VISUAL SPEECH RECOGNITION USING A 3D CONVOLUTIONAL NEURAL NETWORK

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

Matt Rochford

November 2019

COMMITTEE MEMBERSHIP

TITLE: Visual Speech Recognition Using a 3D Convolutional Neural Network

AUTHOR: Matt Rochford

DATE SUBMITTED: November 2019

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

COMMITTEE MEMBER: Dennis Derickson, Ph.D.
Professor of Electrical Engineering, Department Chair

COMMITTEE MEMBER: Clay McKell, Ph.D.
Lecturer of Electrical Engineering

ABSTRACT

Visual Speech Recognition Using a 3D Convolutional Neural Network

Matt Rochford

Main stream automatic speech recognition (ASR) makes use of audio data to identify spoken words, however visual speech recognition (VSR) has recently been of increased interest to researchers. VSR is used when audio data is corrupted or missing entirely and also to further enhance the accuracy of audio-based ASR systems. In this research, we present both a framework for building 3D feature cubes of lip data from videos and a 3D convolutional neural network (CNN) architecture for performing classification on a dataset of 100 spoken words, recorded in an uncontrolled environment. Our 3D-CNN architecture achieves a testing accuracy of 64%, comparable with recent works, but using an input data size that is up to 75% smaller. Overall, our research shows that 3D-CNNs can be successful in finding spatial-temporal features using unsupervised feature extraction and are a suitable choice for VSR-based systems.

ACKNOWLEDGMENTS

Thanks to:

- My parents for giving me all the opportunity to succeed in life.
- My brothers for being the absolute boys.
- Megan for keeping me motivated.
- My awesome roommates throughout college that made it memorable.
- Jane Zhang for being an amazing mentor and advisor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Problem Statement

Automatic speech recognition (ASR) is the ability to translate spoken words into text using computers. ASR incorporates knowledge and algorithms from linguistics, computer science, and electrical engineering to identify and process human voices [32]. Google Speech's ASR engine is the most accurate on the market today and has an average error rate of 16%, and as low as 2% for some datasets [28]. Main stream ASR, such as Google Speech, uses data from audio signals in combination with machine learning algorithms. However, when audio data is corrupted, such as in noisy environments, performance suffers greatly.

Visual speech recognition (VSR) is the ability to extract spoken words from a video of an individual talking without the use of audio data, which can otherwise be known as lip reading. This topic has been of focus to researchers in situations where audio data may be corrupted, such as in a noisy restaurant or train, or entirely missing, such as old school silent films or off-microphone exchanges between athletes or politicians. When used in conjunction with tradition audio-based speech recognition visual speach can further increase ASR performance, especially in noisy environments such as in a car or a noisy restaurant where audio-based solutions deteriorate.

In recent years machine learning algorithms have been applied to the problem of ASR most commonly seen in technology like Amazon's 'Alexa' or Apple's 'Siri'. The main challenge of ASR is the need for strong statistical models since it would not be feasible to teach a machine learning model all the words in the English language (of which

there are over 140,000). Instead, traditional models make use of phonemes, which are the distinct sounds present in speech that make up words. Models identify phonemes and attempt to properly assemble them together to identify spoken words [32].

Hidden Markov Models (HMMs) have been successfully used since the 1970s for ASR [42]. HMMs use statistical methods to determine the most likely state that follows the current state. Since 2008 though, Recurrent Neural Networks (also known as Long Short Term Memory (LSTM)) have replaced HMMs as the premier statistical model for ASR. The advantage RNNs have over HMMs is that they use information from more than just the current state when making a prediction on the next state. The ability for models to incorporate more data when selecting the most likely sequence of phonemes have seen rises in ASR recognition rates [28].

In the English language there are approximately 44 phonemes [1] (this number is approximate due to differences in accent, pronunciation, etc.), some examples are the *p*, *oo*, and *sh* sounds. Deep learning models (DLMs) detect phonemes and use them to determine the most likely words being spoken. The biggest difficulty that arises during this deep learning stage is that phonemes are rarely detected at 100% likelihood so it requires choosing the most likely sequence of phonemes. This makes the use of contextual information extremely important in determining the highest probability word being spoken [1].

For visual speech recognition, word prediction is even more challenging due to the smaller quantity of visemes available relative to phonemes. Visemes are the visual equivalent of phonemes and are the shapes the mouth makes when making certain sounds. In the English language there are only approximately 14 visemes since multiple phonemes correspond to the same viseme [35] [23]. For example the phonemes *p*, *b*, and *m* all have the same viseme associated with them, as shown in Figure 1.1.

This makes the process of predicting the spoken word even more challenging than in traditional audio-based ASR.

#	phonemes	example	#	phonemes	example
1	p, b, m	put, bed, mill	8	n, l	lot, not
2	f, v	far, voice	9	r	red
3	T,D	think, that	10	A:	car
4	t, d	tip, doll	11	e	bed
5	k, g	call, gas	12	I	tip
6	tS,dZ,S	chair,join, she	13	Q	top
7	s, z	sir, zeal	14	U	book

Figure 1.1: Chart of Phonemes Associated with the Same Viseme Number [23]

1.2 Motivation and Approach

This work presents an approach for determining the spoken word from a visual dataset of known words. Rather than tackling the entire problem of VSR as several other papers have attempted, we attempt to build a model that can choose between a dataset of 500 different words when presented with training samples of each. This research could potentially be used to assist more advanced VSR models when confidence values of choosing between words is low.

Rather than relying on traditionally used statistical machine learning models we focus our efforts on training a Convolutional Neural Network (CNN) to recognize sequences of lip movements as words. This shifts our problem to more of a traditional object

recognition problem, seen commonly in computer vision, that CNNs have had great success with. CNNs apply filters to an image, essentially looking for certain patterns or shapes present in the image. This information is then used to make a prediction on what can be seen in the input image.

Instead of using a traditional CNN, which has 2-Dimensional layers and is used primarily on images (2-Dimensional collections of pixel data), we will use a 3-Dimensionals CNN, where convolutional layers are 3 dimensional in nature, by adding a time dimension to the input data. This is based on the observation that dynamic information encoded in lip movements contribute to improved intelligibility in lip reading. Hence temporal features extracted with 3D-CNNs should improve ASR performance. The 3D-CNN will be trained on a dataset of 500 different words and its performance will then be evaluated on a set of test videos containing words from the dataset the network has not seen before.

In order to train a model to recognize spoken words, accurately detecting lips in the frames of a video is essential. We will develop a pipeline for assembling 3D data packages of lip frames from short videos taken from a dataset of words from television programs. High-level face detection algorithms will first be used to assist in building the lip detector framework. The facial detection algorithms used are based upon the principles of Histogram of Oriented Gradients (HOG) and Support Vector Machines (SVMs), which are commonly used algorithms in the field of computer vision. A top-level flowchart of this process can be seen in Figure 1.2

1.3 Scope of Thesis

The goal of this thesis is twofold: develop a framework for processing videos into a collection of lip frames and train a 3D-CNN to detect words from a test dataset at an

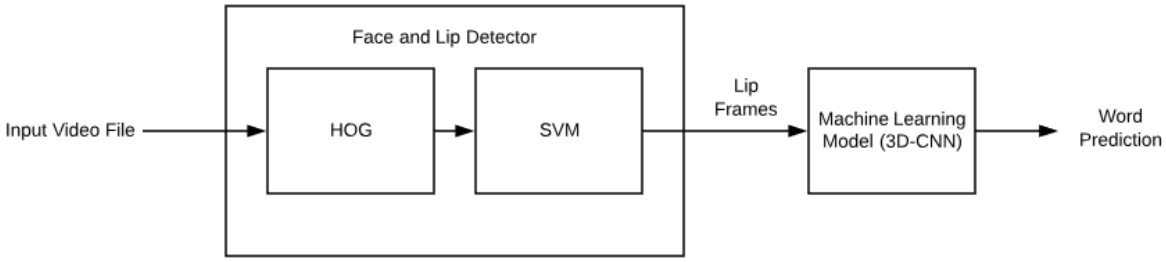

Figure 1.2: Top Level Flowchart

accuracy close to that achieved in recent VSR research, at least 60%. Lip detection algorithms and framework must be applied to the thousands of videos in the dataset in order to prepare the data needed during the machine learning phase. The machine learning phase will then attempt to train the 3D-CNN using the lip data to identify spoken words.

This thesis is organized as follows: Background, Methods, Implementation, Testing and Results, and Conclusions. The background section will go in depth on both the computer vision and machine learning principles behind this thesis. Methods will discuss the general problem solving approach. Implementation will present the specific implementation of solving this problem from a software and mathematical standpoint. Testing and Results will discuss the models performance on a test set of unseen videos of learned words. Conclusions will present the biggest difficulties and challenges faced during this thesis as well as the most promising discoveries and results.

Chapter 2

BACKGROUND

2.1 Overview

This chapter outlines the relevant theory and literature used throughout this thesis. The theoretical areas of the thesis are broken down into two larger parts: computer vision and machine learning. Computer vision is used to address the face detection and lip detection. The face detection algorithm employs Histogram of Oriented Gradients (HOG) for feature selection and Support Vector Machine (SVM) for classification. Machine learning consists of the implementation of the 3D convolutional neural network and its training and subsequent testing. Lastly, the most relevant literature on Visual Speech Recognition will be reviewed.

2.2 Face and Lip Detection

Facial detection has seen great advancements in the past decade and its use can be seen in everyday technology, such as auto-focus on smartphone cameras. Software is used to detect the faces in a digital image and then the camera is focused on those detected faces. Any facial identification system uses a face detection algorithm for its first stage; you need to detect a face before you can begin to identify it!

Face detection can be considered a specific type of object detection; object detection is used to locate all items that belong to a certain class in an image. In this instance the class is human faces and the algorithm is tasked with finding the location of each

face and putting a bounding box around each detected face as verification as seen in Figure 2.1.

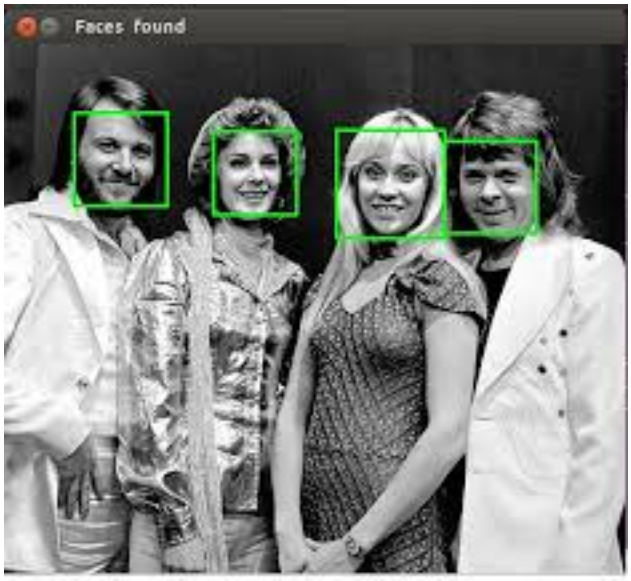

Figure 2.1: Example Output of a Face Detection Program [16]

Algorithms used for face detection include Classifier-Cascades (Viola-Jones), Eigenfaces, Cross-correlation, Neural Networks, Principal Component Analysis, and many more [39] [14]. This paper will use an algorithm based on Histogram of Oriented Gradients and Support Vector Machine and will provide a background and review on each [34].

2.2.1 Digital Image Representation

Before reviewing the algorithms used in face detection it is important to understand how images are represented in a digital world. The first step in this understanding is to think of an image as a 2D array of pixels, where pixels are the smallest possible units that make up an image. This can be shown in the picture of the letter 'a' in Figure 2.2 . Each square in the grid represents one pixel and each pixel has a value

associated with it that describes its color. The left grid shows what the image would look like and the right grid shows the corresponding 2D array of pixel values.

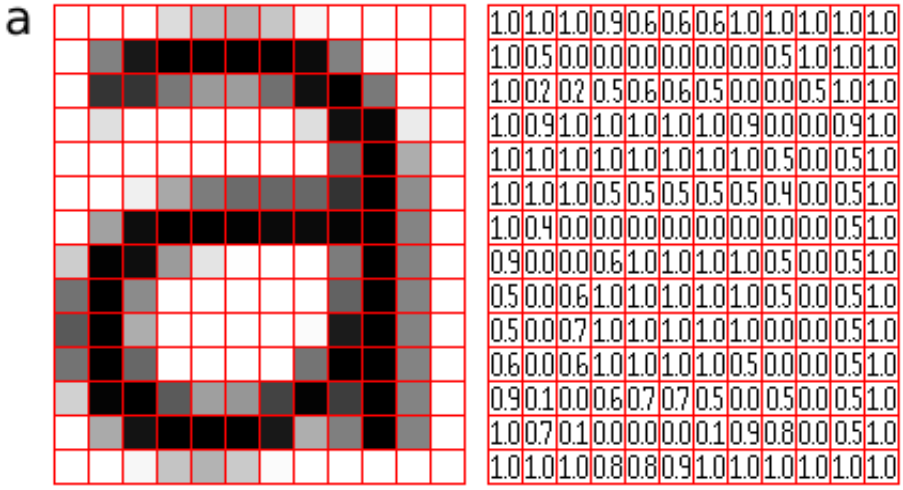

Figure 2.2: Digital Image Representation [9]

For this example, the pixel values are normalized between 0 and 1 but in a standard 8-bit image the values would be between 0 and 255. A pixel value of 0 corresponds to completely black and a pixel value of 255 (or 1 in the normalized case) corresponds to completely white. Since this is a gray scale image the array is 2D which corresponds to the height and width of the image (in pixels). The image of the letter 'a' in the example is a 14x12 array.

For color images the array is 3D instead of 2D. It shares the same height and width elements of gray scale images but now has three 'channels', or 2D layers, with one channel each for the red pixel value, green pixel value, and blue pixel value (RGB). RGB is just one of multiple color spaces used for encoding digital images but is by far the most common so the review will be limited to the RGB color space. In a color image the three pixel values at each location combine to form the color you see in the image. A visual depiction is included in Figure 2.3 to help readers visualize the 3D array representation of color images [12].

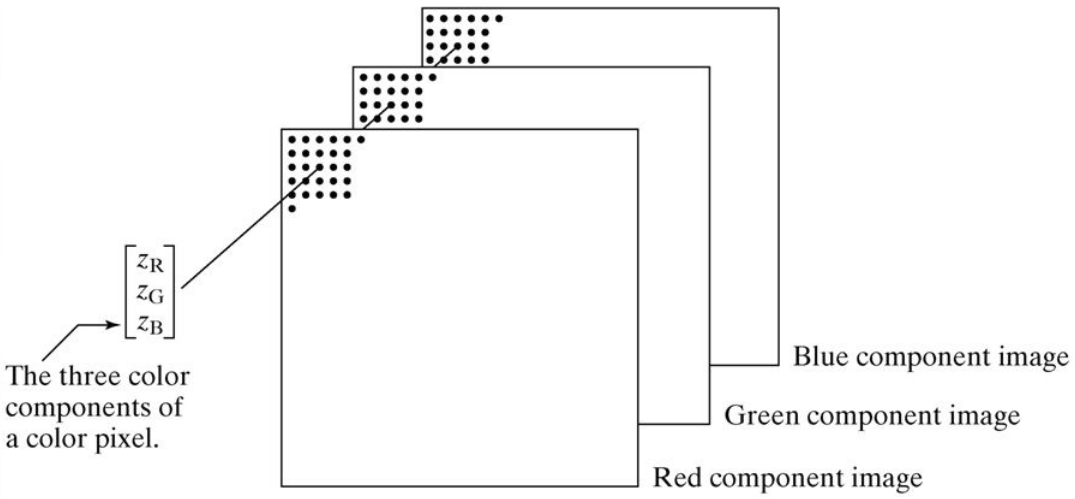

Figure 2.3: Representation of Digital Color Images as 3D Array [12]

Every subsequent concept in computer vision builds on this idea of treating images as arrays of pixel values. Doing this allows images to be broken down into a set of numeric data which can then be used in computations with various algorithms.

2.2.2 Histogram of Oriented Gradients

Histogram of Oriented Gradients, known in the computer vision world as HOG, is a commonly used feature representation for object detection. Its use for human detection was first presented in [34] in 2005. HOG has shown great success in person detection and has been further applied to the problem of face detection. Although HOG has seen an increase in use for face detection it is worth mentioning that the Viola-Jones object detection framework, which makes use of Haar features, is still the predominantly used algorithm for face detection. The HOG technique is chosen for this paper for two reasons: it has easier to use open-source implementations and is more easily adapted to perform lip detection instead of face detection.

The first step in HOG involves computing the gradient vector at each pixel location, also known as the image gradient [18]. Computing the gradient vector at each location

involves comparing the adjacent pixel values and finding the direction that the image is becoming lighter, or increasing in pixel value. To demonstrate this, consider the example shown in Figure 2.4 where the gradient vector will be calculated for the pixel highlighted in red.

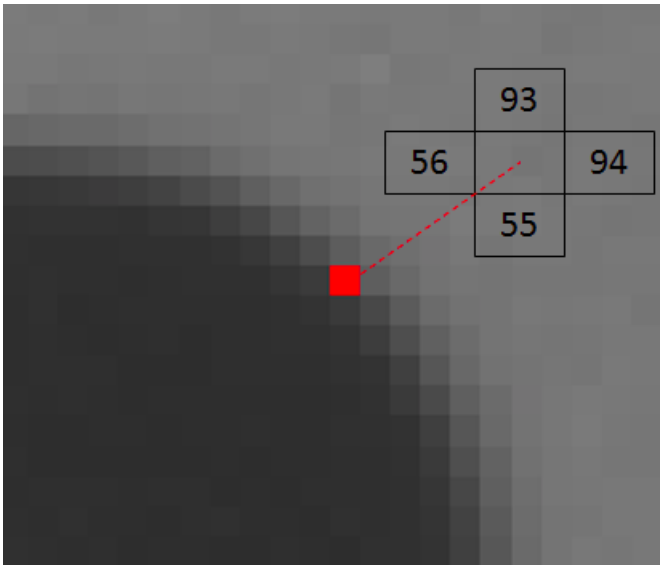

Figure 2.4: Calculating the Gradient Vector [18]

The pixel values are increasing upwards with a magnitude of $93 - 55 = 38$, and towards the right with a magnitude of $94 - 56 = 38$. Combining these two changes as vectors in the x and y directions gives a resultant gradient vector shown in Figure 2.5. This process is repeated for every pixel location in the image, except for edge pixels.

After calculating all of the necessary gradient vectors the next step is to break the image down into cells and assemble histograms of the gradient vectors in each cell [19]. First, the size of each cell must be defined, a common size used in HOG algorithms is an 8x8 cell (64 pixels total) such as the one highlighted in red in the image shown in Figure 2.6. Then a histogram of each gradient vector in the cell is assembled as shown in Figure 2.7 (except the bin values would range from 0-360 for this application) where the bin numbers refer to vector orientation in degrees, and the counts in each bin are

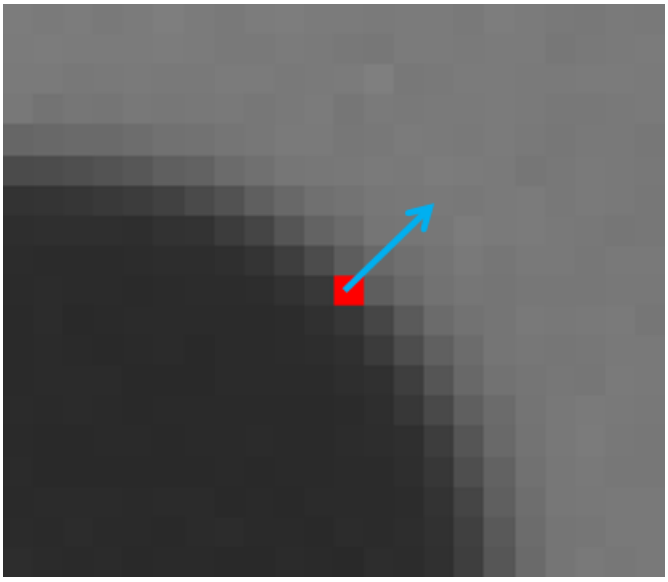

Figure 2.5: Resultant Gradient Vector [18]

the sum of the magnitudes of each gradient vector in that direction. A bin size of 20 degrees is typically chosen (as seen in Figure 2.7).

Each cell is then represented by the vector orientation that is most frequent in the histogram. This process is then repeated for every cell in the image until each cell has a gradient vector to represent the pixel change in that localized area of the image. This process significantly reduces the dimensionality and size of the image data when compared to using the raw pixel values. When performed on an image of a human face the output is shown in Figure 2.8. The HOG representation basically shows the flow of light across the image from dark to light (increasing pixel values).

The output of the HOG is a feature vector containing magnitudes and directions at each location in the image. The size of the feature vector is based on the cell size used to generate the individual histograms. This feature vector can then be input into a deep learning algorithm, such as a support vector machine. For face detection specifically, the algorithm will have been trained using other HOG feature vectors

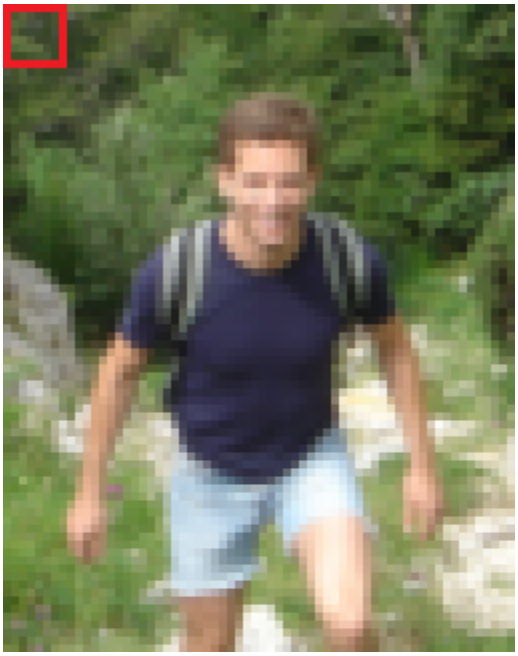

Figure 2.6: 8x8 Cell for HOG [19]

from images with faces in it and will then decide if the HOG feature vector is a positive or negative match to the class of images it has been trained on.

2.2.3 Support Vector Machine

Support Vector Machine, commonly referred to as SVM, is a widely used classification algorithm in the world of machine learning. It is a supervised learning model that performs binary classification, which classifies an input as belonging to one of two possible classes. The key feature of the SVM is that it finds the optimal hyperplane between two classes by utilizing an allowable margin for overlap between the two classes [48] [8].

In order to better understand the mechanics behind SVM, it iss best to start with a simple example of a 2D dataset. Consider the dataset shown in Figure 2.9, the goal of the SVM is to draw a line that best separates the two classes. Doing so will then allow the SVM to classify future points by determining what side of the classifier line

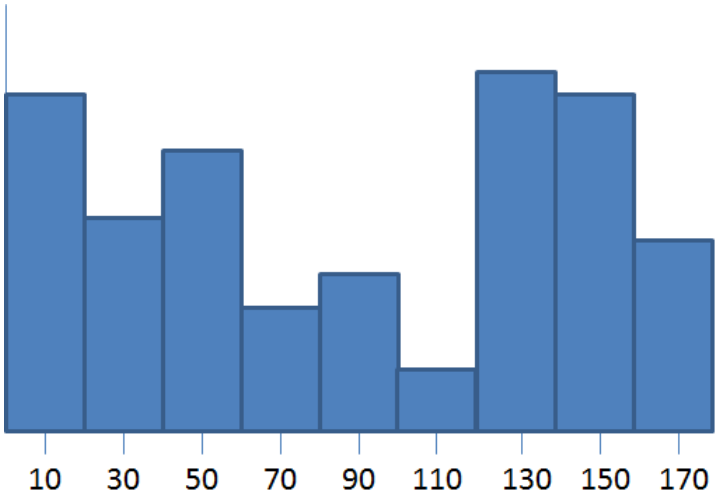

Figure 2.7: Histogram Used in HOG [19]

it lies on. Drawing a line to separate the two classes is easy enough and the green line on the figure is just one of many options, but how can SVM decide which line is the optimal line? The key concept of SVM is that the classifier boundary should be as far away from each class of data point as possible; notice how the green line is about equal distance from the black data points and the blue data points. This is achieved by carefully considering the margin when calculating the classifier.

To understand the math behind SVM consider a similar example shown in Figure 2.10. First let us focus on the derivation of the boundary equation. The classifier equation is the same for other linear classifiers: $g(x) = \mathbf{w}^T\mathbf{x} + b$, where $\boldsymbol{x}$ is the input data, which would be a vector with dimension 2 in this example, $\boldsymbol{w}$ is the weight vector that gives the boundary equation its slope, and b is the bias that shifts the equation up or down. When $g(x) > 0$ the classifier chooses $\boldsymbol{x}$ to belong to class 1 and when $g(x) < 0$ the classifier chooses $\boldsymbol{x}$ to belong to class 2. So then $g(x) = 0$ is clearly the boundary equation for the classifier as shown in Figure 2.10 (the bolded line in the middle) [51].

Figure 2.8: Input Image (left) and Output of HOG (right) [15]

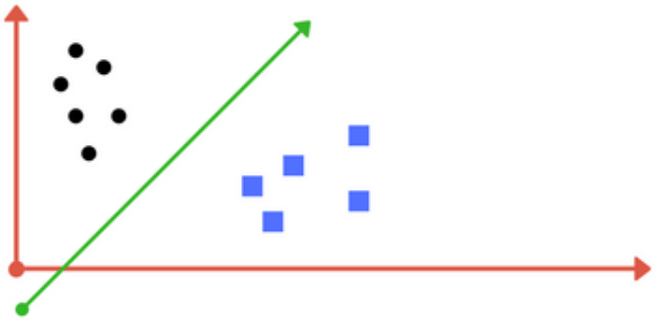

Figure 2.9: SVM Example for 2D Dataset [8]

The next step in the SVM process is to determine the equation for the margin lines. Since the goal of SVM is to find the line ($\boldsymbol{w}$ and b values) that best separates the classes, the classifier equation is applied to the points closest to the boundary. For the blue points closest to the point the equation $\mathbf{w}^T\mathbf{x} + b = 1$ is used and for the red points the equation $\mathbf{w}^T\mathbf{x} + b = -1$ is used [51]. This is done to generate a margin that can be used to best separate the two different classes.

The margin is then given by the following equation (and can also be seen on Figure 2.10, where $\mathbf{x}_+$ and $\mathbf{x}_-$ are the data points from each class closest to the margin:

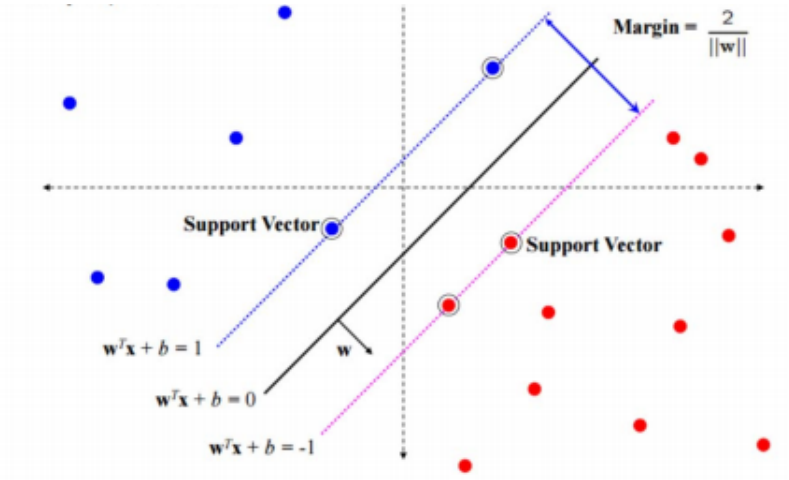

Figure 2.10: SVM Example with Boundary Equations [51]

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

The next step in the derivation is to find the boundary equation by maximizing the margin between classes. This is considered a constrained optimization problem and is formulated using the Lagrange multiplier [51].

The primal formulation is then, where y_i are the individual training samples and α is the Lagrange Multiplier:

$$\min L(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i [y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1]$$

Rewriting the primal formulation with the weights in terms of the training data, y_i, and substituting in leads to the dual formulation:

$$\max L(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

subject to: $\sum_{i=1}^{n} \alpha_i y_i = 0, \alpha_i \geq 0$

These equations are then solved using quadratic programming and the discriminant function is found to be:

$$g(\mathbf{x}) = \sum_i \alpha_i y_i (\mathbf{x}_i^T \mathbf{x}) + b$$

Sometimes attempting to find a boundary with zero misclassifications is not possible due to noise and outliers in the data to be classified, this issue is solved by the introduction of a *soft* margin. This introduces a new parameter C which changes the optimization problem to:

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

subject to $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i$

In this situation C acts as a regularization parameter that controls the effect of the constraints and ξ is a 'slack' variable that is used to determine when a sample is either a margin violation or a misclassification. A small C value will allow constraints to be easily ignored which results in a large margin, where as a large C makes constraints much harder to ignore which results in a small margin. A hard margin where all constraints must be followed is the case where C is equal to infinity, which is simply the base case first presented [51].

In the previous two examples a 2D space was considered which meant that the boundary equation was a 1D line. The SVM algorithm can be used in higher dimension spaces as well but instead of a boundary line it will be a boundary hyperplane. The SVM used in the face detection algorithm operates in a much higher dimension space and as such utilizes a boundary hyperplane.

For the specific problem of face detection, a sliding window search strategy was employed where classification is performed on different windows in the image. SVM compares the feature vector generated by HOG to other feature vectors of known classes. First, HOG is applied to training images of faces and images of non-faces. Each image is then associated with an output feature vector. These output feature vectors are then used to train the SVM for classification. The trained SVM will then perform the classification on HOG feature vectors.

If the sliding window search method identifies multiple potential bounding boxes, as it commonly does, only the best fit box will be returned. Choosing of the best fit box is done via non-maximum suppression (NMS). NMS starts with the bounding box that has the highest score, which in the case of the SVM would be the distance from the boundary equation, and suppresses all other boxes that overlap it [45].

2.2.4 Lip Detection

Following the face detection lip detection is performed by detecting the key facial landmarks using a shape predictor. The shape predictor to be used is based off of [41] and makes use of an ensemble of regression trees. This method is chosen because its speed allows for real time detection. An example of the output of a shape predictor used in conjunction with a face detector is shown in Figure 2.11.

The figure shows the initial face detected highlighted with the green region of interest box. The shape predictor is then used to identify facial landmark points which are identified by red dots. From here, the red dots corresponding to the lip region can then be used to determine the region of interest for the lip data and crop that area out of the image. Next, a review of regression trees is given.

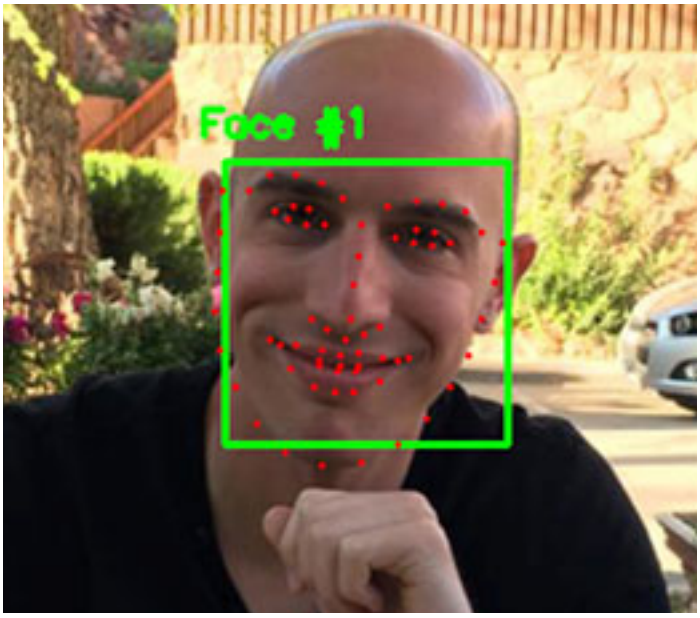

Figure 2.11: Example Output of Facial Landmark Detection [17]

The algorithm starts with an estimate of facial landmarks based on the bounding box returned from the face detector and the mean location of landmarks from the training images. Each regressor function is then used to predict an update to the estimated location of the facial landmarks based on relative pixel intensities. The process is iterated until a combined cascade of regressors gives a sufficient accuracy level [34].

The vector **S** will be used to denote the (x, y) locations of the facial landmarks and is considered the 'shape' that is trying to be predicted. The current estimate of the shape is denoted by $\hat{S}^{(t)}$. Each regressor r_t in the cascade predicts an update vector that is added to the current estimate using:

$$\hat{S}^{(t+1)} = \hat{S}^{(t)} + r_t(I, \hat{S}^{(t)})$$

The initial regressor starts by making use of a training set of manually labeled facial landmarks, S_i, on images, I_i. The first regression function, r_0, is based off a triplet

of a face image, I_{π_i}, an initial shape estimate, $\hat{S}_i^{(0)}$, and a target update step, $\Delta\hat{S}_i^{(0)}$, each defined by:

$$\pi_i \epsilon \{1, ..., n\}$$

$$\hat{S}_i^{(0)} \epsilon \{S_1, ..., S_n\} \backslash S_{\pi_i}$$

$$\Delta S_i^{(0)} = S_{\pi_i} - \hat{S}_i^{(0)}$$

r_0 is then determined using gradient tree boosting with a sum of square error loss using the following algorithm:

Algorithm 1 Learning r_t in the cascade

1. Initialise $f_0(I, \hat{S}^{(t)}) = argmin \sum_{i=1}^{N} \left\| \Delta S_i^{(t)} - \gamma \right\|^2$
2. for $k = 1, ..., K$
(a) for $i = 1, ..., N$

$$r_{ik} = \Delta S_i^{(t)} - f_{k-1}(I_{\pi_i}, \hat{S}_i^{(t)})$$

(b) Fit regression tree to targets r_{ik} giving weak regression function $g_k(I, \hat{S}_i^{(t)})$
(c) Update (v is learning rate)

$$f_k(I, \hat{S}^{(t)}) = f_{k-1}(I, \hat{S}^{(t)}) + v g_k(I, \hat{S}^{(t)})$$

3. Output $r_t(I, \hat{S}^{(t)}) = f_K(I, \hat{S}^{(t)})$

Now that r_0 is learned, the next regressor r_1 is then learned using the updated training data by setting $t = 0$ and using:

$$\hat{S}_i^{(t+1)} = \hat{S}_i^{(t)} + r_t(I_{\pi_i}, \hat{S}_i^{(t)})$$

$$\Delta S_i^{(t+1)} = S_{\pi_i} - \hat{S}_i^{(t+1)}$$

This new set of training data is then used with the previously defined algorithm to learn the next regression function. The process continues until the cascade of regressors when combined reach an acceptable accuracy. Each regression function forms the basis for a tree based regressor.

At each split node in the tree a decision is made based on the thresholding difference between two pixels at locations **u** and **v**. It is ideal to choose two points who have the same position relative to its shape as **u** and **v** have to the mean shape. This is achieved by warping the image using the following functions:

$$\delta x_u = u - \bar{x}_{k_u}$$

Where k_u is the index of the facial landmark in the mean shape that is closest to u and δx_u is the offset. The the position in an image I_i that is most similar to **u** in the mean shape image is given by:

$$u' = x_{i,k_u} + \frac{1}{s_i} R_i^T \delta x_u$$

Where s_i and R_i are the scale and rotation matrix that transforms S_i to $\bar{S}$, the mean shape. R_i and s_i are found to minimize:

$$\sum_{j=1}^{p} \|\bar{x}_j - (s_i R_i x_{i,j} + t_i)\|^2$$

Now each split in the decision tree is based on 3 parameters $\theta = (\tau, \mathbf{u}, \mathbf{v})$ and is applied to each training sample using:

$$h(I_\pi, \hat{S}_i^{(t)}, \theta) = \begin{cases} 1 & I_{\pi_i}(u') - I_{\pi_i}(v') > \tau \\ 0 & otherwise \end{cases}$$

The decision at each node in the tree is based on thresholding the difference in intensity values between two pixels. This helps insensitivity to changes in lighting as opposed to one pixel thresholding. The drawback is that there are quadratically more pairs than single pixels. This issue is eased by taking the prior structure of image data into account. This is achieved by using an exponential prior that is based on the distance between the pixels used in the split and the follwing:

$$P(u, v) \propto e^{-\lambda \|u-v\|}$$

The proportional term is effective in reducing error rate and for reducing computation time as well. This math that makes the basis of the shape predictor is based on [41], which showed that this algorithm can obtain an error rate as low as 0.049.

2.2.5 Data Processing

The last step before going on to the model training is the preparation of the data extracted using the lip detector. Since the goal is to form a 3D array by adding a time dimension to the already existing spatial dimensions, each frame is converted to grayscale from color. If the images were not converted to grayscale each frame would already be 3D and the time dimension would then make the output array 4D in nature. Minimal research has been done on 4-Dimensional CNNs and few implementations exist.

The lip data detected in each frame is a bounding box around the lips, which is then cropped from the rest of the frame, forming a 2D array of pixel values. Each frame of lip data in the video will likely be of different size so the 2D data arrays need to be resized to a common size in order for them to be compatible for stacking. After each frame is resized they are all stacked together to form a 3D array. Each frame of lip data is 2D and there are multiple frames in each video which adds a third dimension, time, to the data. Figure 2.12 shows a visualization of the stacking of 2D arrays.

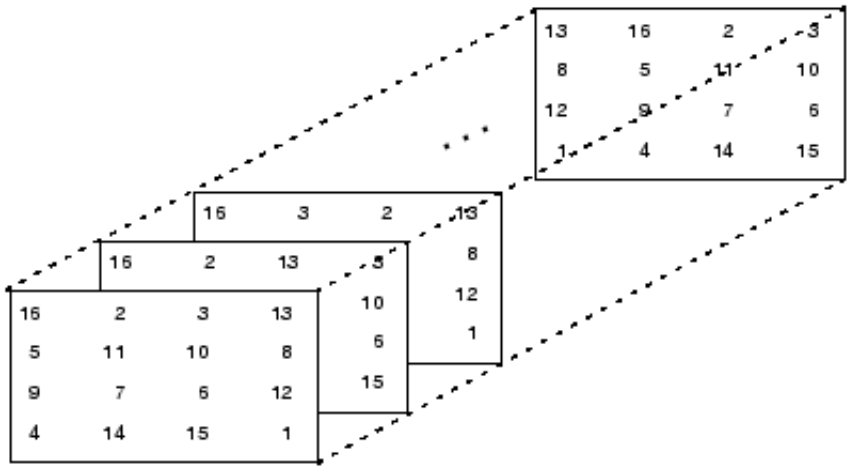

Figure 2.12: Stacking 2D Arrays to form a 3D Array [24]

2.3 Convolutional Neural Networks

The purpose of this next section is to explain the background on Convolutional Neural Networks (CNNs), which is the machine learning model that will be used in speech recognition. CNNs are a variation of standard neural networks (NNs); while in standard neural networks the layers of neurons are 1D in nature, they are 2D layers in CNNs. This makes CNNs the obvious choice when solving image related problems. Since digital images are 2D arrays of pixel data, as explained in the digital image representation section, the 2D layers of CNNs work extremely well in conjunction with these images.

2.3.1 Neural Networks

Although there is a large difference between traditional neural networks and CNNs due to their layer composition, there are a lot of similarities in the construction and training of each. Before we can fully understand the principles behind CNNs it is important to first understand NNs and their basics.

The most common use for NNs is to take a set of input data and then classify it into different groups. They work by having layers of *neurons*, modeled off the neurons in a human brain, that connect to each other and help the network make decisions and learn patterns in data [6]. An overview of a simple neural network structure can be seen in Figure 2.13.

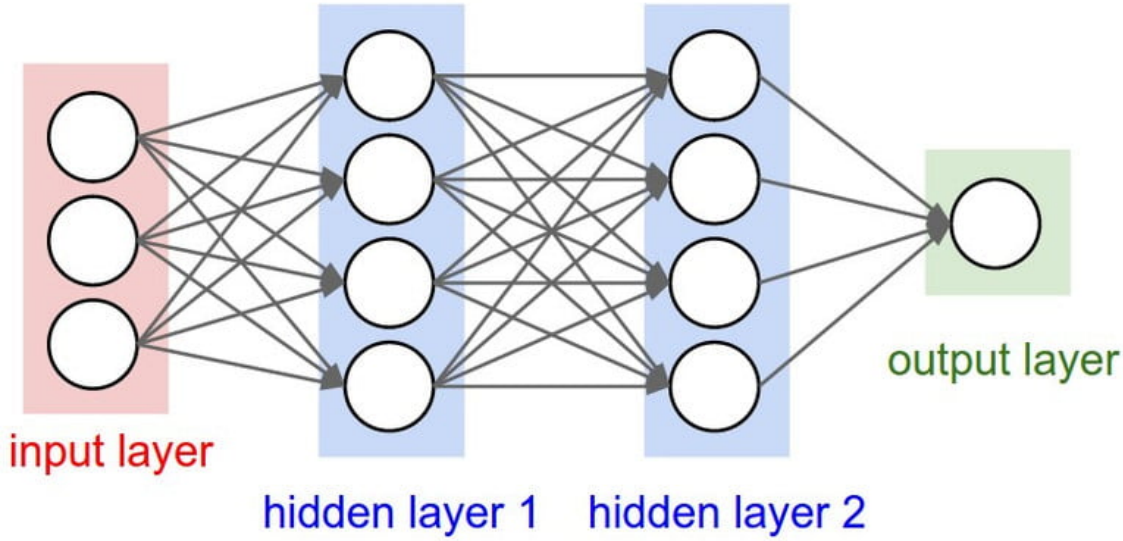

Figure 2.13: Architecture of a Simple Neural Network [26]

In Figure 2.13, the white circles represent the individual neurons that make up each layer of the network. The connections between neurons, represented as grey arrows in the image, are called *weights*, and their values are updated as the model is trained. These weights are the most important part of the network and are what allows the network to make decisions about input data. All NNs have an input layer (shown

in red), an output layer (shown in green), and a variable amount of hidden layers (shown in blue).

Input layers are simply just the direct input values of the data being fed to the network. The output layer is the layer tasked with performing classification of the input data and the size usually corresponds to the number of classes in the dataset. If the task of the NN is to predict a number based on a picture of a handwritten digit, the 'classes' would be the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Since there are ten different classes the NN would have 10 output neurons. The output neuron with the highest value after the input is fed into the network would be the predicted class. The hidden layers are what gives the NN more complexity and allows it to perform more difficult tasks. More hidden neurons equals more parameters which allows the network to learn more complex patterns in the data.

Another important note about NNs is that each neuron is connected to every neuron in the previous layer and every neuron in the next layer. Consider the example shown in Figure 2.14 which shows 5 input neurons connected to two hidden neurons.

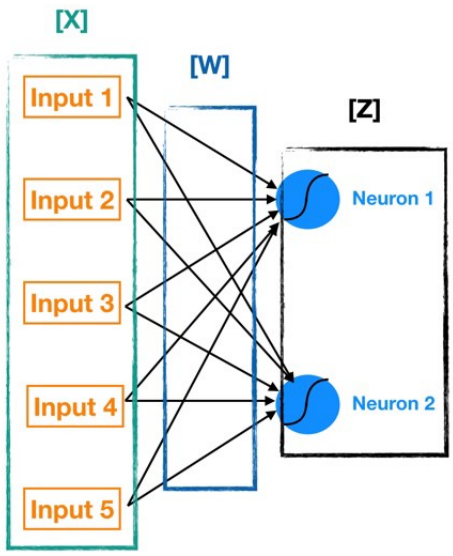

Figure 2.14: Weight Connection Between Neurons [25]

The values in the input neurons are denoted by $x_1, ..., x_5$, while the values in the hidden neurons are denoted by z_1 and z_2. Let the weights be denoted by w_{ij} for the connection between input i, and neuron j in the next layer. The value z_1 can then be expressed as a summation of the inputs and weights according to:

$$z_1 = w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + w_{41}x_4 + w_{51}x_5$$

The value of the weights is determined during training which will be covered in a later section. The z_1 value is then put into an activation function and the output of that function is the value held in the neuron. Activation function are used to increase the non-linearity of the network and help it realize more complex functions and patterns. Some commonly used activation functions for neural networks include the sigmoid function and the rectified linear unit (ReLU). These functions are included below [25].

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-}x} = \frac{e^x}{e^x+1}$
ReLU: $R(z) = x^+ = max(0, x)$

Graphs of these two activation functions are included in Figure 2.15 to demonstrate the non-linear nature of each function.

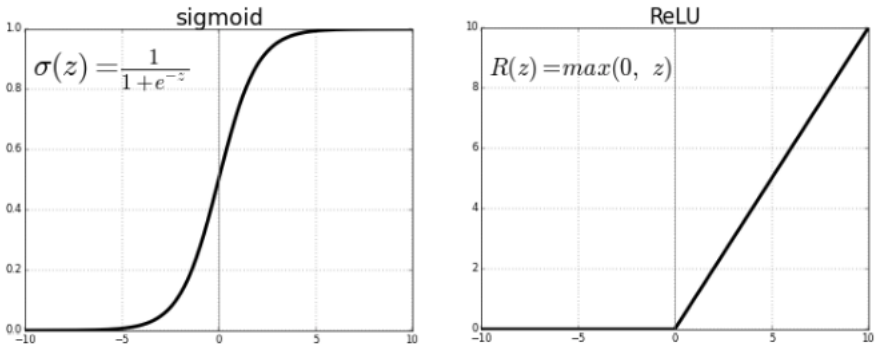

Figure 2.15: Graph of Sigmoid and ReLU Activation Functions [4]

2.3.2 Model Architecture

Now that the basics of neural nets have been covered, it is now appropriate to review the theory on CNNs, as they are the machine learning model that will be implemented in this paper. First, we will cover the architecture of CNNs and how they relate to NNs.

Firstly, consider the CNN architecture shown in Figure 2.16, this network is used to classify images into categories such as car, truck, van, bicycle, etc. The network consists of multiple different types of layers that each behave differently. There are three main types of layers: Convolutional, Pooling, and Fully Connected. This differs from a NN where each layer operates the same but has a variable amount of neurons [10].

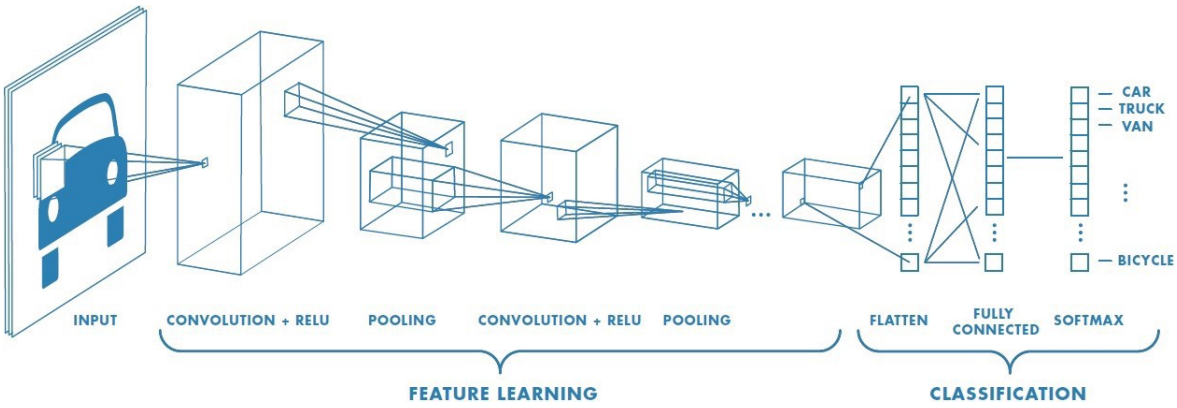

Figure 2.16: Architecture of a Common CNN [10]

The first layer is a convolutional layer that is connected directly to the raw pixel values of the input image (the input image and first layer must have the exact same shape). This layer is the core of all CNNs and the most important element of the network. They work by convolving a 2D filter around an image (or the output of a previous layer). Consider the example shown in Figure 2.17, the blue square is the filter and the red square is the image that the filter is being convolved around. A

filter is a 2D array of numbers (the values of which are determined during training just like weights in a traditional neural network), also called a convolutional kernel, and in this case the filter size is 3x3 (which is the most commonly used size, with 5x5 the next most common).

The filter values and the image values in the neighborhood around the center pixel are multiplied via dot product and added to replace the center pixel (the purple square in the right array). This process is repeated for every area on the image that the filter can fit on and the output values are stored in a new array that is the result of the convolution. All of these values are then input into an activation function, which for CNNs is most commonly the ReLU activation function, and the result is the output of the convolutional layer [11].

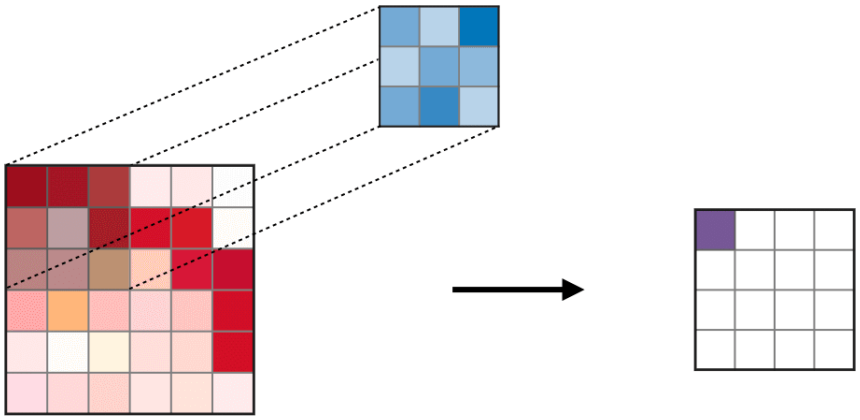

Figure 2.17: Example of a Filter Being Convolved Around an Image [11]

After one or multiple convolutional layers follows a pooling layer. The most commonly used pooling layer is max pooling, but average pooling is also occasionally used (in LeNet specifically [11]). Pooling layers are used to extract dominant features and reduce the dimensionality of the data, which saves computing power [10]. Most traditional max pooling layers take a 2x2 area and replace it with the max value of

the area. This reduces the size of the data by half in each direction. A visual example is included in Figure 2.18 that demonstrates both max and average pooling.

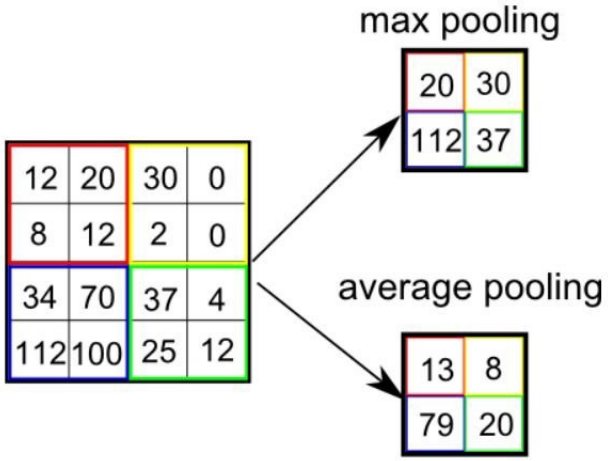

Figure 2.18: Max and Average Pooling Example [10]

The combination of input, convolution, ReLU activation, and pooling layers is what makes up the *feature learning* section of the CNN; this section performs unsupervised feature extraction. The second section of the CNN performs classification. This classification section of the CNN is actually just a standard neural net as described previously. This can be seen in the right section of 2.16.

Since the output from the feature learning section of the CNN is 2D but a standard neural net only accepts 1D inputs, the 2D output must be *flattened*, as seen in the first layer of the classification section in Figure 2.16. The flatten layer converts the 2D data array to a 1D data array. After the data has been flattened it is then input to the classification NN as described in the previous section. In CNNs these standard NN layers are called *fully connected* layers since each neuron is connected to every neuron in the layers before and after it.

Lastly classification is performed using a softmax layer. A softmax layer simply outputs the neuron with the highest value as the class. So if neuron 2 has the highest value then the CNN chooses class 2 as the most probable selected class for the input

image [11]. Softmax is chosen instead of a traditional max because it also rescales output values to be between 0 and 1, where all the output values summed together equal 1. Doing this allows the values to also be interpreted as a prediction confidence rating. If the model predicts class 2 with a value of 0.65 it is much more certain of the prediction than if it predicts class 2 with a value of 0.21.

2.3.3 Model Training

Model training enables a neural network to recognize patterns and make decisions or classifications from data. It is essential in machine learning. Before we can talk about the mathematics behind CNN model training we must first discuss what the model will be trained with. Data used for training a model is called the *training dataset* and data used for testing a model (after it has been trained) is called the *test dataset*. It is important that samples used in the training dataset are not also present in the test dataset. Showing the model samples from the test set invalidates results because it is important to grade neural networks on their ability to classify unseen data.

Another consideration when training NNs is to have as much quantity of training data as possible. This is because sometimes neural networks can become overfit to their training data and lose their ability to generalize data. This means that some networks can show extremely high training accuracy (the number of samples that it predicts the correct class when compared to their training label), such as 99% or even 100%, but when applied to a test dataset perform significantly worse. What happens is that the network learns how to recognize the difference between the exact training data rather than learning the general pattern used in recognition. Using as large of a training dataset as possible typically means an increase in training variance which helps reduce the issue of overfitting.

Training datasets are generally much larger than test datasets. In addition to training and testing datasets there are validation sets. Validation sets are used to check the performance of the model after each training cycle, allowing ML engineers to tune hyper parameters on the fly, such as filter size, stride, or filter count, but is not used for model learning. The test set is used after training has been completed to grade the performance of the network. Although it is not uncommon for validation and test sets to be used interchangeably since neither set is shown to the model during training. A traditional approach is to use 70% of the data for training and 30% for testing and validation but this is just a suggestion and other splits have worked well too [3]. This split is visualized in Figure 2.19.

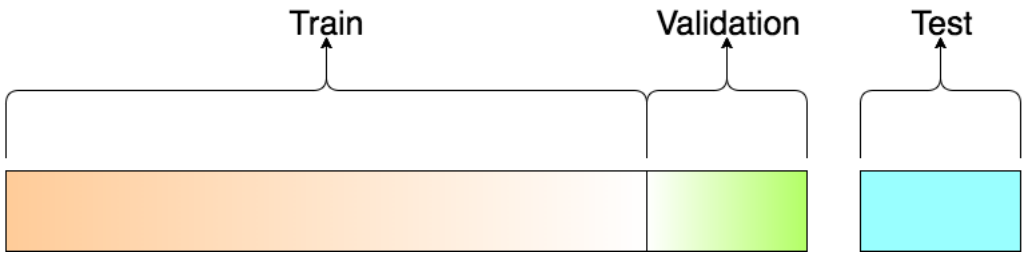

Figure 2.19: Split of Data into Training, Validation, and Test Datasets [3]

Model training consists of two important steps. First an input is fed into the model and moves forward through each layer to the end of the network. Then the expected output is compared to the actual output and the *loss* is calculated. This loss is then propagated backwards through the network to update the weights in a process known as *back-propagation*.

Lets start with how the loss is calculated. All training data in a supervised learning method, which is what is used to train neural nets, has a label associated it which tells the model which class the data acutally belongs to. For example, if a dataset has 10 different classes and a training data belongs to class 3 then it would have a label vector of:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

As can be seen there is a 1 in the 3rd location in the label vector, which is the class that the training sample belongs to, and a 0 in all the other locations since the sample is not a member of those classes. Next, the sample data is input into the network and moves forward through the network. For example say that the sample output is a vector of:

$$\begin{bmatrix} 0.22 & 0.11 & 0.76 & 0.30 & 0.09 & 0.16 & 0.19 & 0.23 & 0.13 & 0.32 \end{bmatrix}^T$$

This sample would be correctly classified because the value in the third location is the highest value so the model would choose class 3 as its predicted class. Although the model predicted the sample class correctly there is a still an error vector associated with the training sample. The error vector is calculated based on the desired loss function being used (of which there are many). For a multi-class classification problem (where the number of classes ¿ 2) a standard loss function used is categorical cross-entropy, which is what is used in this paper. Cross entropy is also called log loss, and uses the following formula where y_d is the desired output and y_a is the actual output [20]:

$$loss(y_d, y_a) = -\log|1 - y_d - y_a|$$

This loss is calculated for every individual entry in the output vector. A plot of the loss function when the desired output is 1 is included in Figure 2.20. As can be seen

when the actual value is close to 1 the loss is small and when the error is far from 1 it is much larger.

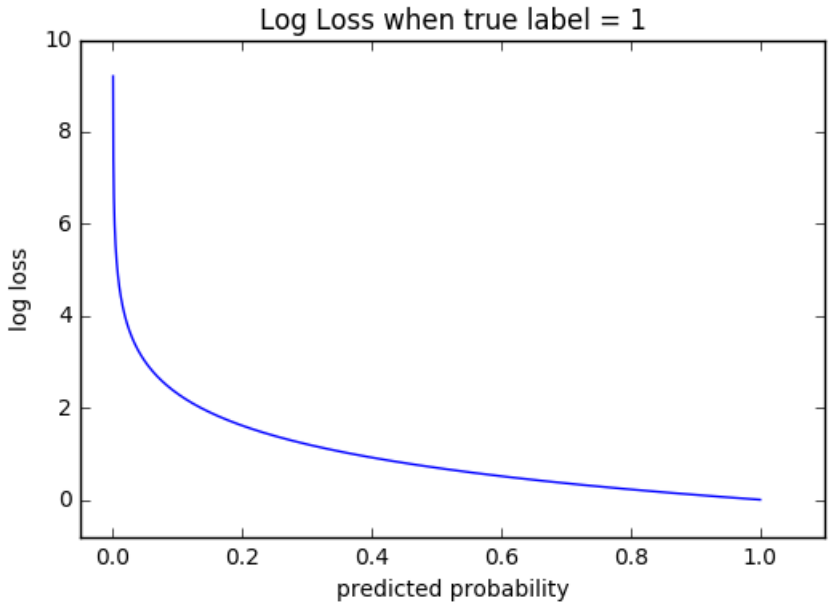

Figure 2.20: Cross Entropy Loss Function when Desired Output is 1 [20]

After the loss is calculated at each neuron in the final layer of the network, this value is then propagated backwards through the network, correcting the weights between each neuron. The goal of this is to correct the weights that are most responsible for the loss. The principle of back propagation is based on gradient descent [37].

As defined in the previous section, the neuron value before the activation function (otherwise known as the induced local field $v(n)$) for a neuron j is:

$$v_j(n) = \sum_i w_{ji}(n) y_i(n)$$

Where $y_i(n)$ is the output of the neurons in the previous layer and $w_{ji}(n)$ is the associated weight vector for each connection from neuron i to neuron j. The output,

$y_j(n)$, of the neuron after applying the activation function for neuron j, $\varphi_j(n)$, is then:

$$y_j(n) = \varphi_j(v_j(n))$$

Each neuron has an instantaneous error energy associated with it defined using the least-mean squared algorithm, where e_j is the error function being used:

$$\varepsilon_j(n) = \frac{1}{2}e_j^2(n)$$

Next step is to find the partial derivative of the error energy with respect to the weight vector. This is done using the chain rule of calculus:

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = \frac{\partial \varepsilon(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

One thing to note is that the ReLU activation function is not differentiable at 0. This is solved by setting the derivative equal to 0 at that point. Next, differentiating both sides of the instantaneous error energy equation leads to:

$$\frac{\partial \varepsilon(n)}{\partial e_j(n)} = e_j(n)$$

Using a simplified loss function of $e_j(n) = y_d(n) - y_j(n)$, the partial derivative is then:

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

Then differentiating the previously defined equation of $y_j(n) = \varphi_j(v_j(n))$ gives:

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi'(v_j(n))$$

Lastly, differentiating the previously defined equation of $v_j(n) = \sum_i w_{ji}(n) y_i(n)$ gives:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

Now substituting these individual partial derivatives into the chain rule equation leads to:

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = -e_j(n) \varphi_j{}'(v_j(n)) y_i(n)$$

The delta rule is then used to define the weight correction as:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)}$$

In this equation η is the learning rate parameter which is one of the most important parameters in model training. Combining the two previous equations yields:

$$\Delta w_{ji}(n) = -\eta \delta_j(n) y_i(n)$$

Where $\delta_j(n)$ is the local gradient of the neuron and is defined by:

$$\delta_j(n) = \frac{\partial \varepsilon(n)}{\partial v_j(n)} = e_j(n)\varphi_j{}'(v_j(n))$$

As can be seen in the previous equation the local gradient for the neuron is equal to the product of the error signal for that neuron and the derivative of the activation function. From this equation the weight adjustment for each connection can be calculated. Analyzing the equation it can be seen that the error signal is a key component to the weight update. The error signal can be broken into distinct cases: when the neuron is an output node and when it is a hidden node [37].

The output node is a more straightforward case since the desired output is provided by the label associated with the training sample. The loss function is simply the previously defined function of $e_j(n) = y_d(n) - y_j(n)$.

The more complicated case is when the neuron is a hidden node since there is no explicit desired response value such as the training label for output nodes. This means that the error signal must be propagated backwards from the output nodes to the hidden nodes. The first step in this is redefining the formula for the local gradient:

$$\delta_j(n) = -\frac{\partial \varepsilon(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial \varepsilon(n)}{\partial y_j(n)} \varphi_j{}'(v_j(n))$$

Next the partial derivative of the error energy must be calculated using the formula:

$$\varepsilon(n) = \frac{1}{2} \sum_k e_k^2(n)$$

Then differentiating each side results in:

$$\frac{\partial \varepsilon(n)}{\partial y_j(n)} = \sum e_k \frac{\partial e_k(n)}{\partial y_j(n)}$$

Combining this with previously defined equations leads to:

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k{}'(v_k(n))$$

Differentiating the previously defined formula for the induced local field, $v_k(n) = \sum_i w_{ki}(n) y_i(n)$, gives:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

Combing the previous two equations gives:

$$\frac{\partial \varepsilon(n)}{\partial y_j(n)} = -\sum \delta_k(n) w_{kj}(n)$$

Now finally substituting this equation into the error signal equation gives the local gradient for hidden neurons:

$$\delta_j(n) = \varphi_j{}'(v_j(n)) \sum \delta_k(n) w_{kj}(n)$$

Now that the local gradient has been defined we can revisit the formula for weight correction:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

$\Delta w_{ji}(n)$ is the weight correction from neuron i to neuron j. $\delta_j(n)$ is the local gradient as previously defined in the proof. $y_i(n)$ is the input signal from neuron i that feeds into neuron j. η is the learning rate parameter that is controlled during the model training. This parameter has a major impact on the model during training, if it is too small training could take much longer than necessary as weights update too slowly, wasting valuable time. If the learning rate is too large then large values of loss can propagate backwards too strongly through the network and ruin previously defined network weights [37].

The weights are then recalculated based on a concept called *batches*. A batch size of 1 means that the weights are recalculated after every individual training sample. More commonly a batch size of 16 or 32 is used in training [40]. This means that the weight correction, Δw_{ji}, is added together for 16 or 32 samples and then updated each time the batch size is reached.

Each time the full set of training samples is used for weight correction is called an epoch. In general, the more training epochs you can run the better but real life time constraints exist. It is generally chosen to use as many epochs as time and hardware allows for which could be 10, 100, 1000, or more based on the set of training data being used.

After each training epoch the data should be randomized in order to help reduce overfitting. As stated before increases in variance in the data presented during training helps reduce overfitting. Reshuffling the data helps the model avoid focusing on specific patterns in the training data and focus on general patterns instead.

During training, classification accuracy and loss is closely monitored. Once either metric reaches a level deemed acceptable the model training period is completed.

2.3.4 Model Testing

After training the model is tested using the test dataset previously set aside. This dataset is what is used to grade the performance of the model.

When testing the model each sample is pushed forward through the network but no back propagation happens because the model is done learning. After being pushed forward through the network, the model makes a prediction on the class of the input based on the max neuron value in the output layer. This prediction is compared to the known label, since the test dataset is labeled in the same manner as the training and validation sets.

After testing each sample in the test dataset the number of misclassifications is used to calculate the accuracy using the standard formula:

$$Accuracy = \frac{TotalSamples - Misclassifications}{TotalSamples} \times 100$$

2.4 Related Works

The problem of VSR has been explored in [31], [30], [49], [38], and [33], among other works.

Lip Reading in the Wild [31], authored by a team from the University of Oxford, proposes both a pipeline for automated data collection for the building of advanced datasets and CNN architectures that are able to recognize words from the assembled dataset. This proposal is of importance to the VSR research community due to the successful assembly of a new dataset in an uncontrolled environment. The authors of [31] develop a framework for generating videos (of fixed length) of specific spoken

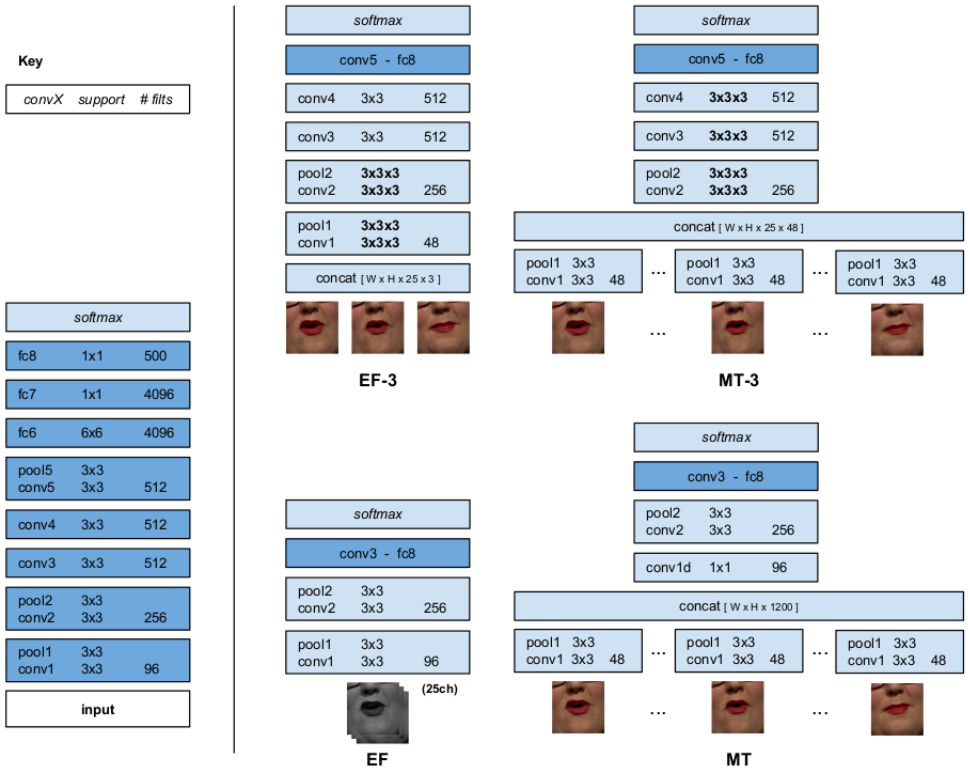

Figure 2.21: Architectures Used in *Lip Reading in the Wild* Paper [31]

words taken from TV programs. This differs greatly from most VSR datasets that are recorded and assembled in a controlled lab environment, and is a better approximation of the videos seen in real-life classification tasks.

The second proposal of [31] is to investigate the performance of different CNN architectures for recognition of the previously assembled dataset. They test the performance of 4 different architectures, two 2D-CNNs and two 3D-CNNs, and achieve a top recognition rate of 61.1% using a 2D CNN architecture with multiple input towers that are concatenated after performing one convolution on each frame. The architectures used in [31] are shown in Figure 2.21.

Interestingly, the authors of [31] consider their architectures 2D and 3D even though the inputs to each are 3D and 4D respectively. The EF-3 and MT-3 architectures take a collection of color images thus having the dimension of WxHx25x3 where W is the

width of each frame, H is the height of each frame, 25 is the number of total frames per video (the temporal dimension), and 3 is the multi-channels of the color image. Our paper uses the conventional notation that CNNs with 3D inputs are considered '3D CNNs' especially since they perform 3D-Convolutions at the layer level.

The architecture used in this work is most similar to the EF architecture. The main differences between our architecture and the EF architecture is the number of pooling layers, number and size of fully connected layers, and the number of frames that make up the input to the network. We use significantly less than 25 frames due to the observation that the duration of most words is less than 25 frames, and by including too many frames recognition rate could actually be worsened by the inclusion of information unrelated to the target word.

Lip Reading Sentences in the Wild [30], authored by the same team as [31], takes the work done in [31] and extends it a step further. Where [31] focuses on assembling a dataset of spoken words from TV programs, [30] focuses on assembling a dataset of spoken sentences from TV programs. [30] uses essentially the same pipeline as [31] with minor modifications to assemble the new dataset. The task of recognizing the spoken words is made much more difficult by the fact that the data is now sentences of variable length rather than videos of fixed length containing one word. This requires a more sophisticated ML model(s) than the one used in [31] and also requires the use of audio data during training which was not used in [31].

The architecture in [30] makes use of essentially two different machine learning stages, as is commonly seen in VSR tasks that require decoding of sentences. The first machine learning stage is used as an unsupervised feature detector that features two models in parallel, one for audio data and one for visual data. In [30] the visual section of the first stage is a CNN in conjunction with Long Short Term Memory modules (LSTM, also known as Recurrent Neural Networks) and the audio section involves

a linear cosine transform (known commonly as Mel Frequency Cepstrum, MFCC) followed by LSTM modules. The second section of the model uses the outputs from the audio and visual halves as an input into a second group of LSTM modules and attempts to assemble words and match visual and audio data [30].

This structure of two or more separate ML models is commonly used in VSR tasks that require full decoding of sentences. This model structure can be seen in Figure 2.22. The second half of this problem, assembling words from phonemes or visemes, is similar to that in standard ASR and is explored in [46] and [44]. Both these papers show the success of using Hidden Markov Models (HMMs) for speech recognition.

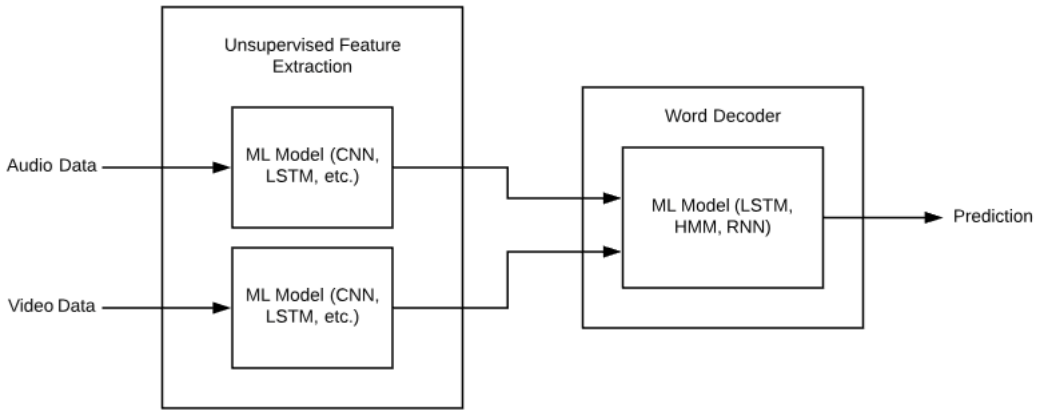

Figure 2.22: Overview of Traditional AV-ASR Programs

3D Convolutional Neural Networks for Cross Audio-Visual Matching Recognition [49], is authored by a team out of West Virginia University, and is similar in nature to [30]. [49] also uses the datasets assembled and presented in [31] and [30]. The work of [49] attempts to build a network that maps audio and visual data to the same feature space. It uses two non-identical 3D CNNs that are connected by a contrastive loss function. One CNN is used for visual data and the other is used for audio data. The contrastive loss function's goal is to pull genuine data pairs (audio and visual data

from the same video) closer together and push impostor pairs (audio and visual data from different videos) farther apart.

The work of this paper most closely follows that of [31] and [49]. The preprocessing pipeline of [49] that is used to extract lip data from videos is closely followed in this work. Additionally, the 3D-CNN used for visual feature extraction is used as a starting point network architecture for performing classification. The work of this paper relates to [31] because we attempt to use a 3D-CNN to perform feature extraction and recognition as opposed to just feature extraction, like in [49]. Essentially, the work of [49] is adapted to perform the same task attempted in [31].

Chapter 3

METHODS & IMPLEMENTATION

3.1 Overview

This section will lay out the actual software and mathematics used to implement the model used for visual speech recognition. A block diagram of the overall work is shown in Figure 3.1.

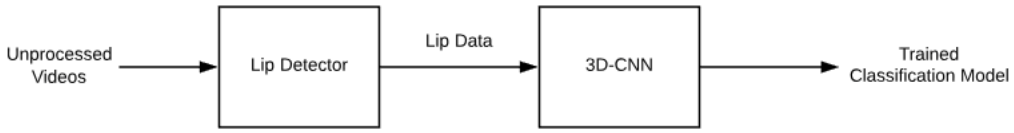

Figure 3.1: Overview of Thesis Work

3.2 Dataset

Before any code can be written, the first actual step in the project is to decide and compile the dataset to be used. Compiling a comprehensive VSR dataset is a complicated task due to the difficulty of finding and labeling data. In this situation the data is a video of a speaker and the label is the spoken word or group of words. This task has been discussed in [31] and [30].

The dataset to be used in this project is the one presented in [31] and is titled *Lip Reading in the Wild* (LRW). The LRW dataset was compiled by J. S. Chung and A. Zisserman from the University of Oxford and is presented in [31]. Table 3.1 compares

this dataset to various other datasets used in VSR research, and is pulled directly from [31].

Name	Env.	Output	I/C	# class	# subj.	Best perf.
AVICAR	In-car	Digits	C	10	100	37.9%
AVLetter	Lab	Alphabet	I	26	10	43.5%
CUAVE	Lab	Digits	I	10	36	83.0%
GRID	Lab	Words	C	8.5	34	79.6%
OuluVS1	Lab	Phrases	I	10	20	89.7%
OuluVS2	Lab	Phrases	I	10	52	73.5%
OuluVS2	Lab	Digits	C	10	52	-
LRW	TV	Words	C	500	1000+	-

Table 3.1: Comparision of Various Datasets [31]

The major differences between LRW and other datasets is the environment, number of classes, and number of subjects. Each video in the LRW dataset is cut out of a TV program on BBC, which is what gives it its *In the Wild* moniker since videos are not recorded in a controlled lab environment (AVICAR is the only other dataset not recorded in a lab). I/C stands for isolated and continuous, samples in the LRW dataset are single words cut out of an entire sentence, further adding to the more natural and in the wild nature of the data. LRW also has 500 different spoken words making it the largest of the available datasets and has over 1000 different speakers which is significantly more than any other dataset. For a combination of these reasons, the LRW dataset has been one of the most popular datasets for speech recognition research tasks.

The LRW dataset is broken up into training, testing, and validation subsets and consists of 500 classes. Each class is a single spoken word. Each subset contains samples from every class. The training subset was compiled using TV clips from 1/1/2010 to 8/31/2015 and contains 800-1000 samples per class. The validation subset was compiled using TV clips from 9/1/2015 to 12/24/2015 and contains 50 samples per class. Lastly, the test subset was compiled using TV clips from 1/1/2016

to 9/30/2016 and also contains 50 samples per class [31]. From the date range it is clear that there is no overlap between subsets since all videos are pulled from new broadcasts. These statistics are shown in Table 3.2.

Set	Dates	# class	# per class
Train	1/1/2010-8/31/2015	500	800-1000
Validation	9/1/2015-12/24/2015	500	50
Test	1/1/2016-9/30/2016	500	50

Table 3.2: LRW Dataset Statistics [31]

Each sample is a video containing 29 frames, at a frame rate of 25 frames per second, for a total duration of 1.16 seconds each. For each sample the word occurs in the middle of the video. Included with each video is metadata that gives the duration of the word, which could be used to determine the exact starting and end frames. All videos are included as *mp4* files and are labeled using the format 'WORD.mp4', such as ACCORDING.mp4 for a sample thats class is the word 'according' [31].

3.3 Classification and Accuracy Goals

The goal of this thesis is to classify the first 100 words included in the LRW dataset using a 3D-CNN. A subset of the dataset is considered rather than the entire 500 word dataset due to memory constraints. Classification of a 100 word dataset in an uncontrolled environment is still a challenging task and is still a more complex problem than classifying many of the datasets presented in Table 3.1. A list of the 100 words is presented in Table 3.3. The dataset still includes difficult to distinguish pairs such as AGAIN-AGAINST, ALLOW-ALLOWED, AMERICA-AMERICAN, ATTACK-ATTACKS, BENEFIT-BENEFITS, CHANGE-CHANGES, and CHARGE-CHARGES, which keeps the classification task a challenging problem.

ABOUT	ABSOLUTELY	ABUSE	ACCESS	ACCORDING
ACCUSED	ACROSS	ACTION	ACTUALLY	AFFAIRS
AFFECTED	AFRICA	AFTER	AFTERNOON	AGAIN
AGAINST	AGREE	AGREEMENT	AHEAD	ALLEGATIONS
ALLOW	ALLOWED	ALMOST	ALREADY	ALWAYS
AMERICA	AMERICAN	AMONG	AMOUNT	ANNOUNCED
ANOTHER	ANSWER	ANYTHING	AREAS	AROUND
ARRESTED	ASKED	ASKING	ATTACK	ATTACKS
AUTHORITIES	BANKS	BECAUSE	BECOME	BEFORE
BEHIND	BEING	BELIEVE	BENEFIT	BENEFITS
BETTER	BETWEEN	BIGGEST	BILLION	BLACK
BORDER	BRING	BRITAIN	BRITISH	BROUGHT
BUDGET	BUILD	BUILDING	BUSINESS	BUSINESSES
CALLED	CAMERON	CAMPAIGN	CANCER	CANNOT
CAPITAL	CASES	CENTRAL	CERTAINLY	CHALLENGE
CHANCE	CHANGE	CHANGES	CHARGE	CHARGES
CHIEF	CHILD	CHILDREN	CHINA	CLAIMS
CLEAR	CLOSE	CLOUD	COMES	COMING
COMMUNITY	COMPANIES	COMPANY	CONCERNS	CONFERENCE
CONFLICT	CONSERVATIVE	CONTINUE	CONTROL	COULD

Table 3.3: 100 Word Subset of the LRW Dataset

The full training set (800-1000 samples per class) for the first 100 words will be used. It was chosen to use all the samples of a subset rather than use all classes but with less samples. Using less samples per class has been shown to have a strong negative affect on neural net accuracy [31].

The validation accuracies presented in [31] will serve as the benchmark as it has the current state-of-the-art results on the LRW dataset using a CNN. The results from the four CNN architectures tested in [31] are 44% and 46% for the two 3D architectures and 57% and 61% for the two 2D architectures. These results are for classifying the entire 500 word dataset.

The authors of [31] also tested each model on a 333 word subset of the LRW dataset. The subset removes words that are plurals of another word (ex. BENEFIT and BENEFITS) in the dataset and words that have the same visemes (as explained in the introduction) such as BILLION and MILLION or WORSE AND WORST. The

accuracy's of the models in classifying the 333 word subset all see substantial increases and the new highest accuracy is 65%.

Rather than remove plurals and viseme ambiguities this thesis includes them since the dataset is already being restricted to a 100 word subset. This keep the problem both complex and challenging when compared to other VSR works. It is expected that word combinations such as BENEFIT-BENEFITS, CHANGE-CHANGES, and CHARGE-CHARGES, all of which are present in the 100 word subset, will make successful classifications more difficult. Due to this slightly different dataset used in comparison with [31] it is difficult to draw exact comparisons but the goal is to achieve similar performance classifying the 100 word dataset with pluralities to the results of the 333 word dataset.

3.4 Hardware

All processing and model training was performed on a Nvidia GeForce 2080 Ti GPU with an Intel Zion 2.1Ghz processor, 16-core CPU, with 32GB of RAM. The machine had Ubuntu 18.04.3 64-bit operating system with kernel version 4.15.0-65 and CUDA version 10.0.13.

3.5 Software

Firstly, the entire project will be written in the Python programming language. Python has been the go to language for many machine learning researchers in recent times. One of the main reasons for this is the plethora of already defined libraries and frameworks available for researchers.

NumPy is a scientific computation library that is excellent for array based operations. Much of machine learning is based in linear algebra concerning vectors and arrays, which makes NumPy hugely popular and important. In addition to this library, *SciPy* can be used for further advanced computation and *Scikit* for implementating specific ML algorithms and data mining [29].

In addition to these libraries are Keras and Tensorflow. Tensorflow is the premier open source library for handling complex back-end ML operations. It is written in C++ and Python, and also developed and maintained by Google Brain, giving it great reliability and flexibility. Keras is another open sourced library specifically for neural networks and written in Python. Keras is primarily used in conjunction with Tensorflow, as a front-end API that is extremely easy to use with Tensorflow as the powerful back-end. This combination of Keras and Tensorflow makes Python the premier language when dealing specifically with neural networks [29].

In addition to the neural network implementation is the lip detector module. The lip detector module is based off the *dlib* face detector. *Dlib* is an open source computer vision and machine learning library written in C++ with easy to use Python APIs. Dlib is one of the most widely used face detection softwares for two reasons: it is fully open source and is available for both commercial and academic use, and it is also one of the fastest implementations available since it is written completely in C++ [13].

All processing of video and image files will be handled with *OpenCV*. OpenCV is the premier open source computer vision library, and is called *cv2* within the Python environment. It is written in C and C++ and has implemented APIs in many languages included Python. It also has support for Keras and Tensorflow. It was released over 10 years ago and is the largest open source computer vision project in existence and has extensive support for many applications [2].

Other specific, but minor, software packages used during implementation will be mention as they are used. The main software packages used are the previously mentioned Numpy, Dlib, OpenCV, and Keras with Tensorflow.

3.6 Lip Detection

This section will discuss the actual implementation from a software standpoint for building the lip detection framework. This includes the lip detector module to be applied to each video, the framework for applying it to every sample and storing the result, and the framework for making the accompanying labels for each sample.

3.6.1 Lip Detector Module

The lip detector module is built primarily off the *dlib* face detection package as described previously. It will also make use of the *OpenCV* library for handling video files, *Numpy* for converting videos and images to usable arrays and storage of those arrays, and also *imutils*, an open source library developed by Adrian Rosebrock of Pyimagesearch, which is a series of OpenCV convenience functions [21].

The first step our module performs is to load dlib's predefind face detector. The face detector then requires a path to a shape predictor. The shape predictor is a '.dat' file provided directly from dlib. This is performed using the following code:

```
predictor_path = 'shape_predictor_68_face_landmarks.dat' # Define path
detector = dlib.get_frontal_face_detector() # Initialize detector
predictor = dlib.shape_predictor(predictor_path) # Load predictor
```

For this project we are using the 68 point facial landmark detector as can be seen in the first line of the code above. Dlib's face detector makes use of the HOG and SVM algorithms outlined previously in the background chapter. The 68 point detector can be seen in Figure 3.2.

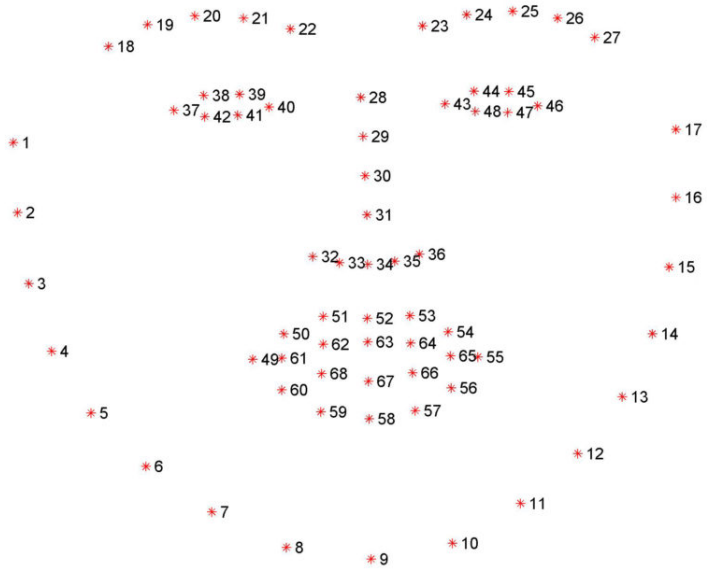

Figure 3.2: 68 Point Facial Landmark Detector Used by Dlib [17]

This detector is chosen over smaller, faster detectors, such as the 5 point predictor, due to the later need of isolating just the lip region. This is easily done by referring to a subset of the predictor points.

After the detector has been initialized the facial landmark predictor loaded, the video path is provided to an OpenCV and is then loaded into the environment. The video object is then opened and a while loop is used to iterate through each frame of the video. This is performed using the following code:

```
video = cv2.VideoCapture(video_path) # Read video in
while(video.isOpened()):
    ret, frame = video.read() # Capture frame
```

In addition to the actual video frame, the `video.read()` function returns a boolean value that is used to check if the frame exists. If the frame exists, it is then time to process the frame and apply the face detector. An example of all the frames in a sample video is included in Figure 3.3

Figure 3.3: Frames in a Sample Video

The first thing to do with each frame is convert it from color to grayscale. This is because since each frame will be stacked together to build the 3D data array the individual frames will need to be 2D. Stacking color frames would make a 4D array which requires more computing with minimal increases in performance. Each frame will also be resized to a standard size before being placed in the lip detector. The frame will then be input into the previously initialized lip detector. The code for this is as follows:

```
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) # Convert to grayscale
frame = cv2.resize(frame,(224,224)) # Resize image
faces = detector(frame,1) # Detect faces in image
```

The frames are all resized to be 224 pixels by 224 pixels. The 1 argument to the detector function means to upsample the image 1 time. Since we know each sample has exactly one speaker, if the detector detects more than 1 face or 0 faces then that video sample is discarded. This is acceptable due to the abundance of training data for every class.

The face object that is returned from the detector is then input into the shape predictor to identify the locations of the landmark. The output of the shape predictor is converted to a Numpy array for convenience, using the imutils library. This is shown in the following code.

```
shape = predictor(frame,face) # Identify facial landmarks
shape = face_utils.shape_to_np(shape) # Convert shape to numpy array
```

This shape array that contains locations of the facial landmarks is then used to locate the lips in the image. Since from Figure 3.2 the landmarks that correspond to the lips are 48-68, only that subset of the shape array is used. That subset is then converted to a numpy array and used as an input in OpenCV's bounding rectangle function. A margin of 10 pixels is included to grab excess information around the lips as just using the strict landmarks tends to be too tight of a crop and some information is lost. This bounding rectangle is then applied to the frame to grab the required lip data. Lastly, the lip data is resized to a common size for concatenation compatibility. The code for this is included below.

```
(x, y, w, h) = cv2.boundingRect(np.array([shape[48:68]])) # Find bounding box
margin = 10 # Extra pixels to include around lips
lips = frame[y-margin:y + h + margin, x-margin:x + w + margin] # Grab lips
lips = cv2.resize(lips,(100,60)) # Resize to standard size
```

The lip frames are resized to 60 by 100 pixels because this will match the first layer of the CNN that it will be input in to. A collection of lips from each frame is shown in Figure 3.4.

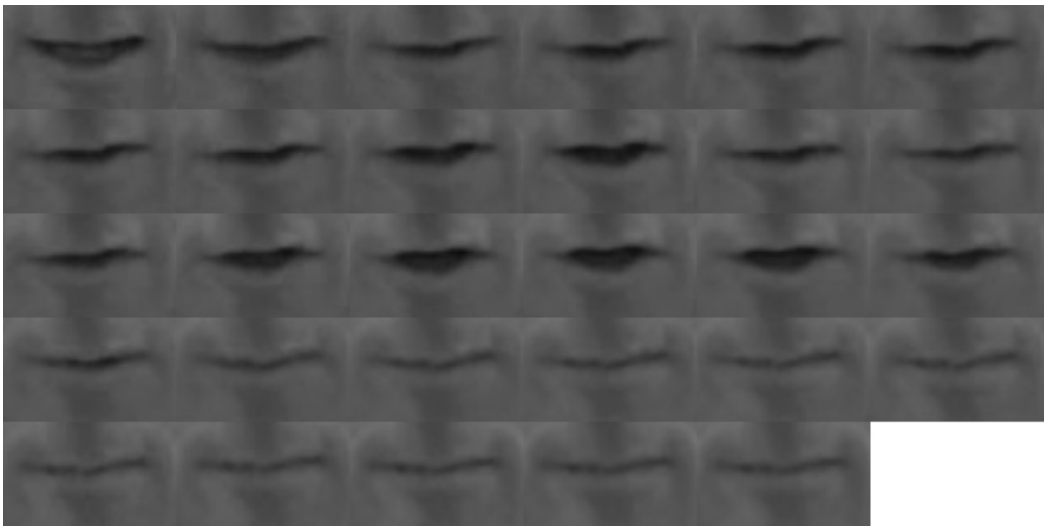

Figure 3.4: Detected Lips in Each Video Frame

The last step of the lip detector is to stack the lip frames together to create a 3D array. The lip frames are stacked together using Numpy's `dstack` function, which stacks 2D arrays to form a 3D array. Afterwards, the array is reshaped using Numpy's `moveaxis` function. The array needs to be reshaped in order to be compatible with the first layer of the CNN. The lip detector module accepts a path to a video file as the argument and returns the 3D array of lip frames as a numpy array. A software diagram for the lip detector module is included in the software diagrams appendix.

3.6.2 Processing Framework

The tree structure of the dataset is shown in Figure 3.5. Only the first two classes are shown, which are 'About' and 'Absolutely', the actual directory structure extends for all 500 classes. Inside each class are 3 folders: test, train, and val, and each folder contains the actual 'mp4' sample files and a corresponding 'txt' file that contains the metadata.

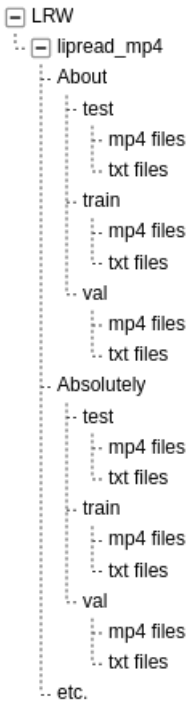

Figure 3.5: Directory Structure for LRW Dataset

The first function in the processing framework goes through each subfolder in the directory and applies the lip detector module to each mp4 file. If no face is detected in the 'mp4' file, the program simply skips the file, otherwise the output of the module is the previously mentioned numpy array.

This numpy array is then sliced using standard array slicing methods in Python. The array is sliced to grab the middle 11 frames of the video. This is done to isolate the

actual frames that correspond to the word being spoken. Since only part of the video is the target word, it makes sense to discard the extraneous information that does not pertain to the target word. 11 frames are chosen because it is the closest to the average frame length per word as seen in Table 3.4.

	Duration (s)	# of Frames
Max	0.8	20
Min	0.09	2.25
Average	0.424	10.6

Table 3.4: Duration of Words in LRW Dataset

The resulting numpy array is then stored using the command `np.save(location,numpy_array)`. The location for the stored array is the current directory that the input 'mp4' file is located. This is chosen for simplicity.

Next, it makes sense to move all '.npy' files to their own directory for future convenience when assembling the training, validation, and test sets. Each numpy folder is moved to a corresponding 'test','train', or 'val' folder based on which folder it is currently stored. The new folders are located on the same level as the 'lipread_mp4' directly, just below the main 'LRW' directory as seen in Figure 3.5.

3.7 Model Training

3.7.1 Label Maker

The first step of preparing the data to be used for training is making the associated labels to accompany the data. Since there are 500 classes in the dataset all the labels will be vectors of length 100. The labels are made corresponding to the class directories located in the 'lipread_mp4' directory. Each label is a vector of all 0's with

a 1 in the location that corresponds to the class. For example, the word 'About', which is the first class in the dataset will have a label vector of:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & ..... \end{bmatrix}^T$$

Furthermore the label vector for 'Absolutely', the second class in the dataset will have a label vector of:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & ..... \end{bmatrix}^T$$

Each label vector is stored as a '.npy' file using the previously mentioned numpy save command. They are all stored in a 'labels' directory located on the same level as 'lipread_mp4', just below the 'LRW' directory.

3.7.2 Assemble Dataset

Next, the dataset needs to be assembled according to Keras CNN compatibility. Keras models expect datasets to be one array where the dimension of the array is one larger than the dimension of each training sample. For example if the training dataset is 100 images of size $[50, 200]$ then the dimension of the training array will be a 3D array of size $[100, 50, 200]$.

For our project each input into the CNN will have size $[11, 60, 100, 1]$, the 4th dimension of 1 is needed for CNN compatibility, since convolution operations will expand that dimension it needs to be initialized before training. This leads to an array size of $[X, 11, 60, 100, 1]$, where X is the number of samples in each set. For the 100 class subset the test set contains 4919 samples, the training set contains 95991 samples, and

the validation set contains 4937 samples. Each numpy array is built using Python's built in 'append' function.

At this point it is important to normalize all the data to be between 0 and 1, this is a step that is usually performed when training neural networks in order to keep loss from destabilizing the model. Every value in the array is divided by 255.0 to convert to the proper range (since pixel values range from 0 to 255).

Dividing by 255.0 converts each value from a unsigned 8-bit integer (uint8) to a float. The standard float type in Python is a 64-bit value (float64). This means that each dataset now takes up 8 times as much memory. Using the standard float type for the 100 class dataset leads to a memory requirement of:

$$95991 \times 11 \times 60 \times 100 = 6,335,406,000 Floats \times 8 Bytes/Float = 47.2 GB$$

This is larger than the memory available as described in the hardware section. To solve this issue every float value is converted to a 16-bit value (float16) which still allows for 4 decimal points of precision.

3.7.3 Model Architecture and Initialization

From this point on, all model intialization, training, testing, etc. is done using the Keras API with Tensorflow-GPU backend. The 'model_init' function takes an input shape as its only parameter. It starts by initializing a model using the Keras Sequential API. Sequential models are linear stacks of layers and are the traditional choice when building CNNs.

The base architecture is based off the architecture described in [49] which the authors use for feature extraction. This base architecture will then be altered multiple times in an attempt to increase accuracy. Each of the individually altered architectures and their results will be presented in the Results chapter.

The base architecture consists of four 3D-convolutional layers, three 3D-max pooling layers, and three fully connected layers. The convolutional layers all use a kernel size of 2, which in 3D is a 2x2x2 cube, a stride size of 1, and ReLu activation. The number of filters start at 16 and increase by a factor of two in each subsequent layer (32, 64, 128). The max pooling layers use a pool size and strides of (1,2,2) with the 1 being in the temporal dimension.

After the last convolution layer is a flatten layer that converts the data to 1D for compatibility with fully connected layers. The first two fully connected layers use ReLu activation and have 1024 and 512 neurons respectively. The final fully connected layer uses softmax activation for classification and has 100 neurons to match the number of classes. After each of the first two fully connected layers a 0.5 dropout is applied in order to reduce overfitting.

The base model architecture is shown in Table 3.5 and the code to initialize the base CNN architecture is shown below that.

```
model = Sequential()

model.add(Convolution3D(input_shape=shape,filters=16,kernel_size=3,strides=1,activ
model.add(MaxPooling3D(pool_size=(1,3,3),strides=(1,2,2)))

model.add(Convolution3D(filters=32,kernel_size=3,strides=1,activation='relu'))
model.add(MaxPooling3D(pool_size=(1,3,3),strides=(1,2,2)))
```

Layer	Input Size	Output Size	Kernel	Stride
Conv1	11x60x100x1	9x58x98x16	3x3x3	1
Pool1	9x58x98x16	9x28x48x16	1x3x3	1x2x2
Conv2	9x28x48x16	7x26x46x32	3x3x3	1
Pool2	7x26x46x32	7x12x22x32	1x3x3	1x2x2
Conv3	7x12x22x32	5x10x20x64	3x3x3	1
Pool3	5x10x20x64	5x4x9x64	1x3x3	1x2x2
Conv4	5x4x9x64	3x2x7x128	3x3x3	1
Flatten	3x2x7x128	5376	-	-
FC1	5376	1024	-	-
FC2	1024	512	-	-
FC3	512	100	-	-

Table 3.5: Base Model Architecture

```
model.add(Convolution3D(filters=64,kernel_size=3,strides=1,activation='relu'))
model.add(MaxPooling3D(pool_size=(1,3,3),strides=(1,2,2)))

model.add(Convolution3D(filters=128,kernel_size=3,strides=1,activation='relu'))

model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(100, activation='softmax'))
```

3.7.4 Model Training Script

The model training script begins by calling the previously defined function to build the training and validation dataset and labels arrays. Next it uses the other previously defined function to initialize the Keras CNN model.

Next the model is compiled using the standard Keras 'model.compile' function. A stochastic gradient descent optimizer is used with learning rate equal to 0.02 and Nesterov momentum. The learning rate of 0.02 was the largest rate that was able to successfully train the models from scratch. Since the model architectures used are not well-known architectures with pretrained models available they must all be trained from scratch. A categorical cross entropy loss function was used as is standard for multi-class classification problems.

From here the model is trained using the 'model.fit' function with shuffle set to True. Shuffling the data randomizes the order of the samples presented to the model each training epoch. This is another parameter used to help reduce overfitting.

Model training is monitored by considering both training and validation accuracy. Models with viable architectures started to show sharp increases in accuracy after three epochs. Each model was trained until the validation accuracy didn't improve for five epochs.

Chapter 4

TESTING AND RESULTS

4.1 Face and Lip Detection

The face and lip detection program was applied to all video samples for the first 100 classes of the LRW dataset. Once a face was detected, the lips were easily extracted using the dlib shape predictor landmarks, so all videos that had detected faces had corresponding detected lips. Detecting faces on the other hand did produce some errors. There were two types of face detection errors: no faces detected and more than one face detected (since every video was only one speaker). Samples that had detection errors were discarded from the datasets. Even after discarding the samples with detection errors, there was still enough samples to successfully train the model.

The statistics of the face detection program are shown in Table 4.1. The correct classification rate of 98.62% resulted in enough samples needed to help reach model accuracy goals. The no face error resulted about 3 times as often as the multiple face error but both were relatively infrequent compared to the amount of correct classifications.

The numer of samples of each dataset after processing is shown in 4.2. Each sample was a 4D Numpy array of size (11,60,100,1). The shape of each dataset was then a

Type	Number	Percentage
Face Correctly Detected	105,847	98.62%
No Face Detected	1,112	1.04%
Multiple Faces Detected	373	0.35%

Table 4.1: Face Detection Statistics

5D array with shape (x,11,60,100,1) where X is the number of samples per dataset. Since the validation dataset is slightly larger than the test dataset it is used for testing. In practice, when the final trained model was tested on both the test and validation datasets the results were negligible. For example, for model #1 the final validation accuracy was 56.98% and test accuracy was 57.32% and for model #6 the final validation accuracy was 63.68% and test accuracy was 63.90%.

Dataset	Number of Samples
Training	95,991
Testing	4,919
Validation	4,937

Table 4.2: Samples per Dataset

4.2 Model Architectures and Results

4.2.1 Model #1

As mentioned in the implementation chapter, multiple different architectures were tested. Each architecture was a modified version of the base architecture presented in the implementation chapter. The base architecture will be considered Model #1 and is shown in Table 3.5.

This model had 6,373,124 parameters of which 5,506,048 were connections between the last convolution layer and the first fully connected layer. The output of the convolutional section of the network was a vector of dimension 5,376.

Model #1 showed promising results for it being the first architecture tested. It showed a peak validation accuracy of 58.44% which occurred after 12 epochs. Since this was the first model it was used as a test for the training script on multiple machines and training accuracy data from scratch was not recorded.

4.2.2 Model #2

Model #2 was a minor modification on the first model that consisted of adding one more pooling layer and convolutional layer. After the 4th convolutional layer (the last convolutional layer of the base architecture) another 3D max pooling layer was added, followed by a 5th 3D convolutional layer. In order to accommodate the dimensionality loss from the additional layers, the kernel sizes of both the pooling layers and convolutional layers were changed from 3 to 2. This also allowed us to investigate the effectiveness of kernels of size 2, as most research has been done on CNNs with kernel size 3. The architecture of Model #2 is shown in Table 4.7.

The

Layer	Input Size	Output Size	Kernel	Stride
Conv1	11x60x100x1	10x59x99x16	2x2x2	1
Pool1	10x59x99x16	10x29x49x16	1x2x2	1x2x2
Conv2	10x29x49x16	9x28x48x32	2x2x2	1
Pool2	9x28x48x32	9x14x24x32	1x2x2	1x2x2
Conv3	9x14x24x32	8x13x23x64	2x2x2	1
Pool3	8x13x23x64	8x6x11x64	1x2x2	1x2x2
Conv4	8x6x11x64	7x5x10x128	2x2x2	1
Pool4	7x5x10x128	7x2x5x128	1x2x2	1x2x2
Conv5	7x2x5x128	6x1x4x256	2x2x2	1
Flatten	6x1x4x256	6144	-	-
FC1	6144	1024	-	-
FC2	1024	512	-	-
FC3	512	100	-	-

Table 4.3: Model #2 Architecture

Adding these two layers greatly reduced the number of parameters in the network. The output of the convolutional section was now 6,144 (from 44,800) which greatly reduced the total parameters from 46,538,708 to 7,217,364. Despite the reduction in total parameters, Model #2 actually showed an increase in performance. It achieved

a peak validation accuracy of 64.84% which occurred after 22 epochs. The training and validation accuracy's after each epoch are shown in Figure 4.1.

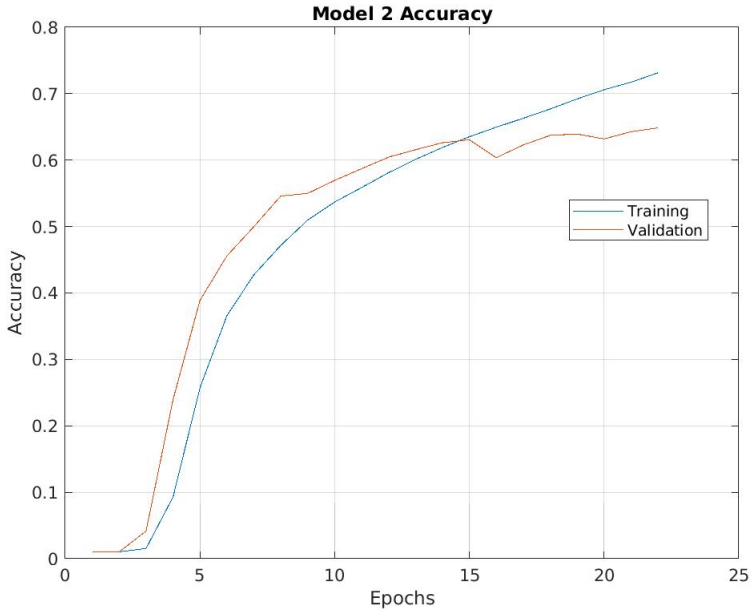

Figure 4.1: Model #2 Accuracy Results

4.2.3 Model #3

Model #3 used the most different architecture of any tested in this paper. It consisted of banks of two convolutional layers followed by one max pooling layer rathen than alternating convolutional and pooling layers. This architecture was inspired from the VGG architecture that has shown promising results in image classification tasks.

The architecture consists of two 3D convolutional layers followed by a max pooling layer, this is then repeated two more times. Then it is followed by a convolutional, max pooling, and convolutional layer similar to the structure used in the base architecture. This was the max amount of convolutions that could be performed based on the sample array size. This resulted in a convolutional section output of size 2,304 and 3,328,324 total parameters. The architecture of Model #3 is shown in Table 4.4.

Layer	Input Size	Output Size	Kernel	Stride
Conv1	11x60x100x1	10x59x99x16	2x2x2	1
Conv2	10x59x99x16	9x58x98x16	2x2x2	1
Pool1	9x58x98x16	9x29x49x16	1x2x2	1x2x2
Conv3	9x29x49x16	8x28x48x32	2x2x2	1
Conv4	8x28x48x32	7x27x47x32	2x2x2	1
Pool2	7x27x47x32	7x13x23x32	1x2x2	1x2x2
Conv5	7x13x23x32	6x12x22x64	2x2x2	1
Conv6	6x12x22x64	5x11x21x64	2x2x2	1
Pool3	5x11x21x64	5x5x10x64	1x2x2	1x2x2
Conv7	5x5x10x64	4x4x9x128	2x2x2	1
Pool4	4x4x9x128	4x2x4x128	1x2x2	1x2x2
Conv8	4x2x4x128	3x1x3x256	2x2x2	1
Flatten	3x1x3x256	2304	-	-
FC1	2304	1024	-	-
FC2	1024	512	-	-
FC3	512	100	-	-

Table 4.4: Model #3 Architecture

Model #3 had about half as many parameters as Model #2 but performed similarly. It achieved a peak validation accuracy of 61.09% after 30 epochs. The training and validation accuracy's after each epoch are shown in Figure 4.2.

4.2.4 Model #4

Model #4 uses the base architecture and adds two more convolutional layers and a max pooling after the end of the convolutions. It consists of three convolutional layers in a row each increasing in filter number. It starts with the 128 filter layer that is the last convolutional layer of the base network and then follows that with a 256 filter layer and then a 512 filter layer. The architecture of Model #4 is shown in Table 4.5.

These convolutions reduce the dimension of the convolutional section output to 10,240 and 12,460,756 total parameters. This network achieved a peak validation accuracy

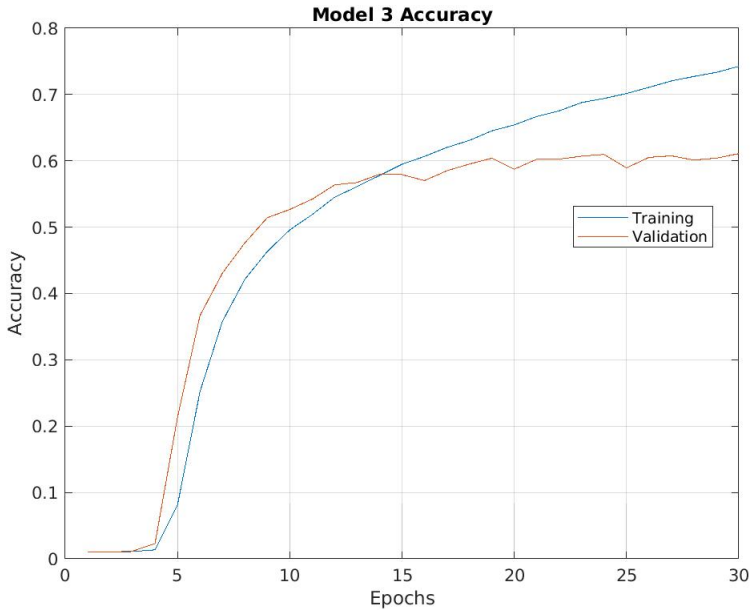

Figure 4.2: Model #3 Accuracy Results

of 63.93% which occurred after 19 epochs. The training and validation accuracy's after each epoch are shown in Figure 4.3.

4.2.5 Model #5

Model #5 uses a similar architecture to Model #3 in that it uses banks of two convolutional layers followed by a max pooling layer. This set of three layers is repeated four times, with each set increasing the number of filters in the convolutional layer. The architecture of Model #5 is shown in Table 4.6.

This architecture had the least amount of parameters with 2,410,692 and a convolutional output dimension of 1,536. It achieved a peak validation accuracy of 60.75% which occurred after 39 epochs. The training and validation accuracy's after each epoch are shown in Figure 4.4.

Layer	Input Size	Output Size	Kernel	Stride
Conv1	11x60x100x1	10x59x99x16	2x2x2	1
Pool1	10x59x99x16	10x29x49x16	1x2x2	1x2x2
Conv2	10x29x49x16	9x28x48x32	2x2x2	1
Pool2	9x28x48x32	9x14x24x32	1x2x2	1x2x2
Conv3	9x14x24x32	8x13x23x64	2x2x2	1
Pool3	8x13x23x64	8x6x11x64	1x2x2	1x2x2
Conv4	8x6x11x64	7x5x10x128	2x2x2	1
Conv5	7x5x10x128	6x4x9x256	2x2x2	1
Conv6	6x4x9x256	5x3x8x512	2x2x2	1
Pool4	5x3x8x512	5x1x4x512	1x2x2	1x2x2
Flatten	5x1x4x512	10240	-	-
FC1	10240	1024	-	-
FC2	1024	512	-	-
FC3	512	100	-	-

Table 4.5: Model #4 Architecture

4.2.6 Model #6

The last model test uses the same architecture as Model #2 but has double the number of filters in each convolutional layer. The architecture of Model #6 is shown in Table **??**. This model performed very similarly to Model #2 and achieved a peak validation accuracy of 64.49% which occurred after 24 epochs. This model had 14,553,924 total parameters, about double that of Model #2. The training and validation accuracy's after each epoch are shown in Figure 4.5.

4.3 Analysis of Results

Table 4.8 compares the results of all six architectures tested. Model #2 had the best performance with a validation accuracy of 64.84%. Since all the models were derived from the same base model they all exhibited similar performance.

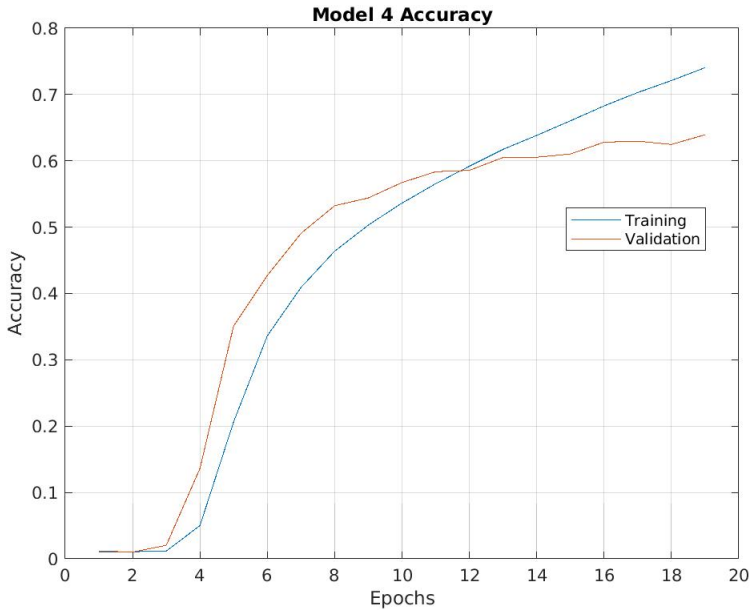

Figure 4.3: Model #4 Accuracy Results

When total parameters were increased the model reached peak validation accuracy in less epochs, demonstrating a strong inverse relationship. The only model to deviate from this pattern was Model #6, which actually took more epochs than the comparable Model #2 that had half as many parameters. Doubling the filter count in each layer for Model #6 vs Model #2 had a negligible effect on accuracy but at the cost of doubling the parameters.

Adding extra convolutional layers past 5 did little to help the accuracy of the model. Models #3 and #5 had the most convolutional layers at 8 which also resulted in the least amount of parameters. This is because the majority of CNN parameters are in the fully connected layers and by applying more convolutions the dimensionality of the convolutional section output is reduced.

Figures 4.6 shows the individual plots for epoch vs validation accuracy of each model on the same plot.

Layer	Input Size	Output Size	Kernel	Stride
Conv1	11x60x100x1	10x59x99x16	2x2x2	1
Conv2	10x59x99x16	9x58x98x16	2x2x2	1
Pool1	9x58x98x16	9x29x49x16	1x2x2	1x2x2
Conv3	9x29x49x16	8x28x48x32	2x2x2	1
Conv4	8x28x48x32	7x27x47x32	2x2x2	1
Pool2	7x27x47x32	7x13x23x32	1x2x2	1x2x2
Conv5	7x13x23x32	6x12x22x64	2x2x2	1
Conv6	6x12x22x64	5x11x21x64	2x2x2	1
Pool3	5x11x21x64	5x5x10x64	1x2x2	1x2x2
Conv7	5x5x10x64	4x4x9x128	2x2x2	1
Conv8	4x4x9x128	3x3x8x128	2x2x2	1
Pool4	3x3x8x128	3x1x4x128	1x2x2	1x2x2
Flatten	3x1x4x128	1536	-	-
FC1	1536	1024	-	-
FC2	1024	512	-	-
FC3	512	100	-	-

Table 4.6: Model #5 Architecture

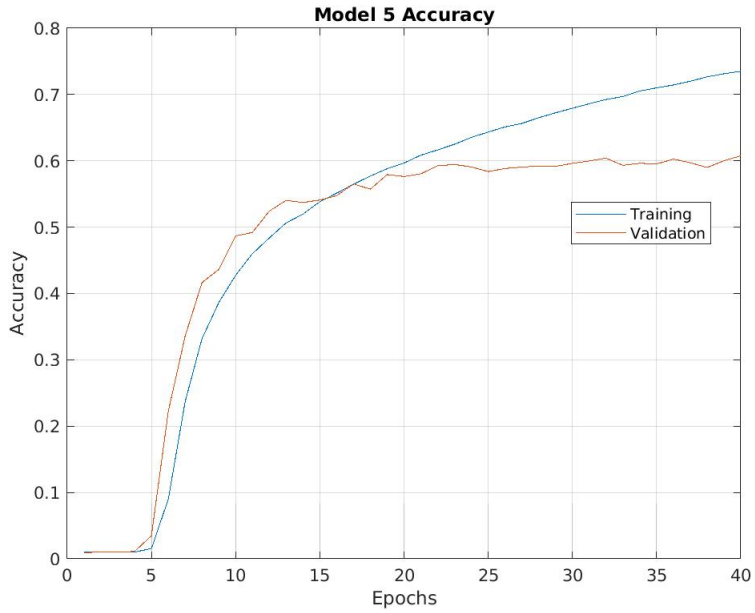

Figure 4.4: Model #5 Accuracy Results

Layer	Input Size	Output Size	Kernel	Stride
Conv1	11x60x100x1	10x59x99x32	2x2x2	1
Pool1	10x59x99x16	10x29x49x32	1x2x2	1x2x2
Conv2	10x29x49x16	9x28x48x64	2x2x2	1
Pool2	9x28x48x32	9x14x24x64	1x2x2	1x2x2
Conv3	9x14x24x32	8x13x23x128	2x2x2	1
Pool3	8x13x23x64	8x6x11x128	1x2x2	1x2x2
Conv4	8x6x11x64	7x5x10x256	2x2x2	1
Pool4	7x5x10x128	7x2x5x256	1x2x2	1x2x2
Conv5	7x2x5x128	6x1x4x512	2x2x2	1
Flatten	6x1x4x512	12288	-	-
FC1	12288	1024	-	-
FC2	1024	512	-	-
FC3	512	100	-	-

Table 4.7: Model #6 Architecture

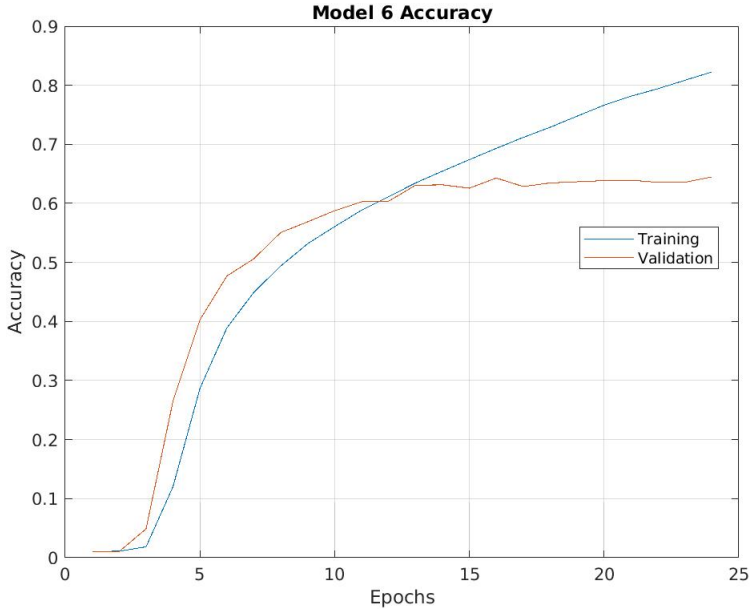

Figure 4.5: Model #6 Accuracy Results

Model #	Peak Accuracy	Epochs for Max Acc.	Total Parameters	Conv Layers
Model #1	58.44%	-	6,373,124	4
Model #2	64.84%	22	7,217,364	5
Model #3	61.09%	30	3,328,324	8
Model #4	63.93%	19	12,460,756	6
Model #5	60.75%	39	2,410,692	8
Model #6	64.49%	24	14,553,924	5

Table 4.8: Comparison of Different Architectures Performances

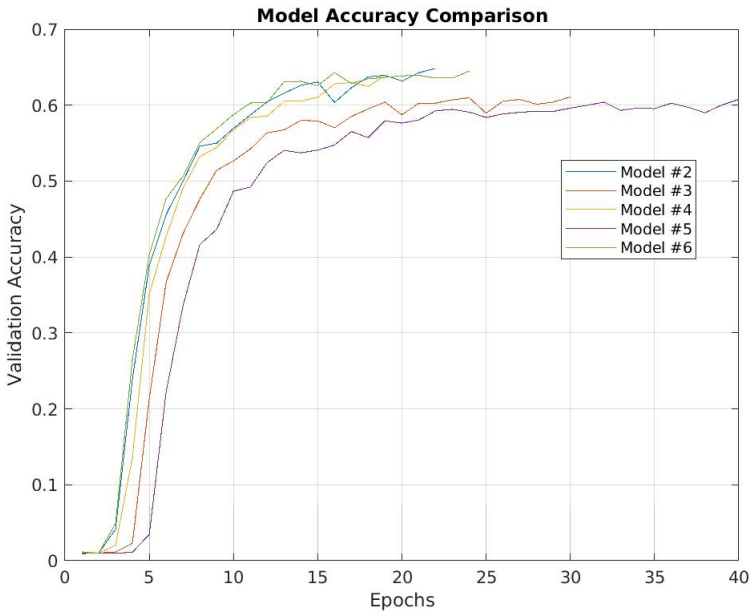

Figure 4.6: Model Accuracy Comparison

Chapter 5

CONCLUSION

5.1 Summary of Work

In this work we presented both a processing framework for extracting lip data from videos and a 3D-CNN architecture for performing visual speech recognition on a dataset of 100 similar words recorded in an uncontrolled environment. We showed that our model achieved 64.8% validation accuracy which is in line with other recent work done on this problem. Most notably our results are similar to those presented in [31], the only current paper to use 3D-CNNs for classification of the LRW dataset.

The results of the face detection program provided a processing framework that can achieve greater than 98% success rate in extracting lip frames from videos and outputting a 3D array of data. This processing framework will be available for use by the open-source community via Github. This framework excels in it's simplicity in that it takes only a path to a video file and returns an array of lip data. This framework is also easily adjusted to return any other face subregion from the 68-point dlib facial landmark set. For example, using landmarks 37-48 can be used to return a 3D data packet centered on the eyes, which could be useful for applications such as gaze detection.

The results of the model training and testing show that 3D-CNNs can be used for unsupervised extraction of temporal features. It is hard to make a direct comparison with the results presented in [31] due to the differences in datasets but our results still show that 3D-CNNs can be used to perform VSR. Furthermore, the authors of [31] used 25 frames from each video with each lip frame being of size 124x124 and in

our work we used an input size of 11 frames, with each frame of size 60x100, a 82% reduction.

The architecture presented in model #2 performed the best of all architectures tested. This result when compared to the performance of other architectures leads to the conclusion that 5 convolutional layers worked best for this classification task. It showed a 6% increase in accuracy compared to the 4 convolutional layer network and the performance of other networks with 6 or more convolutional layers did not outperform it. The results also showed that ending with a convolutional layer rather than with a pooling layer produced better results.

Though this could also be explained by the difference in parameters before the start of the fully connected layers. Models #2, #4, and #6 showed the 3 best accuracy's and they also had the most parameters before the start of the fully connected layers, but model #2 had the least of the three and had the best performance of the three so it is hard to say definitively if that was the reason.

We also showed with our architectures presented that using a kernel of size 2x2x2 can be successful in comparison with many works that use the more traditional kernel size of 3x3x3, which is what was used in both [31] and [49].

The results in Table 4.8 also show a few observations regarding total parameters. Models with more parameters reached peak validation accuracy faster in a nearly linear relationship, with Model #6 being the only data outlier. The best performances occurred with 5, 5, and 6 convolutional layers respectively leading to the conclusion that 5 or 6 is the most ideal number of convolutional layers, which can be further backed up by the results of [31] where all the architectures presented used 5 convoltuional layers.

5.2 Limitations

The main limitations of this work had to do with the time required to process each video and the memory required to store the dataset during model training. Also, the complex nature of CNN architectures leaves many possible architecture types untested.

Processing each video to extract and assemble the lip data took about 2 seconds on our machine. When considering the entire LRW dataset which consists of about 1050 videos per class, and 500 classes, this results in a processing time of about 12 days, when running continuously. The main reason for the relatively slow processing speed is that the face detection algorithm can not be easily sped up by the use of GPUs. Using the HOG+SVM face detection technique had the benefit of the simple implementation and easy adaption for lip detection, but has a massive drawback in speed when compared to a CNN based detector that can utilize a GPU. If designing a real-time application for VSR it would make sense to either redesign a processing framework based around CNN face detection or to write low level code that would allow the HOG+SVM technique to be run on a GPU. Even if the HOG+SVM technique could be run on a GPU, there is no guarantee that it would show substantial increases in speed as GPUs are not optimized for all operations (they are mainly used currently for neural network training among other things).

The memory constraint was caused by the size of the training dataset array. Even using the smallest float precision that python would allow the dataset still took up over 50% of the available memory. In order to train a model using the entire 500 word dataset either more memory would need to be available for use or the program would have to be rewritten in a more memory efficient language such as C++. Performing

classification on only the first 100 words was chosen both for this limitation and the one regarding processing time.

The last limitation has to do with the complex nature of CNNs and the many options present when designing them. Due to the multiple different layer types (convolution, pooling, fully connected, etc.) and parameters for each type (kernel size, stride, filter count, etc.) it is difficult to test a comprehensive set of models. To some degree choosing model architectures can be more of an art than an exact science due to the nearly limitless combinations of layer types and parameters.

5.3 Future Work

5.3.1 Accuracy Improvements

5.3.1.1 Model Architecture Changes

Due to the previously mentioned complexities when choosing a convolutional neural network architecture, there are many different tweaks that could be tested in order to attempt to increase the model accuracy. Of these possibilities include different pooling layers, convolutional layer parameters, and fully connected layer sizes.

In terms of pooling layers there are both different types of pooling layers that could be used as well as different parameters for max pooling layers. Other pooling options include average pooling layers and convolutional layers with larger strides. Average pooling layers are used less commonly than max pooling layers as they do a worse job at extracting the more important features like edges and corners [10]. There is also the option of using large stride convolutional layers which has been explored in [47]. This method was briefly attempted in this work but did not immediately yield

positive training results and was abandoned for the continued use of max pooling layers.

In terms of convolutional layers, there are many parameters that can be tweaked while still maintaining the overall structure of the network. This work used kernel sizes of two rather than the more traditional size of three in order to allow room for more convolutional layers in the network. One option could be to use a larger lip frame size than 60x100 which could allow for the same network architecture but with convolution kernel sizes of three or greater. Another option would be to use padding, which would allow for convolutions to be performed without a loss of dimensionality. Also, the effect of changing filter count was briefly tested in Model #6 but showed very similar results to Model #2. This was the only attempt at focusing on the changing of filter counts and only double the number of filters was tested.

Lastly, the number and size of the fully connected layers could easily be altered. Only the last layer size of 100 neurons must be kept the same due to the number of classes in the LRW subset being tested on. There is the option of either increasing or decreasing the number of neurons in the first two fully connected layers as well as adding another fully connected layer. Increasing the number of neurons increases the complexity of the network which traditionally leads to higher training accuracy but not always higher validation accuracy. Having too complex of a fully connected network can easily lead to overfitting and actually decrease validation accuracy. Still, it could be a worthwhile pursuit as this work never adjusted the size of the fully connected layers at all.

5.3.1.2 Image Preprocessing

A common technique for increasing the accuracy of image classification models is to apply different filters to images before model training. Filters such as Sobel, Gabor, and Difference of Gaussian have been shown to improve the results of face recognition models by increasing contrast near edges and other key areas [50].

Other preprocessing methods such as mean normalization could possibly help results. The only preprocessing down in this work was a simple normalization to scale pixel values between 0 and 1 by dividing each value by 255. This work focused more on model architectures rather than processing techniques and leaves a lot of possibilities for improvement for future researchers willing to test different processing techniques.

5.3.1.3 Included Frames

This work decided to use the middle 11 frames of each video to form the 3D arrays used to train the CNN models. The two other papers used for inspiration used 25 frames [31] and 9 frames [49], with each producing relatively decent results. This leaves the possibility that the optimal value of included frames is still yet to be determined and could become the focus of a future work.

The reasoning for the use of 11 frames in this work was to use the average number of frames per word but most words fall either below or above this value. Words with frame lengths less than 11 end up including extraneous information that is not related to the target word. On the other hand, words with frame lengths greater than 11 have a loss of information that could end up being vital to differing it from similar words. Especially words that are longer than 11 frames and also include plural words in the

same dataset (such as BENEFIT and BENEFITS), could cause high misclassification rates.

5.3.2 Application for Full Scale VSR

The problem of full scale visual speech recognition is much more complicated than the problem approached in this work. As mentioned in the introduction full scale VSR requires the use of statistical models such as LSTM to determine the most likely sequence of visemes. Since training a 3D-CNN to detect all 150,000+ words in the English language is not a realistic application, a different approach must be made.

This work shows that 3D-CNNs can be successful in extracting temporal features to use for speech recognition which leads to the idea that a 3D-CNN can be a valid choice for a viseme detector. Using this model in conjunction with a framework that breaks videos down into smaller segments can be a practical approach to improving VSR.

Since there are less than 15 visemes in the English language, there is a high probability that the 3D-CNN can detect visemes with higher than the 64% accuracy reported in this work. Using this model as a viseme detector which is then used as an input to a LSTM model has the possibility of improving upon the state-of-the-art for stand alone VSR. The challenging part of this concept would be the successful building of a framework that can extract segments of video that correspond to single visemes. Nevertheless this is an exciting possibility for improvement in a field that still has much room for growth.

5.3.3 Enhancement of Existing ASR Systems

The other main reason for VSR research is the effect of combining it with main stream ASR to increase accuracy. This is especially true for ASR in noisy environments such as in a restaurant or other public settings, which can be considered one of the main weaknesses of current audio-based ASR systems.

This results of this work show that 3D-CNNs have the potential to be state-of-the-art in VSR, which when used in conjunction with current ASR systems can increase accuracy in previously difficult environments. Since the need for ASR will always outweigh the need for stand alone VSR systems, this is the most likely application of the work shown here. Using the idea presented in the previous section on full scale VSR, this model can be used to improve current ASR systems. By combining audio data with video data, researchers have been able to improve the performance of audio-based ASR systems [43] [36].

A 3D-CNN would be especially good for ASR systems with limited number of words to classify. Examples of this would be classifying spoken numbers such as 1 through 10, or spoken letters A through Z. Our work showed that a 3D-CNN can perform classification on a dataset with 100 similar words, leading to the observation that it would also perform well in the two described scenarios which would have 10 and 26 classes respectively. Also for systems with a limited number of voice commands, around 100 or less, a 3D-CNN would be suitable.

BIBLIOGRAPHY

[1] The 44 phonemes in english. https://www.dyslexia-reading-well.com/44-phonemes-in-english.html. (Accessed on 09/10/2019).

[2] About. https://opencv.org/about/. (Accessed on 09/21/2019).

[3] About train, validation and test sets in machine learning. https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7. (Accessed on 09/18/2019).

[4] Activation functions in neural networks - towards data science. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. (Accessed on 09/17/2019).

[5] Avicar project: Audio-visual speech recognition at the university of illinois at urbana-champaign. http://www.isle.illinois.edu/sst/AVICAR/. (Accessed on 09/21/2019).

[6] A beginner's guide to neural networks and deep learning — skymind. https://skymind.ai/wiki/neural-network. (Accessed on 09/16/2019).

[7] Cal Poly Github. http://www.github.com/CalPoly.

[8] Chapter 2 : Svm (support vector machine) theory - machine learning 101 - medium. https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72. (Accessed on 09/11/2019).

[9] Chapter1.digital image representation.
http://pippin.gimp.org/image_processing/chap_dir.html. (Accessed on 09/10/2019).

[10] A comprehensive guide to convolutional neural networks the eli5 way.
https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. (Accessed on 09/17/2019).

[11] Cs 230 - convolutional neural networks cheatsheet.
https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks. (Accessed on 09/17/2019).

[12] Digital image processing in the name of god digital image processing lecture6: Color image processing m. ghelich oghli by: M. ghelich oghli - ppt download. https://slideplayer.com/slide/8752313/. (Accessed on 09/10/2019).

[13] dlib c++ library. http://dlib.net/. (Accessed on 09/21/2019).

[14] Face detection algorithms and techniques.
https://facedetection.com/algorithms/. (Accessed on 09/10/2019).

[15] Face recognition with deep learning - machine intelligence.
https://www.hackevolve.com/face-recognition-deep-learning/. (Accessed on 09/11/2019).

[16] Face recognition with python, in under 25 lines of code real python.
https://realpython.com/face-recognition-with-python/. (Accessed on 09/10/2019).

[17] Facial landmarks with dlib, opencv, and python - pyimagesearch. https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/. (Accessed on 09/23/2019).

[18] Gradient vectors chris mccormick. http://mccormickml.com/2013/05/07/gradient-vectors/. (Accessed on 09/11/2019).

[19] Hog person detector tutorial chris mccormick. https://mccormickml.com/2013/05/09/hog-person-detector-tutorial/. (Accessed on 09/11/2019).

[20] Loss functions ml cheatsheet documentation. https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html. (Accessed on 09/19/2019).

[21] My imutils package: A series of opencv convenience functions - pyimagesearch. https://www.pyimagesearch.com/2015/02/02/just-open-sourced-personal-imutils-package-series-opencv-convenience-functions/. (Accessed on 09/23/2019).

[22] Online latex equation editor - create, integrate and download. https://www.codecogs.com/latex/eqneditor.php. (Accessed on 09/13/2019).

[23] Phonemes and visemes - cryengine 3 manual - documentation. https://docs.cryengine.com/display/SDKDOC2/Phonemes+and+Visemes. (Accessed on 09/10/2019).

[24] Practical 9 scientific programming lab 1 version 0.1.0. https://sciprolab1.readthedocs.io/en/latest/practical9.html. (Accessed on 09/13/2019).

[25] Understanding neural networks - towards data science. https://towardsdatascience.com/understanding-neural-networks-19020b758230. (Accessed on 09/16/2019).

[26] What is an artificial neural network? here's everything you need to know — digital trends. https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/. (Accessed on 09/16/2019).

[27] What is automatic speech recognition (asr)? - definition from techopedia. https://www.techopedia.com/definition/6044/automatic-speech-recognition-asr. (Accessed on 10/03/2019).

[28] Which automatic transcription service is the most accurate? 2018. https://medium.com/descript/which-automatic-transcription-service-is-the-most-accurate-2018-2e859b23ed19. (Accessed on 10/03/2019).

[29] Why is python so good for ai, machine learning and deep learning? — netguru blog on python. https://www.netguru.com/blog/why-is-python-so-good-for-ai-machine-learning-and-deep-learning. (Accessed on 09/21/2019).

[30] J. S. Chung, A. Senior, O. Vinyals, and A. Zisserman. Lip reading sentences in the wild. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3444–3453, July 2017.

[31] J. S. Chung and A. Zisserman. Lip reading in the wild. In S.-H. Lai, V. Lepetit, K. Nishino, and Y. Sato, editors, *Computer Vision – ACCV 2016*, pages 87–103, Cham, 2017. Springer International Publishing.

[32] M. Cooke, J. Barker, S. Cunningham, and X. Shao. An audio-visual corpus for speech perception and automatic speech recognition. *The Journal of the Acoustical Society of America*, 120(5):2421–2424, 2006.

[33] A. Czyzewski, B. Kostek, P. Bratoszewski, J. Kotus, and M. Szykulski. An audio-visual corpus for multimodal automatic speech recognition. *Journal of Intelligent Information Systems*, 49(2):167–192, Oct 2017.

[34] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. In C. Schmid, S. Soatto, and C. Tomasi, editors, *International Conference on Computer Vision & Pattern Recognition (CVPR '05)*, volume 1, pages 886–893, San Diego, United States, June 2005. IEEE Computer Society.

[35] T. Ezzat and T. Poggio. Visual speech synthesis by morphing visemes. *International Journal of Computer Vision*, 38(1):45–57, 2000.

[36] R. Goecke. Audio-video automatic speech recognition: an example of improved performance through multimodal sensor input. 2006.

[37] S. S. Haykin. *Neural networks and learning machines*. Pearson Education, Upper Saddle River, NJ, third edition, 2009.

[38] G. Hinton, l. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29:82–97, 11 2012.

[39] E. Hjelmås and B. K. Low. Face detection: A survey. *Computer vision and image understanding*, 83(3):236–274, 2001.

[40] E. Hoffer, I. Hubara, and D. Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.

[41] V. Kazemi and J. Sullivan. One millisecond face alignment with an ensemble of regression trees. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1867–1874, 2014.

[42] S. E. Levinson. Continuously variable duration hidden markov models for automatic speech recognition. *Computer Speech & Language*, 1(1):29–45, 1986.

[43] G. Potamianos, C. Neti, J. Luettin, and I. Matthews. Audio-visual automatic speech recognition: An overview. *Issues in audio-visual speech processing*, 01 2004.

[44] L. R. Rabiner and B. H. Juang. Hidden markov models for speech recognition — strengths and limitations. In P. Laface and R. De Mori, editors, *Speech Recognition and Understanding*, pages 3–29, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[45] R. Rothe, M. Guillaumin, and L. Van Gool. Non-maximum suppression for object detection by passing messages between windows. In *Asian conference on computer vision*, pages 290–306. Springer, 2014.

[46] G. Saon and J. Chien. Large-vocabulary continuous speech recognition systems: A look at some recent advances. *IEEE Signal Processing Magazine*, 29(6):18–33, Nov 2012.

[47] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[48] J. A. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[49] A. Torfi, S. m. Iranmanesh, N. Nasrabadi, and J. Dawson. 3d convolutional neural networks for cross audio-visual matching recognition. *https://arxiv.org/abs/1706.05739*, PP, 09 2017.

[50] S. Wang, W. Li, Y. Wang, Y. Jiang, S. Jiang, and R. Zhao. An improved difference of gaussian filter in face recognition. *Journal of Multimedia*, 7, 12 2012.

[51] J. Zhang. Support vector machine. (Accessed on 09/13/2019).

APPENDICES

Appendix A

LIP DETECTOR MODULE CODE

```
##############################################################################################

# Lip Detector Module
# Matt Rochford

# This file defines the function to be used for lip detection in videos.
# Input is a path to an mp4 file (or other video type supported by cv2.VideoCapture).
# Output is a 3D data packet of frames of lip data stored as a numpy array.

##############################################################################################

# Import the necessary packages
from imutils import face_utils
import numpy as np
import cv2
import dlib
import time

##############################################################################################

# Define path to dlib predictor
predictor_path = 'shape_predictor_68_face_landmarks.dat'

##############################################################################################
```

```
# Lip detection function
def lip_detector(video_path):

    # Uncomment this line for use with timing block at bottom of function
    #t0 = time.time()

    # Initialize dlib's face detector (HOG-based) and create facial landmark predictor
    detector = dlib.get_frontal_face_detector()
    predictor = dlib.shape_predictor(predictor_path)

    # Read video in as mp4 file
    video = cv2.VideoCapture(video_path)

    # Check if video opened
    if (video.isOpened()==False):
        print('Error opening video file: ' + video_path)
        return # Return from function if video does not open

    lip_frames = [] # Initialize variable to store lip frames
    count = 0 # Initialize counter to track frames

    # Read video frame by frame
    while(video.isOpened()):

        ret, frame = video.read() # Capture frame
        if ret == True: # if frame exists

            count = count + 1 # Increment counter

            if 10 <= count <= 20: # If frame is in target time area then detect faces
            # This line can be removed to process all frames in video
            # Keeping this line only processes the middle 11 frames of video
```

```
# Convert frame to grayscale and resize to a standard size
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
frame = cv2.resize(frame,(224,224))

# detect face in the image
faces = detector(frame, 1)

if len(faces) > 1: # If more than one face detected print error and return
   #print('Error: Multiple faces detected in video')
   return 2 # Return error code 2
elif len(faces) == 0: # If no face detected print error and return
   #print('Error: No face detected in video')
   return 0 # Return error code 0

else: # If one face detected perform lip cropping
   for face in faces:

      # Determine facial landmarks for the face region and convert to a NumPy array
      shape = predictor(frame, face)
      shape = face_utils.shape_to_np(shape)

      # Extract the lip region as a separate image
      (x, y, w, h) = cv2.boundingRect(np.array([shape[48:68]]))
      margin = 10 # Extra pixels to include around lips
      lips = frame[y-margin:y + h + margin, x-margin:x + w + margin]
      lips = cv2.resize(lips,(100,60))

      # Create a stack of extracted frames
      if len(lip_frames) == 0:
         lip_frames = lips
      else:
         lip_frames = np.dstack(((lip_frames),(lips)))
```

```
        else: # If no frame left break from loop
            break

    # Release video object
    video.release()
    # Close any open windows
    cv2.destroyAllWindows()

    # Reshape array for CNN layer compatibility
    lip_frames = np.moveaxis(lip_frames,-1,0)

    # Uncomment this block to keep track of processing time per video
    #t1 = time.time()
    #total = t1-t0
    #print('Lip detection time: '+str(total))

    return lip_frames

#################################################################################################

# REFERENCES:

# Code for reading and editing video files is adapted from 'Learn OpenCV'
# https://www.learnopencv.com/read-write-and-display-a-video-using-opencv-cpp-python/

# Code for lip detection is based off Dlib library and an implementation from PyImageSearch
# https://www.pyimagesearch.com/2017/04/10/detect-eyes-nose-lips-jaw-dlib-opencv-python/
```

Appendix B

PROCESSING FRAMEWORK CODE

```
##############################################################################################

# Video Preprocessing Framework
# Matt Rochford

# This is the main function that implements the lip detector on each video
# file and stores the output numpy arrays. Also includes all helper functions.

##############################################################################################

# Import modules
from lip_detector import lip_detector
import os
import numpy as np
import time

##############################################################################################

# Define Paths
input_folder = '../LRW/lipread_mp4/' # Path to folders of each class
output_folder = '../LRW/' # Included for help when storing output numpy arrays

##############################################################################################

# This function forms a subset of the LRW dataset consisting of the first 100 classes.
# The output is a list of each class name to then be used in deciding which classes
# to process.
```

```
def data_subset():

    # Initialize counter f
    i = 0

    # Initialize array for storing class names
    subset_strings = []

    # Loop through each class directory e.g. ABOUT, ABSOLUTELY, ACCESS
    for directory in os.listdir(input_folder):

        # Only include the first 100 classes
        if i < 100:
            subset_strings.append(directory)

        # Increment counter
        i = i + 1

    # Return list of 100 class strings
    return subset_strings

##########################################################################################

# This function applies the lip detector function to all the files in the input folder
# and saves the output numpy files to the output folders.

def process_videos():

    # Form list of subset strings
    subset = data_subset()

    # Initialize variables for performance evaluation
```

```
mult_faces_tot = 0
no_faces_tot = 0
face_tot = 0
i = 1

# Loop over each file in entire dataset

# Loops over each word directory Ex. ABOUT, ABSOLUTELY, ACCESS, etc.
for directory in os.listdir(input_folder):

    # Start timer to keep track of processing time per class
    t0 = time.time()

    print('Processing '+directory+' class #'+str(i))

    i = i + 1

    # Initialize variables for per class evaluation
    mult_faces = 0
    no_faces = 0
    face = 0

    # If class is in the 100-class subset
    if directory in subset:

        # Loops over each sub directory, test train val
        for sub_directory in os.listdir(input_folder+directory):

            # Loops over each file in final directory
            for file in os.listdir(input_folder+directory+'/'+sub_directory):

                # Only use mp4 files
                if file.endswith('.mp4'):
```

```
# Generate label for output data file
label = os.path.splitext(file)[0] # Grab file base name
label = label + '.npy'

# Check if file has already been processed and update counter
if os.path.exists(output_folder+sub_directory+'/'+label):
    face = face + 1

# Otherwise process file
else:
    # Form full input path using directory and individual file
    path = input_folder+directory+'/'+sub_directory+'/'

    # Perform lip detection and return data array
    data_array = lip_detector(path+file)

    # If face detection error, keep track for evaluation
    if isinstance(data_array,int):
        if data_array == 2:
            mult_faces = mult_faces + 1
        elif data_array == 0:
            no_faces = no_faces + 1

    # If face was detected successfully
    else:

        # Update counter
        face = face + 1

        # Define path to output folder
        path = output_folder+sub_directory+'/'
```

```
                                # Save output numpy array to output folder
                                np.save(path+label,data_array)

            # Update total evaluation stats with class stats
            mult_faces_tot = mult_faces_tot + mult_faces
            no_faces_tot = no_faces_tot + no_faces
            face_tot = face_tot + face

            # Print detection rate for each class
            print('Correctly detected faces in '+directory+': '+str(face))
            print('No faces detected in '+directory+': '+str(no_faces))
            print('Multiple faces detected in '+directory+': '+str(mult_faces))

            # Reset per class counters
            mult_faces = 0
            no_faces = 0
            face = 0

            # Print class processing time
            t1 = time.time()
            total = t1-t0
            print('Processing time: '+str(total))

    # Print total detection rate
    print('Correctly detected faces total: '+str(face_tot))
    print('No faces detected total: '+str(no_faces_tot))
    print('Multiple faces detected total: '+str(mult_faces_tot))


##############################################################################################

# This function is used to calculate max min and average word duration for help with
# training. Average word duration is used to determine amount of frames to include for
```

```
# training.

def duration():

    # Initialize array to store word durations
    duration = []

    # Loop over each file in entire dataset

    # Loops over each word directory Ex. ABOUT, ABSOLUTELY, ACCESS, etc.
    for directory in os.listdir(input_folder):

        # Loops over each sub directory, test train val
        for sub_directory in os.listdir(input_folder+directory):

            # Loops over each file in final directory
            for file in os.listdir(input_folder+directory+'/'+sub_directory):

                # Only read txt files
                if file.endswith('.txt'):

                    data = [] # Initialize dummy variable to store text data
                    path = input_folder+directory+'/'+sub_directory+'/'+file # Define path to fi

                    with open(path,'rt') as myfile:

                        for line in myfile:
                            data.append(line) # Append lines to dummy variable

                    time = float(data[-1].split()[1]) # Grab duration value
                    duration.append(time) # Add duration to total array

    print('Mean, max, and min word durations:')
```

```
    print(np.mean(duration))
    print(np.max(duration))
    print(np.min(duration))

##########################################################################################

if __name__ == "__main__":
    #duration()
    process_videos()
```

Appendix C

MODEL TRAINING AND TESTING CODE

```
#############################################################################################

# 3D CNN
# Matt Rochford

# This file contains functions used for loading and assembling datasets, and initializing,
# training, and testing the 3D CNN model used for visual speech recognition.

#############################################################################################

# Ignore numpy and tensorflow compatibility warnings
import warnings
warnings.simplefilter(action="ignore", category=FutureWarning)

# Import tensorflow and ignore some errors
import os
import tensorflow as tf
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)


# Import other modules
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, load_model
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution3D, MaxPooling3D
from keras import optimizers
import numpy as np
```

```
#################################################################################################

# Define path to datasets
input_folder = '../LRW/'

#################################################################################################

# This function forms a subset of the LRW dataset consisting of the first 100 classes.
# The output is a list of each class name to then be used in deciding which classes
# to process.

def data_subset():

    # Initialize counter f
    i = 0

    # Initialize array for storing class names
    subset_strings = []

    # Loop through each class directory e.g. ABOUT, ABSOLUTELY, ACCESS
    for directory in os.listdir(input_folder):

        # Only include the first 100 classes
        if i < 100:
            subset_strings.append(directory)

        # Increment counter
        i = i + 1

    # Return list of 100 class strings
    return subset_strings
```

```
#################################################################################################

# This function create labels for each class and saves them as .npy files to the proper
# directory.

def label_maker():

    # Make string of classes to be used in dataset
    subset = data_subset()

    # Initialize counter for label making
    i = 0

    folder = input_folder+'lipread_mp4/'

    # Loops over each word directory Ex. ABOUT, ABSOLUTELY, ACCESS, etc.
    for directory in subset:

        # Hard code label of size 100 for each word in dataset subset
        # Change to 500 if using entire LRW dataset
        label = np.zeros((100), dtype=int)

        # Assign value of 1 at proper location
        label[i] = 1

        # Create name for label file
        name = directory+'_label.npy'

        # Create path to labels folder
        path = input_folder+'labels/'

        # Save numpy file to labels folder
        np.save(path+name,label)
```

```
        # Increment counter
        i = i + 1

##############################################################################################

# This function builds a 4D data array to use for model training. Key is which dataset to be
# assembled: test, train, or val. The output is one array of the entire dataset with ordered
# labels.

def build_4D_arrays_and_labels(key):

    # Generate string of classes for data subset
    subset = data_subset()
    i = 0

    # Initialize arrays for data and labels
    dataset = []
    labels = []

    # Define path to proper dataset folder: test,train,val
    path = input_folder+key+'/'

    # For each numpy array file from lip detector output
    for file in os.listdir(path):

        name = file.split('_')
        # Only add files in data subset
        if name[0] in subset:

            # Load numpy array
            data = np.load(path+file)
```

```
            # Grab correct frames if full video was processed earlier
            if data.shape[0] != 11:
                data = data[9:20,:,:]

            # Add 4th dimension to numpy array for CNN layer compatibility
            data = np.expand_dims(data,axis=-1)
            # Normalize training data to 0~1 range
            data = data / 255.0
            # Convert to shortest float for memory allocation demands
            data = np.float16(data)

            # Grab file base word
            word = file.split('_')[0]
            # Make label file string
            label_file = word+'_label.npy'
            # Load label array
            label = np.load(input_folder+'labels/'+label_file)

            # Append data to list
            dataset.append(data)
            labels.append(label)

    # Convert lists to numpy arrays
    dataset = np.stack(dataset)
    labels = np.stack(labels)

    # Print output stats
    print(key+' dataset size and label size')
    print(dataset.shape)
    print(labels.shape)

    # Save numpy arrays to save processing time when no new data is added
    np.save(key+'_data.npy',dataset)
```

```
    np.save(key+'_labels.npy',labels)

    return dataset, labels

##############################################################################################

# This function is used to load a dataset that was already assembled using the
# 'build_4D_arrays_and_labels' function. Used to save time when no changes to dataset
# have been made.

def load_dataset(key):
    dataset = np.load(key+'_data.npy')
    labels = np.load(key+'_labels.npy')
    return dataset,labels

##############################################################################################

# Function to initialize 3D CNN
# Only input to function is the shape of the input numpy array used for training.
# Formatted as a 3 item tuple, which is also the output of object.shape.
# Returns the tensorfow model object and prints the model summary.

# This current architecture is the one that produced the best results with a
# validation accuracy of 63~64%

def model_init(shape):

    print('Initializing CNN Model using Keras')

    model = Sequential()

    model.add(Convolution3D(input_shape=shape,filters=16,kernel_size=2,strides=1,activation='relu
    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))
```

```
    model.add(Convolution3D(filters=32,kernel_size=2,strides=1,activation='relu'))
    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))

    model.add(Convolution3D(filters=64,kernel_size=2,strides=1,activation='relu'))
    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))

    model.add(Convolution3D(filters=128,kernel_size=2,strides=1,activation='relu'))
    model.add(MaxPooling3D(pool_size=(1,2,2),strides=(1,2,2)))

    model.add(Convolution3D(filters=256,kernel_size=2,strides=1,activation='relu'))

    model.add(Flatten())
    model.add(Dense(1024, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(100, activation='softmax'))

    print(model.summary())

    return model

#############################################################################################

# Main function for training the keras model. This function creates training labels,
# calls array building functions, initializes model, and then trains it.

def train_model():

    # Make training labels
    label_maker()
```

```
# Build training dataset
print('Building training dataset')
#train_data,train_labels = build_4D_arrays_and_labels('train')
train_data,train_labels = load_dataset('train')

# Build validation dataset
print('Building validation dataset')
#val_data,val_labels = build_4D_arrays_and_labels('val')
val_data,val_labels = load_dataset('val')

# Initialize Keras model
model = model_init(train_data[1].shape)

# Load Keras model (use if you want to continue training on an already made model)
#model = load_model("model.h5")

# Nesterov momentum increases learning rate throughout training
sgd = optimizers.SGD(lr=0.02,nesterov=True)

# Compile using categorical cross entropy loss function
model.compile(optimizer=sgd,loss='categorical_crossentropy',metrics=['accuracy'])

# Train model using keras fit
print('Training Keras Model')

# Use for loop for training to help save weights after each epoch
for i in range(20):

    # Train model
    model.fit(train_data,train_labels,epochs=5,validation_data=(val_data,val_labels),shuffle=

    # Save model after each epoch
    model.save("model.h5")
```

```
#################################################################################################

# This function is used for testing the final performance of the trained model.

def test_model():

    # Build testing dataset
    print('Building testing dataset')
    #test_data,test_labels = build_4D_arrays_and_labels('test')
    test_data,test_labels = load_dataset('test')

    # Load model
    model = load_model("model.h5")

    # Evaluate model with test dataset
    [loss,accuracy] = model.evaluate(test_data,test_labels)
    accuracy = accuracy*100

    # Print test results
    print('Testing loss: '+str(loss))
    print('Testing accuracy: '+str(accuracy))



#################################################################################################

if __name__ == '__main__':
    train_model()
    #test_model()
    #model = model_init((11,60,100,1))
```

Appendix D

SOFTWARE DIAGRAMS

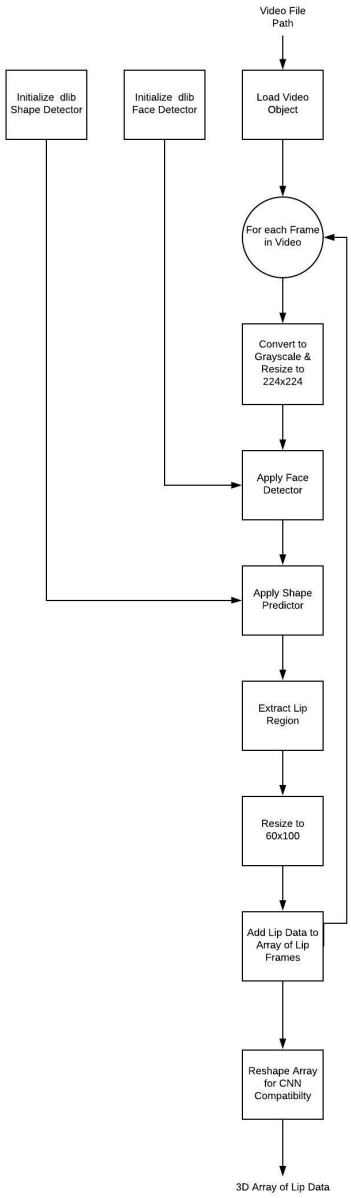

Figure D.1: Software Diagram for Lip Detector Module