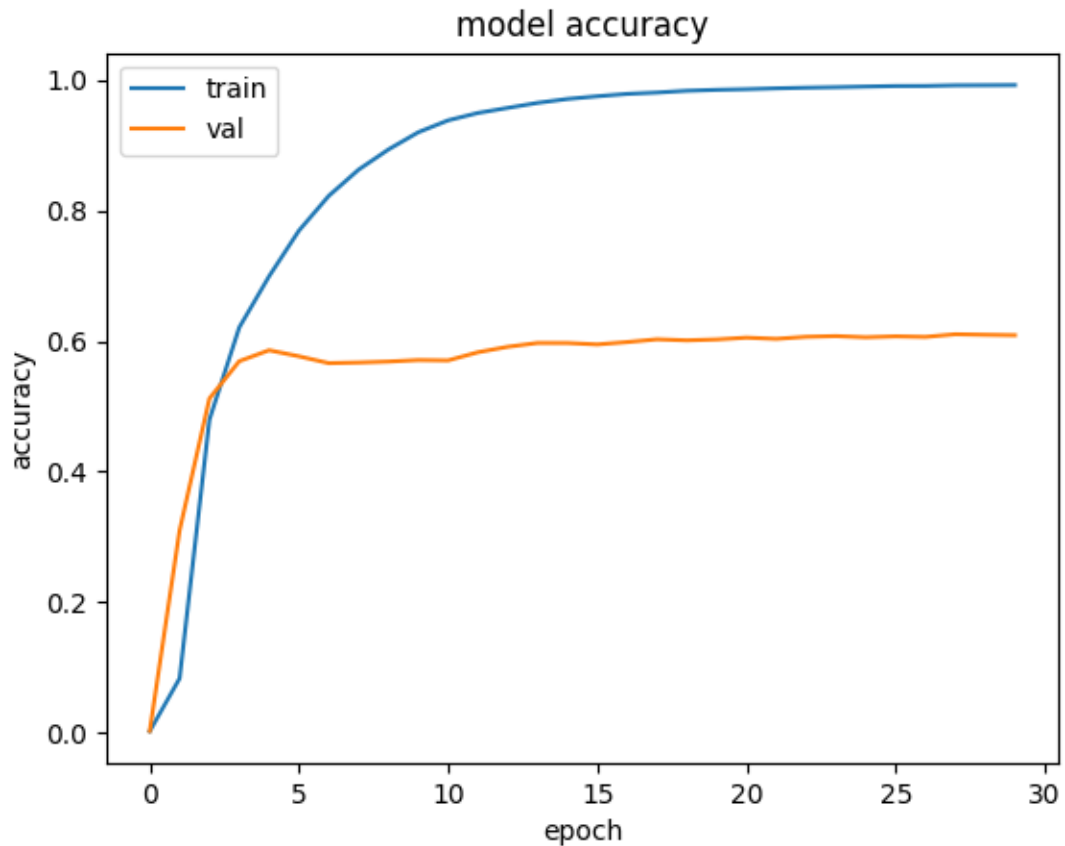
#### 8.2.4 Architecture #4: Multiple Towers

This architecture improves upon architecture #3 (VGG based) by a small amount. Since this architecture is based on architecture #3, the difference in the results should be minor.

The objective of the MT architecture is to increase the error tolerance of our neural network. Since the test accuracy between architecture #3 and multiple towers is very similar (MT is better by 0.5%), this implies that the preprocessing of video data is relatively error free.

Peak Training Accuracy	Peak Validation Accuracy	Testing Accuracy	Learning Algorithm
99.27%	61.05% at epoch 28	60.43%	Adam, LR=0.001, Epsilon = 0.1

**TABLE 12. ARCHITECTURE #4 RESULTS**



**FIGURE 35. ACCURACY CHANGES PER EPOCH FOR ARCHITECTURE #4**

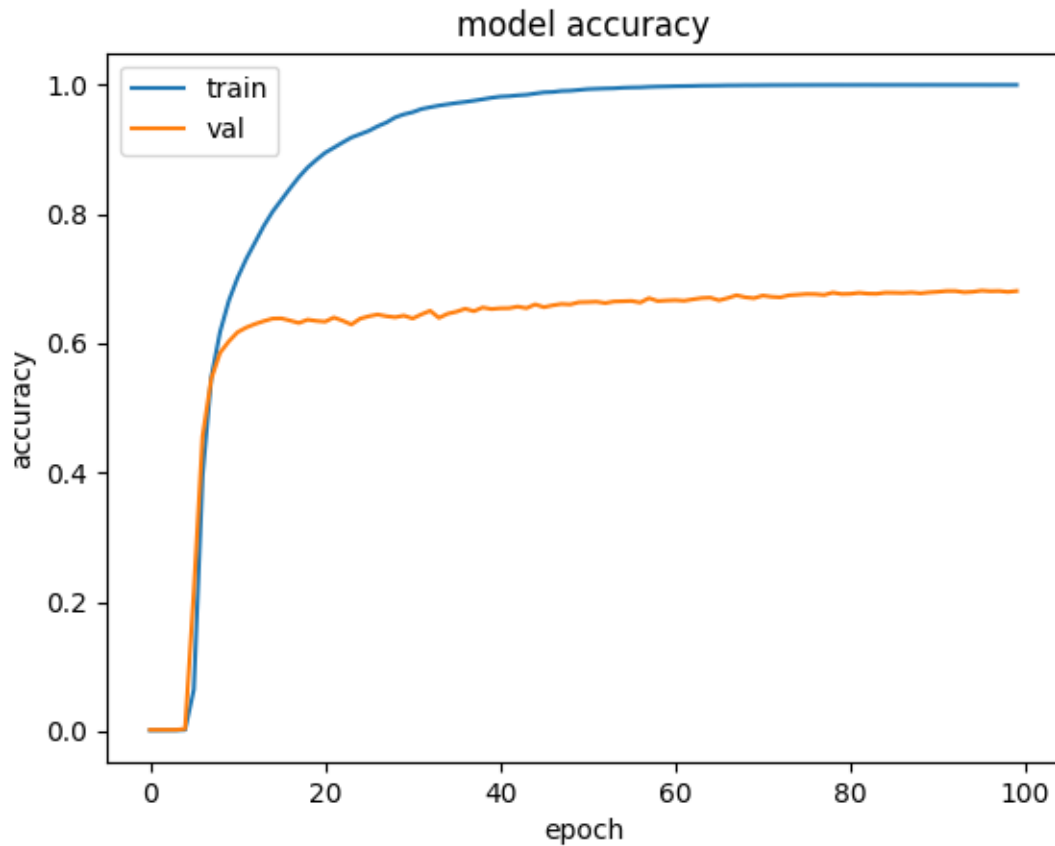
#### 8.2.5 Architecture #5: LSTM-CNN

It took many trials and errors to find out the number of epochs to run for this neural network. In the end, 100 epochs were found to be the right number of epochs to use. Towards the end, training accuracy was reaching 100%, meaning that more training would not increase accuracy any longer. For 100 epochs, this neural network took the second longest to run out of all the different architectures at 34 hours. While training time is long, testing time is similar to other neural network architectures.

Despite the lengthened run time, this neural network outperformed the best non-LSTM architecture by a significant margin. It outperformed the baseline architecture by a staggering 14% and outperformed the best CNN only architecture by 7%.

Peak Training Accuracy	Peak Validation Accuracy	Testing Accuracy	Learning Algorithm
100%	68.17% at epoch 96	67.14%	Adam, LR=0.005, Epsilon = 0.1

**TABLE 13. ARCHITECTURE #5 RESULTS**



**FIGURE 36. ACCURACY CHANGES PER EPOCH FOR ARCHITECTURE #5**

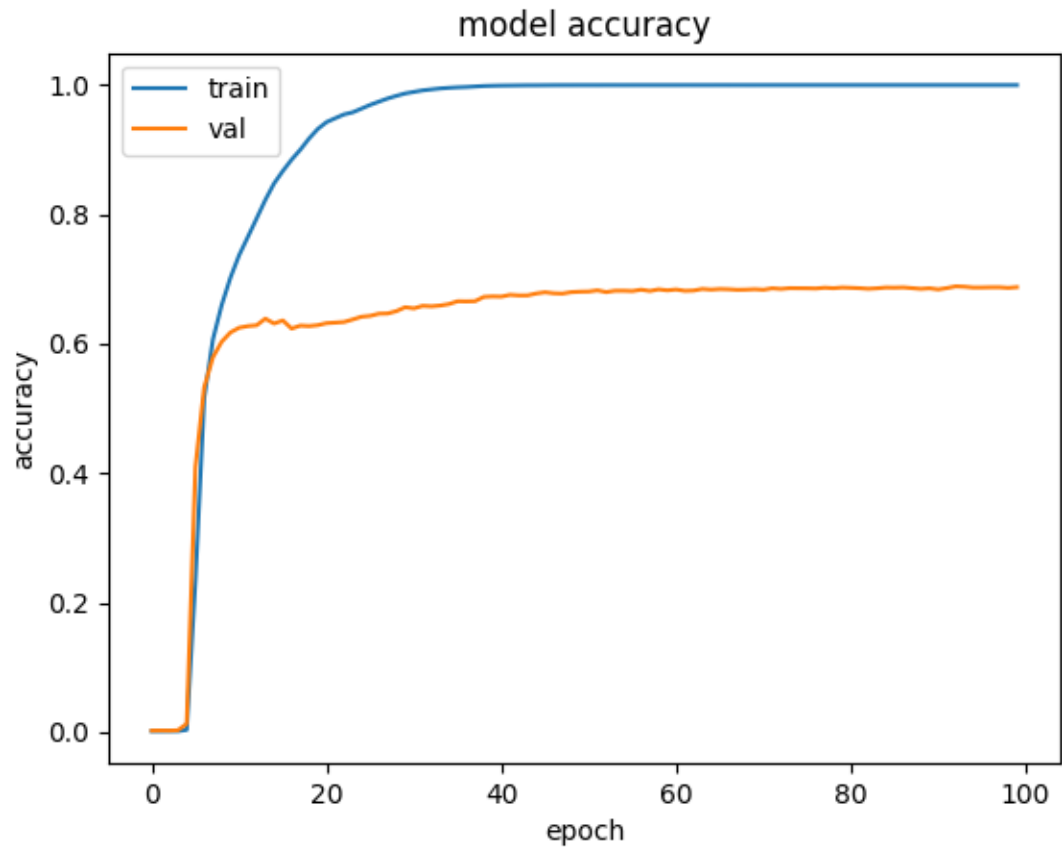
#### 8.2.6 Architecture #6: CNN and Bidirectional LSTM

The bidirectional LSTM architecture performed better than the CNN-LSTM architecture, if only by a bit. Similar to architecture #5 (CNN-LSTM), this architecture is run for 100 epochs with the same learning algorithm. This took the longest to run out of all the architecture because of the number of epochs needed to run as well as the added complexity compared to architecture #5. This architecture outperformed the CNN-LSTM architecture by 0.7%, which is a small but appreciable gain.



Peak Training Accuracy	Peak Validation Accuracy	Testing Accuracy	Learning Algorithm
100%	68.88% at epoch 93	67.93%	Adam, LR=0.005, Epsilon = 0.1

**TABLE 14. ARCHITECTURE #6 RESULTS**



**FIGURE 37. ACCURACY CHANGES PER EPOCH FOR ARCHITECTURE #6**

### 8.2.7 Comparison of Different Architectures

Architecture	Test Accuracy	Parameters
Matt's	53.02%	26,296,932
Matt's + Extra Dense Layers	47.77%	28,396,132
VGG based 3D CNN	59.95%	27,148,372
Multiple Towers with VGG	60.43%	27,148,084
LSTM + VGG	67.14%	15,253,300
Bi-LSTM + VGG	67.93%	37,293,376

**TABLE 15. OVERALL TEST ACCURACY AND PARAMETERS**

For the architectures, architecture #6 performed the best while architecture #2 performed the worse. Despite the extra layers for architecture #2, performance is not good. For the non-LSTM results, the architecture #4 had the highest accuracy. While it is shown that our lip framing algorithm is good, the MT architecture manages to catch some errors, giving a 0.5% improvement in accuracy over architecture #3.

The Bi-LSTM architecture also took the longest to run. Having the greatest number of parameters meant that it took just under two days to fully run 100 epochs. The improvement over the regular LSTM architecture nets around 0.8%, which is a small but appreciable gain.

The learning rates for each architecture had to be carefully tuned. Other than the first architecture where the classic SGD optimizer is used, the Adam optimizer is used for everything else. Following TensorFlow’s suggestion to use epsilon equal to 0.1 for image training helped immensely and was used for all architectures that utilized Adam. The learning rates used were typically the default value of 0.001. The LSTM module needed a slightly higher learning rate or 0.005 than others to learn effectively. For LSTM, there were issues when learning rate was too high and learning began to diverge. In one case, the LSTM architecture “unlearned” everything halfway through training.

### 8.3 Using Difference of Frames

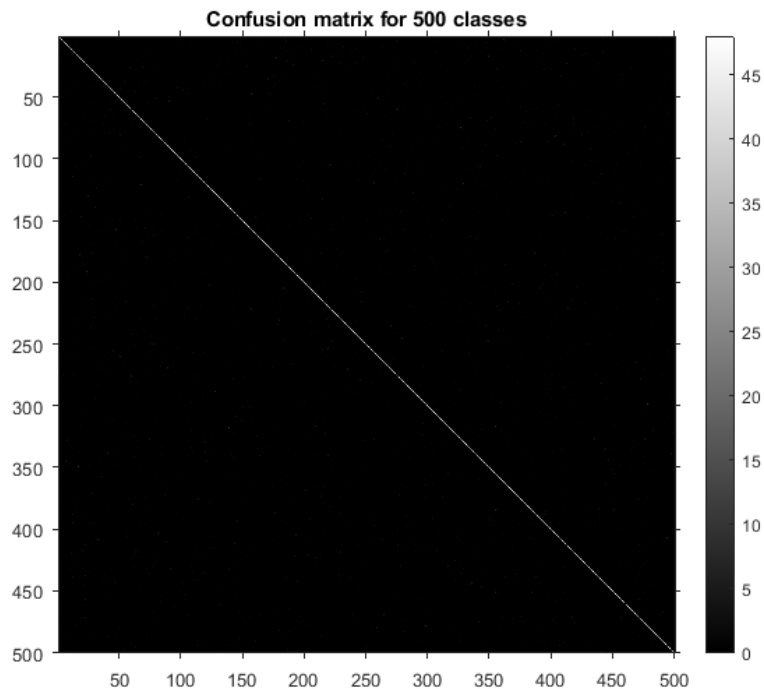
Data Type	Peak Training Accuracy	Peak Validation Accuracy
Raw grayscale	100%	68.17%
Frame Difference	100%	66.66%

**TABLE 16. COMPARING TWO DIFFERENT DATASET TYPES**

The different datasets were run using the best performing architecture, which is architecture #6. The raw grayscale data performed 1.5% better overall compared to taking the frame differences.

#### 8.4 Confusion Matrix

The confusion matrix is used to check for commonly confused word pairs. The python script runs both *evaluate()* and *predict()* to evaluate the accuracy and to predict the results given the data. Using the scikit-learn library, the output of *predict()* is then plotted against actual results for the confusion matrix. The output of the confusion matrix function is printed to a text file, which is then read by MATLAB. Since there are so many classes, the analysis of the confusion matrix is done in MATLAB.



**FIGURE 38. CONFUSION MATRIX FOR 500 CLASSES**

The top ten confused word pairs are (note that many words are tied in rank):

1	MILLION	BILLION
2	HOUSING	HOUSE
3	HAPPENED	HAPPEN
4	REPORT	REPORTS
4	PRICE	PRESS
4	SPENT	SPEND
4	LIVING	GIVING
5	STILL	UNTIL
5	INDUSTRY	HISTORY
5	PARENTS	POWERS

**TABLE 17. MOST CONFUSED WORD PAIRS**

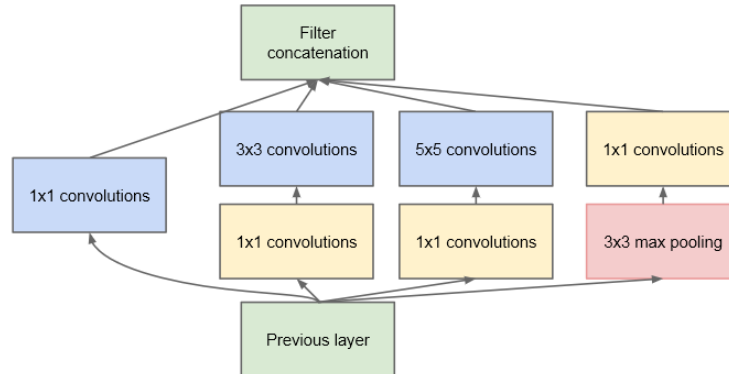
The first category of confused words are words that are plurals of each other. An example is “benefit” and “benefits”. Another category of confused words are words that share all the same syllables except for a single word. An example is “billion” and “million”, where the “illion” part of the words are shared. The last category of confused words is when spoken sound is done with tongue movement instead of lip movement. This includes words like “parents” and “powers”, where the “ts” and “ers” sounds are produced by tongue movement instead of lip movement.

### 8.5 Failed approaches not included in results

One failed approach, which is not shown in the results, were to use a pre-built neural network architecture in Keras as a front end to the LSTM modules. Architectures tried were: DenseNet201, InceptionV3, and ResNet50. These all have high image recognition accuracy and I expected that to carry over to video classification accuracy. However, there are limitations to these pre-built neural networks. For example, Google’s InceptionV3 is not able to take images smaller than 75x75, which one of our dimensions is 60 pixels, making it incompatible with Inception V3. The results for ResNet50 and DenseNet201 performed far worse than even our worst model at around 40% accuracy. This is not to mention the added complexity of neural networks such as ResNet50 and

DenseNet201 that significantly increases runtime. It took almost two hours to run a single epoch compared to the usual 45 minutes to one hour per epoch for the other architectures.

Another failed approach would be my custom implementation of Google's Inception module. I implemented the Inception V3 module shown below.



**FIGURE 39. INCEPTION MODULE WITH DIMENSION REDUCTION [35]**

This neural network is much more complicated compared to VGG-16. Instead of one linear CNN, the Google Inception module is comprised of a few parallel CNNs concatenated at the end. I was only able to implement a single module due to the memory requirements for the complicated Inception V3 architecture. For that, many inception modules are needed along with global pooling layers. With 27 layers, it is far deeper than any neural network I have. This custom version of Inception has 405,825,210 parameters, making it significantly heavier compared to other neural networks I implemented.

The performance was disappointing, probably due to the massively scaled downsize of the Inception implementation. Running on my sample dataset (10 words instead of the full 500-word library) only yielded 83% accuracy, compared to architecture #6 that scored 95% in my sample dataset, the results are very poor.

It is difficult to explain why these more complex and deep pre-built neural networks would perform so poorly compared to the relatively simple neural networks I concocted. The only explanation I can offer is that the neural networks are simply just not compatible for usage with video data. Google's Inception is more suitable for classifying images rather than detecting movement of lips.

## 9. Conclusion

### 9.1 Summary

This thesis has detailed different approaches to pre-processing and neural network architecture types. The best results came from pre-processing the images to be grayscale and using a stacked LSTM neural network. During testing, the accuracy was 67.14%, higher than human lip-reading ability (around 45%) and higher than recent works (61.1%). Other works that utilize LSTMs achieve higher accuracy than my thesis does but uses a far smaller word pool.

The hardware provided by the computer science department allowed me to spend resources freely, especially system RAM and graphics card memory. The whole dataset of 500 words, preprocessed, used up to 160 gigabytes of RAM. To put that in perspective, a high-end desktop computer usually has 32 gigabytes of RAM, far less than the 160 gigabytes I needed. The graphics memory allowed me to explore deeper architectures and use larger batch sizes to speed up training.

For this thesis, I was able to fully utilize the entire LRW dataset of 500 words. Because of this, I am able to compare my work directly to Chung and Zisserman's work.

Compared to other datasets, LRW provided high quality recordings and was the easiest to work with. All of the videos are cut to be exactly 29 frames long and are already sorted to test, training, and validation groups.

The lip frames extraction program run using Dlib had over 95% success rate overall.

Because the number of dropped videos is low, no further investigation into the error was

done. The images are first processed to be grayscale, then some other effects are done to potentially enhance the images for the neural network.

The focus of this thesis is the research of different neural network architectures and the effect that different parameters has. Parameters such as learning algorithm, batch size, convolutional kernel size, and dropout are adjusted to find what is optimal. Overall, the Adam optimizer is found to be the best learning algorithm for a majority of the neural networks. The optimal batch size is simply the default size. Most architectures utilized a 3x3 convolutional kernel. Some dropout is used to balance between training speed and overfit reduction.

Many different architectures are tested in this thesis. The best 3D-CNN architecture, the multiple towers architecture, has around 60.43% test accuracy. This architecture is based off the VGG16 architecture. While it is no longer a cutting-edge architecture, its simple structure makes it ideal to implement. The VGG architecture is notorious for being memory intensive, so the number of filters used per layer is reduced. From testing, what makes the difference in accuracy is convolutional layers and not fully connected layers. Fully connected layers have little relevance in finding filters that detect lip features while convolutional layers do. Since my lip frames are of size 60x100 pixels, the maximum number of pooling layers that can be done is five.

My LSTM architecture trumps the best 3D-CNN architecture by a large margin, having 67.14% test accuracy. The 3D-CNN's ability to perceive temporal data is primarily through the kernel size used. A 3x3x3 convolutional cube only considers data one frame ahead and behind it. However, an LSTM can consider all past frames input. Because it is



able to look further behind, or ahead in the case of bidirectional LSTMs, the results are far superior to 3D-CNNs.

## 9.2 Challenges

On the preprocessing end, the approach of taking differences between frames in order to obtain lip movement did not work well as an input for the neural network. The accuracy dropped by around 4 percent compared to using raw grayscale data. Since people tend to move when they speak, the lips move along with the face. Taking the difference between frames may not only capture lip movement but also facial movement. Perhaps aligning the lips so that the edges of the lips are in the same locations for every frame may solve this issue. However, I was not able to find a way to achieve this.

Adjusting the neural network took a lot of time. To adjust a small parameter such as learning rate, the neural network data must be loaded into memory, then the graphics card needs to run the neural network. To combat this, I created a smaller dataset with only 10 words. This allowed me to prototype different parameters such as learning rate, batch size, and even neural network architecture much quicker than loading the entire 500 word dataset. Small adjustments to learning rate was made, usually in increments of 0.005, to find what is optimal for a certain neural network. For larger and deeper neural networks, it seems that larger learning rates were needed. After testing a variety of batch sizes, it seems that the default batch size of 32 was the best in terms of the number of epochs to fully train and time per epoch.

In the failed attempts section, I discuss several obstacles of implementing deeper and more complex neural networks to feed into the LSTM. CNNs such as DenseNet201 and ResNet50 all suffered in accuracy when in theory a more complex neural network is

supposed to give me better results. I attempted a customized implementation of Google's InceptionV3. This neural network ended up being large in size while having lower accuracy than a much simpler implementation. Here, a theoretically superior CNN than my custom VGG16 model performed worse when processing video data. I'm unsure of why that is the case. The only reason that I have is that these neural networks are simply not compatible with my data type as they are meant to classify images, not a series of images (video).

### 9.3 Future Work

There are an almost infinite number of neural network architectures that can be tried. While there are general rules and guidelines to follow for a good neural network, there are no "best" combination of neural network layers to use. Moreover, a CNN does not have to be linear in structure, as demonstrated by Google's Inception modules. Although my attempts have not worked in implementing a parallel type CNN such as Google's Inception, I believe that one can make this type of neural network function well for video data. Also, a more popular approach to LSTMs recently have been to include attention modules.

Attention modules are usually placed after LSTM modules. Its purpose is to decide which parts of the input data are more important than others. For example, certain words in a sentence are more important than others in deciphering its meaning. The attention module's job is to decide what those words are and assign more weight to the key words and assign less weight to less important words. Attention modules are much more recent in development, therefore there is no consensus on a "best" type of attention module. In research papers, people often write their own customizations of the attention module.

A paper written in 2017, *Attention is All You Need*, addresses the main concern with LSTMs: speed of training. Since LSTMs read data sequentially, they take longer to train. Transformers instead read all data at once. Attention modules are used to assign weights to all input data. For example, for video-based ASR, weight matrices are assigned to every frame, weighing the frame data to each other with the weight matrix. The transformer not only outperforms LSTMs, but also requires less computation to train. A transformer system adapted to video-based ASR will be a great option to explore in the future.

Ideally, video-based ASR systems will be integrated with audio-based ASR systems. Video-based ASR is resistant to audio noise and audio-based ASR is resistant to visual noise. Combining both types of ASR systems will result in an overall more robust system. There are two ways to integrate these systems. There is an approach that combines both video and audio data using LSTMs with attention modules. The video and audio data are processed separately via LSTMs (audio data) and LSTM and CNNs (video data). These outputs are combined, passed through an attention module, then through an LSTM module into a multi-layer perceptron. The attention module weights the outputs of the video and audio-based ASR separately. The audio and video data are combined to decipher a word.

Another approach, which is simpler, would be to process audio and video data separately. A decision will be made based on how much we trust the different systems to accurately decipher the data. Since audio-based ASRs are more researched and have higher accuracy, more weight can be given to audio-based ASRs compared to video-based ASRs.

when making a decision based on the input data. In this approach, audio and video data are kept separately and are only used when deciding the word.

Extending beyond single word recognition would be to perform ASR on phrases or even sentences. Knowing the context of a word in a sentence can potentially reduce the misidentification chance of similar words.

## BIBLIOGRAPHY

1. “Dlib Face Recognition.” *Dlib C++ Library*, [dlib.net/face\\_recognition.py.html](http://dlib.net/face_recognition.py.html). Accessed 4 Dec. 2020.
2. Olah, Christopher. “Understanding LSTM Networks.” *Understanding LSTM Networks -- Colah's Blog*, 27 Aug. 2015, [colah.github.io/posts/2015-08-Understanding-LSTMs/](http://colah.github.io/posts/2015-08-Understanding-LSTMs/). Accessed 4 Dec. 2020.
3. Taylor, Ashley. “Introduction to Digital Images.” *Image-1 Introduction to Digital Images*, [web.stanford.edu/class/cs101/image-1-introduction.html](http://web.stanford.edu/class/cs101/image-1-introduction.html). Accessed 4 Dec. 2020.
4. Eckhardt, Karsten. “Choosing the Right Hyperparameters for a Simple LSTM Using Keras.” *Medium*, Towards Data Science, 29 Nov. 2018, [towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046](https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046). Accessed 4 Dec. 2020.
5. Bilogur, Aleksey. “Tuning Your Learning Rate.” *Kaggle*, Kaggle, 30 July 2018, [www.kaggle.com/residentmario/tuning-your-learning-rate](http://www.kaggle.com/residentmario/tuning-your-learning-rate). Accessed 4 Dec. 2020.
6. Brownlee, Jason. “Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.” *Machine Learning Mastery*, 20 Aug. 2020, [machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/](http://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/). Accessed 4 Dec. 2020.
7. Berthon, J. C. “Containers, Volumes and File Permissions.” *Ice and Fire – by J-C Berthon*, 25 July 2018, [www.berthon.eu/2018/containers-volumes-and-file-permissions/](http://www.berthon.eu/2018/containers-volumes-and-file-permissions/). Accessed 4 Dec. 2020.

8. Dille, Nicholas. "Handling File Permissions When Writing to Volumes from #Docker Containers." *Nicholas Dille*, 16 July 2018, [dille.name/blog/2018/07/16/handling-file-permissions-when-writing-to-volumes-from-docker-containers/](https://dille.name/blog/2018/07/16/handling-file-permissions-when-writing-to-volumes-from-docker-containers/). Accessed 4 Dec. 2020.
9. Erin, Christopher. "Today I Learned." *Creating a Bind Mount with `Docker Volume`*, 2019, [til.hashrocket.com/posts/enm4qz5zxw-creating-a-bind-mount-with-docker-volume](https://til.hashrocket.com/posts/enm4qz5zxw-creating-a-bind-mount-with-docker-volume). Accessed 4 Dec. 2020.
10. "Docker Pull." *Docker Documentation*, 26 Nov. 2020, [docs.docker.com/engine/reference/commandline/pull/](https://docs.docker.com/engine/reference/commandline/pull/). Accessed 4 Dec. 2020.
11. "Convolutional Neural Networks (CNN): Step 4 - Full Connection." *SuperDataScience*, 17 Aug. 2018, [www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-4-full-connection](https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-4-full-connection). Accessed 4 Dec. 2020.
12. Bhande, Anup. "What Is Underfitting and Overfitting in Machine Learning and How to Deal with It." *Medium*, GreyAtom, 18 Mar. 2018, [medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76](https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76). Accessed 4 Dec. 2020.
13. Kilgarriff, Emmett, et al. "NVIDIA Turing Architecture In-Depth." *NVIDIA Developer Blog*, 25 Aug. 2020, [developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/](https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/). Accessed 4 Dec. 2020.
14. Rosebrock, Adrian. "Detect Eyes, Nose, Lips, and Jaw with Dlib, OpenCV, and Python." *PyImageSearch*, 18 Apr. 2020, [www.pyimagesearch.com/2017/04/10/detect-eyes-nose-lips-jaw-dlib-opencv-python/](https://www.pyimagesearch.com/2017/04/10/detect-eyes-nose-lips-jaw-dlib-opencv-python/). Accessed 4 Dec. 2020.

15. Aggarwal, Raghav. "Bi-LSTM." *Medium*, Medium, 4 July 2019, [medium.com/@raghavaggarwal0089/bi-lstm-bc3d68da8bd0](https://medium.com/@raghavaggarwal0089/bi-lstm-bc3d68da8bd0). Accessed 4 Dec. 2020.
16. Czyzewski, A., Kostek, B., Bratoszewski, P. *et al.* An audio-visual corpus for multimodal automatic speech recognition. *J Intell Inf Syst* **49**, 167–192 (2017). <https://doi.org/10.1007/s10844-016-0438-z>
17. Chung, Joon Son & Zisserman, Andrew. (2017). Lip Reading in the Wild. 87-103. 10.1007/978-3-319-54184-6\_6.
18. Kingma, Diederik & Ba, Jimmy. (2014). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations.
19. Kim, Joshua & Liu, Chunfeng & Calvo, Rafael & McCabe, Kathryn & Taylor, Silas & Schuller, Björn & Wu, Kaihang. (2019). A Comparison of Online Automatic Speech Recognition Systems and the Nonverbal Responses to Unintelligible Speech.
20. Simonyan, Karen & Zisserman, Andrew. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 1409.1556.
21. Lawrence, Steve & Giles, C. & Tsoi, Ah. (2001). What Size Neural Network Gives Optimal Generalization? Convergence Properties of Backpropagation.
22. Srivastava, Nitish & Hinton, Geoffrey & Krizhevsky, Alex & Sutskever, Ilya & Salakhutdinov, Ruslan. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*. 15. 1929-1958.
23. Wand, Michael & Koutnik, Jan & Schmidhuber, Jurgen. (2016). Lipreading with long short-term memory. 6115-6119. 10.1109/ICASSP.2016.7472852.
24. McCandlish, Sam & Kaplan, Jared & Amodei, Dario & Team, OpenAI. (2018). An Empirical Model of Large-Batch Training.

25. Rochford, Matthew. (2019). Visual Speech Recognition Using a 3D Convolutional Neural Network.
26. Amit Garg, Jonathan Noyola and Sameep Bagadia. (2016). "Lip reading using CNN and LSTM." (2016).
27. Fohr, D., Odile Mella and I. Illina. "New Paradigm in Speech Recognition: Deep Neural Networks." *ICIS 2017* (2017).
28. Altieri, Nicholas & Pisoni, David & Townsend, James. (2011). Some normative data on lip-reading skills (L). *The Journal of the Acoustical Society of America*. 130. 1-4. 10.1121/1.3593376.
29. Lee, Bowon & Hasegawa-Johnson, Mark & Goudeseune, Camille & Kamdar, Suketu & Borys, Sarah & Liu, Ming & Huang, Thomas. (2004). AVICAR: Audio-visual speech corpus in a car environment. *Proc. Annu. Conf. Int. Speech Commun. Assoc. (INTERSPEECH)*.
30. Team, Keras. "Keras Documentation: Model Training APIs." *Keras*, [keras.io/api/models/model\\_training\\_apis/](https://keras.io/api/models/model_training_apis/). Accessed 4 Dec. 2020.
31. Zhang, Jane. "SIFT: Scale Invariant Feature Transformation." 26 Nov. 2020.
32. Zhang, Jane. "Support Vector Machines ." 11 Nov. 2020.
33. "Tf.keras.optimizers.Adam : TensorFlow Core v2.3.0." *TensorFlow*, 23 Oct. 2020, [www.tensorflow.org/api\\_docs/python/tf/keras/optimizers/Adam](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam). Accessed 4 Dec. 2020.
34. Team, Keras. "Keras Documentation: TimeDistributed Layer." *Keras*, [keras.io/api/layers/recurrent\\_layers/time\\_distributed/](https://keras.io/api/layers/recurrent_layers/time_distributed/). Accessed 4 Dec. 2020.



35. C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9, doi: 10.1109/CVPR.2015.7298594.

## APPENDIX

### A. LIP DETECTION MODULE

```
1. #####
2.
3. # Lip Detector Module
4. # Alvin Lin
5. # This file defines extracts lip landmarks, then crops it using OpenCV
6. # Includes the option to take the difference between lip frames
7. # Returns lip frames. Should be grayscale and cropped!
8.
9. #####
10.
11. # Import the necessary packages
12. from imutils import face_utils
13. import numpy as np
14. import cv2
15. import dlib
16. import time
17.
18. #####
19.
20. # Define path to dlib predictor
21. predictor_path = 'shape_predictor_68_face_landmarks.dat'
22.
23. #####
24.
25. fgbg = cv2.createBackgroundSubtractorMOG2(history=10,varThreshold=2,detectShadows=False)
26.
27. # Lip detection function
28. def lip_detector(video_path):
29.
30.     # Initialize dlib's face detector (HOG-based) and create facial landmark predictor
31.     detector = dlib.get_frontal_face_detector()
32.     predictor = dlib.shape_predictor(predictor_path)
33.
34.     # Read video in as mp4 file
35.     video = cv2.VideoCapture(video_path)
36.
37.     # Check if video opened
38.     if (video.isOpened()==False):
39.         print('Error opening video file: ' + video_path)
40.         return # Return from function if video does not open
41.
42.     lip_frames = [] # Initialize variable to store lip frames
43.     count = 0 # Initialize counter to track frames
44.
45.     # Read video frame by frame
46.     while(video.isOpened()):
47.
48.         ret, frame = video.read() # Capture frame
49.         if ret == True: # if frame exists
50.
51.             count = count + 1 # Increment counter
52.
53.             # Convert frame to grayscale and resize to a standard size
54.             frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
55.             frame = cv2.resize(frame,(224,224))
```

```

56.
57.     # detect face in the image
58.     faces = detector(frame, 1)
59.
60.     if len(faces) > 1: # If more than one face detected print error and return
61.         #print('Error: Multiple faces detected in video')
62.         return 2 # Return error code 2
63.     elif len(faces) == 0: # If no face detected print error and return
64.         #print('Error: No face detected in video')
65.         return 0 # Return error code 0
66.
67.     else: # If one face detected perform lip cropping
68.         for face in faces:
69.
70.             # Determine facial landmarks for the face region and convert to a NumPy array
71.             shape = predictor(frame, face)
72.             shape = face_utils.shape_to_np(shape)
73.
74.             # Extract the lip region as a separate image
75.             (x, y, w, h) = cv2.boundingRect(np.array([shape[48:68]]))
76.             margin = 10 # Extra pixels to include around lips
77.             lips = frame[y-margin:y + h + margin, x-margin:x + w + margin]
78.             lips = cv2.resize(lips,(90,90))
79.
80.             # Create a stack of extracted frames
81.             if len(lip_frames) == 0:
82.                 lip_frames = lips
83.             else:
84.                 lip_frames = np.dstack(((lip_frames),(lips)))
85.         else: # If no frame left break from loop
86.             break
87.
88.     # Release video object
89.     video.release()
90.
91.     # Close any open windows
92.     cv2.destroyAllWindows()
93.
94.     # Reshape array for CNN layer compatibility
95.     lip_frames = np.moveaxis(lip_frames,-1,0)
96.
97.
98.     # This segments subtracts the lip frames and pads with an empty frame
99.     # sub_lips = np.zeros([29,60,100])
100.
101.     # for i in range (1,29):
102.     #     temp = np.subtract(lip_frames[i,:,:],lip_frames[i-1,:,:])
103.     #     sub_lips[i,:,:] = temp
104.
105.     # sub_lips = sub_lips[1:28,:,:]
106.
107.
108.     return lip_frames
109.
110. #####
111.
112. # REFERENCES:
113. # Code for reading and editing video files is adapted from 'Learn OpenCV'
114. # https://www.learnopencv.com/read-write-and-display-a-video-using-opencv-cpp-python/
115. # Code for lip detection is based off Dlib library and an implementation from PyImageSearch
116. # https://www.pyimagesearch.com/2017/04/10/detect-eyes-nose-lips-jaw-dlib-opencv-python/

```

## B. LIP FRAMES TO NUMPY

```
1. #####
2.
3. # Video Preprocessing Framework
4. # Alvin Lin
5. # This is the main function that implements the lip detector on each video
6. # This file creates the labels to label each file
7. # file and stores the output numpy arrays. Also includes all helper functions.
8. # Utilize multiprocessing for speed increases
9.
10. #####
11.
12. # Import modules
13. from lip_detector import lip_detector
14. from multiprocessing import Pool
15. import os
16. import numpy as np
17. import time
18.
19. #####
20.
21. # Define Paths
22. input_folder = '../Thesis/images/lipread_mp4/' # Path to folders of each class
23. output_folder = '../Thesis/images/lipread_npy/' # Included for help when storing output numpy arrays
24.
25. #####
26.
27. # This function forms a subset of the LRW dataset consisting of the first 100 classes.
28. # The output is a list of each class name to then be used in deciding which classes
29. # to process.
30.
31. def data_subset():
32.
33.     # Initialize counter f
34.     i = 0
35.
36.     # Initialize array for storing class names
37.     subset_strings = []
38.
39.     # Loop through each class directory e.g. ABOUT, ABSOLUTELY, ACCESS
40.     for directory in os.listdir(input_folder):
41.
42.         # Only include the first 100 classes
43.         if i < 500:
44.             subset_strings.append(directory)
45.
46.         # Increment counter
47.         i = i + 1
48.
49.     # Return list of 100 class strings
50.     return subset_strings
51.
52. #####
53.
54. # This function applies the lip detector function to all the files in the input folder
55. # and saves the output numpy files to the output folders.
56. # This function is modified to include labeling of classes
57.
58. def process_videos(directory):
59.
```

```

60. # Form list of subset strings
61. mult_faces = 0
62. no_faces = 0
63. face = 0
64. # Loops over each word directory Ex. ABOUT, ABSOLUTELY, ACCESS, etc.
65. # If class is in the 100-class subset
66. if directory in subset:
67.
68.     # Loops over each sub directory, test train val
69.     for sub_directory in os.listdir(input_folder+directory):
70.
71.         # Loops over each file in final directory
72.         for file in os.listdir(input_folder+directory+'/'+sub_directory):
73.
74.             # Only use mp4 files
75.             if file.endswith('.mp4'):
76.
77.                 # Generate label for output data file
78.                 label = os.path.splitext(file)[0] # Grab file base name
79.                 label = label + '.npy'
80.
81.                 # Check if file has already been processed and update counter
82.                 if os.path.exists(output_folder+sub_directory+'/'+label):
83.                     face = face + 1
84.
85.                 # Otherwise process file
86.                 else:
87.                     # Form full input path using directory and individual file
88.                     path = input_folder+directory+'/'+sub_directory+'/'+
89.
90.                     # Perform lip detection and return data array
91.                     data_array = lip_detector(path+file)
92.
93.                     # If face detection error, keep track for evaluation
94.                     if isinstance(data_array,int):
95.
96.                         if data_array == 2:
97.                             mult_faces = mult_faces + 1
98.                         elif data_array == 0:
99.                             no_faces = no_faces + 1
100.
101.                     # If face was detected successfully
102.                     else:
103.
104.                         # Update counter
105.                         face = face + 1
106.
107.                     # This part adds the extra dimensions
108.                     # Multithreads the build_4D_arrays_and_labels partially, done for speed
109.                     # Add 4th dimension to numpy array for CNN layer compatibility
110.                     data_array = np.expand_dims(data_array,axis=-1)
111.
112.                     # Normalize training data to 0~1 range
113.                     data_array = data_array / 255.0
114.
115.                     # Convert to shortest float for memory allocation demands
116.                     data_array = np.float16(data_array)
117.
118.                     # Define path to output folder
119.                     path = output_folder+sub_directory+'/'+
120.

```

```

121.             # Save output numpy array to output folder
122.             np.save(path+label,data_array)
123.
124. #####
125.
126. # This function creates labels for each class (e.g. AFRICA, AMERICA) and saves them as .npy files
127. # This is written to the output file called "labels"
128. # Before feeding into the NN, this file is combined with videos processed
129.
130. def label_maker():
131.
132.     print('Creating labels...')
133.     # Make string of classes to be used in dataset
134.     subset = data_subset()
135.
136.     # Initialize counter for label making
137.     i = 0
138.
139.     # Loops over each word directory Ex. ABOUT, ABSOLUTELY, ACCESS, etc.
140.     for directory in subset:
141.
142.         # Hard code label of size 10 for each word in dataset subset
143.         # Change to 500 if using entire LRW dataset
144.         label = np.zeros((500), dtype=int)
145.
146.         # Assign value of 1 at proper location
147.         label[i] = 1
148.
149.         # Create name for label file
150.         name = directory+'_label.npy'
151.
152.         # Create path to labels folder
153.         path = output_folder+'labels/'
154.
155.         # Save numpy file to labels folder
156.         np.save(path+name,label)
157.
158.         # Increment counter
159.         i = i + 1
160.
161. #####
162.
163. # This function builds a 4D data array to use for model training. Key is which dataset to be
164. # assembled: test, train, or val. The output is one array of the entire dataset with ordered
165. # labels.
166.
167. def build_4D_arrays_and_labels(key):
168.
169.     # Generate string of classes for data subset
170.     subset = data_subset()
171.     i = 0
172.
173.     # Initialize arrays for data and labels
174.     dataset = []
175.     labels = []
176.
177.     # Define path to proper dataset folder: test,train,val
178.     path = output_folder+key+'/'
179.
180.     # For each numpy array file from lip detector output
181.     for file in os.listdir(path):

```

```

182.     name = file.split('_')
183.
184.     # Only add files in data subset
185.     if name[0] in subset:
186.
187.         # Load numpy array
188.         data = np.load(path+file)
189.
190.         # Grab correct frames if full video was processed earlier
191.         #if data.shape[0] != 11:
192.         #    data = data[9:20,:,:]
193.
194.         # Grab file base word
195.         #word = file.split('_')[0]
196.
197.         # Make label file string
198.         label_file = name[0]+'_label.npy'
199.
200.         # Load label array
201.         label = np.load(output_folder+'labels/'+label_file)
202.
203.         # Append data to list
204.         dataset.append(data)
205.         labels.append(label)
206.
207.     # Convert lists to numpy arrays
208.     dataset = np.stack(dataset)
209.     labels = np.stack(labels)
210.
211.     # Print output stats
212.     print(key+' dataset size and label size')
213.     print(dataset.shape)
214.     print(labels.shape)
215.
216.     # Save numpy arrays to save processing time when no new data is added
217.     np.save(key+'_datasq.npy',dataset)
218.     np.save(key+'_labelssq.npy',labels)
219.     print('done')
220.
221. #####
222.
223. if __name__ == "__main__":
224.
225.     t0 = time.time()
226.     # Initialize variables for per class evaluation
227.     mult_faces = 0
228.     no_faces = 0
229.     face = 0
230.     subset = data_subset()
231.
232.     # file labeling
233.     label_maker()
234.
235.     #maximum of 100 threads to use
236.     pool = Pool(processes=100)
237.
238.     #process multiple subsets at a time (e.g. 'AFRICA' and 'AMERICA' at the same time)
239.     for i in range(len(subset)):
240.         print(subset[i])
241.         pool.apply_async(process_videos, args = (subset[i], ))
242.

```

```

243. pool.close()
244. pool.join()
245.
246. # Build validation dataset
247. print('Building validation dataset')
248. build_4D_arrays_and_labels('val')
249.
250. # Build training dataset
251. print('Building training dataset')
252. build_4D_arrays_and_labels('train')
253.
254. # Build testing dataset
255. print('Building testing dataset')
256. build_4D_arrays_and_labels('test')
257.
258. t1 = time.time()
259. total = t1-t0
260. print('Processing time: '+str(total))

```

### C. BI-DIRECTIONAL LSTM-CNN

```

1. # bi-LSTM-CNN
2. # Alvin Lin
3. # This file creates tensor slices using the dataset
4. # Defines an bidirectional LSTM
5. # Trains, validates, and tests data
6. # Also outputs text files for confusion matrix
7.
8. #####
9.
10. # Import tensorflow and ignore some errors
11. import os
12. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
13. import tensorflow as tf
14. tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
15.
16. # Import other modules
17. from keras.preprocessing.image import ImageDataGenerator
18. from keras.models import Sequential, load_model
19. from keras.layers import TimeDistributed, Attention
20. from keras.layers.core import Dense, Dropout, Activation, Flatten
21. from keras.layers.convolutional import Convolution2D, MaxPooling2D
22. from keras.layers import Dense, Dropout, LSTM, BatchNormalization, Bidirectional
23. from keras import optimizers
24.
25. # Plot data
26. from matplotlib import pyplot as plt
27.
28. # Confusion Matrix
29. from sklearn.metrics import classification_report, confusion_matrix
30. import itertools
31.
32. # Handy things to have
33. import numpy as np
34. import time
35.
36. #####
37.
38. # This function forms a subset of the LRW dataset consisting of the first 100 classes.
39. # The output is a list of each class name to then be used in deciding which classes

```



```

40. # to process.
41.
42. def data_subset():
43.
44.     input_folder = '../Thesis/images/lipread_mp4/' # Path to folders of each class
45.     # Initialize counter f
46.     i = 0
47.
48.     f = open("datasubset.txt","w")
49.
50.     # Loop through each class directory e.g. ABOUT, ABSOLUTELY, ACCESS
51.     for directory in os.listdir(input_folder):
52.
53.         # Only include the first 100 classes
54.         if i < 500:
55.             f.write(str(directory) + '\n')
56.
57.         # Increment counter
58.         i = i + 1
59.
60.     # Return list of 100 class strings
61.     f.close()
62.
63. #####
64.
65. # This function is used to load a dataset that was already assembled using the
66. # 'build_4D_arrays_and_labels' function. Used to save time when no changes to dataset
67. # have been made.
68.
69. def load_dataset(key):
70.
71.     BATCH_SIZE = 32
72.     #SHUFFLE_BUFFER_SIZE = 10000
73.
74.     dataset = np.load(key+'_datasub.npy')
75.     labels = np.load(key+'_labelssub.npy')
76.
77.     # Convert away from one-hot encoding
78.     # Uncomment to train
79.     notonehot = tf.argmax(labels,axis=1)
80.
81.     assert dataset.shape[0] == labels.shape[0]
82.
83.     # Use tf.data.Dataset to create tensor slices
84.     dataset = tf.data.Dataset.from_tensor_slices((dataset,labels)).batch(BATCH_SIZE)
85.     #shuffled = combined.shuffle(SHUFFLE_BUFFER_SIZE)
86.
87.
88.     return dataset
89.
90. #####
91.
92. # Function to initialize Bidirectional LSTM-CNN (Architecture #6)
93. # Time Distributed to process video per frame
94. # Bidirectional LSTM
95. # Returns model summary, including number of parameters
96.
97.
98. def model_init():
99.
100.    print('Initializing CNN Model using Keras')

```

```

101.
102. model = Sequential()
103.
104. model.add(TimeDistributed(Convolution2D(input_shape=(29,60,100,1),filters=16,kernel_size=3,strides=1,
padding='same',activation='relu'))))
105. model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
106.
107. model.add(TimeDistributed(Convolution2D(filters=32,kernel_size=3,strides=1,padding='same',activation='r
elu'))))
108. model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
109.
110. model.add(TimeDistributed(Convolution2D(filters=64,kernel_size=3,strides=1,padding='same',activation='r
elu'))))
111. model.add(TimeDistributed(Convolution2D(filters=64,kernel_size=3,strides=1,padding='same',activation='r
elu'))))
112. model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
113.
114. model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
'relu'))))
115. model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
'relu'))))
116. model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
117.
118. model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
'relu'))))
119. model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
'relu'))))
120. model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
121.
122. model.add(TimeDistributed(Flatten()))
123.
124. model.add(Bidirectional(LSTM(1024, return_sequences=True, dropout=0.2)))
125. model.add(Bidirectional(LSTM(1024, dropout=0.2)))
126.
127. model.build((32,29,60,100,1))
128. print(model.summary())
129.
130. # model.add(Dense(2048, activation='relu'))
131. # model.add(Dense(1024, activation='relu'))
132. model.add(Dense(500, activation='softmax'))
133.
134. return model
135.
136. #####
137. # Main function for training the keras model. This function creates training labels,
138. # calls array building functions, initializes model, and then trains it.
139.
140. def train_model():
141.
142.     # Build training dataset
143.     print('Building training dataset')
144.
145.     # Load training dataset
146.     train = load_dataset('train')
147.     #train_data,train_labels = load_dataset('train')
148.
149.     # Build validation dataset
150.     print('Building validation dataset')
151.
152.     #switch to load_dataset for quicker reruns!
153.     val = load_dataset('val')

```

```

154. #val_data,val_labels = load_dataset('val')
155.
156. print('Finished loading dataset')
157.
158. # Use if one GPU is taken
159. with tf.device("/GPU:0"):
160.     # Initialize Keras model
161.     model = model_init()
162.
163.     # Load Keras model (use if you want to continue training on an already made model)
164.     #model = load_model("model.h5")
165.
166.     # Adam optimizer
167.     sgd = optimizers.Adam(lr=.005, epsilon=0.1)
168.
169.     # Compile using categorical cross entropy loss function
170.     history = model.compile(optimizer=sgd,loss='categorical_crossentropy',metrics=['accuracy'])
171.
172.     # Train model using keras fit
173.     print('Training Keras Model')
174.
175.     # Train model
176.     history = model.fit(train,epochs=100,validation_data=(val),verbose=1)
177.
178.     model.save('model_biLSTMsub.h5')
179.
180.     print('Done training')
181.
182.     # Returns plot of accuracy and validation over epochs
183.     # Plot Model
184.     print(history.history.keys())
185.     # Accuracy plot
186.     plt.figure(0)
187.     plt.plot(history.history['accuracy'])
188.     plt.plot(history.history['val_accuracy'])
189.     plt.title('model accuracy')
190.     plt.ylabel('accuracy')
191.     plt.xlabel('epoch')
192.     plt.legend(['train', 'val'], loc='upper left')
193.     # Save plot
194.     plt.savefig('biLSTMsub_accuracy.png')
195.     # Loss plot
196.     plt.figure(1)
197.     plt.plot(history.history['loss'])
198.     plt.plot(history.history['val_loss'])
199.     plt.title('model loss')
200.     plt.ylabel('loss')
201.     plt.xlabel('epoch')
202.     plt.legend(['train', 'val'], loc='upper left')
203.     # Save plot
204.     plt.savefig('biLSTMsub_loss.png')
205.
206. #####
207. #This function is for testing of trained model
208.
209. def test_model():
210.
211.     # Build testing dataset
212.     print('Building testing dataset')
213.
214.     # returns test data and non-onehot labeled data for confusion matrix

```

```

215. test, labelnotonehot = load_dataset('test')
216.
217. # Load model
218. model = load_model("model_biLSTM.h5")
219.
220. # Evaluate model with test dataset
221. [loss,accuracy] = model.evaluate(test)
222. accuracy = accuracy*100
223.
224. # Predict data for confusion matrix
225. print('Predicting...')
226. predictions = model.predict_classes(test)
227. confuse = confusion_matrix(labelnotonehot,predictions)
228. np.savetxt("confusion_LSTMsub.txt",np.rint(confuse))
229.
230. # Print test results
231. print("Testing loss: "+str(loss))
232. print("Testing accuracy: "+str(accuracy))
233.
234.
235. #####
236.
237. if __name__ == '__main__':
238.
239.     t0 = time.time()
240.     train_model()
241.     #test_model()
242.     t1 = time.time()
243.     total = t1-t0
244.     print('Processing time: '+str(total))

```

#### D. LSTM-CNN

```

1. # Function to initialize LSTM-CNN (Architecture #5)
2. # Only input to function is the shape of the input numpy array used for training.
3. # Returns the tensorflow model object and prints the model summary.
4.
5.
6. def model_init():
7.
8.     print('Initializing CNN Model using Keras')
9.
10.    model = Sequential()
11.
12.    model.add(TimeDistributed(Convolution2D(input_shape=(29,60,100,1),filters=16,kernel_size=3,strides=1,
padding='same',activation='relu'))))
13.    model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
14.
15.    model.add(TimeDistributed(Convolution2D(filters=32,kernel_size=3,strides=1,padding='same',activation='r
elu'))))
16.    model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
17.
18.    model.add(TimeDistributed(Convolution2D(filters=64,kernel_size=3,strides=1,padding='same',activation='r
elu'))))
19.    model.add(TimeDistributed(Convolution2D(filters=64,kernel_size=3,strides=1,padding='same',activation='r
elu'))))
20.    model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
21.
22.    model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
'relu'))))

```

```

23. model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
    'relu'))))
24. model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
25.
26. model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
    'relu'))))
27. model.add(TimeDistributed(Convolution2D(filters=128,kernel_size=3,strides=1,padding='same',activation=
    'relu'))))
28. model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
29.
30. model.add(TimeDistributed(Flatten()))
31.
32. model.add(LSTM(1024, return_sequences=True, dropout=0.2))
33. model.add(LSTM(1024, dropout=0.2))
34.
35. # model.add(Dense(2048, activation='relu'))
36. # model.add(Dense(1024, activation='relu'))
37. model.add(Dense(500, activation='softmax'))
38.
39. model.build((32,29,60,100,1))
40.
41. print(model.summary())
42.
43. return model

```

## E. Multiple Towers

```

1. # Function to initialize Multiple Towers (Architecture #4)
2. # Only input to function is the shape of the input numpy array used for training.
3. # Returns the tensorflow model object and prints the model summary.
4.
5.
6. def model_init():
7.
8.     print('Initializing CNN Model using Keras')
9.
10.    model = Sequential()
11.
12.    model.add(TimeDistributed(Convolution2D(input_shape=(29,60,100,1),filters=16,kernel_size=3,strides=1,pad
        ding='same',activation='relu'))))
13.    model.add(TimeDistributed(MaxPooling2D(pool_size=(2,2),strides=(2,2))))
14.
15.    model.add(Convolution3D(filters=32,kernel_size=3,strides=1,padding='same',activation='relu'))
16.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
17.
18.    model.add(Convolution3D(filters=64,kernel_size=3,strides=1,padding='same',activation='relu'))
19.    model.add(Convolution3D(filters=64,kernel_size=3,strides=1,padding='same',activation='relu'))
20.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
21.
22.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
23.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
24.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
25.
26.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
27.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
28.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
29.
30.    model.add(Flatten())
31.    model.add(Dense(2048, activation='relu'))
32.    model.add(Dense(1024, activation='relu'))

```

```

33. model.add(Dense(500, activation='softmax'))
34.
35. model.build((32,29,60,100,1))
36. print(model.summary())
37.
38. return model

```

## F. VGG-16 Based Architecture

```

1. # Function to initialize VGG-16 based architecture (Architecture #3)
2. # Only input to function is the shape of the input numpy array used for training.
3. # Returns the tensorflow model object and prints the model summary.
4.
5.
6. def model_init():
7.
8.     print('Initializing CNN Model using Keras')
9.
10.    model = Sequential()
11.
12.    model.add(Convolution3D(input_shape=(29,60,100,1),filters=16,kernel_size=3,strides=1,padding='same',activation='relu'))
13.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
14.
15.    model.add(Convolution3D(filters=32,kernel_size=3,strides=1,padding='same',activation='relu'))
16.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
17.
18.    model.add(Convolution3D(filters=64,kernel_size=3,strides=1,padding='same',activation='relu'))
19.    model.add(Convolution3D(filters=64,kernel_size=3,strides=1,padding='same',activation='relu'))
20.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
21.
22.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
23.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
24.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
25.
26.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
27.    model.add(Convolution3D(filters=128,kernel_size=3,strides=1,padding='same',activation='relu'))
28.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
29.
30.    model.add(Flatten())
31.    model.add(Dense(2048, activation='relu'))
32.    model.add(Dense(1024, activation='relu'))
33.    model.add(Dense(500, activation='softmax'))
34.
35.    print(model.summary())
36.
37.    return model

```

## G. Extended 3D-CNN

```

1. # Function to initialize extended 3D CNN (Architecture #2)
2. # Only input to function is the shape of the input numpy array used for training.
3. # Returns the tensorflow model object and prints the model summary.
4.
5.
6. def model_init():
7.
8.     print('Initializing CNN Model using Keras')
9.

```

```

10. model = Sequential()
11.
12. model.add(Convolution3D(input_shape=(29,60,100,1),filters=16,kernel_size=2,strides=1,activation='relu'))
13. model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
14.
15. model.add(Convolution3D(filters=32,kernel_size=2,strides=1,activation='relu'))
16. model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
17.
18. model.add(Convolution3D(filters=64,kernel_size=2,strides=1,activation='relu'))
19. model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
20.
21. model.add(Convolution3D(filters=128,kernel_size=2,strides=1,activation='relu'))
22. model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
23.
24. model.add(Convolution3D(filters=256,kernel_size=2,strides=1,activation='relu'))
25.
26. model.add(Flatten())
27. model.add(Dense(1024, activation='relu'))
28. model.add(Dense(1024, activation='relu'))
29. model.add(Dense(1024, activation='relu'))
30. model.add(Dropout(0.5))
31. model.add(Dense(512, activation='relu'))
32. model.add(Dropout(0.5))
33. model.add(Dense(500, activation='softmax'))
34.
35. print(model.summary())
36.
37. return model

```

## H. 3D-CNN

```

1. # Function to initialize 3D CNN (Architecture #1)
2. # Only input to function is the shape of the input numpy array used for training.
3. # Returns the tensorflow model object and prints the model summary.
4.
5.
6. def model_init():
7.
8.     print('Initializing CNN Model using Keras')
9.
10.    model = Sequential()
11.
12.    model.add(Convolution3D(input_shape=(29,60,100,1),filters=16,kernel_size=2,strides=1,activation='relu'))
13.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
14.
15.    model.add(Convolution3D(filters=32,kernel_size=2,strides=1,activation='relu'))
16.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
17.
18.    model.add(Convolution3D(filters=64,kernel_size=2,strides=1,activation='relu'))
19.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
20.
21.    model.add(Convolution3D(filters=128,kernel_size=2,strides=1,activation='relu'))
22.    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
23.
24.    model.add(Convolution3D(filters=256,kernel_size=2,strides=1,activation='relu'))
25.
26.    model.add(Flatten())
27.    model.add(Dense(1024, activation='relu'))
28.    model.add(Dropout(0.5))
29.    model.add(Dense(512, activation='relu'))

```

```
30. model.add(Dropout(0.5))
31. model.add(Dense(500, activation='softmax'))
32.
33. print(model.summary())
34.
35. return model
```