Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Deep Lipreading

Author:
Michail-Christos Doukas

Supervisor:
Stefanos P. Zafeiriou

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science / Machine Learning of Imperial College London

September 2017

Abstract

Visual speech recognition, also known as lipreading, is the task of recognizing what is being said solely depending on the lip movements of the speaker. The recent advancements in machine learning and more specifically in deep learning have revolutionized automated lipreading. A significant amount of research in the area of lipreading has been focused on building deep neural networks that decode text using visual information from the mouth region. There are two dominant approaches for dealing with the lipreading problem. The first one considers visual speech recognition as a word or phrase classification task. The lipreading networks that follow this approach process video samples where a single word or short phrase is spoken and predict a word or phrase label from the vocabulary of the dataset. Recently, another approach which faces the lipreading problem by predicting character sequences instead of word labels was developed. These deep networks have the ability to perform text prediction even for video samples which include full sentences.

The purpose of this work is to describe the process of designing, implementing training and evaluating the performance of two deep networks for different visual speech recognition. First, we present a word classification network which learns to predict word labels for short video clips where speakers utter single words. The key components of this system is a 3D and a 2D Convolutional neural network and a Bidirectional Long short-term memory. Subsequently, we use this neural network as a feature extractor in the front end, while we employ an attention-based Long short-term memory transducer for character decoding at the back end of the network. The entire character decoding network learns to translate videos of mouth motion to character sequences. Both networks are trained end-to-end and evaluated using the Lip Reading in the Wild (LRW) dataset, which contains approximately five hundred thousand video samples of speakers uttering five hundred different words. The word classification network is trained in three stages, where the number of target word classes is increased from five, to fifty and five hundred in the last stage. Next, we execute the training process for the character decoding network, starting with the learned parameters acquired from the first network and using scheduled sampling. The two proposed networks yield a word accuracy of 53.2% and 41.3% respectively on the test set of LRW dataset.

Contents

Chapter 1

Introduction

1.1 The Lipreading Problem

Lipreading is the task of understanding speech by analyzing the movement of lips. Alternatively, it could be described as the process of decoding text from visual information generated by the speaker's mouth movement. The task of lipreading relies also on information provided by the context and knowledge of the language. Lipreading, also known as visual speech recognition, is a challenging task for humans, especially in the absence of context. Several seemingly identical lip movements can produce different words, therefore lipreading is an inherently ambiguous problem in the word level. Even professional lipreaders achieve low accuracy in word prediction for datasets with only a few words.

Automated lipreading has been a topic of interest for many years. A machine that can read lip movement has great practicality in numerous applications such as: automated lipreading of speakers with damaged vocal tracts, biometric person identification, multi-talker simultaneous speech decoding, silent-movie processing and improvement of audio-visual speech recognition in general. The advancements in machine learning made automated lipreading possible. However, many attempts that employed traditional probabilistic models did not achieve the anticipated results. Most of these lipreading methods were exclusively used to enchant the performance of audio-visual speech recognition systems in case of low quality audio data.

Lipreading and audio-visual speech recognition in general was revolutionized by deep learning and the availability of large datasets for training the deep neural networks. Lipreading is an inherently supervised problem in machine learning and more specifically a classification task. Most existing deep visual recognition systems have approached lipreading as a word classification task or a character sequence prediction problem. In the first case, a lipreading network receives a video where a single word is spoken and predicts a word label from the vocabulary of the dataset. In the second case, the input video may contain a full sentence (multiple words) and a deep neural network outputs a sequence of characters, which is the predicted text given the input sentence. This type of network performs classification in the character level. The two different approaches to the lipreading problem are depicted in Figure 1.1.

Obtaining inspiration from existing deep lipreading networks, this project aims to describe the process of designing, implementing and training two different deep lipreading networks

and finally evaluating their predictive performance. First, a neural network that performs word classification as shown in the left part of Figure 1.1 is proposed. Then, we propose a second neural network that decodes a sequence of characters from the input video sample, as shown in the right part of Figure 1.1. However, instead of training the second network on videos with full sentences, we use single word videos in a similar way to the first network. Both neural networks are trained and evaluated on the Lip Reading in the Wild (LRW) dataset, described in Chung and Zisserman (2016a), which is a dataset with a vocabulary of five hundred words, consisting of short video clips from BBC TV broadcasts.

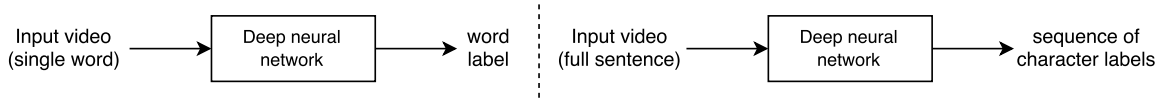

Figure 1.1: A word classification network on the left and a character decoding network on the right.

1.2 Building two Lipreading Networks

This project describes the entire process of designing, implementing, training and evaluating the performance of two different neural networks. Our goal is to train these networks on a lipreading dataset, so that they can operate as lipreading systems. More specifically, these networks should operate as a system with a video input and an text output. When a video from the LRW dataset with a speaker uttering a single word is presented in the input, the neural networks should produce a text prediction in the output. This text should ideally be the spoken word in the input video. It is important to note that we utilize only visual data, the lip movement, and therefore no audio signals are involved in the prediction process.

Considering that all video samples in the LRW dataset are labeled with the word which is spoken throughout, our first thought would be to build a classifier, where the target classes will coincide with the discrete words in the vocabulary of the dataset. Following this thought, we match each one of the five hundred words in the vocabulary of the LRW dataset with a class label and treat lipreading as a word classification problem. We solve the lipreading problem by designing a word classification neural network, which consists of an 3D Convolutional neural network (CNN) followed by multiple parallel 2D Convolutional neural networks with shared weights and a Bidirectional Long short-term memory (BiLSTM) with three hidden layers. At the back end of the network multiple fully connected layers with shared weights are followed by Softmax units in order to perform the classification task. First, the 3D CNN models the spatial and short term temporal dynamics of the lip movements in the input video. The output of the 3D CNN is then unstacked in the time dimension and transfered to a 2D CNN, which is applied in each time step and spatial feature vectors are extracted for each step. Next, this sequence of feature vectors is processed by the BiLSTM which captures both short and long temporal dynamics of the feature sequence. At each time step, the output vector of the BiLSTM is linearly transformed by a fully connected layer and Softmax is employed to obtain a distribution on the five hundred word classes. All distributions produced in the different BiLSTM output steps are combined to form a final one that determines the predicted word. The concepts of 3D and 2D Convolutional neural networks and Bidirectional Long short-term memories are presented in more detail in Section 2.1.

Another way to perceive lipreading would be as a task of predicting a sequence of character

labels instead of word labels. In this case, the character sequence forms the predicted word, which should ideally match the uttered word in the input video. For this purpose, we use the front end of the word classification network as a feature extractor (encoder) and combine it we a character decoder to form a character decoding network. The decoder is made up of a Long short-term memory transducer followed by an attention mechanism and a Multilayer perceptron (MLP) with one hidden layer with a Softmax unit after the output layer. At each output time step the decoder produces a distribution over characters that determines the current character output in the sequence. The attention mechanism is a network which learns to assign weights in the feature vectors of different time steps produced by the BiLSTM in the encoder. Then, the weighted summation of these feature vectors constitutes a context vector which is transfered to the MLP as input. Moreover, the context vector along with the predicted character of the current output time step are transfered to the LSTM transducer. The LSTM network models the dependency of each character with all previous characters in the sequence. The output of the LSTM at each time step is sent to the attention mechanism and contributes to the generation of the previously mentioned weights. When the decoder has executed all output time steps and has produced the corresponding character distributions, one in each step, the network uses them to predict a word. The attention-based LSTM transducer is described in more detail in Section 2.1

The two aforementioned networks were inspired by the lipreading networks presented in Chung et al. (2016) and Stafylakis and Tzimiropoulos (2017) respectively. Our word classification network, which is also used as an encoder in the second network, was inspired by Stafylakis and Tzimiropoulos (2017). In their lipreading system, they used a 3D CNN in the front end and a BiLSTM in the back, in the same way we do. However, they employed a Residual network for spatial feature extraction in the middle. Moreover, in their work Chung et al. (2016) proposed a audio-visual recognition system, whose visual part can operate independently as a lipreading network for character sequence prediction. Their network consists of an encoder and a decoder. We use the same decoder architecture they have proposed in order to build our character decoding neural network.

After determining their structure, the two neural networks are implemented in Python using Tensorflow (Abadi et al. (2015)). Tensorflow is a commonly used library for implementing neural networks in an efficient and robust way, since the training and evaluation process is executed on GPUs. The two networks are trained on the LRW training set and their predictive performance is evaluated on the LRW test set.

1.3 Roadmap

Having discussed the lipreading problem and presented an overview of the two proposed lipreading networks, in the next chapters we deliberate the entire process of designing, implementing, training and evaluating their performance in detail.

In the first half of Chapter 2 we present an overview of some popular and commonly used neural networks. More specifically, the Multilayer perceptrons, the Convolutional neural networks and the Long short-term memories. Moreover, we introduce the attention-based LSTM transducer network, which has been successfully used in speech and visual recognition. All these networks form building blocks of the two proposed lipreading networks and understanding their functionality is an important first step. In the second half of Chapter 2, we provide a summary on existing lipreading systems, from both the pre-deep learning and deep

learning era. We concentrate more on deep neural networks for visual speech recognition, since they are more relevant with the purpose of this project.

The Lip Reading in the Wild (LRW) dataset is discussed in Chapter 3. This is a large dataset from BBC TV with half a million videos in the training set. Each video displays a speaker uttering a single word. The LRW dataset, which was generated by Chung and Zisserman (2016a), is used to train and evaluate the performance of both proposed networks.

In Chapter 4 we present the architecture of the two proposed networks for visual speech recognition. We describe the building blocks of the word classification and the character decoding network and the way they are connected and interact with each other. After presenting and explaining the two neural network models, we direct discussion to their implementation.

Chapter 5 discusses the implementation of the word classification and character decoding network in Python. We present each of the two lipreading networks as a system with a training and an evaluation operation. The lipreading systems consist of two programs, one for training and one for evaluation, which are both implemented in Python with the Tensorflow library. These programs are made up of two Python Processes. The first Process generates batches of samples from the dataset and the second one performs the training or evaluation operation.

After implementing the two lipreading systems in Python, we execute the training operation for each one. In Chapter 6, first we discuss Stochastic gradient descent with L2 regularization and momentum, which is the optimization method used for training, and then we describe the training procedure followed for the two networks.

Subsequently, the predictive performance of the two networks is evaluated on the test set of the LRW dataset and the results of the evaluation operation are discussed in Chapter 7.

In Chapter 8 we discuss the challenges that occurred during the implementation process of the two networks. Moreover, we review the results of their predictive performance. Finally, we point out limitations of the proposed networks and possible ways to overcome them.

Chapter 2

Background and Related Work

2.1 Deep Learning Preliminaries

Lipreading is the point where the speech recognition and computer vision fields meet, and since deep learning advances have greatly affected both fields, lipreading was revolutionized by deep learning. Therefore, it would be useful to summarize fundamental deep learning concepts, which constitute the building blocks of various existing lipreading neural networks.

Deep learning is a part of machine learning and includes both supervised and unsupervised techniques. It uses a sequence of connected non-linear units, known as layers, for feature transformation and extraction. Deep learning algorithms learn multiple levels of data abstraction and representation. Each layer corresponds to a different level of abstraction and higher level features are extracted from lower. Deep learning provides a very powerful framework for supervised learning and classification more specifically. By stacking many layers together, with many units in each layer, a deep network can model non-linear functions and recognize very complicated data patterns in its input. Parametric function approximation is the core idea behind deep learning methods and the training process involves learning parameters which best model the underlying function of the problem.

In the next three subsections we describe some of the most known and commonly used neural networks which have been also successfully used in many lipreading systems: Multilayer perceptrons (MLPs), Convolutional neural networks (CNNs) and Long short-term memories (LSTMs). These algorithms are presented thoroughly in Goodfellow et al. (2016) and most of the concepts discussed below are obtained from this book. Finally, in subsection 2.1.4 we study the concept of attention mechanisms and LSTM transducers presented in Bahdanau et al. (2014a), Chan et al. (2015a) and Chung et al. (2016).

2.1.1 Multilayer Perceptrons (MLPs)

Multilayer perceptrons (MLPs), also known as Feedforward deep networks consist of three layers at least. There is one input layer $\boldsymbol{x}$, one (or more) hidden layer(s) $\boldsymbol{h}$ and one output layer $\boldsymbol{y}$. In the simple case where there is only one hidden layer, the deep network is called shallow or vanilla MLP. Then, we have the network's output defined as

$$\boldsymbol{y} = \boldsymbol{V}\boldsymbol{h} + \boldsymbol{c}, \quad \boldsymbol{y} \in \mathbb{R}^m, \boldsymbol{h} \in \mathbb{R}^k, \boldsymbol{V} \in \mathbb{R}^{m \times k}, \boldsymbol{c} \in \mathbb{R}^m$$

where

$$\boldsymbol{h} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}), \quad \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{W} \in \mathbb{R}^{k \times n}, \boldsymbol{b} \in \mathbb{R}^k$$

is the hidden layer.

The matrices $\boldsymbol{W}, \boldsymbol{V}$ are the weights and the vectors $\boldsymbol{b}, \boldsymbol{c}$ are the biases. The sigmoid function is defined as

$$\sigma(z) = \frac{1}{1 + \exp(-z)}, \quad z \in \mathbb{R}$$

and can be extended to be applied on vectors as well (element-wise). In general this function is called the activation function and could be another non-linear function such as the

- Rectified linear unit (ReLU): $r(z) = \max(0, z)$,
- Hyperbolic tangent: $a(z) = \tanh(z)$.

The hidden layers of an MLP can be considered as features extracted either from the input data layer for the first hidden layer, or from the previous hidden layer for the rest hidden layers of the network:

$$\boldsymbol{h}^{(l)} = \sigma(\boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}), \qquad \text{for hidden layers } l = 2, 3, \ldots.$$

In this way the neural network acts as a feature designer, where the input data are transformed in each layer step to a new feature space, where higher level information of data is obtained. Deep MLPs allow as to model complex non-linear functions. Every hidden layer represents a non-linear space transformation, produced by the activation function, which is in our case is the sigmoid function. By stacking together many hidden layers we could describe an arbitrarily complicated function. Therefore, a classification problem which may not have linearly separated classes in the input space, after applying a sequence of hidden layers may become separable in the last hidden layer space. A shallow MLP can be visualized as shown in Figure 2.1.

In this simple MLP case the trainable variables are the set $\theta = \{\boldsymbol{W}, \boldsymbol{V}, \boldsymbol{b}, \boldsymbol{c}\}$ and the loss function can be defined to be the squared error

$$E(\theta) = \frac{1}{N}\sum_{i=1}^{N} ||\boldsymbol{y}_i^{target} - \boldsymbol{y}_i||^2 = \frac{1}{N}\sum_{i=1}^{N} ||\boldsymbol{y}_i^{target} - \boldsymbol{V}(\sigma(\boldsymbol{W}\boldsymbol{x}_i + \boldsymbol{b})) + \boldsymbol{c}||^2$$

with $(\boldsymbol{x}_i, \boldsymbol{y}_i^{target})$ being a datapoint in the training dataset of N points. This is the function we aim to minimize with respect to the parameters θ. Additionally, we could add a regularization term to the error in order to reduce overfitting. Neural networks are optimized with iterative procedures, such as Stochastic gradient descent (SGD). The optimization procedure, which is essentially the training process of the neural network, heavily depends on the network architecture and is described in more detail in chapter 6. To perform optimization steps, in most cases, we have to compute the gradients of the loss function with respect to the network trainable parameters. This is achieved with a method called back-propagation, a general algorithm, which is not applied only to MLP, but in arbitrarily complex neural networks as well.

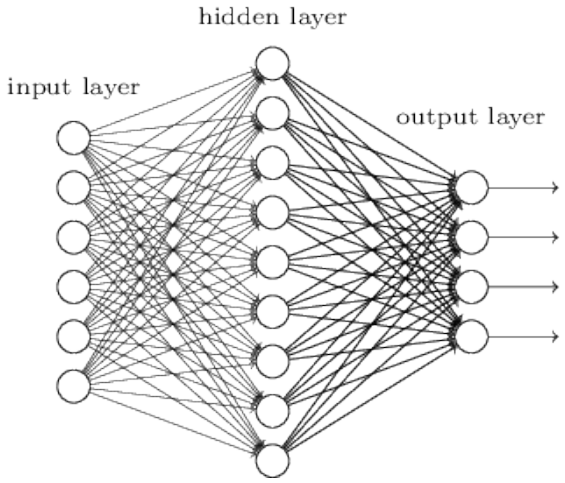

Figure 2.1: Vanilla (shallow) MLP, with one hidden layer. The nodes of every layer correspond to data values (a vector), while the connections represent the weights of the network. This image is taken from the Nielsen (2015) book.

Softmax

In the case of classification, the supervised learning category in which the lipreading problem belongs, we aim to predict class labels and not continuous values. The Softmax function is often employed in the final layer of neural network-based classifiers. For a Multilayer perceptron classifier, the Softmax function transforms the MLP output, which is a vector $\boldsymbol{y} \in R^m$ to a new vector $\boldsymbol{s} \in \mathbb{R}^m$, where $\sum_{j=1}^{m} s_j = 1$. This vector can then be perceived as a categorical probability distribution over m possible outcomes. The Softmax function or normalized exponential function is defined as

$$s_j = s_j(\boldsymbol{y}) = \frac{\exp(y_j)}{\sum_{r=1}^{m} \exp(y_r)} \quad j = 1, \ldots, m.$$

In this way every Softmax output value s_j corresponds to the probability that the input data sample of the MLP belongs in class j.

2.1.2 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) is another class of deep neural networks, which are mainly used in problems where data samples appear to have a grid topology. For example, images are datapoints with a 2D structure and there is a correlation between values of pixels in a neighbourhood. Convolutional neural networks that perform image classification exploit the 2D spatial patterns to extract more meaningful and better quality features. In the same way 3D CNNs are used for capturing spatiotemporal dynamics in sequences of images (videos), since there are correlations in the values of pixels across the time dimension as well.

Convolutional neural networks is a special type of neural networks and share many haracteristics with Multilayer perceptrons, while their main difference is that they use convolution instead of matrix multiplication.

A CNN usually consists of:

- An input layer where 2D or 3D objects are placed (for 2D or 3D CNN respectively), which has one or more channels. For instance, an RGB image requires three input channels.
- A sequence of hidden layers, where each one is computed from the previous one with the operations:
  - convolution,
  - activation function, with ReLU being the one most commonly used,
  - and pooling, which is optional.
- A number of fully connected (FC) layers with one output layer, which is essentially an MLP.
- In case of classification, a Softmax unit is the final layer of the network and generates the distribution over classes.

An example of the architecture of a 2D CNN is demonstrated in Figure 2.2 below.

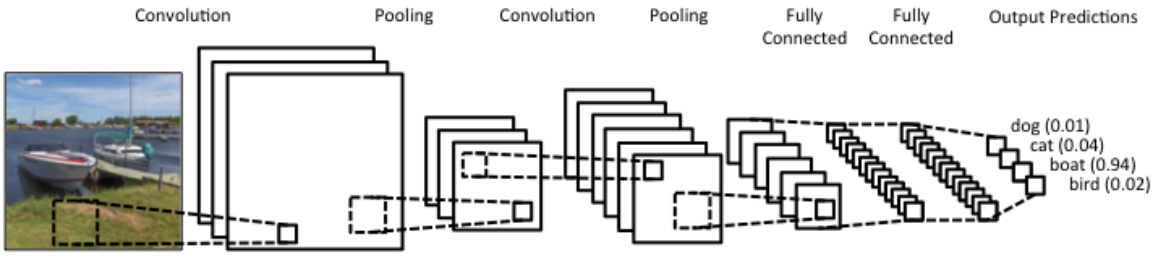

Figure 2.2: This image is taken from https://www.clarifai.com/technology. The network is made up of two convolutional hidden layers, each one with a pooling function, followed by two fully connected layers.

Convolutional layer

Convolution is the main feature of a CNN. Each convolutional layer's parameters consist of a set of learnable filters (or kernels). Each filter has a set of weights and a bias. For a convolutional layer with D_2 filters (or output channels) and D_1 input channels the weights have the form: $\boldsymbol{W} \in \mathbb{R}^{D_1 \times F \times F \times D_2}$, where F is the width and height of each filter. The bias of the convolutional layer is $\boldsymbol{b} \in \mathbb{R}^{D_2}$. In order to compute the convolutional layer output, or local receptive field in a hidden layer, we slide the filters across the previous hidden layer:

$$z^{(l)}_{i,j,r} = \sum_{d=0}^{D_1-1} \sum_{x,y=0}^{F-1} w_{d,x,y,r} \cdot a^{(l-1)}_{(i+x),(j+y),d} + b_r, \quad i = 0, ..., W_2 - 1, j = 0, ..., H_2 - 1, r = 0, ..., D_2 - 1.$$

If the previous hidden layer has dimensions $W_1 \times H_1 \times D_1$ and we use zero padding in the borders of the previous layer of size P_1, then the current hidden layer has dimensions $W_2 \times H_2 \times D_2$ where

$$W_2 = W_1 - F + 2P_1 + 1, \quad H_2 = H_1 - F + 2P_1 + 1$$

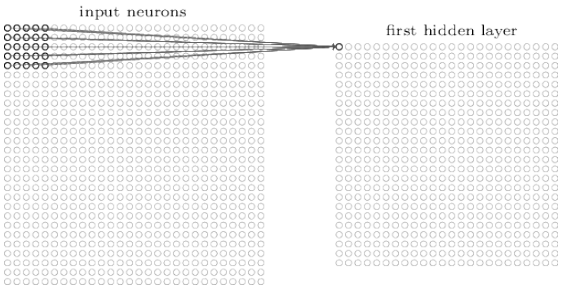

Figure 2.3: The convolutional operation to compute the receptive field at (0,0) in the first hidden layer from the input image. Image taken from Nielsen (2015).

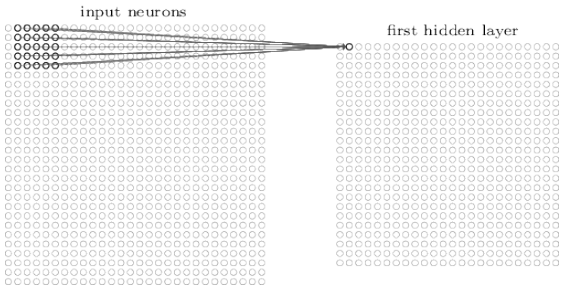

Figure 2.4: Computation of the receptive field at (0,1) in the first hidden layer from the input image. Image taken from Nielsen (2015).

In case we slide filters with a step greater than one, we define a stride term S. Then we have the receptive field in the hidden layer $\boldsymbol{z}^{(l)}$, obtained from the activation $\boldsymbol{a}^{(l-1)} \in \mathbb{R}^{W_1 \times H_1 \times D_1}$ of the previous hidden layer as

$$z^{(l)}_{i,j,r} = \sum_{d=0}^{D_1-1} \sum_{x,y=0}^{F-1} w_{d,x,y,r} \cdot a^{(l-1)}_{(Si+x),(Sj+y),d} + b_r, \quad i = 0, ..., W_2 - 1, j = 0, ..., H_2 - 1, r = 0, ..., D_2 - 1,$$

with

$$W_2 = \frac{W_1 - F + 2P_1}{S} + 1, \quad H_2 = \frac{H_1 - F + 2P_1}{S} + 1$$

Figure 2.3 and Figure 2.4 illustrate how convolution works for stride $S = 1$, in case we have one filter ($D_2 = 1$) and one input channel ($D_1 = 1$) with filter width and height $F = 5$.

ReLU layer

A rectified linear unit (ReLU) function is applied to the local receptive field $\boldsymbol{z}^{(l)}$ of a hidden layer l as follows

$$\boldsymbol{a}^{(l)} = max(0, \boldsymbol{z}^{(l)}), \quad \boldsymbol{z}^{(l)} \in \mathbb{R}^{W \times H \times D},$$

to obtain the activation of the neurons. This step increases the non-linear properties of the overall network. Different activation functions may be also used, such as the hyperbolic tangent and the sigmoid function. However, the ReLU is preferred in various networks, since it helps overcome the the problem of vanishing gradients, which is a common issue in neural networks. During training, when the back-propagation algorithm is used to propagate the error from the last layers of the network to the front layers, the gradient in the first layers takes small values close to zero. This results in slow updates in the weights of the first layers.

Pooling Layer

Pooling is a function commonly applied after the activation function in a hidden layer. Pooling is a non-linear function which acts as a down-sampling mechanism, since it progressively reduces the spatial size of features in the hidden layers. In this way, the number of trainable parameters is reduced in the last layers of the network.

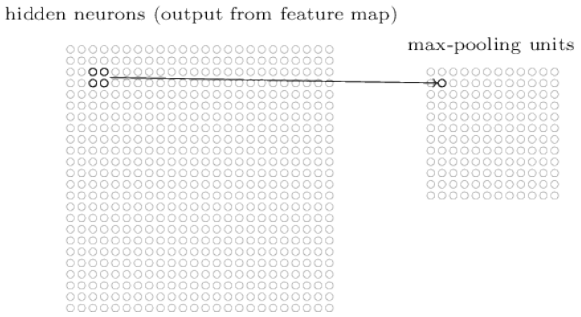

Figure 2.5: Max-pooling function with kernel size 2×2 ($F = 2$) and a stride $S = 2$. In this example the spatial dimensions (width, height) are reduced in half, while the four adjacent values are reduced to one, which is the maximum of them. This image is from Nielsen (2015).

The pooling layer operates independently on every depth channel and resizes it spatially, while the depth dimension remains unchanged. The most commonly used pooling method is max-pooling, which is alleged to produce the best results in practice. In addition to max pooling, the pooling units can use other functions, such as average pooling or L2-norm pooling. In max-pooling 2D subregions of size $F \times F$ are reduced to a single value, the maximum value of the neurons in the subregion. The concept of stride exists in pooling as well. Figure 2.5 shows how max-pooling operator works in case we have a depth dimension of unitary size.

Pooling helps to make the network invariant to translations of the input data, since small variations in the positions of features in the input do not affect the max-pooling output significantly. The exact location of a feature in the input becomes of less importance and the network focuses on the fact that there is a feature in the neighborhood of other features in the input. Therefore, max-pooling can be used as a downsampling method to reduce overfitting.

Fully connected layers

In Convolutional neural networks, the sequence of hidden layers (each one consisting of the convolution operation, ReLU and optionally pooling) is often followed by one ore more fully connected (FC) layers. These fully connected layers with one output layer in the end are essentially a feed-forward network (MLP), where neurons in each layer are connected with all neurons from the previous layer. In order to attach the fully connected layers in the final convolutional hidden layer, the following technique is used: The final convolutional hidden layer is flattened, by transforming the 3D representation $width \times height \times depth$ to a vector. This vector is then treated as the input to the FC layers. The output of the fully connected network coincides with the output of the entire CNN.

2.1.3 Long Short-term Memories (LSTMs)

Recurrent neural networks (RNNs) is another family of neural networks, where connections between hidden units form cycles in the time domain. These networks are mainly used to model the temporal dynamics in sequences of input data. The architecture of RNNs enables the formation of a memory unit in the neural network, which retains information from pre-

vious data in the sequence. Information is passed from one step of the network to the next. A simple RNN is shown if Figure 2.6, where the network is unfolded in time. The input of the network is a time sequence $\boldsymbol{X} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_{t-1}, \boldsymbol{x}_t, \boldsymbol{x}_{t+1}, ..., \boldsymbol{x}_T\}$. The hidden layer vector at time step t is computed as follows:

$$\boldsymbol{h}_t = \tanh(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{b})$$

where the trainable parameters $\boldsymbol{W}, \boldsymbol{U}$ are weight matrices and $\boldsymbol{b}$ is a bias vector.

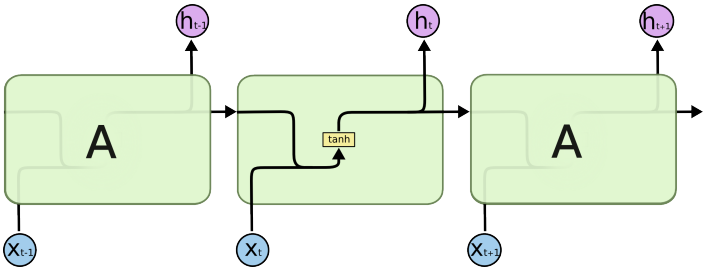

Figure 2.6: A simple RNN with one hidden layer and the hyperbolic tangent activation function. The figure is taken from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long short-term memory (LSTM) is a special Recurrent neural network which revolutionized speech recognition due to its two main advantages:

- LSTM avoids the vanishing gradient problem, in which loss gradients approach zero values in back propagation during training.
- LSTM is capable of learning long-term dependencies, even in cases where there is a large gap between data-points with a strong dependence in the sequence.

The core idea behind LSTM is the cell state, which is the memory unit of the network. In contrast with the RNN where the hidden unit is actually the memory unit and an activation function is applied in every time step to the hidden state, no activation function is applied to the cell state $\boldsymbol{c}_t$ of an LSTM. The network has the ability to add and delete information from the cell state in a controlled way. Gates are a way to control how information is passed to the cell state. The internal operations in a single LSTM layer to obtain the hidden state (which could be the output of the network as well) are shown in Figure 2.7.

The forget gate vector acts as a weight of remembering old information. This vector at time step t is defined as:

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f\boldsymbol{x}_t + \boldsymbol{U}_f\boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$$

Next, there is a mechanism to determine what new information we are going to store in the cell state. The input gate vector decides which values of the cell state we will update. At time step t its value is:

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{U}_i\boldsymbol{h}_{t-1} + \boldsymbol{b}_i)$$

Additionally, a vector of candidate values for the cell state is computed:

$$\bar{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}_c\boldsymbol{x}_t + \boldsymbol{U}_c\boldsymbol{h}_{t-1} + \boldsymbol{b}_c)$$

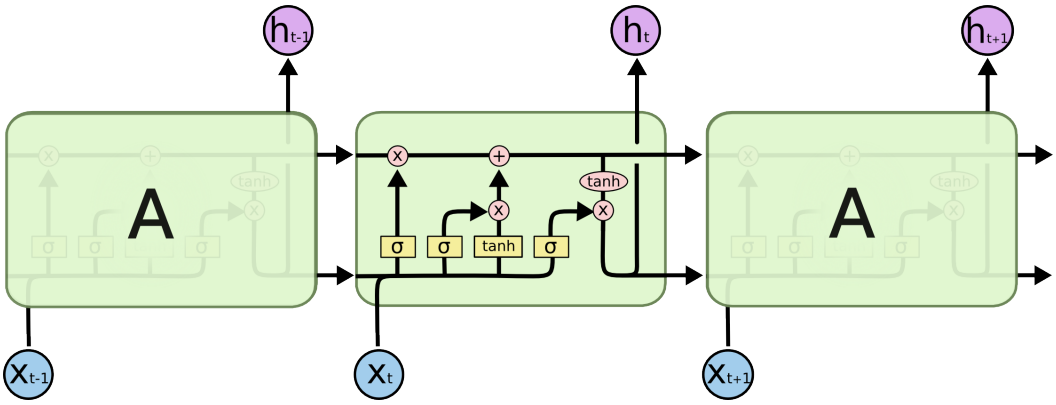

Figure 2.7: An LSTM layer with the operations to compute the hidden state at time step t. The figure is taken from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Then the new cell state will be formed from the previous cell state and the new candidate values

$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \bar{\boldsymbol{c}}_t,$$

where $\circ$ is the Hadamard or element-wise product.

Subsequently, the hidden state (or output) is determined form the new cell state and a candidate output, which is called the output gate vector:

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \boldsymbol{x}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$$

Finally, the new hidden state will be

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t).$$

Therefore the set of trainable parameters of the LSTM is: $\theta = \{\boldsymbol{W}_f, \boldsymbol{W}_i, \boldsymbol{W}_c, \boldsymbol{W}_o, \boldsymbol{U}_f, \boldsymbol{U}_i, \boldsymbol{U}_c, \boldsymbol{U}_o, \boldsymbol{b}_f, \boldsymbol{b}_i, \boldsymbol{b}_c, \boldsymbol{b}_o\}$.

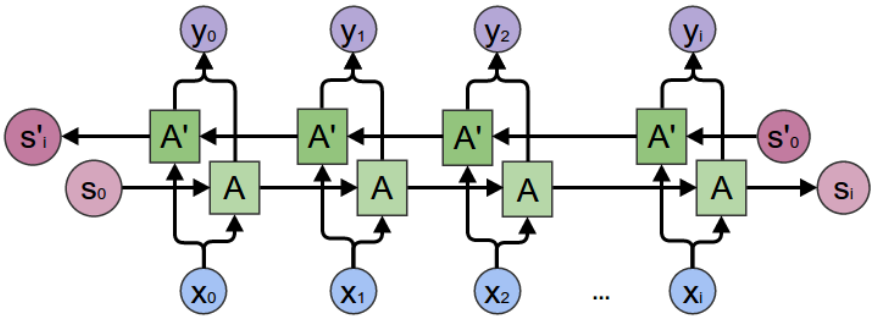

Figure 2.8: A Bidirectional long short-term memory with one hidden layer. The forward LSTM cells unfolded in time are denoted by **A**, while the backward LSTM cells unfolded in time are denoted by **A'**. The figure is taken from: http://colah.github.io/posts/2015-09-NN-Types-FP/

In many cases the input data sequence may have a bidirectional dependence. This means that the data-point $\boldsymbol{x}_t$ could be also related to future values in the stream $\boldsymbol{x}_{t'}$, with $t' > t$. Bidirectional Recurrent Neural Networks (BiRNNs) have to ability to model temporal dynamics in both forward and backward directions. The basic idea of BiRNNs is to connect

two hidden layers of opposite directions to the same output. A special type of BiRNNs is the Bidirectional Long short-term memory (BiLSTM), which is essentially formed by two LSTMs, one forward and one backward layer. The forward LSTM reads the input sequence in regular order $\boldsymbol{x}_1, \dots \boldsymbol{x}_T$ and outputs a sequence of forward hidden states $\overrightarrow{\boldsymbol{h}_1}, \dots \overrightarrow{\boldsymbol{h}_T}$. In the same way the backward LSTM reads the input sequence in reverse order $\boldsymbol{x}_T, \dots \boldsymbol{x}_1$ and produces the backward hidden states $\overleftarrow{\boldsymbol{h}_1}, \dots \overleftarrow{\boldsymbol{h}_T}$. Then, the BiLSTM output $\boldsymbol{y}_1, \dots \boldsymbol{y}_T$ is the concatenation of the two hidden states, where $\boldsymbol{y}_t = [\overrightarrow{\boldsymbol{h}_t}^\top ; \overleftarrow{\boldsymbol{h}_t}^\top]^\top$. Now, the BiLSTM output $\boldsymbol{y}_t$, at time step t, contains information from both the preceding $\boldsymbol{x}_{t'<t}$ and following $\boldsymbol{x}_{t'>t}$ input data in the sequence. Figure 2.8 is a graphical illustration of the BiLSTM model.

2.1.4 Attention-based LSTM Transducer

Attention-based recurrent networks have been successfully applied to neural machine translation on the task of English-to-French translation by Bahdanau et al. (2014b) and to audio-based speech recognition by Chorowski et al. (2015); Chan et al. (2015b). This technique has been recently used for lipreading as well. Chung et al. (2016) proposed a attention-based LSTM transducer as a part of a neural network that produces a sequence of characters in its output based on visual and audio information. The same LSTM transducer network is described in more detail below, since it constitutes a building block of the second proposed lipreading network in this project.

An LSTM transducer with an attention mechanism is a neural network that transforms an input data sequence to an output sequence, where the input-output alignment is learned automatically during training. The network ”learns” which segments in the input sequence are more important and contribute to the generation of every output token in the output sequence.

The input data sequence of the network $\boldsymbol{o} = \{\boldsymbol{o}_1, \dots \boldsymbol{o}_T\}$ is usually a higher level feature description of a raw sequence $\boldsymbol{x}_1, \dots, \boldsymbol{x}_T$ and has been obtained by a feature extraction network or method in general. The output $\boldsymbol{y} = \{y_1, \dots, y_{T_{out}}\}$ is a sequence of tokens. For the lipreading problem these tokens could be characters of the English alphabet with some extra symbols. The <sos> token is used to indicate the beginning of the sequence, the <eos> to indicate the end, while the <pad> token corresponds to a padding symbol to help us deal with output sequences of variable length. The attention-based LSTM transducer architecture is shown in Figure 2.9.

The neural network consists of three parts:

1. An LSTM transducer network (LSTM decoder), with one or more hidden layers,
2. an attention network
3. and a Multilayer Perceptron with a Softmax function in its final layer.

The LSTM is used in the front part of the network to model the dependence of every output token with all previous tokens. At each time step t the LSTM consumes as input the previous output token y_{t-1} and the previous context vector $\boldsymbol{c}_{t-1}$, and produces the LSTM hidden state $\boldsymbol{h}_t$. The context vector is computed by the attention mechanism and is described in more detail below. Therefore, the LSTM decoder can be described as:

$$\boldsymbol{h}_t = LSTM(y_{t-1}, \boldsymbol{c}_{t-1}, \boldsymbol{h}_{t-1})$$

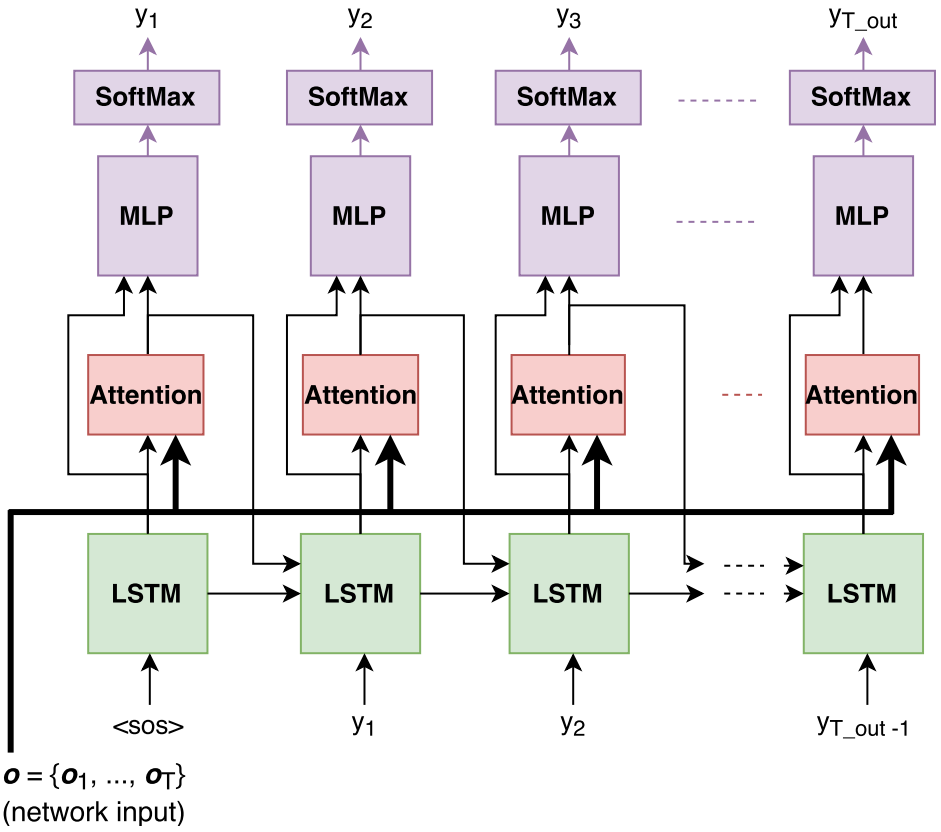

Figure 2.9: The unrolled in time block diagram of the attention-based LSTM transducer network.

The hidden state $\boldsymbol{h}_t$ of the LSTM decoder at time step t is then forwarded to the attention mechanism, along with the network input $\boldsymbol{o}$, in order to compute the context vector $\boldsymbol{c}_t$:

$$\boldsymbol{c}_t = Attention(\boldsymbol{h}_t, \boldsymbol{o})$$

At time step t, for each point in the sequence $\boldsymbol{o}_1, \dots \boldsymbol{o}_T$, the attention network computes the scalar

$$e_{ti} = \boldsymbol{w}^\top \tanh(\boldsymbol{W}\boldsymbol{h}_t + \boldsymbol{V}\boldsymbol{o}_i + \boldsymbol{b}), \quad i = 1, \dots, T.$$

Next, the vector $\boldsymbol{e}_t = \begin{bmatrix} e_{t1} \\ \vdots \\ e_{tT} \end{bmatrix}$ is "squashed" using the normalized exponential function

$$a_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^{T} \exp(e_{tj})}, \quad i = 1, \dots, T.$$

Then, the normalized vector $\boldsymbol{a}_t = \begin{bmatrix} a_{t1} \\ \vdots \\ a_{tT} \end{bmatrix}$ contains the weight for each vector $\boldsymbol{o}_i$, with $i = 1, \dots, T$ in the input sequence $\boldsymbol{o}$. In this way, every element a_{ti} represents the importance of $\boldsymbol{o}_i$ for the determination of the network's output at time step t. The context vector at this

time step is defined as the weighted sum of the vectors from the input sequence:

$$\boldsymbol{c}_t = \sum_{i=1}^{T} a_{ti} \boldsymbol{o}_i$$

Subsequently, the context vector is concatenated with the LSTM hidden state and the result $[\boldsymbol{c}_t^\top ; \boldsymbol{h}_t^\top]^\top$ forms the input of the Multilayer perceptron. The output of the MLP is then captured by a SoftMax unit to compute a distribution vector $\boldsymbol{p}_t \in [0,1]^{num\_tokens}$ on the different possible tokens (classes). Then, the output of the entire neural network at time step t is the class with the maximum probability:

$$y_t = \text{argmax}(\boldsymbol{p}_t)$$

In the next time step $t+1$, the previous context vector $\boldsymbol{c}_t$ and the predicted class y_t are passed to the LSTM decoder as inputs in order to calculate the next output y_{t+1}. During training, instead of using the predicted token y_t, we use the ground truth class label y_t^{target} from the dataset. However, this practice may lead to poor performance during inference, since the system is trained to calculate the next token based on the correct current token, which is not available in prediction. One way to overcome this problem could be the implementation of a probabilistic policy during training, where the predicted token is passed to the LSTM in the next time step with some probability, instead of using the ground truth every single time.

A commonly used method to determine the predicted output sequence $\boldsymbol{y} = \{y_1, \ldots, y_{T_{out}}\}$ during inference is beam search. According to beam search algorithm, we store w of the most probable hypothesis in every time step, rather than keeping only the one with the maximum probability. Then, we test every single one of these hypothesis, by using it as input to the LSTM decoder in the next time step. Again in next time step, we keep only w of the best hypothesis so far. In this way, we compare in total $num\_tokens \times w$ possible output subsequences of the network at every time step and choose w with the highest joint probability. The key idea behind this method is that we may get a better prediction if we are not completely "greedy" by always propagating to the LSTM the token with the highest probability in the next time step. By trying subsequences of tokens that may not seem optimal in the beginning we may discover an output sequence $y_1, \ldots, y_{T_{out}}$ with a higher joint probability. The hyperparameter w is called the beam search width and controls the search space.

2.2 A Review on Lipreading Systems

2.2.1 Lipreading as part of Audio-visual Speech Recognition

Lipreading, also known as visual speech recognition, is a problem closely related with automatic speech recognition (ASR). Automatic speech recognition is the ability of a computer to convert words or sentences in spoken language into text. Human-machine interaction has increased rapidly throughout the last decades (e.g. smartphones, smart TVs), therefore speech recognition performance has become crucial for achieving effective interaction between user and machine. Speech recognition applications include voice search, voice dialing, data entry, call routing and speech-to-text processing. ASR systems use mainly audio signals to predict what the speaker says. Audio-based speech recognition works by processing voice

signals and using the extracted audio features to translate speech to text. Despite the success of audio-based ASR systems, there are cases where audio is noisy or absent. Audio-visual speech recognition (AVSR) is thought to be one of the most promising solutions for reliable speech recognition, as it combines audio signals with visual information to produce the text that corresponds to the spoken word or sentence.

Most lipreading methods have been developed as part of AVSR systems, since they enabled the addition of visual information to the audio-based speech recognition techniques in order to enhance the predicting ability of these systems. Therefore, a detailed examination of various AVSR systems along with lipreading methods could help us determine the key features of a robust lipreading system.

2.2.2 Pre-deep Learning Approaches

Prior to the utilization of deep learning techniques, most of the work was focused on extracting better visual features, which were usually modeled by Hidden Markov Model (HMM) classifiers. Additionally, research effort was concentrated on audio-visual speech recognition systems, since pure lipreading systems did not achieve high predictive performance. Therefore, a great amount of work was focused on the development of better audio-video fusion schemes. Traditional machine learning approaches for lipreading and AVSR systems were reviewed thoroughly in Zhou et al. (2014b). Many of them aimed to extract hand-crafted features which are no longer used in state-of-art deep learning approaches. Visual feature extraction and audio-visual fusion in traditional machine learning AVSR systems is briefly discussed below.

Visual feature extraction

In their work, Zhou et al. (2014b) categorized the development of visual feature extraction methods from a problem-oriented perspective.

Speaker dependency was the first problem to tackle in visual feature construction. Much effort was done in the direction of searching for a linear transformation that results in a lower-dimensional subspace, where speaker dependency is suppressed. Linear discriminant analysis (LDA), a widely utilized and well studied dimensionality reduction method was widely used. Potamianos et al. (2001) applied first an image transformation (e.g. PCA) on mouth region images and removed the mean from the feature vectors. Then, they applied LDA to obtain feature vectors of a smaller dimension. Later, they extended their method by applying another LDA transformation on the concatenation of consecutive feature vectors, which was the output of the previous LDA transformation. The extracted feature for the second Linear discriminant analysis transformation was named 'HiLDA' and contained temporal information as well. Later, Zhou et al. (2014a) identified two dominating sources of variation in images of the mouth region. Variations were caused by the appearance variability among speakers, as well as by a speaker uttering different words. Therefore, the process of generating a video sequence was modeled as:

$$\boldsymbol{x}_t = \boldsymbol{\mu} + \boldsymbol{F}\boldsymbol{h} + \boldsymbol{G}\boldsymbol{w}_t + \boldsymbol{\epsilon}_t,$$

where $\boldsymbol{x}_t$ is the image at time t, which is assumed to be obtained by the latent speaker variable $\boldsymbol{h}$, the latent utterance variable $\boldsymbol{w}_t$, an image mean $\boldsymbol{\mu}$ and the normally distributed

noise e_t. A model $\mathcal{M}$ was fitted to image sequences $\{\boldsymbol{x}_t\}$ for a particular utterance through measuring the posterior $p(\{\boldsymbol{w}_t\}, \boldsymbol{h}|\{\boldsymbol{x}_t\}, \mathcal{M})$ and the MAP estimation $\{\hat{\boldsymbol{w}}_t\}$ was used as visual feature for the corresponding sequence $\{\boldsymbol{x}_t\}$.

Pose Variation is another important problem to consider for visual feature construction. Speakers are sometimes filmed from different views, which can significantly affect the appearance of the mouth region and diminish features quality. Most of the methods concerning pose variation either extracted features from non-frontal camera views (pose-dependent features) and trained a system for every view, or attempted to transform pose-dependent features (PDFs) into a common pose-independent features (PIFs) space, so that they are comparable. The first technique may suffer from lack of representative data, since non-frontal poses may not be enough to train the system. Among many methods proposed to design PIFs, a linear transformation technique was proposed by Lucey et al. (2007, 2008). They used linear regression to transform PDFs from an unwanted camera view to a desired common view. If $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_N]$ contains the training samples of the unwanted view and $\boldsymbol{T} = [[\boldsymbol{t}_1, 1]^\top, ..., [\boldsymbol{t}_N, 1]^\top]$ are the corresponding desired view features, the linear transformation to project features from PDF to the PIF space turns out to be

$$\boldsymbol{W} = \boldsymbol{T}\boldsymbol{X}^\top(\boldsymbol{X}\boldsymbol{X}^\top + \lambda \boldsymbol{I})^{-1},$$

where λ is the regulariser hyperparameter used to control overfitting. Now given a sample $\boldsymbol{x}$ of the unwanted view, its transformed vector was computed as $\boldsymbol{t} = \boldsymbol{W}\boldsymbol{x}$.

Visual data do not only contain spatial information of the mouth region of a talking person, but temporal information as well, which describes the dynamical process of speech. As mentioned before, Potamianos et al. (2001) proposed to concatenate consecutive feature vectors and employ LDA to obtain the final "HiLDA" features that encoded temporal information. However a linear transformation is not sufficient to encapsulate the dynamic process that generates data. Ong and Bowden (2011) proposed temporal signatures as a way to capture the temporal information. They represented a static image as a binary feature vector. First they constructed a temporal pattern, which is a binary image formed by stacking feature vectors extracted from a fixed number of successive video frames. They defined a temporal signature as an arbitrary set of "one" locations within a temporal pattern. In order to detect temporal signatures within an input temporal pattern, weak classifiers were used. Next, strong classifiers were constructed from the weak classifiers using AdaBoost algorithm, for utterances recognition. In their work Pachoud et al. (2008) extracted visual features directly from 3D images. For this purpose, video segments were separated into several "macro-cuboids". Each macro-cuboid was afterwards split into cuboids. Then, the SIFT descriptor for cuboids was used to compute visual features from them. Many other sophisticated methods were proposed to model the temporal dynamics of uttering. Instead of modeling the local pixel-level spatio-temporal structure, these methods modeled the structure at the frame level and enforced it in visual features.

Audio-visual fusion

Audio and video fusion is a complex problem and may be performed either at the feature or the decision level. At the feature level, the visual and audio features are combined to create a new set of features, which include information from both sources. On the contrary, the decision level fusion combines the distribution results of the Hidden Markov Model classifiers, which are trained separately for the individual modalities (one classifier for the audio and

another for the video stream). Dynamic AV fusion (DAVF) is a method applied in the decision level and allows the system to adjust to noisy audio or video sources, and favor the one source over the other dynamically. The weights for each one of the two streams are calculated online from the current audio and visual data, using modality reliability measures. The signal-to-noise ratio (SNR) is considered one of the most straightforward reliability measures and was used in many research works. SNR can be computed as the ratio between the power of the speech signal and the power of noise. Given the reliability measures, there have been various functions designed to map the measures to the two stream weights. Papandreou et al. (2008) proposed a method to model the uncertainties of the observed stream features and introduced a high performance hybrid model to assign weights for the two modalities.

2.2.3 Deep Lipreading Networks

Prior to the deep learning era in lipreading, visual features were handcrafted, but this trend started to change when deep neural networks became popular for visual feature extraction. However, the first attempts for deep lipreading or deep audio-visual speech recognition systems kept using a traditional HMM classifier in the core of the system.

Noda et al. (2014) proposed a Convolutional Neural Network (CNN) as a visual feature extraction mechanism, to predict phonemes. The CNN was trained with images of the speaker's mouth region with phoneme labels as target values. The trained Convolutional network which consisted of multiple convolutional layers, was then used to extract visual features. Furthermore, a Hidden Markov Model (HMM) was used for modeling the temporal dependencies of the generated phoneme label sequences. The evaluation results demonstrated that the visual features acquired by the CNN significantly outperformed those acquired by conventional feature extraction approaches.

Mroueh et al. (2015) presented methods in deep multimodal learning for fusing audio and visual modalities for audio-visual speech recognition. They first studied an approach where two uni-modal deep neural networks (DNNs) were trained separately on the audio and the visual features. For the audio features, they extracted Mel-frequency cepstral coefficients (MFCC) from the audio signal and stacked nine consecutive MFCC frames together. Next, they used Linear discriminant Analysis (LDA) to map them to a lower dimension space and stacked nine LDA frames together around the frame of interest to form the final audio feature. Visual features where extracted in a similar way, by first extracting level 1 and level 2 scattering coefficients on the mouth region, then using LDA for dimensionality reduction and finally stacking nine LDA frames together around the frame of interest to form the final visual feature. Every audio or video frame was labeled with one phonem class. Subsequently, two separate DNNs were trained under the cross-entropy objective using the stochastic gradient descent (SGD), one with the audio and one with the visual previously extracted features. For fusion, a joint audio-visual feature representation was formed by concatenating the output layers of the two deep neural networks. Then, a deep or a shallow (softmax only) network was trained in this fused space up to the target phoneme classes. Finally, they addressed the joint training problem, by introducing a bilinear bimodal DNN to further improve their results.

Another audio-visual speech recognition (AVSR) system was proposed by Noda et al. (2015). In this work, they utilized deep neural networks for the construction of audio and visual features, although the fusion mechanism was HMM-based. For the audio feature extraction, a deep denoising autoencoder was proposed. In order to train the autoencoder, they prepared

deteriorated sound data with different signal-to-noise-ratios (SNRs) levels and extracted log mel-scale fiterbank (LMFB) or MFCC features from all audio signals. The deep denoising autoencoder was trained to construct clean audio features from deteriorated features by feeding the deteriorated dataset as input and the corresponding clean dataset as the target of the network. To optimize the deep autoencoder, the Hessian-free optimization algorithm was used. For visual features, one CNN for every speaker was trained to predict phoneme labels from input images. Each CNN contained seven layers: three convolutional layers each one followed by a max or average-pooling layer and one fully connected layer with a Softmax function in the end. The CNNs were optimized to maximize the multinomial logistic regression objective of the correct label, using a stochastic gradient descent method. After training the CNNs, the desired visual features were generated by recording the outputs from the last layer of the CNN, when the image sequence of the mouth region corresponding to a single word was provided as input. Finally, a multi-stream HMM (MSHMM) was applied for integrating the acquired audio and visual HMMs, which were independently trained with the previously extracted audio and visual features. The outputs of the two unimodal classifiers (HMMs) were merged, using dynamic stream weight adaptation to determine a final word classification.

Tamura et al. (2015) proposed an AVSR system for Japanese audio-visual data classification in eleven classes, each class corresponding to a word. The key idea was to combine basic visual features and then use in with a deep neural network to extract deep bottleneck high-performance features. In the first phase a simple HMM-based AVSR system was built. For audio features, they first prepared conventional Mel-Frequency Cepstral Coefficients (MFCCs) and then an audio GMM-HMM was trained. The time-aligned transcription labels were then obtained using the audio GMM-HMMs and the training speech data. After that, visual GMM-HMMs were trained, applying bootstrap training and using visual PCA features with the labels. In the second phase, one audio DNN and one visual DNN with bottleneck hidden layers were built. For the visual DNN training, five basic visual features (PCA, DCT, LDA, GIF and COORD, a Shaped-based feature) were calculated from image frames. For the transformation methods that require class labels on images, such as LDA, they used Japanese visemes. These basic features were then concatenated into a single feature vector, along with their first and second time derivatives. For every frame, they stacked five previous and five incoming features in addition to the current feature vector. Additionally, the audio DNN was trained using audio features, with each audio feature being concatenated MFCC vectors from consecutive frames. After training, DNNs could be employed as feature extractors. Deep bottleneck audio features (DBAFs) and deep bottleneck visual features (DBVFs) were extracted from visual data and audio signals, respectively. Next, audio and visual GMM-HMMs were rebuilt using the DBAFs and the DBVFs. Moreover, multi-stream HMMs were employed to control contributions of audio and visual modalities and stream weights were empirically optimized. Finally, voice activity detection (VAD) was performed, in order to avoid recognition errors in silence periods for lipreading.

Subsequently, various lipreading systems which employed pure deep neural network techniques were developed. Petridis and Pantic (2016) proposed a lipreading network based on a deep autoencoder to extract Deep Bottleneck features (DBNFs) for visual speech recognition directly from pixels. The deep bottleneck visual feature extraction algorithm consisted of three stages. First an autoencoder, which consisted of an encoder DNN and a decoder DNN was trained. The encoding layers were trained in layer-wise manner using Restricted Boltzmann Machines (RBMs) and then the decoding layers were initialised with the weights of the encoding layers in reverse order. Afterwards, the whole autoencoder was fine-tuned

for optimal image reconstruction. The bottleneck layer which was placed between the encoder and the decoder was treated as the deep bottleneck visual feature. In the next stage, the decoder was removed and replaced by two hidden layers connected with a softmax layer for classification. A k-means clustering was used on the images since viseme target labels were not available for every image in the dataset. Next, the network was fine-tuned, while DCT features were appended to the bottleneck layer. This joint training reduced the redundant information in the deep bottleneck features and made them complementary to the DCT features. In the last stage, the DBNFs and the DCT features were augmented with their first and second derivatives and used as input to an LSTM classifier, which modeled the temporal dynamics. The whole network predicted word labels from an image sequence. Since LSTMs are trained per frame, during inference they applied majority voting to assign a label to the sequence.

In they work Chung and Zisserman (2016a) presented a multi-stage pipeline for automatically collecting and processing a very large-scale visual speech recognition dataset, which is refered as Lip Reading in the Wild (LRW) dataset. First subtitle text was extracted from the broadcast video and then force-aligned to the audio signal. The result was filtered by double-checking against the commercial IBM Watson Speech to Text service. In the next stage, the shot boundaries were determined, a HOG-based face detection method was performed and a KLT tracker was used to track the face. Finally, facial landmarks were determined in every frame of the face track in order to detect the mouth region and determine if the person is talking. This pipeline for automated large scale data collection tackled the small-lexicon problem, but their network still performed a word level classification task, hence the word boundaries had to be known beforehand. The authors developed and compared four neural network models for lipreading with their major difference being the way they ingested the input frames. These CNN architectures shared the configuration of VGG-M Chatfield et al. (2014). The four Convolutional networks developed and trained were named: Early Fusion (EF), 3D Convolution with Early Fusion (EF-3), Multiple Towers (MT) and 3D Convolution with Multiple Towers (MT-3). The EF network ingested a T-channel image, where each one of the channels encodes an individual frame in greyscale, while EF-3 was receiving color images and the convolutional and pooling filters operated and moved along all three dimensions, performing 3D convolution. In the MT network, there were T towers that shared the first convolutional layers (shared weights), each of which takes an input frame. Finally, MT-3 was the same with MT in the first layer, but from the second convolutional layer 3D convolutions were performed, in the same way with EF-3. One the one hand, EF-3 and EF assumed registration between frames, and since these models performed time domain operations, they captured local motion direction and speed. On the other hand, MT-3 and MT, both delayed all time-domain operations, until the second convolutional layer, which gave tolerance against small registration imperfections. The reason for experimenting with 3D convolutions was that intuitively a 3D convolution would be able to match well a spatio-temporal feature. Despite this intuition, their results showed that the 2D convolutions performed better than 3D.

Later, Chung and Zisserman (2016b) proposed another lipreading network containing a CNN connected with an uni-directional LSTM classifier. They used OuluVS2 Anina et al. (2015) dataset, which consists of 10 short phrases spoken by different subjects, for training and evaluation. The LSTM network ingested the visual features from the CNN at each time-step, which were produced on a 5-frame sliding window, and returned the classification result at the end of the sequence. The network was trained with the stochastic gradient descent optimization method, with a Softmax log loss measure, which was computed in the last

time-step. The Convolutional network had been pre-trained on ImageNet Russakovsky et al. (2014).

In all previous lipreading systems, models were trained to perform only word or phoneme classification rather than sentence-level sequence prediction. Assael et al. (2016), proposed LipNet, a model that mapped a variable-length sequence of video frames to text, making use of spatiotemporal convolutions, a recurrent network, and the connectionist temporal classification loss, trained entirely end-to-end. The LipNet architecture started with three Spatiotemporal convolutional neural networks (3D-CNNS), which could process video data by convolving across time, as well as the spatial dimensions. Then, the features extracted were processed by two Bi-GRUs, which are Bidirectional recurrent neural networks. The two Bi-GRUs ensured that at each time-step, the features on their output depended on the STCNNs output of all previous time-steps. Subsequently, a linear transformation was applied at each time-step, followed by a Softmax over the vocabulary (characters) and then the Connectionist Temporal Classification (CTC) loss was applied. Given that the Softmax unit outputs a sequence of discrete distributions over the class labels (characters plus a special ”blank” token), CTC computed the probability of this sequence by marginalizing over all sequences that are defined as equivalent to it. This simultaneously removed the need for alignments and addressed the variable-length sequences issue. The Lipnet network was evaluated on GRID dataset (Cooke et al. (2006)), which contains short sentences drawn from a specific simple grammar.

Chung et al. (2016) faced lipreading as an open-world problem, where there are no constrains in the number of words in the sentences. The proposed neural network was designed to recognize sentences only from visual, audio or both sources. Prediction is done in a character level, since the network's output is a sequence of character tokens. Additionally, a multi-stage pipeline for automatically generating a large-scale dataset (LRS) for audio-visual speech recognition was proposed, similar to the one in Chung and Zisserman (2016a). The deep network consisted of three components: an image encoder which generated visual features, an audio encoder for audio features and a character decoder. The image encoder consisted of a CNN based on the VGG-M model Chatfield et al. (2014), which outputs visual features consumed by a three layer LSTM network at every input time step. The network ingested the image frame sequence in reverse time order. The audio encoder was built as a three layer LSTM which received MFCC features in reverse time order. Finally, the character decoder contained a three layer LSTM decoder with a dual attention mechanism, which determined which part of the image and audio encoder output sequence should be considered for the character in every output time step. The dual attention mechanism produced two context vectors (one for the video and one for the audio) which encapsulated the information required to produce the next step character output. These context vectors along with the LSTM output were passed to an MLP with a Softmax function that generated the probability distribution of the output character, at every output time step. Next, the context vectors and the predicted output character were transfered to the LSTM decoder as inputs for the next time-step character prediction. The whole network was jointly trained using curriculum learning and scheduled sampling, as well as multimodal training. Moreover, a beam search algorithm was employed for the prediction of the character sequence output during evaluation. The neural network was evaluated on the GRID Cooke et al. (2006), LRW Chung and Zisserman (2016a) and LRS datasets and surpassed the performance of all previous lipreading systems.

Stafylakis and Tzimiropoulos (2017) recently developed a deep network for word-level vi-

sual speech recognition. The system consisted of a 3D Convolutional neural network followed by a Residual Network, which provided input at every time step to a two-layer Bidirectional Long Short-Term Memory. Then a SoftMax layer was applied for all time steps. The 3D CNN was composed of a convolutional layer with 3-dimensional filters followed by Batch Normalization and Rectified Linear Units (ReLU). The spatiotemporal convolutional layer was used for capturing the short-term dynamics of the mouth region. Then, a max-pooling layer was used to reduce the spatial size. In the next stage, the features extracted from the 3D CNN were unstuck in the time dimension and passed to a Residual network, one per time step. The Residual network dropped the spatial dimensionality with max pooling layers, until its output was a one dimensional tensor per time step. These tensors were passed to a bidirectional LSTM with a Softmax layer, one at each time step, where the temporal dynamics were captured. The word label was repeated at every time step so that the overall loss was defined as the summation of losses over all time steps. Some variations of the network were explored and trained end-to-end on the LRW dataset Chung and Zisserman (2016a). The best configuration achieved an improvement in the word prediction accuracy on the LRW dataset over the state-of-art network of Chung et al. (2016).

Chapter 3

Dataset

3.1 Dataset Overview

The two lipreading neural networks designed and implemented in this project were trained and evaluated using the Lip Reading in the Wild (LRW) dataset, which was generated by Chung and Zisserman (2016a). The dataset consists of up to a thousand utterances of five hundred different words. Hundreds of speakers appear in the videos, providing strong speaker independence for the lipreading systems. The dataset consists of five hundred directories, each one corresponding to single word in the vocabulary and containing video samples of people uttering this word. Each word directory contains the ”train”, ”test” and ”val” directories, which contain the training, test and validation samples respectively. Each ”train” directory holds between 800 and 1000 video samples of the word, while the ”test” and ”eval” directories enclose 50 video samples each.

Figure 3.1: A sample of speakers in the LRW dataset.

An example of video frames from the LRW dataset is demonstrated in Figure 3.1. All videos are 29 frames in length and centered in the speaker's face. A single word in spoken in each video sample. This word occurs in the middle of the video and is between five and twelve characters long. Every video sample is stored in a .mp4 file format, while a metadata text file encloses information of the word duration for each video. Using the word duration we can determine the first and the last frame in which the speaker utters the word. Therefore, each video sample has a different number of actual frames, since the number of frames depends on the word duration in the video. Consequently, the neural networks should be able to handle frame sequences of various lengths. The maximum frame sequence length is 29.

3.2 The Pipeline for LRW Dataset Generation

All video samples were extracted from British BBC television programs, using a multi-stage pipeline, which automatically collected and processed this very large-scale visual speech recognition dataset. This pipeline was implemented and presented by Chung and Zisserman (2016a). As they describe in their work, the pipeline consists of the following stages:

Stage 1: The first issue they considered is what type of television programs are more appropriate for the collection of a lipreading dataset. For this reason they chose news and current affairs programs, where there is large variety of speakers. In this way the dataset could contain as many different speakers as possible.

Stage 2: After selecting the television programs, the audio and the subtitles should be aligned in order to get a timestamp, the exact point where a word starts and ends, for every word in the videos. First, subtitle text was extracted from the broadcast video using standard OCR methods. Secondly, they used an aligner that is based on the Viterbi algorithm to compute the maximum likelihood alignment between the audio and the text. Finally, the result was checked against the IBM Watson Speech to Text service.

Stage 3: In the subsequent stage, the shot boundaries were determined, by comparing color histograms across successive frames. Boundary shot detection was essential in order to run a face tracking algorithm next. Face detection was performed on every video frame by utilizing a HOG-based face detection method and all face detections of the same person were grouped across frames using a KLT tracker.

Stage 4: After tracking the face region, they employed a facial landmark detection method. These landmarks were useful to determine whether the person was speaking or not and extract the mouth region in case the speaker was indeed uttering a word.

Stage 5: Last but not least, the most frequently occurring words were selected and the corresponding videos were divided into a training, test and an validation set.

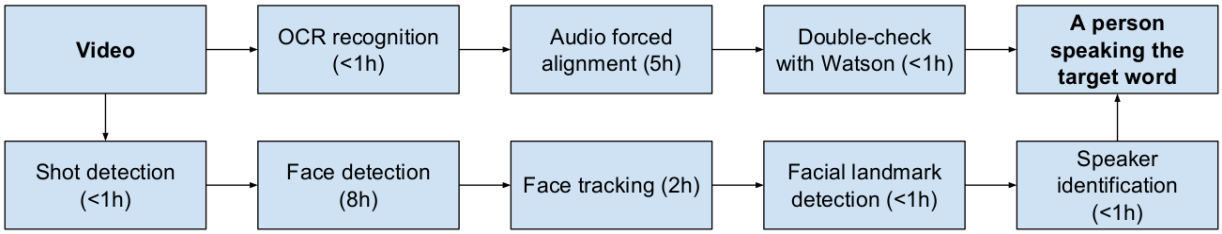

Figure 3.2: The multi-stage pipeline diagram from Chung and Zisserman (2016a).

Chapter 4

Neural Network Architecture

Lipreading can be considered as a classification problem and most existing visual recognition systems perform word level classification on short videos, in which speakers utter single words. The Lip Reading in the Wild (LRW) dataset has the form of videos labeled with a word. Therefore, a straightforward solution would be a neural network which classifies videos (frame sequences) to words. The architecture of the designed and implemented deep classifier is presented in detail in Section 4.1.

Another way to deal with the lipreading problem is instead of predicting word labels for a sequence of frames, to predict a sequence of characters. The goal of this approach is to predict the characters which form the word spoken in the video. This viewpoint of lipreading is considered more general, since the visual recognition system is not confined by the number of different words in the dataset. However the development of such a system appears to be come challenging and less straightforward. In Section 4.2, an encoder-decoder deep visual recognition system that performs character level classification is proposed.

Both neural networks were implemented in Python's Tensorflow (Abadi et al. (2015)). The implementation details are discussed in Chapter 5.

4.1 Word Classification Neural Network

The deep lipreading classifier, is a neural network which could be described as a function

$$\boldsymbol{p} = NeuralNetwork(\boldsymbol{X})$$

where $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\}$ is the input sequence of frames (video data) and $\boldsymbol{p} \in [0,1]^{500}$ is a probability distribution over the five hundred possible words in the LRW dataset. Every frame $\boldsymbol{x}_t$ is considered to be a 112×112 standardized grayscale image of the mouth region. Therefore, the network's input $\boldsymbol{X}$ is a 4D tensor of size $T \times 112 \times 112 \times 1$ and $\boldsymbol{x}_i \in \mathbb{R}^{112 \times 112 \times 1}$. The size of the last dimension is one, since every image has a depth of one (grayscale) and $T = 29$ is the maximum number of frames in the sequence. However, since every video sample in the dataset has a different number of useful frames in which the target word is spoken, we have to extract first these frames and then apply padding for the rest time slots, so that every input sample has length T. The output distribution $\boldsymbol{p}$ is a vector of size 500, where each value p_i is the probability that the speaker is uttering the word i in the input

frame sequence, or in other words that the input sample belongs to class i. An example of a frame sequence before the application of image standardization is shown in Figure 4.1.

Figure 4.1: A frame sequence of length 14. Each frame is 112×112, grayscale and centered around the mouth region of the speaker. In this case $\boldsymbol{x}_1, \ldots \boldsymbol{x}_{14}$ correspond to the images above after applying image standardization, while $\boldsymbol{x}_{15}, \ldots, \boldsymbol{x}_T$ are filled in with zeros.

The neural network is composed of four smaller networks:

1. A 3D Convolutional neural network.
2. A 2D Convolutional neural network.
3. A Bidirectional Long short-term memory (LSTM) network.
4. A fully connected layer with a Softmax unit.

The block diagram of the neural network is illustrated in Figure 4.5, at the end of the section.

4.1.1 3D CNN

A 3D Convolutional neural network, which is the first component of the lipreading network, applies spatiotemporal convolution to the frame sequence in the input. Spatiotemporal convolutional layers are capable of capturing the short-term dynamics of the mouth region and extracting features in the time dimension.

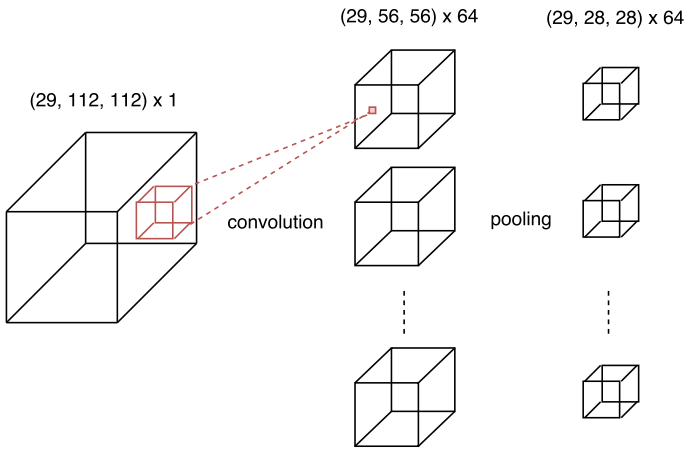

Figure 4.2: The 3D CNN with a convolutional and a max-pooling layer.

The 3D CNN we used in the word classification network is made up of one convolutional layer followed by a max-pooling layer. The 4D tensor $\boldsymbol{X} \in \mathbb{R}^{29\times112\times112\times1}$, that encloses the frame sequence of the mouth region, consists the input of the 3D CNN. The convolutional layer applies 64 filters to the input. Each filter has a set of weights of the form $5 \times 7 \times 7$ (time/width/height) and a bias. The stride is set to one for the time dimension and two for the two spatial dimensions, which means that the width and the height dimensions will be reduced in half. The result of the convolution operation is then followed by a ReLU function to form the hidden layer $\boldsymbol{H}^{3D} \in \mathbb{R}^{29\times56\times56\times64}$, where

$$\boldsymbol{H}^{3D} = \text{ReLU}(\text{conv3D}(\boldsymbol{X})).$$

Next, a max-pooling layer further reduces the spatial dimension in half, by using a $1 \times 2 \times 2$ window. The pooling operation is followed by batch normalization. A detailed description of the implemented batch normalization method is provided in Chapter 5. The output of the 3D CNN $\boldsymbol{O}^{3D}$ is a tensor of size $29 \times 28 \times 28 \times 64$. Finally, this tensor is unstacked in the time dimension to form a sequence $\boldsymbol{o}_1^{3D}, \ldots, \boldsymbol{o}_T^{3D}$, where each $\boldsymbol{o}_t^{3D}$ tensor forms an input tensor for the 2D CNN in the time steps $t = 1, \ldots, T$, with $T = 29$.

4.1.2 2D CNN

A Convolutional neural network is the next building block of the lipreading system. CNNs have been widely used for lipreading since they constitute a robust visual feature extraction method. The output sequence $\boldsymbol{o}_1^{3D}, \ldots, \boldsymbol{o}_T^{3D}$ from the 3D CNN is passed to the CNN. Then for each time step t the feature $\boldsymbol{o}_t^{3D}$ from the sequence is processed to form the visual feature $\boldsymbol{o}_t^{2D}$:

$$\boldsymbol{o}_t^{2D} = \text{CNN}(\boldsymbol{o}_t^{3D}), \quad \text{for } t = 1, ..., T \text{ and } \boldsymbol{o}_t^{2D} \in \mathbb{R}^{512}$$

At this point, is important to highlight that the same CNN is applied to features $\boldsymbol{o}_t^{3D}$, for all input time steps t. For an arbitrary time step t, the tensor $\boldsymbol{o}_t^{3D} \in \mathbb{R}^{28\times28\times64}$ is the CNN's input and $\boldsymbol{o}_t^{2D} \in \mathbb{R}^{512}$ is its output. The 2D CNN can be also perceived as multiple parallel 2D CNNs with shared weights. The structure of the Convolutional neural network is demonstrated in Figure 4.3.

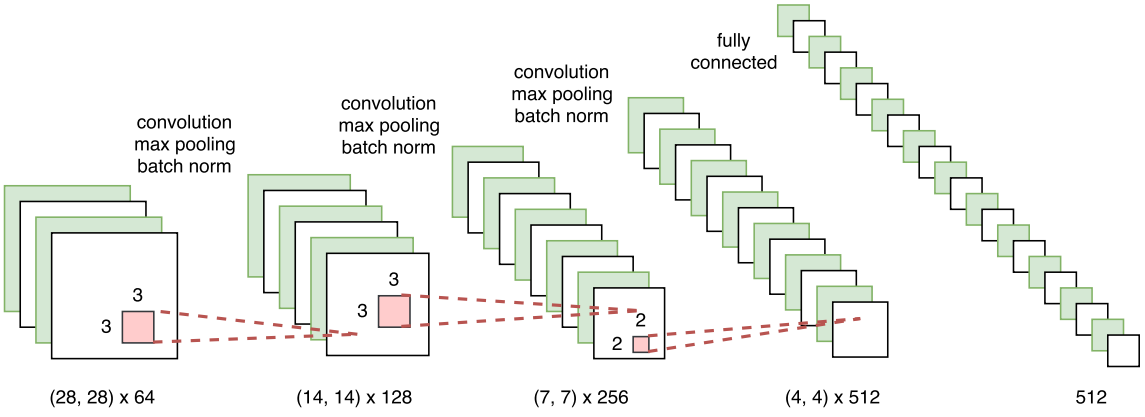

Figure 4.3: The 2D CNN architecture.

In first hidden layer, a convolution operation is applied in the input $\boldsymbol{o}_t^{3D} \in \mathbb{R}^{28\times28\times64}$ using a filter with 3×3 spatial dimensionality, 64 input channels and 128 output channels. Then,

a ReLU activation function is applied and a max-pooling function with a 2×2 frame reduces the width and height in half. Next, batch normalization is performed. The result is the hidden state $\boldsymbol{H}^{(1)} \in \mathbb{R}^{14\times14\times128}$. The second hidden layer performs similar operations to $\boldsymbol{H}^{(1)}$ in order to obtain the hidden state $\boldsymbol{H}^{(2)} \in \mathbb{R}^{7\times7\times256}$. In the third hidden layer the spatial dimension of the convolutional filter is reduced to 2×2. Again, convolution, ReLU, max-pooling and batch normalization take place in this layer and the result is a tensor with the third hidden state $\boldsymbol{H}^{(3)} \in \mathbb{R}^{4\times4\times512}$. Subsequently, this tensor which is a 3-dimensional object is flattened to a vector $\boldsymbol{h}^{(3flat)} \in \mathbb{R}^{4\cdot4\cdot512}$. The final layer is a fully connected layer which drops the size of the CNN output to 512 features:

$$\boldsymbol{o}_t^{2D} = \boldsymbol{W}\boldsymbol{h}^{(3flat)} + \boldsymbol{b}, \quad \boldsymbol{o}_t^{2D} \in \mathbb{R}^{512}.$$

The output $\boldsymbol{o}_t^{2D}$ of the CNN at time step t constitutes a visual feature vector, which contains spatial and short term temporal features of the frame $\boldsymbol{x}_t$ from the neural network's input sequence. After computing the CNN output for all time steps, the feature sequence $\boldsymbol{o}_1^{2D}, \ldots, \boldsymbol{o}_T^{2D}$ is transfered to a Bidirectional LSTM.

4.1.3 BiLSTM

Recurrent neural networks and especially LSTMs have been repeatedly used as the core network of many recently developed lipreading systems. The ability of LSMTs to model long term temporal dynamics is the main reason they have been employed in lipreading and speech recognition systems in general. Bidirectional LSTMs are able to capture dependencies on data-points from both previous and future time steps in their input sequence.

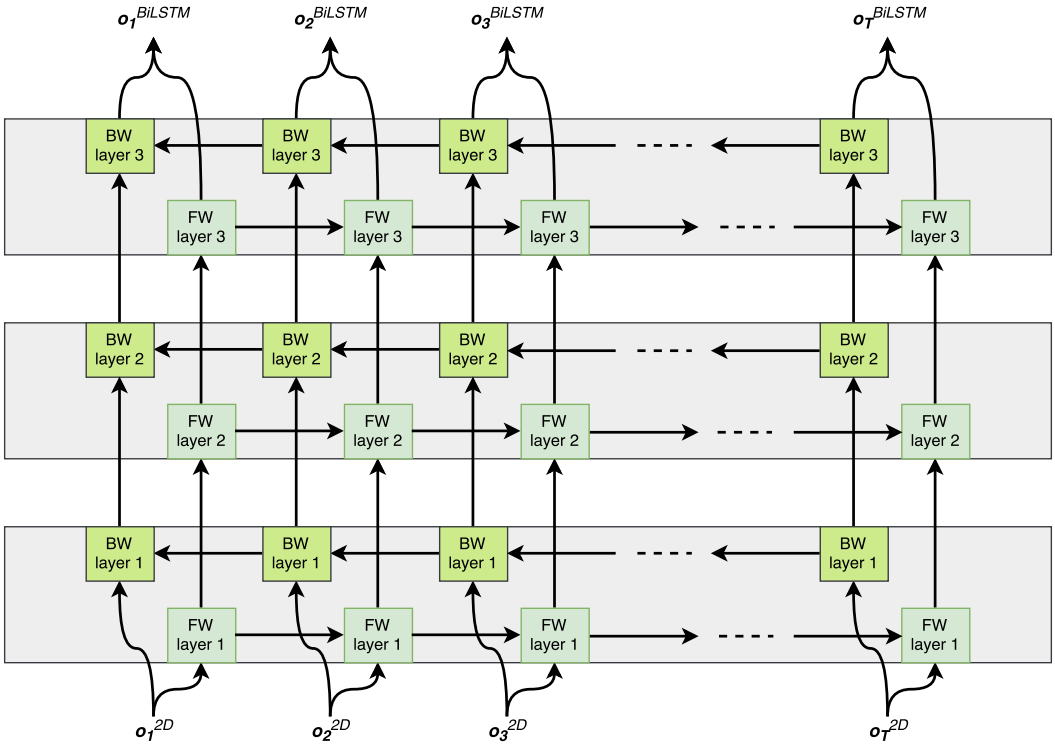

Figure 4.4: The three layered Bidirectional Long short-term memory architecture, unfolded in time.

The BiLSTM component we have chosen for the lipreading system is shown in Figure 4.4. It consists of three hidden layers, where the output of each layer is forwarded as input to the next one. Each hidden layer has a size of 256 units. The input sequence $\boldsymbol{o}_1^{2D}, \ldots, \boldsymbol{o}_T^{2D}$ to the BiLSTM network is obtained by the 2D CNN of the previous stage. Each hidden layer consists of a forward LSTM cell and a backwards LSTM cell, which model the past and future temporal dependencies respectively. The forward part of the BiLSTM consumes the input sequence and produces a feature sequence $\boldsymbol{o}_1^{FW}, \ldots, \boldsymbol{o}_T^{FW}$, which is essentially the output of the final hidden forward layer. The backward LSTM consumes the input sequence in reverse time order and outputs the feature sequence $\boldsymbol{o}_1^{BW}, \ldots, \boldsymbol{o}_T^{BW}$. Then the total output sequence $\boldsymbol{o}_1^{BiLSTM}, \ldots, \boldsymbol{o}_T^{BiLSTM}$ is the concatenation of the two individual sequences:

$$\boldsymbol{o}_t^{BiLSTM} = [\boldsymbol{o}_t^{FW^\top}; \boldsymbol{o}_t^{BW^\top}]^\top, \text{with } \boldsymbol{o}_t^{FW}, o_t^{BW} \in \mathbb{R}^{256} \text{ and } \boldsymbol{o}_t^{BiLSTM} \in \mathbb{R}^{512}, t = 1, \ldots, T.$$

The output sequence $\boldsymbol{o}_1^{BiLSTM}, \ldots, \boldsymbol{o}_T^{BiLSTM}$ is considered to be a sequence of features, where each visual feature vector $\boldsymbol{o}_t^{BiLSTM}$ corresponds to a higher level representation of the raw video frame $\boldsymbol{x}_t$. However $\boldsymbol{o}_t^{BiLSTM}$ holds information from the entire input video, since it encapsulates short and long term temporal dependencies.

4.1.4 Fully Connected Layer with Softmax

At this point, we have obtained a sequence of features $\boldsymbol{o}_1^{BiLSTM}, \ldots, \boldsymbol{o}_T^{BiLSTM}$, which represents the input video with the mouth region of a speaker uttering one word. Our goal is to use this sequence to predict a class label among 500 possible words. Since this is a classification task, first we have to produce a distribution on the class labels. Therefore, every vector $\boldsymbol{o}_t^{BiLSTM}$ is transformed to a new vector using linear fully connected layer:

$$\boldsymbol{o}_t^{FC} = \boldsymbol{W}_{FC}\boldsymbol{o}_t^{BiLSTM} + \boldsymbol{b}_{FC}, \text{ where } \boldsymbol{o}_t^{FC} \in \mathbb{R}^{500}, t = 1, \ldots, T.$$

Then, we apply a Softmax function to obtain the desired distribution vector on classes, one for every time step:

$$\boldsymbol{p}_t = \text{Softmax}(\boldsymbol{o}_t^{FC}), \quad t = 1, \ldots, T,$$

where

$$\sum_{i=1}^{500} p_{ti} = 1, \quad t = 1, \ldots, T.$$

The next stage would be to combine the distributions $\boldsymbol{p}_t$ for all time steps t and compute a mixed distribution $\boldsymbol{p}$. In order to do that we define the weights $v_t, t = 1, \ldots, T$ such that

- $v_t = 1$ if $t \leq t_0$,
- $v_t = 0$ if $t > t_0$,

where t_0 is the actual number of frames (without padding) in the input sequence of the network $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$. This means that $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t_0}$ hold the video frames, while $\boldsymbol{x}_{t_0+1}, \ldots, \boldsymbol{x}_T$ are padding frames, set to zero. Therefore, for every time step t, if $\boldsymbol{x}_t$ holds an actual frame

from the video, the weight v_t has to be one and zero in case that $\boldsymbol{x}_t$ is used for padding. Then the mixed distribution is given by the weighted sum of the individual distributions:

$$\boldsymbol{p} = \frac{\sum_{t=1}^{T} v_t \boldsymbol{p}_t}{\sum_{t=1}^{T} v_t}, \quad \boldsymbol{p} \in [0,1]^{500}$$

The distribution $\boldsymbol{p}$ on word labels is the output of the entire neural network, while the predicted label y for the input video is the word with the highest probability according to the model: $y = \text{argmax}(\boldsymbol{p})$, with $y \in \{1, \ldots, 500\}$.

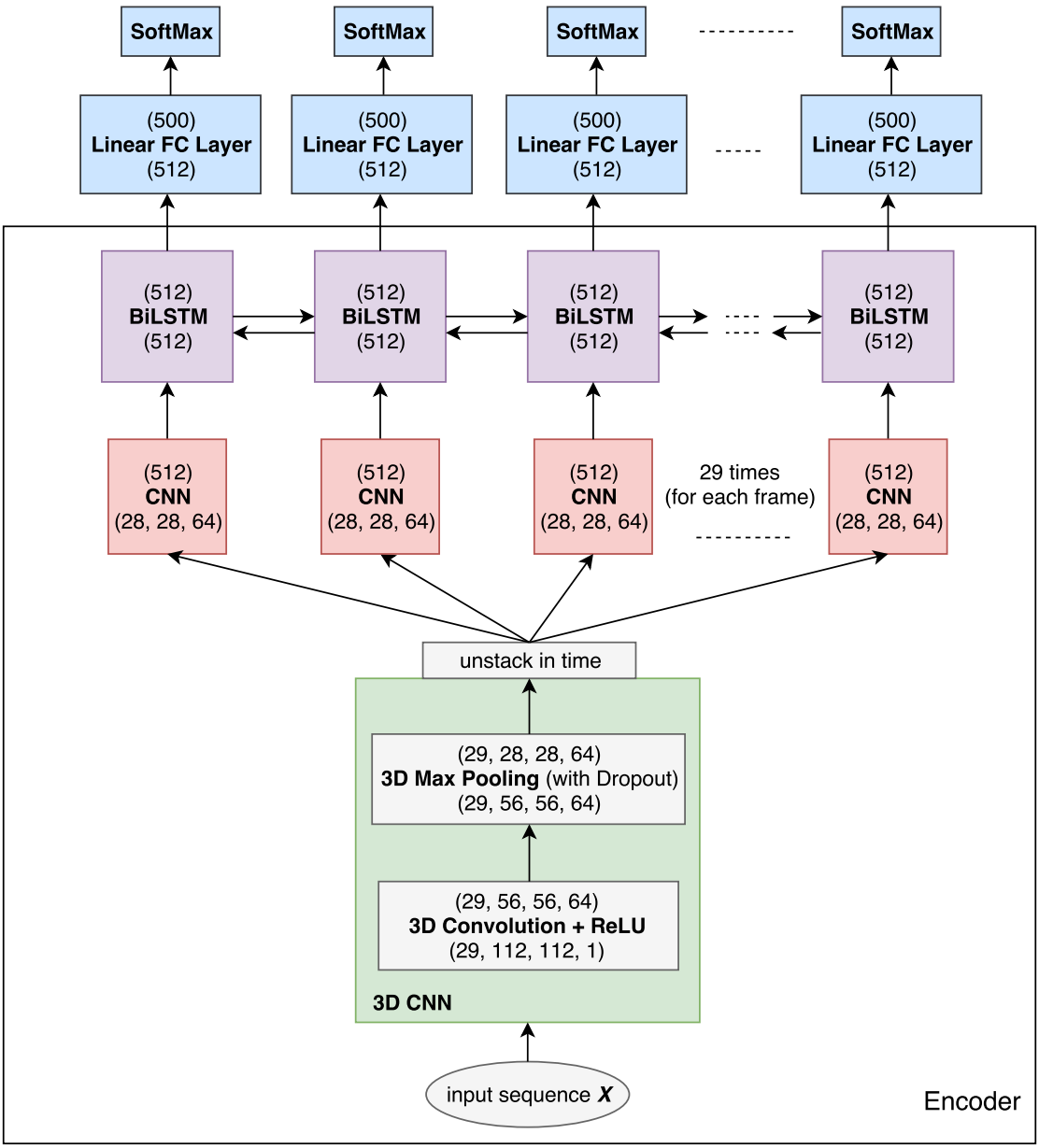

Figure 4.5: The block diagram of the word classification neural network.

4.2 Character decoding Neural network

Instead of treating the lipreading problem for Lip Reading in the Wild dataset as a task of assigning word labels to videos, a different approach which was inspired by Chung et al. (2016) could be followed. Classification can be performed in the character level by building a neural network which outputs a sequence of probability distributions on characters from the English alphabet. Similarly to the first proposed network in Section 4.1, this network receives a sequence of frames from the video but outputs a sequence of probability distributions:

$$\boldsymbol{P} = NeuralNetwork(\boldsymbol{X})$$

where $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\}$ and $\boldsymbol{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{T_{out}}\}$. Every distribution $\boldsymbol{p}_t$, $t = 1, \ldots, T_{out}$ in the output sequence is a 29-dimensional vector, since we have 26 letters from the alphabet and three special characters <sos>, <eos> and <pad>. The length of the output sequence is $T_{out} = 13$, since the longer word in the LRW dataset in 12 characters long. The one extra spot in the end of the sequence is a always set to <eos>. The sequence $\{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{T_{out}}\}$ is used to extract the predicted word, by choosing in every output time step the character with the highest probability:

$$y_t = \text{argmax}(\boldsymbol{p}_t) \in \{1, \ldots, 29\}, \quad t = 1, \ldots, T_{out},$$

where

$$\boldsymbol{p}_t = [p_{t1}, \ldots, p_{t29}]^\top \in [0, 1]^{29}$$

is the probability vector for output step t. Next, the predicted word is computed by using the sequence $y_1, \ldots, y_{T_{out}}$ and the mapping:

$$1 \rightarrow \text{'a'}, 2 \rightarrow \text{'b'}, \ldots, 26 \rightarrow \text{'z'}, 27 \rightarrow < \text{sos} >, 28 \rightarrow < \text{eos} > \text{ and } 29 \rightarrow < \text{pad} > .$$

The input frame sequence $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\}$ is constructed in the same way as described in Section 4.1, such that every element $\boldsymbol{x}_i$ in the sequence is a grayscale standardized image in the space $\mathbb{R}^{112 \times 112 \times 1}$.

The proposed neural network consists of two blocks:

- An image Encoder.
- A character Decoder.

4.2.1 Image Encoder

This part of the neural network is responsible for extracting high level visual features. For this purpose the encoder network from the word classification neural network from Section 4.1 is utilized. As can be seen in Figure 4.5, if we remove the final fully connected layer with the Softmax unit, the word classifier we have already built can be perceived as a feature extraction network. Then, the encoder receives the input sequence of frames $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\}$ and produces the sequence of features $\boldsymbol{o}_1^{BiLSTM}, \ldots, \boldsymbol{o}_T^{BiLSTM}$:

$$\boldsymbol{O}^{BiLSTM} = Encoder(\boldsymbol{X})$$

with

$$\boldsymbol{O}^{BiLSTM} = \{\boldsymbol{o}_1^{BiLSTM}, \ldots, \boldsymbol{o}_T^{BiLSTM}\} \text{ and } \boldsymbol{o}_i^{BiLSTM} \in \mathbb{R}^{512}.$$

These extracted features from the frame sequence form the input of the decoder network.

4.2.2 Character Decoder

The encoder is immediately followed by the decoder network. This neural network is responsible for producing the sequence of distributions over characters:

$$\boldsymbol{P} = Decoder(\boldsymbol{O}^{BiLSTM}).$$

The stream of distributions $\boldsymbol{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{T_{out}}\}$ coincides with the output of the whole character decoding neural network. Moreover, the sequence of predicted tokens (character labels) $\boldsymbol{y} = \{y_1, \ldots, y_{T_{out}}\}$ is computed by taking the index with the maximum probability: $y_t = \text{argmax}(\boldsymbol{p}_t), t = 1, \ldots, T_{out}$.

The character decoder is essentially an LSTM transducer with an attention mechanism network, which was described thoroughly in Subsection 2.1.4. The network's architecture is illustrated in Figure 2.9. The LSTM transducer, which is the front part of the decoder models the dependence of the currently predicted character with all previous characters. It is an LSTM network with three hidden layers. In the output time step t, the hidden state of this unidirectional LSTM is:

$$\boldsymbol{H}_t^{LSTM} = LSTM(y_{t-1}, \boldsymbol{c}_{t-1}, \boldsymbol{H}_{t-1}^{LSTM})$$

where $\boldsymbol{H}_t^{LSTM} = \{\boldsymbol{h}_t^{LSTM(1)}, \boldsymbol{h}_t^{LSTM(2)}, \boldsymbol{h}_t^{LSTM(3)}\}$ are the values of the three hidden layer states. Each hidden layer state is a 256-dimensional vector and the output of the LSTM is considered to be the third hidden layer: $\boldsymbol{o}_t^{LSTM} = \boldsymbol{h}_t^{LSTM(3)}$. The input to the LSTM transducer is the context vector $\boldsymbol{c}_{t-1} \in \mathbb{R}^{512}$ from the previous output time step concatenated with the character $y_{t-1} \in \{1, \ldots, 29\}$. During prediction y_{t-1} is the predicted character in the previous step, while during training this character can be either the ground truth from the dataset y_{t-1}^{target} or the prediction y_{t-1}. More details of the LSTM input during training are given in Chapter 6.

The output $\boldsymbol{o}_t^{LSTM}$ of the LSTM transducer along with the output sequence from the encoder network are transfered to the attention mechanism, which produces the context vector $\boldsymbol{c}_t$:

$$\boldsymbol{c}_t = Attention(\boldsymbol{o}_t^{LSTM}, \boldsymbol{O}^{BiLSTM}).$$

Here, we use a variation of the attention mechanism described in Subsection 2.1.4 with one and zero weights in the input sequence $\boldsymbol{O}^{BiLSTM}$. At output time step t, for each feature vector in the sequence $\boldsymbol{o}_1^{BiLSTM}, \ldots, \boldsymbol{o}_T^{BiLSTM}$, the attention mechanism calculates a scalar

$$e_{ti} = \boldsymbol{w}^\top \tanh(\boldsymbol{W}\boldsymbol{o}_t^{LSTM} + \boldsymbol{V}o_i^{BiLSTM}), \quad i = 1, \ldots, T,$$

and the attention weights which reflect the attention concept are computed as

$$a_{ti} = \frac{v_i \exp(e_{ti})}{\sum_{j=1}^{T} v_j \exp(e_{tj})}, \quad i = 1, \ldots, T,$$

forming the vector $\boldsymbol{a}_t = [a_{t1}, \ldots, a_{tT}]^\top$.

The weights $\boldsymbol{v} = [v_1, \ldots, v_T]^\top$ are the same with these presented in Section 4.1, where

- $v_i = 1$ if $i \leq t_0$,
- $v_i = 0$ if $i > t_0$

and t_0 is the number of elements in the input sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ of the lipreading network which contain actual frames from the input video and are not used for padding. Then the context vector at output time step t is computed as the weighted sum of the features extracted by the encoder network:

$$\boldsymbol{c}_t = \sum_{i=1}^{T} a_{ti} \cdot \boldsymbol{o}_i^{BiLSTM}$$

In the next stage of the network, the LSTM transducer's output is concatenated with the context vector. The result is then processed by a Multilayer perceptron with a single hidden layer and an output layer with a Softmax unit. The result is a probability distribution vector $\boldsymbol{p}_t \in [0, 1]^{29}$ given by

$$\boldsymbol{p}_t = \text{Softmax}(MLP([\boldsymbol{c}_t^\top ; \boldsymbol{o}_t^{LSTM^\top}]^\top))$$

or

$$\boldsymbol{p}_t = \text{Softmax}(\boldsymbol{W}_{MLP}(ReLU(\boldsymbol{W}_{MLP}^{hid}[\boldsymbol{c}_t^\top ; \boldsymbol{o}_t^{LSTM^\top}]^\top + \boldsymbol{b}_{MLP}^{hid})) + \boldsymbol{b}_{MLP}).$$

After repeating this process for all output time steps $t = 1, \ldots, T_{out}$, we obtain the sequence $\boldsymbol{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{T_{out}}\}$, which forms the output of the the entire character decoding neural network.

The block diagram of this second lipreading network is illustrated in Figure 4.6.

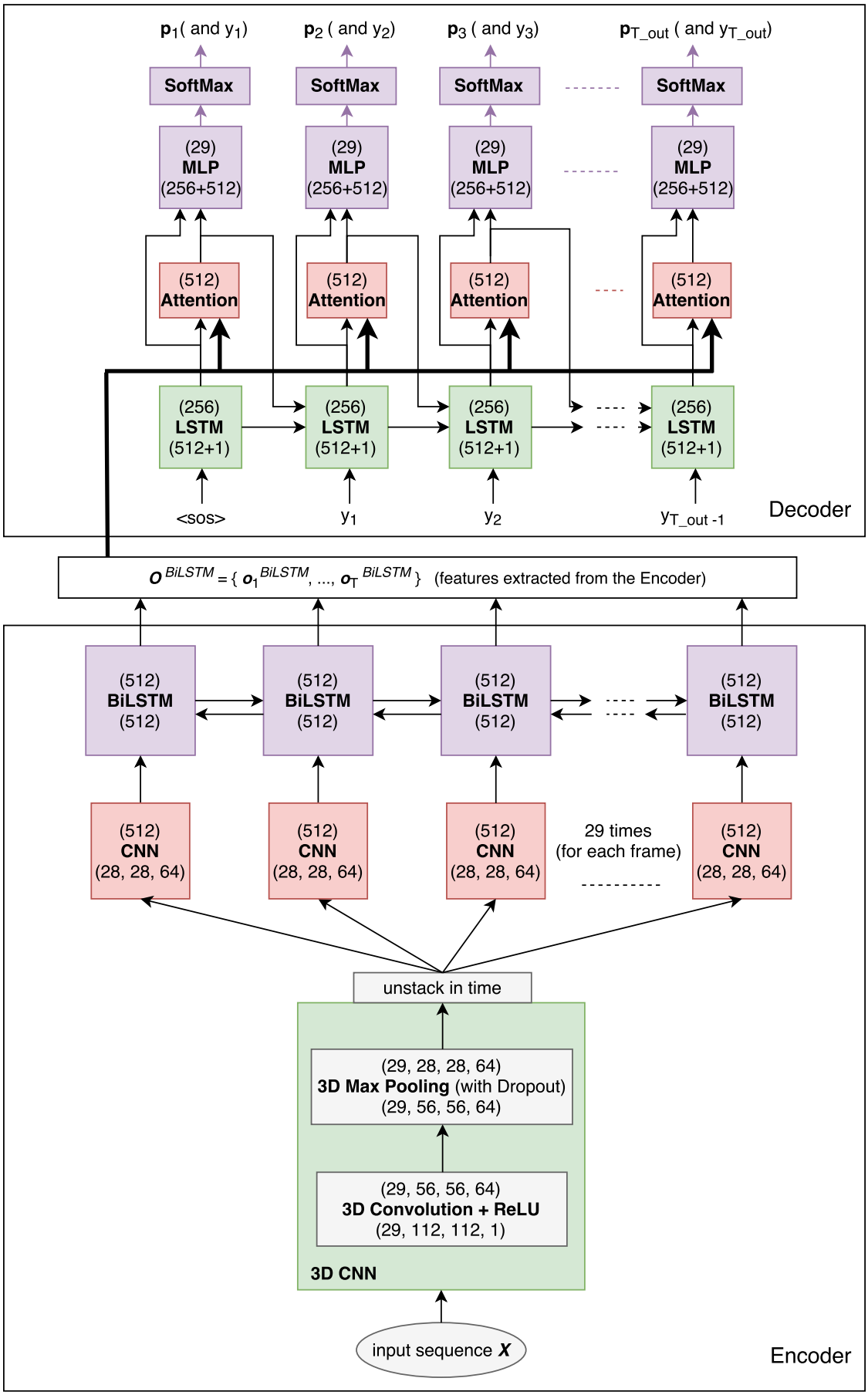

Figure 4.6: The block diagram of the character decoding neural network.

Chapter 5

Tensorflow Implementation

In Chapter 4 we presented two different lipreading systems. The first one is a neural network for word classification, while the second one is a character decoding neural network. Both networks where implemented in Tensorflow (Abadi et al. (2015)).

5.1 Tensorflow

Tensorflow is widely used for machine learning and deep neural network applications and was originally developed by researchers and engineers working on the Google Brain Team. Tensorflow is an open source library, where computation is performed using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent tensors transfered between nodes. Tensors could be defined as organized multidimensional arrays of numerical values. Tensorflow has APIs for constructing and executing Tensorflow graphs in several languages such as Python, Java, C++ and Go. As mentioned in Tensorflow's website "The Python API is at present the most complete and the easiest to use, but other language APIs may be easier to integrate into projects and may offer some performance advantages in graph execution". Therefore, the source code for the entire project was written in Python. Data flow graphs in Tensorflow can be deployed in one or more CPUs or GPUs, hence for performance efficiency both lipreading systems were trained and evaluated on a Nvidia Tesla K80 GPU.

5.2 Lipreading System Architecture

5.2.1 The Reader Class

The Lip Reading in the Wild (LRW) dataset consists of almost half a million training samples and a total size size of approximately 71 Gigabytes. Each sample is a .mp4 video with its corresponding metadata text file. Since .mp4 files are compressed, the actual size of the raw images contained in the video is expected to be larger than the size of the .mp4 file. Therefore, it would be obviously impossible to fit the whole dataset in the DRAM of the machine.

For this reason, the most straightforward solution would be to read each time only a few

files/video samples from the disk, process them by the neural network and then deallocate them from the memory. Moreover, this approach fits well together with the concept of using mini batches of data to train neural networks. Instead of using the entire training dataset in a single optimization step during the training procedure of a neural network, only a small batch (subset) of samples is propagated through the network. Therefore, one solution could be to create a reader that generates a batch of samples from the dataset and subsequently sends this batch to the network. Furthermore, one could notice that the process of reading .mp4 files from the disk and converting them to the appropriate form to generate a mini batch could be parallelized to an extend, even if reading the disk is a serial process.

The entire functionality of creating a batch of samples was implemented by the Reader class in Python. This class contains the methods:

- __init__(): This is the constructor of the class. It receives a boolean argument indicating whether the Reader operates on the training or the test dataset.
- generate_all_sample_paths(): When this method is invoked on a Reader object, it creates four lists, where the first one contains the full paths of all video files in the dataset and the second one contains the paths of all metadata files. Additionally, a list with the word labels that correspond to each video path is generated along with a list of strings with the word spoken in each video. All these four lists are then shuffled while we are keeping an alignment, such that the information on every single video appears in the same index of the lists. These lists are fields of the Reader class.
- _generate_video_sample(): This method returns a sample from the dataset in the appropriate form for the neural network. Since we implement a reading mechanism we multiple execution Threads, we use a lock on the lists created by the generate_all_sample_paths() method, so that each Thread pops a unique video and metadata path from the lists. In this way, each Python Thread running this method gets a random video and metadata path along with the word label information which has not been extracted by any other Thread. Python's locks guarantee the uniqueness of input samples to the neural network during training and evaluation. Therefore, first a lock is acquired and the full path of the .mp4 file and its metadata file along with the word label and the sting containing the word is popped out of the lists. Then, the lock is released and both the video and the text file are opened. We use the skvideo.io library in order to extract the raw frames from the .mp4 file, since there is no reader in Tensorflow for video manipulation. Afterwards, the information in the metafile is used to find the first and the last frame where the target word is spoken. In addition, the mouth region is extracted in a naive way, since we crop all images to the same position and then we convert them to grayscale frames. Next, all these frames from the first one to the last one are stacked together to a numpy array and zero padding is employed, such that all frame sequences are of the same length $T = 29$. Then, the sting that contains the spoken word in the video is transformed to an integer array of length $T_{out} = 13$, where each integer corresponds to a character in the string. Finally, if the sting is shorter that 12 characters, the integer that corresponds to <pad> (padding token) is placed in the rest indexes of the array, while the last index takes the value of <eos> (end of string). The actual (without padding) length of the frame sequence and the label of the word in the video (value between 1 and 500) are also returned by this method, along with the numpy array with the data and the array with the character labels.

- get_input_batch(): When this method is called by a Thread, it returns Q samples. Since more than one Threads run this method at the same time, this method is designed to return a part of the batch, or $Q = batch\_size/num\_threads$ data samples. The method first ensures that there are enough video sample paths left which have not yet been extracted form the list. This is achieved by using a lock to read and write the number of remaining paths in the list of video paths. In this way reading and writing the number of remaining paths becomes an atomic operation. Next, the method that returns a single sample _generate_video_sample() is called Q times and all these Q samples obtained are stacked together. Additionally, a boolean value is returned, which indicates whether or not we are processing parts of the last batch in the dataset.

At this point we have a Python class to read segments of a batch of samples. First, one object of the Reader class should be instantiated and the method generate_all_sample_paths() should be invoked on this object. Then we should create a number of Threads ($num\_threads$), each one running the get_input_batch() method of the Reader object. The next step would be to describe the **batch producer Python Process**, which generates batches, and the **training (or evaluation) Python Process**, which receives batches and feeds them to the network.

5.2.2 An Overview of the Lipreading Application

Tensorflow provides a Queue mechanism for asynchronous computation. A typical architecture is to use a Queue to prepare inputs for a neural network model where:

- Multiple Threads push samples from the dataset in the Queue.
- A single Thread extracts a number of samples from the Queue, enough to form a batch, and forwards the batch to the neural network for the training of evaluation operation.

In order to generate Treads in Python, the Threading library is utilized. However, the internals of the main Python interpreter, CPython, negate the possibility of true multi-threading due to a process known as the Global Interpreter Lock (GIL). The GIL is necessary because the Python interpreter is not thread safe. There is a globally enforced lock when trying to safely access Python objects from within threads. At any one time only a single thread can acquire a lock for a Python object.

The proposed Tensorflow architecture with one Queue, many enqueue Threads and one dequeue Thread was initially implemented. However, as we expected the performance with regard to the execution time was very poor, since the enqueue Threads were preempted very frequently and the main dequeue Thread was scheduled for execution. This back-and-forth and the frequent context switches introduced a great delay in the execution of the program.

In order to actually parallelize the procedures of batch creation and execution of the training (or evaluation) operation in the network, we can instead use the Multiprocessing library of Python. We can disengage the two procedures by designing our Tensorflow implementation as follows:

- A Python Process generates batches of samples, which constitute the neural network's input, and writes them in a multiprocessing queue.
- Another Python Process, which performs the training or evaluation operation, receives the batches from the queue and feeds them to the model of the neural network.

The block diagram of the lipreading application with the batch producer Process, the training (or evaluation) Process and a multiprocessing queue is shown in Figure 5.1.

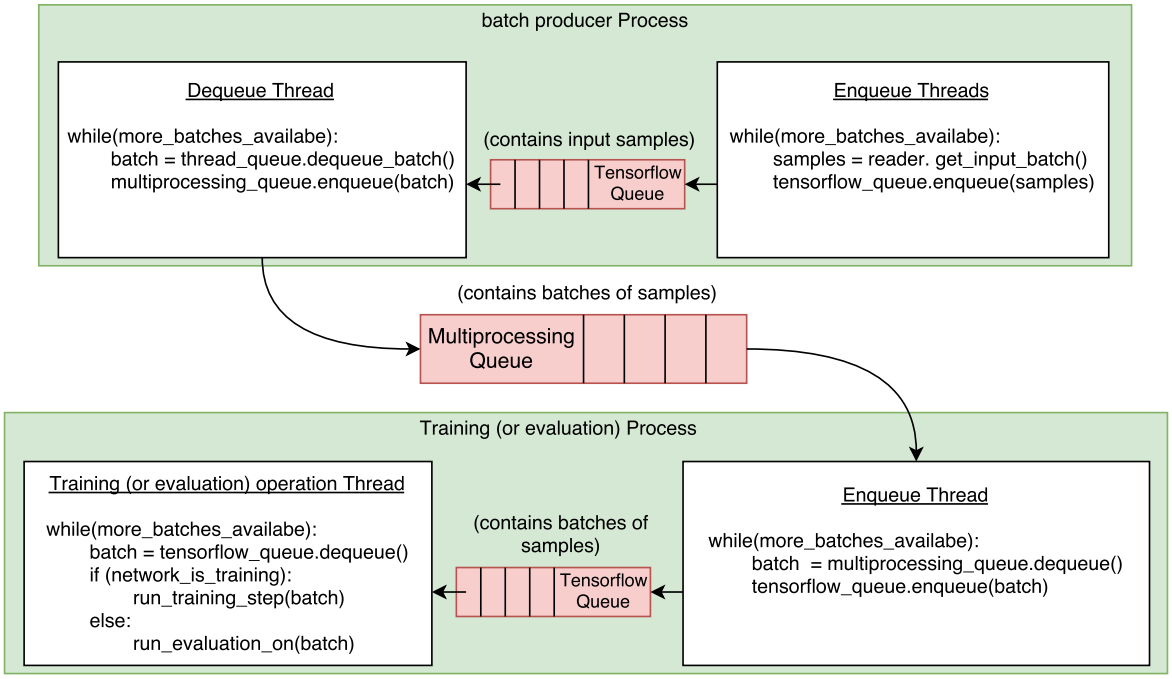

Figure 5.1: The batch producer Process writes mini batches to the multiprocessing queue, while the training (or evaluation) process removes them from the queue and feeds them to the network.

Batch producer Process

The batch producer Process consists of multiple Threads, each one invoking the get_input_batch() method on a Reader Python object and then pushing the Q samples in a Tensorflow FIFO queue. The queue that dequeues elements in first-in first-out order is implemented by the class tf.FIFOQueue() and since each element in the queue is an input sample, the method invoked to enqueue Q samples is enqueue_many().

When there are enough samples to form a batch, another Thread of the same Process extracts $batch\_size$ samples from the queue. This is done by defining a dequeue operation on the Tensorflow queue using the method dequeue() and the tf.train.batch() function on the dequeue operation to extract $batch\_size$ samples. Subsequently, the same Thread pushes the batch to another queue from the Multiprocessing library, which serves as a communication channel between the two Processes. This queue is an object of the Multiprocessing.Queue() class. The put() method of the class is invoked to place an element, which is a batch, in the multiprocessing queue.

In general, Python Treads are useful when a Process is I/O intensive and do not benefit CPU intensive tasks. Since the batch producer Process reads video samples from the hard disk it makes sense to deploy multiple Threads for this operation. However, we need only one Thread to read the intraprocess Tensorflow queue and remove batches in order to send them the training (or evaluation) Process. This dequeue Thread acts as an intermediate node, so that the enqueue Threads do not keep the multiprocessing queue constantly occupied by

writing the input samples there directly.

Training (or evaluation) Process

The training (or evaluation) Process is composed of two Threads. The first one is responsible for removing batches from the multiprocessing queue with the get() method and placing them to an intraprocess Tensorflow queue, which is a tf.FIFOQueue().

The second Thread extracts batches from the Tensorflow queue with the dequeue() method and transfers them to the neural network. At this point, each sample in the batch is a sequence of grayscale frames around the mouth region of the speaker. However, before feeding a batch to the neural network, this Thread performs a preprocessing operation on data. Image standardization is a commonly used technique when Convolutional neural networks are used to extract visual features. This preprocessing operation creates more uniform images, a desired property in CNNs, since the same sets of weights (filters) are applied to different regions of the image. In this way gradients are kept in control during optimization. Standardization is a linear transformation which scales an image to have zero mean and unit norm. If $\boldsymbol{x} = [x_{ij}]$ is an image, then each pixel is updated to the new value

$$\frac{(x_{ij} - \mu)}{\sigma'}, \text{ with } \sigma' = \max(\sigma, \frac{1}{\sqrt{num\_pixels}})$$

where μ is the mean pixel value and σ is the standard deviation of the pixels across the image. This operation is performed in Tensorflow with tf.image.per_image_standardization(). After performing the standardization and depending on the operation we want to apply on the neural network, this Thread uses a batch to execute an optimization step during training, or evaluate the predictive performance of the network during evaluation.

Summarizing, the CPU and GPU intensive task, which is the training or evaluation operation on the network, is performed by one Thread, while the other Thread fetches the mini batches of samples from the Multiprocessing queue. The training and evaluation Processes are described in more detail in Section 5.4.

5.3 Neural Networks Implementation

In the previous section we described the higher level architecture of the lipreading application. We focused our attention on the reading procedure, where mini batches of samples from the training or test set are generated and transfered to the desired lipreading network.

However, the core of our lipreading system are the two models of the neural networks presented in Chapter 4. Both the training and evaluation Process use these two models to execute operations on the networks. We describe the model of each network in Python as a function. In Tensorflow, a neural network is described as a forward model, where input values are propagated through the components of the network and are eventually transformed to output values. We do not have to describe the back propagation steps, since the Tensorflow API provides functions to compute the derivatives of the loss function with respect to the trainable variables and also brings a variety of already implemented optimization methods.

5.3.1 Word Classification Network in Tensorflow

```
def WordClassificationNeuralNetwork(data, labels, length, num_words, is_training):
    ...
    return loss, logits
```

The Python function which describes the model of the first proposed lipreading network is shown above. The input values are:

- **data:** This is a 5D tensor of size $[batch\_size, T, image\_width, image\_height, 1]$, where $T = 29$ is the number of input time steps (the maximum number of frames in a sequence) and the size of each frame is $image\_width \times image\_height \times 1$ or in our case $112 \times 112 \times 1$. Obviously, this object encloses a batch of input frame sequences for the network.
- **labels:** This is a 1D tensor (vector) of size $[batch\_size]$. It contains the target word labels for each sample in the batch, which are values in the set $\{0, \ldots, \text{num\_words} - 1\}$.
- **length:** A tensor of size $[batch\_size]$, where every entry is the actual length in time of the corresponding frame sequence in the batch.
- **num_words:** This is a scalar indicating the number of different spoken words in the dataset. This argument is useful in case we want to experiment with subsets of the LRW dataset.
- **is_training:** A boolean value which indicates whether the model will be used for training or evaluation.

The output values of the model are:

- **loss:** This is the total loss value in the batch, which is the sum of the weight decay loss and the cross entropy of the target distribution given from the labels and the predicted distribution computed by the network.
- **logits:** In general, a logit is the vector in which a Softmax function is applied to obtain a distribution. In our case, logits is a 2D tensor of size $[batch\_size, \text{num\_words}]$, which holds the logit vector for each sample in the batch. Therefore, if we apply the Softmax function on this tensor, we will obtain the predicted distribution on word labels for each sample in the batch.

The body of the function consists of the two components:

1. An encoding procedure, where visual features are extracted from the input batch. The encoder is implemented by a Python class with the name Image_Encoder which has an encode() method. When this method in invoked on a Image_Encoder, it computes the fields of the class lstm_outputs and lstm_final_state, which correspond to the outputs of the BiLSTM.
2. A fully connected layer, which is implemented with the Python class FullyConnected_layer and has the methods fc_layer() and loss(). When fc_layer() is invoked the all_logits field of the class FullyConnected_layer is calculated. Finally, the loss() method returns the total loss on the batch.

Therefore, the complete model is:

```
def WordClassificationNeuralNetwork(data, labels, length, num_words, is_training):
    encoder = Image_Encoder(data, length, is_training)
    encoder.encode()
    lstm_outputs = encoder.lstm_outputs
    fcl = FullyConnected_layer(lstm_outputs, labels, length, num_words, is_training)
    fcl.fc_layer()
    loss = fcl.loss()
    logits = fcl.all_logits
    return loss, logits
```

Image_Encoder class

The Image_Encoder class with the encode() method computes the output of the Bidirectional LSTM. When encode() is called, first the network's input is passed through a 3D CNN, then through multiple copies if the same 2D CNN and finally through a BiLSTM.

3D CNN: The input to the 3D CNN is a [$batch\_size$, T, 112,112, 1] vector, with $T = 29$. The 3D CNN first performs of a convolutional operation which is done with tf.nn.conv3d(). Then a ReLU activation function tf.nn.relu() is applied to the result of the convolution. In the next stage, we use max-pooling with Tensorflow's tf.nn.max_pool3d(). Finally, only in case we train the network, dropout is applied to the max-pooling layer using tf.nn.dropout(). Dropout is a regularization technique for reducing overfitting in neural networks. Individual nodes are either dropped out of the net with probability $1 - p$ or kept with probability p. The output of the 3D CNN is a tensor of size [$batch\_size$, T, 28, 28, 64], since the spatial dimensions are reduced in half twice.

2D CNN: The same 2D CNN is applied to every time step of the 3D CNN output. Therefore, the 2D CNN receives a tensor of size [$batch\_size$, 28, 28, 64] as input. The CNN is made up of three similar hidden layers, each one applying a convolution, an activation function, max-pooling and batch normalization. The 2D convolution is performed using the tf.nn.conv2d() function of Tensorflow. We use the tf.nn.bias_add() function to add the bias parameter of each filter. Then, the tf.nn.relu() is employed to compute the activation in the neurons. Subsequently, tf.nn.max_pool() is called on the activation and the result is directed to the batch_norm() function. This function performs global batch normalization to a 4D tensor, using the mean $\boldsymbol{\mu}$ and the variance $\boldsymbol{s}$ of the tensor across the first three dimensions. Moreover $\boldsymbol{\mu}, \boldsymbol{s}, \boldsymbol{\gamma}, \boldsymbol{\beta}$ are 1D tensors with the same size of the last dimension of the 4D tensor that is normalized. The result is

$$n_{bxyf} = \gamma_f \cdot \frac{(u_{bxyf} - \mu_f)}{s_f} + \beta_f$$

where $\boldsymbol{n} = [n_{bxyf}]$ is the globally normalized version of the 4D tensor $\boldsymbol{u} = [u_{bxyf}]$. The 1D tensors $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are trainable parameters of the network. During training the mean $\boldsymbol{\mu}$ and the variance $\boldsymbol{s}$ are computed using Tensorflow's tf.nn.moments() and their values are used to compute the normalized tensor with tf.nn.batch_normalization(). Then, we compute and store a moving average of the mean and variance:

$$\boldsymbol{moving\_\mu} = d \cdot \boldsymbol{moving\_\mu} + (1 - d) \cdot \boldsymbol{\mu}$$

and

$$\boldsymbol{moving\_s} = d \cdot \boldsymbol{moving\_s} + (1 - d) \cdot \boldsymbol{\mu},$$

where $d = 0.95$ is the decay rate. During inference (evaluation) the moving mean and variance are utilized for batch normalization, instead of the actual values from the data. Moreover, each one of the multiple copies of the same CNN used in different time steps has its own moving average values. After computing the three identical hidden layers with convolution, ReLU, max-pooling and batch normalization, the CNN input has been transformed to a 4D tensor or size [$batch\_size$, 4, 4, 512]. Dropout is performed on this tensor and then it is reshaped to a 2D tensor of size $[batch\_size, 4 \cdot 4 \cdot 512]$. Finally, a fully connected layer is calculated with tf.matmul() and tf.nn.bias_add(). The result is a 2D tensor of size [$batch\_size$, 512], which is also the output of the 2D CNN.

BiLSTM: At the point where the 2D CNN output has been computed for all $T = 29$ time steps, we stack the results in the time dimension and form a 3D tensor of size [$batch\_size$, T, 512]. This tensor is transfered to the Bidirectional LSTM. A basic LSTM cell is obtained in Tensorflow with the tf.contrib.rnn.BasicLSTMCell(). We choose a hidden size of 256 units. Moreover, we can create a cell with dropout using the wrapper tf.contrib.rnn.DropoutWrapper(). However, we employ an LSTM cell with dropout only during training and we use a basic one during evaluation. Next, tf.contrib.rnn.MultiRNNCell() is called twice, to obtain a forward and a backward LSTM with three hidden layers. The bidirectional LSTM is created with tf.nn.bidirectional_dynamic_rnn(), which performs the computation and returns the desired forward and the backward outputs. Each one of them is a tensor of size [$batch\_size$, T, 256] and their concatenation in the last dimension produces the final result, which is written to the field lstm_outputs of the Image_Encoder class. An important feature of the tf.nn.bidirectional_dynamic_rnn() is that we can pass as argument a tensor with the actual lengths in time of all samples in the batch. In this way, for each sequence in the batch, the BiLSTM is unrolled in time only for the given lengths.

The structure of the Image_Encoder class is presented below:

```
class Image_Encoder(object):
    def __init__(data, length, is_training):
        self._data = data
        ...
    def encode(self):
        def batch_norm(...):
            ...
        def CNN_3D(...):
            ...
        def CNN(...):
            ...
        def BiLSTM(...):
            ...
        CNN_3D_out = CNN_3D(self._data)
        CNN_input = tf.unstack(CNN_3D_out, axis=1)
        CNN_output = []
        for t in range(0, T):
            CNN_output.append(CNN(CNN_input[t]))
        lstm_output = BiLSTM(CNN_output)
```

FullyConnected_layer class

The FullyConnected_layer class receives the output tensor of size [batch_size, T, 512] from the Bidirectional LSTM and performs a linear transformation to the last dimension. This is done with Tensorflows tf.matmul(), which multiplies the input for each time step of size [batch_size, 512] with a matrix of trainable weights. After adding a trainable bias term, the result is a set of logits with size [batch_size, num_words]. We apply the same linear transformation to all time steps in the method fc_layer() and we obtain a 3D tensor of logits, which is stored to the field all_logits of the FullyConnected_layer class and has a size of [batch_size, T, num_words]. In case we use the entire LRW dataset, num_words $= 500$.

The loss() method first computes the average cross entropy in the entire batch across the time steps. If all_logits is the 3D tensor obtained by the fc_layer() method, then d^{pred} is another tensor of the same size with the predicted distributions for each sample and time step in the batch:

$$\boldsymbol{d}^{pred} = \text{Softmax(all\_logits)}$$

Moreover, if labels is a tensor of size [$batch\_size$], which contains the target word label for each sample in the batch, we can generate the 2D tensor of distributions $\boldsymbol{d}^{target}$ of size [$batch\_size$, num_words] by setting

- $d_{ij}^{target} = 1$, if $j =$ labels[i] and
- $d_{ij}^{target} = 0$, if $j \neq$ labels[i]

for $i = 0, \ldots, batch\_size - 1$ and $j = 0, \ldots,$ num_classes $- 1$.

Then the cross entropy loss is defined as

$$\text{loss} = -\sum_{i=0}^{batch\_size-1} \frac{\sum_{t=0}^{T-1} v_{it} \Big(\sum_{j=0}^{\text{num\_words}-1} d_{ij}^{target} \cdot \log(d_{itj}^{pred}) \Big)}{\sum_{t=0}^{T-1} v_{it}},$$

where $\boldsymbol{v} = [v_{it}]$ is a set of weights, with $v_{it} = 1$, if t is a time step with an actual video frame for sample i in the batch, or $v_{it} = 0$ in case t corresponds to a time step with padding for sample i in the batch.

This loss value is computed with the function tf.contrib.seq2seq.sequence_loss() in Tensorflow. Finally, the total loss is calculated by adding the weight decay loss from the trainable variables of the network, since we use L2 regularization. The function tf.nn.l2_loss() computes the squared L2 norm of a trainable variable of the network. Then, the total loss is defined as:

$$\text{total\_loss} = \text{loss} + \sum_{var}^{\text{trainable variables}} \lambda \cdot ||var||_2^2$$

where λ is the regularization hyperparameter. Therefore, each time we create a new trainable variable var, we compute the weight loss $\lambda \cdot ||var||_2^2$ and add it to a Tensorflow collection with tf.add_to_collection(). The cross entropy loss is added to the collection as well. Then, the total_loss is calculated with tf.add_n() applied on the result form tf.get_collection(). This total_loss value is finally returned by the loss() method of the FullyConnected_layer class.

5.3.2 Character Decoding Network in Tensorflow

```
def CharacterDecodingNeuralNetwork(data, labels, length, is_training):
    ...
    return loss, logits
```

The Python function which describes the model of the second proposed lipreading network is demonstrated above. The input values are:

- **data:** This is a 5D tensor of size $[batch\_size, T, image\_width, image\_height, 1]$, where $T = 29$ is the number of input time steps (the maximum number of frames in a sequence) and the size of each frame is $image\_width \times image\_height \times 1$ or in our case $112 \times 112 \times 1$.
- **labels:** This is a 2D tensor of size $[batch\_size, T_{out}]$, with $T_{out} = 13$ being the number of output time steps. It contains the target character labels for each sample in the batch and each output time step, which are values in the set $\{0, \ldots, num\_characters - 1\}$, with $num\_characters = 29$
- **length:** A tensor of size $[batch\_size]$, where every entry is the actual length in time of the corresponding frame sequence in the batch.
- **is_training:** A boolean value which indicates whether the model will be used for training or evaluation.

The output values of the model are:

- **loss:** This is the total loss value in the batch, which is the sum of the weight decay loss and the cross entropy of the target distribution given from the labels and the predicted distribution computed by the network.
- **logits:** This is a 3D tensor of size $[batch\_size, T_{out}, num\_characters]$, which holds the logit vector for each sample in the batch and each output time step. If we apply the Softmax function on this tensor, we will obtain the predicted distribution on character labels for each sample in the batch and each output time step.

This neural network is made up of an encoder and a decoder. Therefore, the main body of the Python function that implements the model is divided in two parts:

- The first one is a visual feature extraction mechanism. The Image_Encoder class, which was used in the previous network as well, implements the front part of the neural network. The method encode() of the class computes the output and the final state of the BiLSTM. This output is a sequence of features for each sample in the batch.
- The second part is a decoding procedure, which is implemented by the Python class Character_Decoder and has the methods decode() and loss(). The decode() method computes the logits and stores them in the class field all_logits. Next, the loss() method returns the total loss.

The function the implements the character decoding neural network is presented below:

```
def CharacterDecodingNeuralNetwork(data, labels, length, is_training):
    # 1.Encode
    encoder = Image_Encoder(data, length, is_training)
    encoder.encode()
    lstm_outputs = encoder.lstm_outputs
    lstm_final_state = encoder.lstm_final_state
    # 2.Decode
    decoder = Character_Decoder(lstm_output, labels, length,
                    lstm_final_state, is_training)
    decoder.decode()
    # Calculate loss and logits.
    loss = decoder.loss()
    logits = decoder.all_logits
    return loss, logits
```

Character_Decoder class

The Character_Decoder class implements the LSTM transducer with the attention mechanism and the Multilayer perceptron. The decode() method computes the logits tensor of size $[batch\_size, T_{out}, num\_characters]$, with $T_{out} = 13$ and $num\_characters = 29$.

LSTM transducer: When the decode() method is invoked on the Character_Decoder object, first a unidirectional LSTM with three hidden layers is created. Similarly with the BiLSTM in the encoder, a basic LSTM cell is obtained with Tensorflow's tf.contrib.rnn.BasicLSTMCell(). The size of the hidden state is set to 256 units. In case of training we use a wrapper to add dropout with tf.contrib.rnn.DropoutWrapper(). Then the multilayer LSTM cell object is instantiated with tf.contrib.rnn.MultiRNNCell(). Next, we generate the input for the first time step of the LSTM transducer. This is a tensor of size [$batch\_size$, 1 + 512], since for each sample in the batch, the LSTM input is the previous context vectors and a character label from the previous output time step. Since we have no context vectors for the first output time step, the output of the encoder from the first input time step (corresponding to the first video frame) is concatenated with the label of <sos> (start of sequence) to form the input of the LSTM transducer. The first hidden and cell state of the LSTM is set to the be the final hidden and cell state of the backwards LSTM in the encoder. Then, the LSTM is computed for all output time steps with tf.contrib.legacy_seq2seq.rnn_decoder(). This Tensorflow function takes as argument another function, called the loop function. The loop function receives the output of the LSTM from time step t and then computes and returns the input to the LSTM at step $t + 1$. The loop function is called T_{out} times by the tf.contrib.legacy_seq2seq.rnn_decoder().

Loop function: In the body of this function, first the attention_mechanism() function is called, which returns the context vectors, a tensor of size [$batch\_size$, 512]. Then, the output of the LSTM from time step t, which is a tensor of size [$batch\_size$, 256] along with the context vectors is passed to the MLP() function that computes the logits tensor for step t, of size [$batch\_size$, $num\_characters$], with $num\_characters = 29$. This tensor is stacked together with the logits tensors from all previous output time steps $t' < t$ and stored to all_logits. Therefore, when this loop function is called for the last time , the all_logits field of

the Character_Decoder class should hold the logits for all T_{out} output time steps. However, the main operation of the loop function is the creation of the next input for the LSTM, at step $t + 1$. This input in made up of the context vectors and a tensor with character labels from the output time step t. In case of network evaluation, we use the predicted character labels, which are acquired from the logits by applying a Softmax function and choosing the label with the maximum probability. However, in case of training, we choose the predicted labels with a probability p_{pred} or the ground truth labels from the dataset with probability $1 - p_{pred}$. This is an essential strategy during training, since if we chose to use every time the correct labels, the network would fail during evaluation when the ground truth labels are not available. Finally, the tensor with the character labels of size $[batch\_size, 1]$ and the context vectors are concatenated in the last dimension and returned by the loop function.

Attention mechanism: This is the part of the decoder network that produces the context vectors. It uses the entire output sequence of the encoder and the output of the LSTM transducer at each output time step to form the corresponding context vectors. The Python function attention_mechanism() implements the attention layer, which was described in Subsection 4.2.2.

MLP: A multilayer perceptron is the last part of the decoder. At each output time step t, it processes the concatenated tensor of size [batch_size, 256+512] that is acquired from the context vectors and the LSTM transducer output. The hidden layer reduces the MLP's input to a [batch_size, 128] tensor. First the input in multiplied with tf.matmul() and a bias is added with tf.nn.bias_add(). Then, tf.nn.relu() is applied to obtain the hidden state. During training, we also apply dropout on the hidden state. Finally, a linear transformation takes place to form the output layer of the MLP, which is a tensor of size [$batch\_size$, $num\_characters$] with the logits.

The structure of the Character_Decoder class is illustrated below:

```
class Character_Decoder(object):
    def __init__(lstm_output, labels, length, lstm_final_state, is_training):
        ...
    def decode(self):
        def attention_mechanism(...):
            ...
        def MLP(...):
            ...
        def loop_function(lstm_tranducer_out):
            context_vectors = attention_mechanism(...)
            logits = MLP(lstm_tranducer_out, context_vectors)
            self.all_logits.append(logits)
            ...
        self.all_logits = []
        ... = tf.contrib.legacy_seq2seq.rnn_decoder(loop_function, ...)
        self.all_logits = tf.stack(self.all_logits, axis=1)
    def loss():
        ...
```

The loss() method is similar to the homonymous method of the FullyConnected_layer class. First, it computes the cross entropy loss using the labels and the logits. However, here we do not use weights of ones and zeros for the different output steps, since all steps have a

meaningful content. If labels is a tensor of size $[batch\_size, T_{out}]$, which contains the target character label for each sample in the batch and each output time step, we can generate the 3D tensor of distributions $\boldsymbol{d}^{target}$ of size $[batch\_size, T_{out}, num\_characters]$ by setting

- $d_{itj}^{target} = 1$, if $j =$ labels[i,t] and
- $d_{itj}^{target} = 0$, if $j \neq$ labels[i,t]

for $i = 0, \ldots, batch\_size - 1$, $j = 0, \ldots,$ num_characters $- 1$ and $t = 0, \ldots, T_{out}$.

The predicted distributions are obtained with Softmax over the logits:

$$\boldsymbol{d}^{pred} = \text{Softmax(logits)}$$

The average cross entropy across all output time steps for an entire batch is:

$$\text{loss} = -\sum_{i=0}^{batch\_size-1} \frac{\sum_{t=0}^{T_{out}-1} \sum_{j=0}^{\text{num\_characters}-1} d_{itj}^{target} \cdot \log(d_{itj}^{pred})}{T_{out}},$$

This quantity is calculated in Tensorflow with tf.contrib.seq2seq.sequence_loss(). Then the total loss is obtained by adding the weight decay loss to the cross entropy loss, in the same way that was described for the word classification network. Finally, the total loss is returned by the loss() method of the Character_Decoder class.

5.4 Training & Evaluation Implementation

In Section 5.2 we presented the architecture of the lipreading application. It consists of two Python Processes, one that generates batches of samples and another one that executes a training or evaluation operation on one of the two neural networks.

5.4.1 Training Process

Implementing the training procedure for a neural network is a straightforward procedure in Tensorflow. First we build a model of the network as a Python function, which returns a loss value. This procedure was described in Section 5.3. Then we create a training operation in Tensorflow, which is executed later in the main training loop. For this reason, we define a function train(), which returns the training operation. The structure of the train() function is explained below.

The train() function

This is a Python function which receives the total loss, and returns a training operation. In the body of train(), we first set up the learning rate. We use the function tf.train.exponential_decay() of Tensorflow to compute the learning rate of the current optimization step, since the learning rate is decayed exponentially based on the number of steps.

Next, we declare the optimization method. We choose the Stochastic gradient descent with momentum method, since it is a commonly used optimizer for lipreading networks and

neural networks in general. Tensorflow provides the class tf.train.MomentumOptimizer() which implements the desired optimization method.

After instantiating a MomentumOptimizer object, we invoke the compute_gradients() method on it, by passing the total loss obtained from the network's model as an argument to the method. Then, compute_gradients() returns the gradients of the loss with respect to all trainable variables. In order to prevent gradients from exploding we use tf.clip_by_value() to restrain gradient values in the [-12.0, 12.0] interval.

The final step would be to obtain the training operation. When we run this operation on a Tensorflow session, a single optimization step is performed and the values of all trainable variables are updated. The training operation is returned by a call of the apply_gradients() method on the MomentumOptimizer object.

The outline of the train() function is presented below.

```
def train(total_loss, global_step, ...):
    learning_rate = tf.train.exponential_decay(global_step, ...)
    opt = tf.train.MomentumOptimizer(learning_rate, momentum)
    grads = opt.compute_gradients(total_loss)
    grads = [(tf.clip_by_value(grad, −12.0, 12.0), var) for grad, var in grads]
    training_operation = opt.apply_gradients(grads, global_step)
    ...
  return training_operation
```

Two more functionalities which are not shown above are incorporated in the train() function. First, we store information on the values of trainable variables and gradients in every optimization step with Tensorflow's tf.summary.histogram(). Additionally, the learning rate and the total loss are saved in every training step with tf.summary.scalar(). Secondly, a moving average of each trainable variable in the neural networks is computed and stored, so that we can later use it during evaluation. It is suggested that we could obtain higher predictive performance with the moving average value of the trainable variables. The moving average decay was set to 0.9998.

Trainable variables initialization

Another important aspect of the training procedure is the initialization method of trainable variables. Tensorflow provides a variety of initializers, while the most commonly used is the tf.truncated_normal_initializer() that generates a truncated normal distribution, given a mean and a standard deviation value. In truncated normal initialization, values more than two standard deviations from the mean are discarded and re-drawn. This is the recommended initializer for neural network weights and filters. However, Xavier initialization is another popular method for assigning initial values to weight matrices and filters, which aims to keep variance the same in adjacent hidden layers. It has been shown that his initializer keeps the scale of the gradients roughly the same in all layers. Xavier method uses either a uniform distribution and sets its upper and lower boundaries, or a normal distribution and determines its mean and standard deviation. For the word classification network, we use Xavier initialization with a normal distribution to assign values to all weight matrices and filters of the 3D CNN, 2D CNN, BiLSTM and fully connected layer at the end of the network. The same method is used in the character decoding network to set the values of

the weight matrices in the LSTM transducer, the attention mechanism and finally the MLP. In both networks, biases are initialized with the tf.constant_initializer() to the constant value 0.1.

The structure of the training Process

In Section 5.2 we presented the training Python Process for the first time. However, we briefly mentioned the training Process, as part of the entire lipreading application, which in case of training consists of two Python Processes: the batch producer Process and the training Process. The first one generates batches of samples from our dataset and writes them in a multiprocessing FIFO queue, while the second one reads them and executes the training operation on the batch that is an optimization step. The training Process consists of two Python Treads, one that reads batches from the multiprocessing FIFO queue and writes them to an intraprocess Tensorflow queue and another one that runs the main training loop.

The structure of the lipreading application for training one of the two neural networks is shown below:

```
def training_process(network, ...):
    ...
    # Create a multiprocessing FIFO queue
    multiqueue = multiprocessing.Queue()
    # Launch the batch producer process
    multiprocessing.Process(target=batch_producer_process,
                            args = (multiqueue, ...)).start()
    # Get or create the optimization step to start from
    global_step = tf.train.get_or_create_global_step()
    ...
    # Set up the intraprocess Tensorflow queue
    queue = tf.FIFOQueue(...)
    # Get a enqueue operation which is ran by the enqueue Thread
    enqueue_op = queue.enqueue()
    # The enqueue Thread read batches from the multiqueue and writes them to the
    # Tensorflow queue
    threading.Thread(target=enqueue, args=[enqueue_op, ...]).start()
    # This training Tread of the training Process reads a batch from the intraprocess
    # Tensorflow queue
    batch = queue.dequeue()
    batch = image_standardization(batch)
    # Send batch to the neural network's model
    if network == "word classification network":
        loss, logits = WordClassificationNeuralNetwork(batch, ...)
    else:
        loss, logits = CharacterDecodingNeuralNetwork(batch, ...)
    # Get a training operation
    training_operation = train(loss, global_step,...)
    # Create a Tensorflow Session.
    sess = tf.Session(...)
    ...
```

```
    # The training Thread runs the training loop
    # Each iteration executes an optimization step with a different batch
    while more_batches:
        sess.run(training_operation, ...)

def batch_producer_process(multiqueue, ...):
    # Generates the batches and writes them to multiqueue
    ...

def main():
    ...
    network_to_train = ...
    # Launch the training process
    training_process(network_to_train, ...)
```

During training, the main function of the lipreading application first calls the training_process() function. Now, the main Process, which runs the training_process() function creates a multiprocessing queue and launches the second Process which is responsible for generating batches of samples. Subsequently, the training Process sets up the intraprocess Tensorflow queue and starts the Thread that places the batches obtained by the multiprocessing queue to the Tensorflow queue. Next, a Tensorflow graph is built. It first declares that a batch is obtained by the intraprocess queue and then an image standardization in applied on its frames. The batch is then connected to the model of the network we are currently training. The output of the model is a loss value over the batch and a set of logits. A training operation is returned by the train() function that receives the loss value as an argument. At this point, the Tensorflow graph has been built and a session to run the graph has been generated. The final step would be to run the main training loop, where the training operation is executed for a different batch in every iteration. The execution of the training operation using a session has the effect of updating all trainable variables and is therefore equivalent to a single optimization step. By the time we have ran this operation for all batches in the dataset, one epoch has been performed, since one epoch consists of one full training cycle on the training dataset. Therefore, the number of total optimization steps in a single epoch should be equal to the number of training samples divided by the batch size. The lipreading application terminates after training one of the two networks for one epoch (one pass over all the training examples).

Save and restore the trainable variables

However, we may need to run multiple epochs to successfully train a neural network. Additionally, the values of all trainable variables that were learned during training should be stored, so they can be reused for the evaluation of the network's predictive performance. Tensorflow provides a mechanism to save and retrieve a subset or all trainable variables in the network. The tf.train.Saver() class was used to instantiate a saver object. When invoked on a saver object, the restore() method reads the network's variables from a file, while the save() method writes them in a file.

5.4.2 Evaluation Process

The evaluation Process in quite similar to the training Process, with the difference that we do not run the training operation obtained by an optimization class in Tensorflow, but we use the logits from the network's model to assess its predictive performance.

The structure of the lipreading application for evaluating the performance of the two neural networks is shown below:

```
def evaluation_process(network, ...):
    ...
    # Create a multiprocessing FIFO queue
    multiqueue = multiprocessing.Queue()
    # Launch the batch producer process
    multiprocessing.Process(target=batch_producer_process,
                            args = (multiqueue, ...)).start()
    # Get or create the optimization step to start from
    global_step = tf.train.get_or_create_global_step()
    ...
    # Set up the intraprocess Tensorflow queue
    queue = tf.FIFOQueue(...)
    # Get a enqueue operation which is ran by the enqueue Thread
    enqueue_op = queue.enqueue()
    # The enqueue Thread read batches from the multiqueue and writes them to the
    # Tensorflow queue
    threading.Thread(target=enqueue, args=[enqueue_op, ...]).start()
    # This evaluation Tread of the evaluation Process reads a batch from the intraprocess
    # Tensorflow queue
    batch = queue.dequeue()
    batch = image_standardization(batch)
    # Send batch to the neural network's model
    if network == "word classification network":
        loss, logits = WordClassificationNeuralNetwork(batch, ...)
    else:
        loss, logits = CharacterDecodingNeuralNetwork(batch, ...)
    # The evaluation operation returns the number of correct predictions in the batch
    eval_operation = eval(logits, labels,...)
    # Create a Tensorflow Session.
    sess = tf.Session(...)
    ...
    # The evaluation Thread runs the evaluation loop. Each iteration counts how many
    # samples in the batch were predicted correctly by the network
    sum = 0
    while more_batches:
        sum += sess.run(eval_operation, ...)
    accuracy = sum / number_of_evaluation_samples

def batch_producer_process(multiqueue, ...):
    # Generates the batches and writes them to multiqueue
    ...
```

```
def main():
    ...
    network_to_evaluate = ...
    # Launch the training process
    evaluation_process(network_to_evaluate, ...)
```

During evaluation, we are interested in the logits output of the the two networks. In case we are evaluating the performance of the word classification network, logits is a 3D tensor of size [$batch\_size$, T, num_words] and labels a 1D tensor of size [$batch\_size$]. The eval() function, which calculates the number of correct predictions in a batch, does the following. First, a Softmax function is applied on logits to transform them to probability distributions:

$$\boldsymbol{d} = \text{Softmax(logits)}$$

In the next step, the T different distributions that correspond to the various input time steps of the frame sequences in the batch are merged into one distribution $\boldsymbol{p}$, where:

$$p_{ij} = \frac{\sum_{t=1}^{T} v_{it} \cdot d_{itj}}{\sum_{t=1}^{T} v_{it}} \quad i = 1, \ldots, batch\_size \text{ and } j = 1, \ldots, \text{num\_words}.$$

In case the frame from time step t of sample i in the batch corresponds to an actual frame from the video, then we have $v_{it} = 1$. In the other case where the frame t in sample i is used for padding, we have $v_{it} = 0$. Now, the 2D tensor $\boldsymbol{p} = [p_{ij}]$ of size [$batch\_size$, num_words] contains a distribution on word labels for each sample in the batch.

Finally, for each sample in the batch we choose the word label with the highest probability according to $\boldsymbol{p}$ and compare it with the corresponding true label from the labels tensor. If the labels match, the neural network predicted the word spoken in the input video sample correctly. In Tensorflow we use tf.nn.in_top_k() to obtain a 1D tensor with boolean values for each sample in the batch, where True means that the word label of the sample was successfully predicted by the lipreading network.

For the character decoding network, logits is a 3D tensor of size [$batch\_size$, T_{out}, num_characters], while labels is a 2D tensor of size [$batch\_size$, T_{out}]. In the same way with the word classification network, a Softmax function transforms logits to probability distributions on character labels. Next, for each sample and each output time step, we choose the character label with the highest probability and form a 2D tensor of size [$batch\_size$, T_{out}]. Finally, in order to consider the prediction of a character sequence correct, we demand that all character labels in the predicted sequence of length T_{out} are the same with the character labels in the target sequence. If the character labels match in all time steps, the prediction is considered successful.

Chapter 6

Training Procedure

6.1 Stochastic gradient descent

Stochastic gradient descent (SGD) is a stochastic approximation of the gradient descent optimization method for minimizing a loss function. While gradient descent uses the entire dataset to update the trainable parameters in each optimization step, stochastic gradient descent uses only a subset of the dataset, known as a batch. SGD is a commonly used method for optimizing the loss function of a neural network. The loss function for the word classification and the character decoding network was defined in Section 5.3. For both networks, we can write the total loss on a batch with samples $\{(\boldsymbol{X}_i, \boldsymbol{y}_i)\}_{i=1,\dots,batch\_size}$ as a function of the trainable parameters of the network $\boldsymbol{\theta}$:

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{batch\_size} H(\boldsymbol{\theta}, \boldsymbol{X}_i, \boldsymbol{y}_i) + \frac{1}{2} D(\boldsymbol{\theta})$$

where the first term is the the sum of cross entropy losses $H(\boldsymbol{\theta}, \boldsymbol{X}_i, \boldsymbol{y}_i)$ for each sample i in the batch, while the second term $D(\boldsymbol{\theta}) = \lambda \sum_{w \in \boldsymbol{\theta}} w^2$ is the weight decay loss. Here, we denote with $\boldsymbol{X}_i = \{\boldsymbol{x}_{i1}, \dots, \boldsymbol{x}_{iT}\}$ the i-th frame sequence of length $T = 29$ in the batch, while $\boldsymbol{y}_i$ denotes the corresponding word label of sample i for the first network or the sequence of character labels for the second network. The total loss on a batch $J(\boldsymbol{\theta})$ is the objective function we want to minimize. The derivative of the total loss on a batch with respect to a trainable parameter of the network $w \in \boldsymbol{\theta}$ is:

$$\nabla_w J(\boldsymbol{\theta}) = \nabla_w \left(\sum_{i=1}^{batch\_size} H(\boldsymbol{\theta}, \boldsymbol{X}_i, \boldsymbol{y}_i) \right) + \lambda w$$

Then, according to stochastic gradient descent, in optimization step k, each trainable variable $w \in \boldsymbol{\theta}$ will be updated as follows:

$$w^{(k)} := w^{(k-1)} - \eta \nabla_w J(\boldsymbol{\theta}^{(k-1)})$$

or

$$w^{(k)} := w^{(k-1)} - \eta \nabla_w \left(\sum_{i=1}^{batch\_size} H(\boldsymbol{\theta}^{(k-1)}, \boldsymbol{X}_i, \boldsymbol{y}_i) \right) - \eta \lambda w^{(k-1)}$$

However, we use SGD with momentum. In this case we add a momentum term and each trainable variable is updated in the optimization step k with the rule

$$w^{(k)} := w^{(k-1)} - \eta \nabla_w \left(\sum_{i=1}^{batch\_size} H(\boldsymbol{\theta}^{(k-1)}, \boldsymbol{X}_i, \boldsymbol{y}_i) \right) - \eta \lambda w^{(k-1)} + v(w^{(k-1)} - w^{(k-2)})$$

where $\{\boldsymbol{X}_i, \boldsymbol{y}_i\}_{i=1,\dots,batch\_size}$ is the batch we use for the k-th optimization step, η is the learning rate, λ is the L2 regularization parameter of weight decay and v is the momentum parameter. The parameters η, λ and v are hyperparameters of the two lipreading networks, since they are not trainable and their values are determined in another way.

One way to perform hyperparameter optimization is grid search, where a network is trained with various combinations of the hyperparameters and its performance is estimated on the validation set for each configuration. The configuration which leads to the highest performance on the validation set is chosen as optimal. Another popular method is random search where random hyperparameters are chosen to fully train the network. Again the configuration with the best performance on the validation set in considered to be the optimal. However, the huge size of the training set and both lipreading neural networks renders training a highly time consuming operation. Each network requires several days to be trained and grid or random search require training each network multiple times. Therefore, these methods are not suitable for determining the hyperparameters of the two networks. For this reason, some hyperparameters are set to the proposed values from similar lipreading networks such as Chung et al. (2016); Stafylakis and Tzimiropoulos (2017), while others are set to commonly chosen values in deep networks in general.

6.2 Hyperparameters

6.2.1 Learning Rate

The learning rate η is the first hyperparameter we should determine. This parameter depends on the batch size, since the form of the loss function is influenced by the batch size and the learning rate depends on the form of the loss function. In order to train the word classification network we set the initial learning rate to $3 \cdot 10^{-3}$. The same value was used by Stafylakis and Tzimiropoulos (2017) to train their lipreading system that shares many components will our proposed network. The character decoding network is trained with the initial learning rate of $5 \cdot 10^{-3}$. For both lipreading networks, the learning rate is decayed exponentially during training in order to avoid overfitting and ensure convergence.

6.2.2 Batch Size

One of the most important hyperparameters to consider is the batch size. The batch size significantly affects the progress of the optimization algorithm. A large batch means that we have a better approximation of the true loss function we are optimizing. Therefore, we can use a larger learning rate and take bigger steps in each optimization iteration. However, the step size cannot exceed an algorithmic, problem dependent upper bound which depends on the smoothness of the loss function. Therefore, there is a bound in the batch size and increasing its value beyond that bound will not allow us to increase our learning rate and

take bigger steps. A typical batch size used in other existing lipreading networks trained on the LRW dataset is 64, so we choose the same value for our two networks as well.

6.2.3 Weight Decay

Weight decay λ is a regularization hyperparameter that penalizes large values of a network's trainable parameters. The value of λ determines how dominant the term of weight decay loss will be in the loss gradient. Therefore, a large value of λ generates a big weight decay loss and pushes trainable parameters to small values near zero. Generally, the more trainable variables we have in a neural network, the bigger value we should to assign to λ. Weight decay is used to reduce overfitting, since it prevents trainable values from obtaining large values that better explain the training dataset at the expense of the network's predictive performance on unseen samples.

6.2.4 Momentum

Momentum is a technique that helps the optimization method avoid local minima and continue searching parameters with lower loss values. The momentum term increases the size of the steps taken towards the minimum by adding a fraction of the previous weight update to the current one. A large value of momentum enables the optimization methods to convergence faster. However, if the momentum term v is large then the learning rate η should be kept smaller. If both the momentum and learning rate are large, then we might constantly skip the minimum. Additionally, moment is useful if the gradient keeps changing directions. Both lipreading networks were trained with a momentum term $v = 0.9$, since the same value was used for the network proposed by Stafylakis and Tzimiropoulos (2017).

6.2.5 Dropout

Dropout is a regularization technique for reducing overfitting in neural networks. The technique reduces node interactions, leading them to learn more robust features that better generalize to new data. At each optimization/training step, individual neurons of a hidden layer are either dropped out of the network with probability $1 - p$ or kept with probability p. Incoming and outgoing edges to a dropped-out node are also removed. In this way, a different reduced network is left at each optimization step and only this reduced network is trained in that step. Dropout is employed in many hidden layers of both lipreading networks. A dropout with $p = 0.75$ is applied after max-pooling in the last layer of the 3D CNN and in the third hidden layer with convolution of the 2D CNN. Additionally, dropout is applied to all hidden layers of the LSTMs in both lipreading networks, with $p = 1.0$ and $p = 0.9$ later. Finally, dropout with $p = 0.95$ is used in the hidden layer of the MLP in the decoder of the character decoding network.

6.2.6 Epochs

The number of epochs reflects the number of times we go through the entire training dataset and create batches to perform optimization steps. Since the LRW training set consists of half a million samples, one complete pass through the whole training set requires a lot of hours,

so it is very time consuming. Therefore, we keep the number of epochs low, between five and fifteen.

6.3 Word Classification Network Training

The word classification neural network consists of a four smaller networks. In the the front end there is a 3D CNN with a convolutional and a max-pooling layer. The 3D CNN is followed by multiple parallel 2D CNNs with shared variables. The 2D CNNs have three convolutional layers each one followed by a max-pooling operation and one fully connected layer in the end. Finally, a BiLSTM with three hidden layers is followed by a fully connected layer and a Softmax unit.

Designing the classification network and implementing its model along with the training and evaluation operations in Tensorflow was the first part in the process of building a lipreading system. The next step would be to perform the training procedure and learn a set of parameters, so that the network can predict the word spoken in a unseen video from the LRW test set. However, training a deep network with many layers for a classification task with 500 classes could be considered a challenging task.

One popular way of training a deep network similar to the proposed word classification network is by first initializing the trainable variables of CNNs with pretrained values. According to this method, instead of training the entire network form scratch, we could use pretrained components that have been trained for other similar tasks. Then, it would be sufficient to fine-tune the entire network with the lipreading dataset for a small number of epochs.

However, we choose to train the entire word classification network without pretrained subnetworks. In order to achieve this, we start by training the system on a small subset of the LRW dataset with five classes. Then, we use the learned parameters of the 3D CNN, the 2D CNN and the BiLSTM to train the network on fifty classes of the dataset. Finally, the values of the trainable variables we acquired from this process are used as initial values to train the full network on five hundred classes. The values of the weights of the final fully connected layer are initialized with the Xavier initializer in each stage, since this is the only part of the network that changes between the three training stages.

Training stage 1:

In the first training stage, the word classification network is trained on a very small subset of the LRW dataset with five words. These words were: ABOUT, ABSOLUTELY, ABUSE, ACCESS and ACCORDING.

In this stage, we do not use weight decay during optimization ($\lambda = 0$). In addition, dropout is not employed in the hidden layers of the BiLSTM ($p = 1.0$). We choose an initial learning rate of $3 \cdot 10^{-3}$ which is reduced exponentially with the rule:

$$\text{learning rate} = \text{(initial learning rate)} \cdot \text{(decay rate)}^{\lfloor \text{SGD step}/\text{(decay steps)} \rfloor}$$

The decay rate is set to 0.5, while the decays steps are set to three times the total number of SGD steps performed in one epoch. Therefore the learning rate is reduced in half every three epochs. The network was trained for fifteen epochs, which is equivalent to 1155 Stochastic gradient descent steps. Each optimization step required approximately seven seconds.

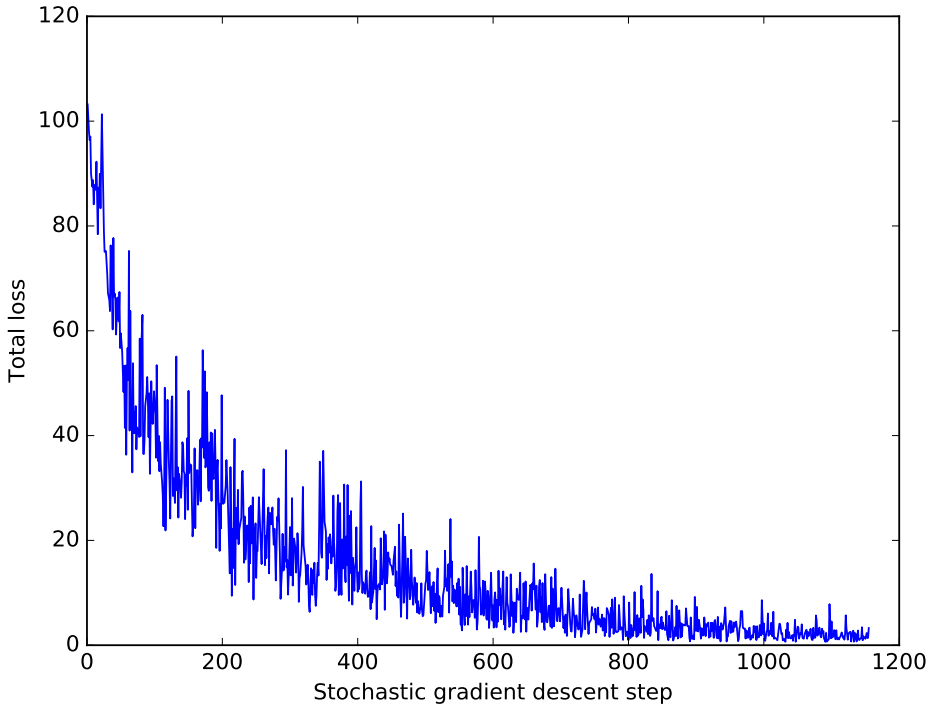

Figure 6.1: The total loss against the number of optimization iterations for fifteen epochs.

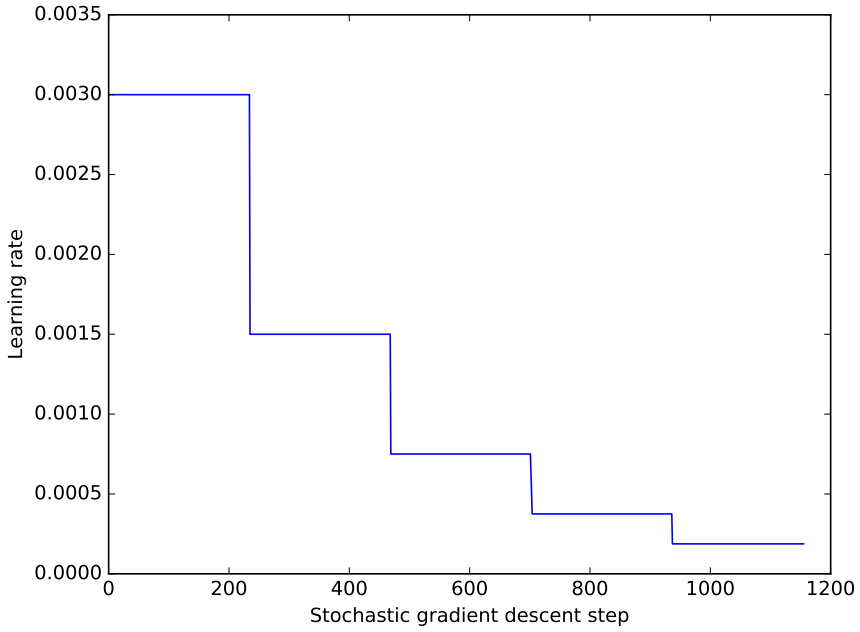

Figure 6.2: The learning rate against the number of optimization iterations for fifteen epochs.

Figure 6.1 shows the total loss across a batch of samples, which is the quantity being optimized during training, against the number of optimization steps. As can be seen from the figure, the total loss had a value near 100 in the beginning of the optimization procedure and was gradually reduced to a value near 2. Another important feature of the graph is the intense fluctuations of the loss value in the first twelve epochs that become less in the final three epochs.

The value of learning rate while training progresses is illustrated in Figure 6.2. As can be observed from the graph, learning rate is decreased in half every three epochs.

Training stage 2:

After training the neural network for five classes of words, we save the learned values of all trainable variables. Next, the final fully connected layer at the back end of the network is removed and replaced with another one which produces logits with fifty values instead of five. Now, the network should perform classification on fifty classes of words. The rest components of the network are initialized with the variable values from the previous training stage and training is launched with the same hyperparameters. The network is trained for fifteen epochs, which is equivalent to 11340 Stochastic gradient descent steps. Again, we divide the learning rate by two every three epochs. Each optimization step required almost seven seconds to be completed. This time could be reduced if we did not save summaries of the trainable variables and the corresponding gradients at each SGD iteration.

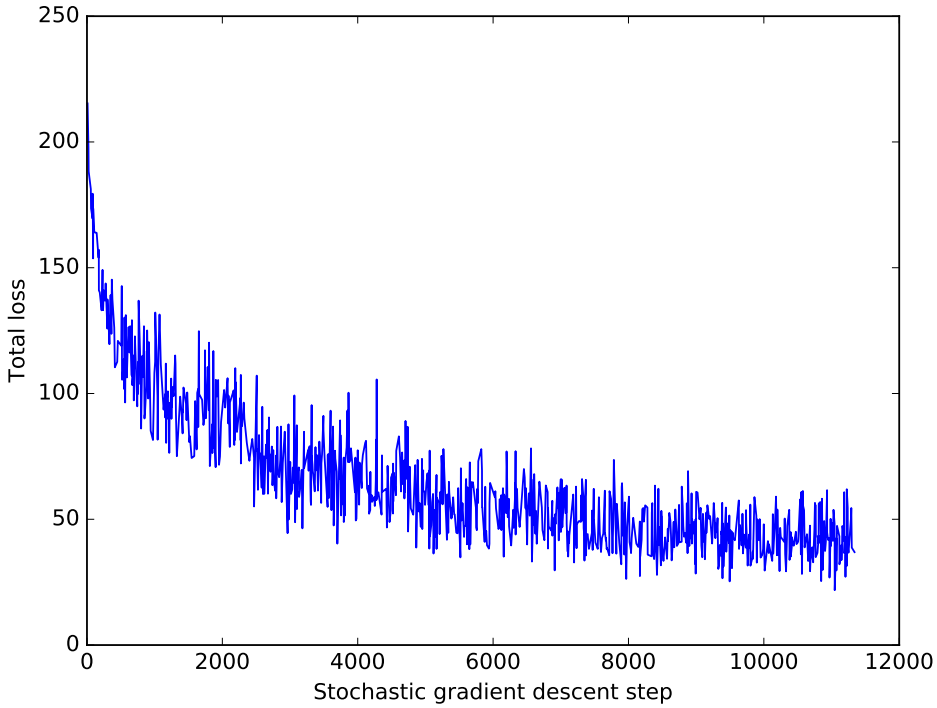

Figure 6.3: The total loss against the number of optimization iterations for fifteen epochs.

The total loss across a batch of samples against the number of optimization iterations is demonstrated in Figure 6.3. As can be observed from the graph, the initial loss value is near 250 and drops near 40 by the end of the training procedure. Moreover, the fluctuations of

the loss value are very intense throughout the optimization process, though the average loss is significantly decreased after the 8000 first steps. Finally, the minimum total loss on a batch is around 30, which is a magnitude greater that the minimum we achieved on the network with five classes of words.

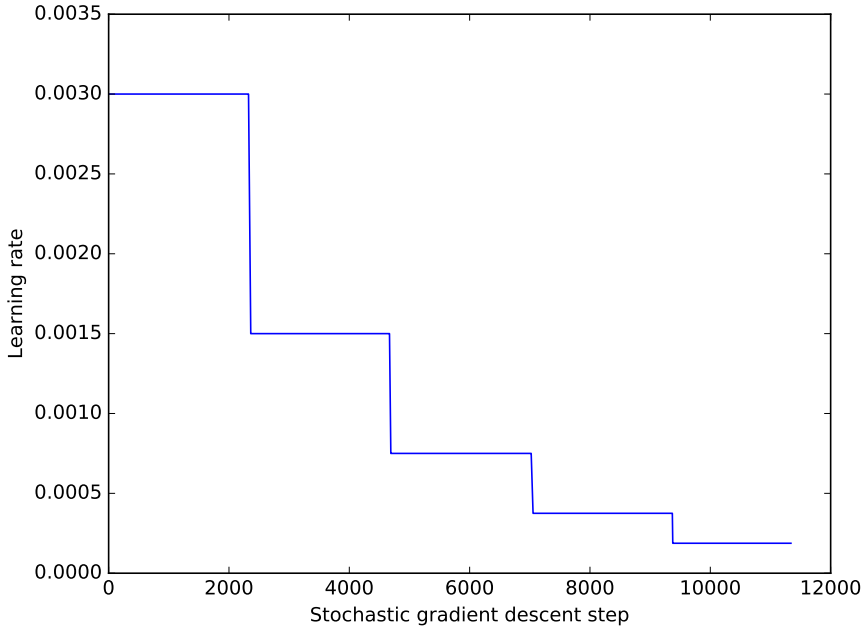

Figure 6.4: The learning rate against the number of optimization iterations for fifteen epochs.

As shown in Figure 6.4, the learning rate is kept to its initial value for the first three epochs and gets reduced to $1.875 \cdot 10^{-4}$ in the final three epochs.

Training stage 3:

In this third and last stage, the entire word classification network is trained end-to-end with all five hundred word classes of the LRW dataset. The parameters of the 3D CNN, 2D CNN and the bidirectional LSTM are initialized with the learned variables from the previous training stage. In contrast with the two previous training stages, we apply dropout with $p = 0.9$ to every hidden layer of both the forward and backward LSMTs. Moreover, weight decay is applied with $\lambda = 10^{-4}$. Since the second training stage was performed with one tenth of the dataset and lasted for fifteen epochs, we do not have to run a long training process for the full dataset. Therefore, we choose to fine-tune the network for six epochs. Furthermore, we employ a different strategy for decaying the learning rate. The learning rate in now decayed every 100 optimization iterations, with a rate of 0.99. This method leads to more stable learning, since each epoch on the full dataset consists of approximately 7500 steps and the learning rate should be constantly reduced to achieve non-diverging loss. The total number of trainable parameters in the network is around 9 million and the full training procedure consisted of 45777 optimization steps. The loss against the number of iterations is shown in Figure 6.5. As can be seen from the graph, the loss value is near 425 in the beginning and gets decreased to values around 175 in the end of training. The learning rate is demonstrated in Figure 6.6. Finally, Figures 6.7 to 6.38 illustrate the histograms of all trainable variables of the word classification network.

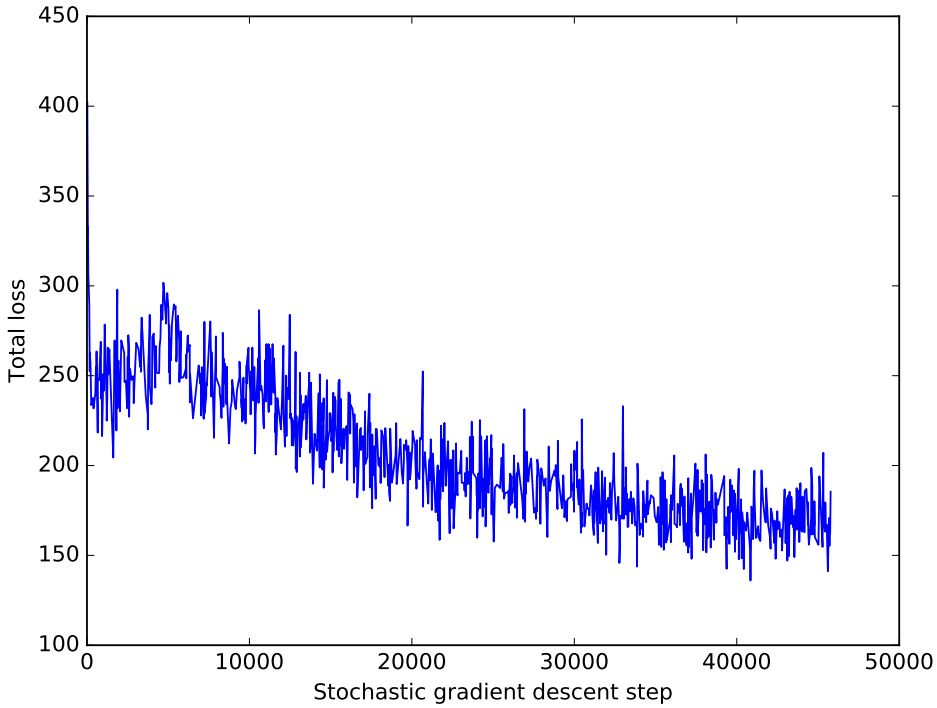

Figure 6.5: The total loss against the number of optimization iterations for six epochs.

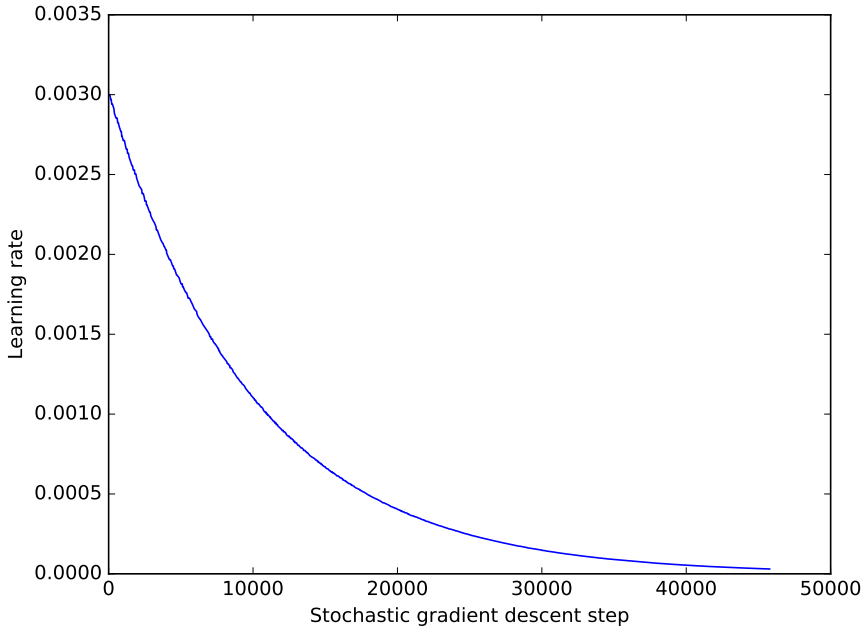

Figure 6.6: The learning rate against the number of optimization iterations for six epochs.

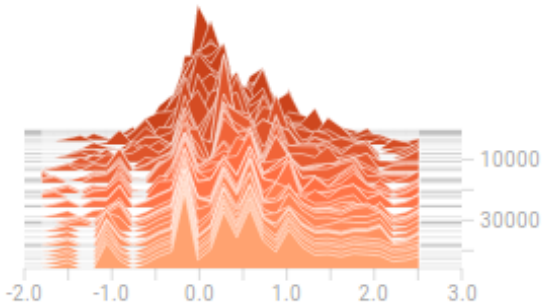

Figure 6.7: Histogram of biases in the first hidden layer of the 3D CNN

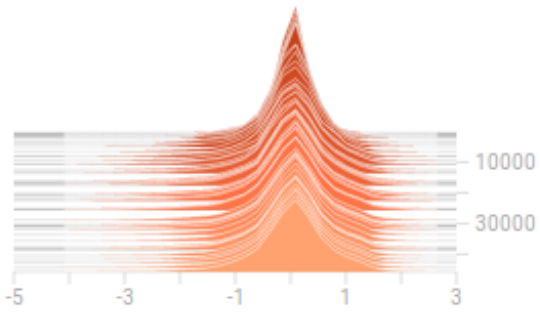

Figure 6.8: Histogram of weights in the first hidden layer of the 3D CNN

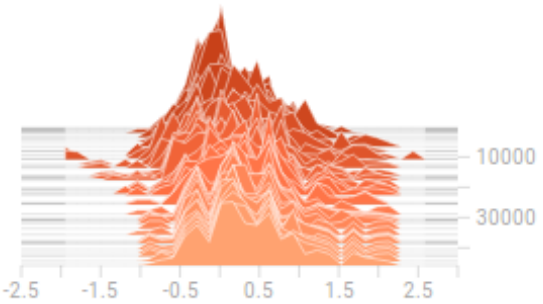

Figure 6.9: Histogram of β terms of batch normalization in the 3D CNN

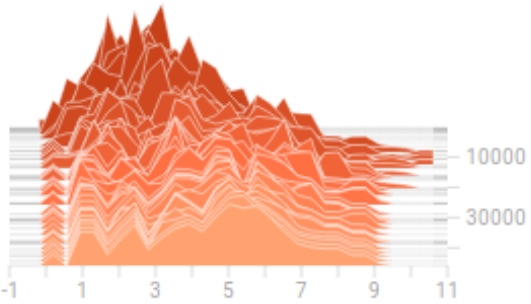

Figure 6.10: Histogram of γ terms of batch normalization in the 3D CNN

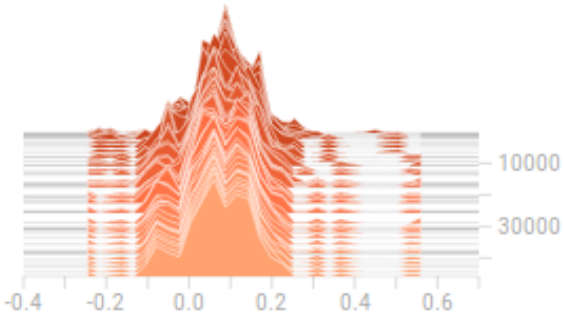

Figure 6.11: Histogram of biases in the first hidden layer of the 2D CNN

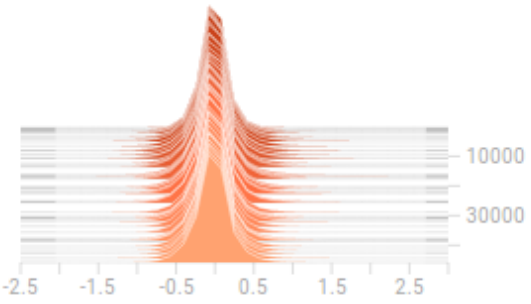

Figure 6.12: Histogram of weights in the first hidden layer of the 2D CNN

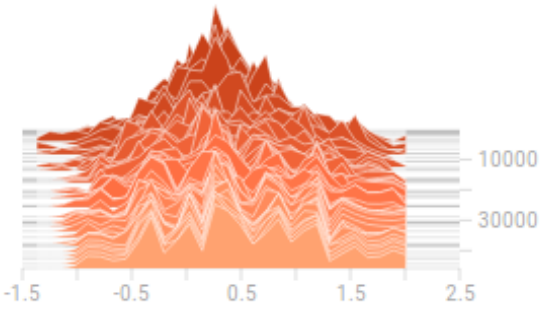

Figure 6.13: Histogram of β terms of batch normalization after the the first layer of the 2D CNN

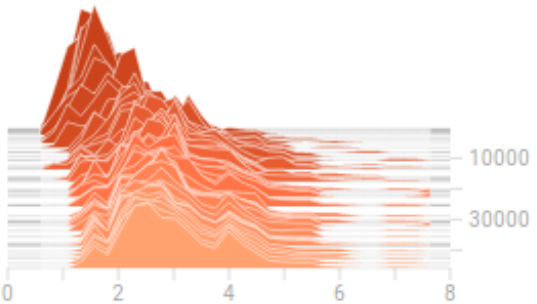

Figure 6.14: Histogram of γ terms of batch normalization after the the first layer of the 2D CNN

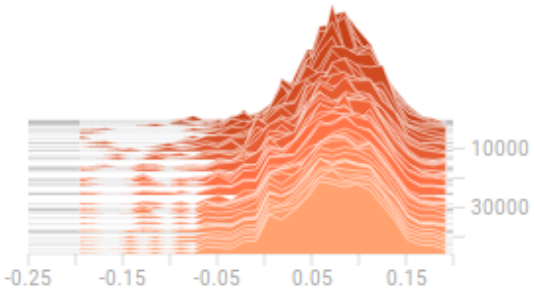

Figure 6.15: Histogram of biases in the second hidden layer of the 2D CNN

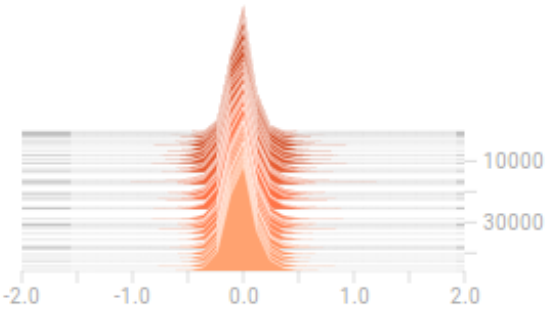

Figure 6.16: Histogram of weights in the second hidden layer of the 2D CNN

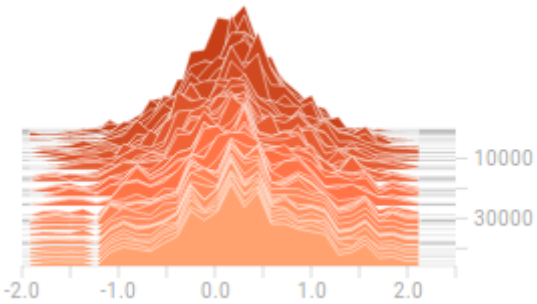

Figure 6.17: Histogram of β terms of batch normalization after the the second layer of the 2D CNN

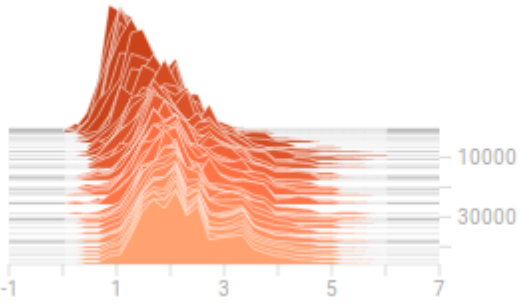

Figure 6.18: Histogram of γ terms of batch normalization after the the second layer of the 2D CNN

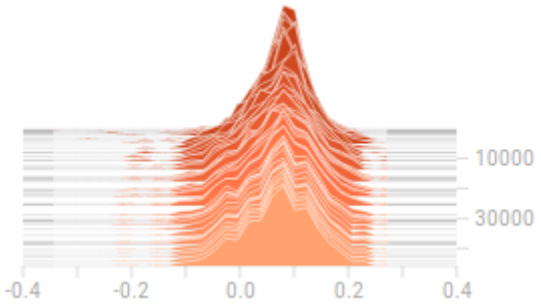

Figure 6.19: Histogram of biases in the third hidden layer of the 2D CNN

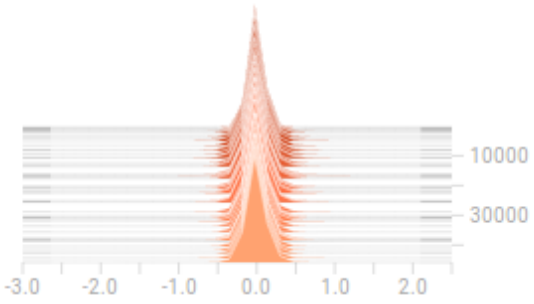

Figure 6.20: Histogram of weights in the third hidden layer of the 2D CNN

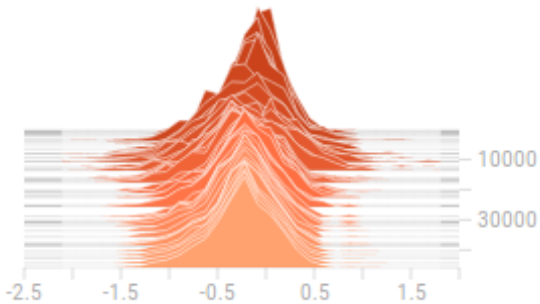

Figure 6.21: Histogram of β terms of batch normalization after the the third layer of the 2D CNN

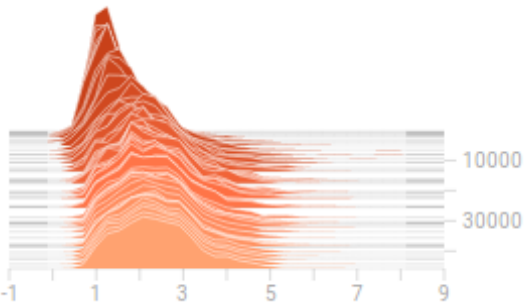

Figure 6.22: Histogram of γ terms of batch normalization after the the third layer of the 2D CNN

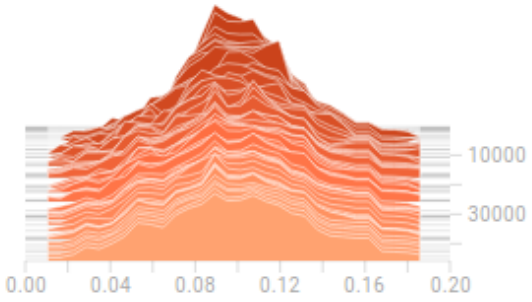

Figure 6.23: Histogram of biases in the fully connected layer of the 2D CNN

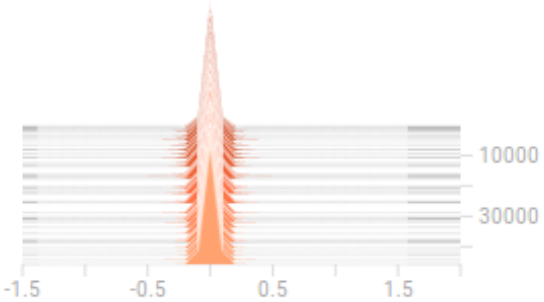

Figure 6.24: Histogram of weights in the fully connected layer of the 2D CNN

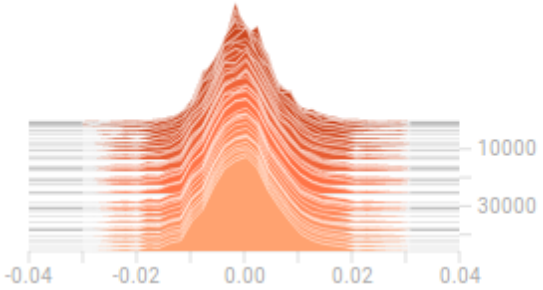

Figure 6.25: Histogram of biases in the first backward hidden layer of the BiLSTM

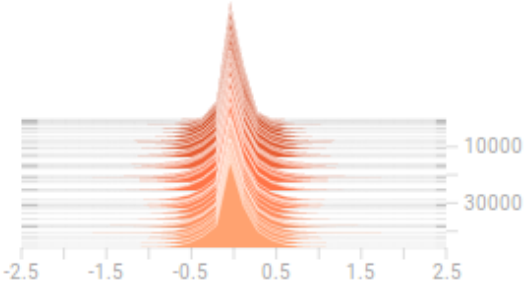

Figure 6.26: Histogram of weights in the first backward hidden layer of the BiLSTM

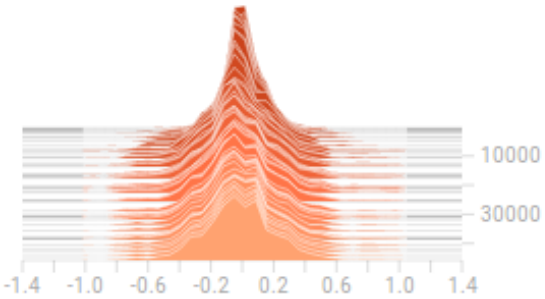

Figure 6.27: Histogram of biases in the second backward hidden layer of the BiLSTM

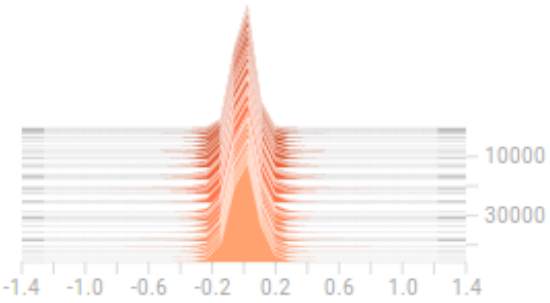

Figure 6.28: Histogram of weights in the second backward hidden layer of the BiLSTM

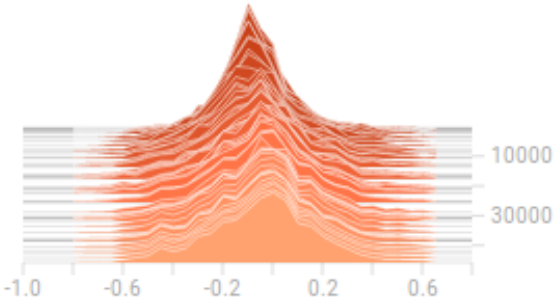

Figure 6.29: Histogram of biases in the third backward hidden layer of the BiLSTM

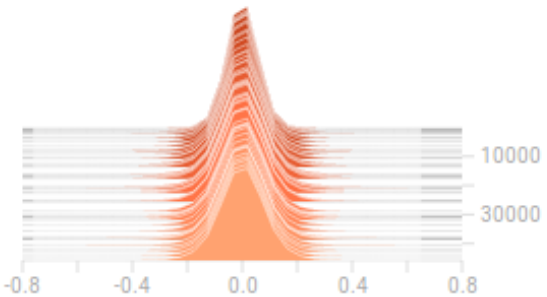

Figure 6.30: Histogram of weights in the third backward hidden layer of the BiLSTM

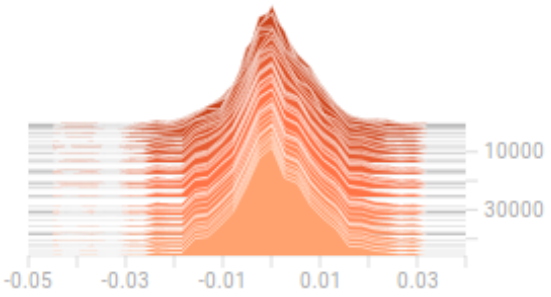

Figure 6.31: Histogram of biases in the first forward hidden layer of the BiLSTM

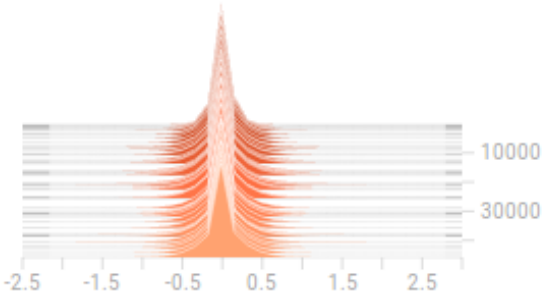

Figure 6.32: Histogram of weights in the first forward hidden layer of the BiLSTM

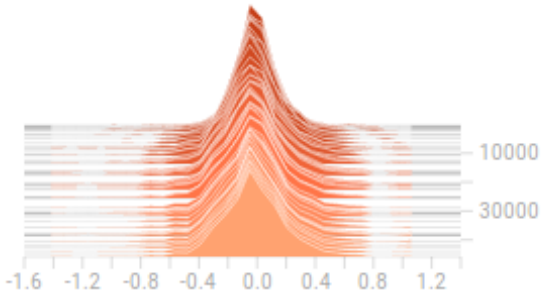

Figure 6.33: Histogram of biases in the second forward hidden layer of the BiLSTM

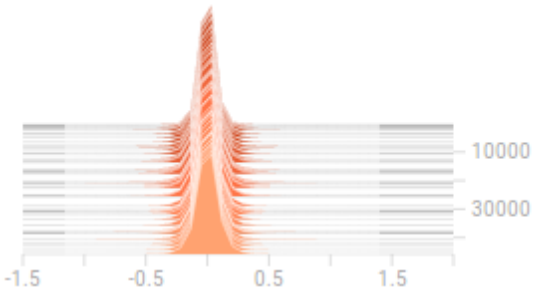

Figure 6.34: Histogram of weights in the second forward hidden layer of the BiLSTM

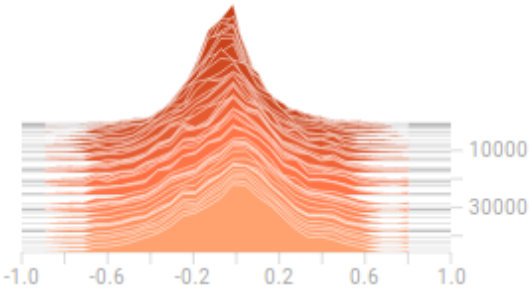

Figure 6.35: Histogram of biases in the third forward hidden layer of the BiLSTM

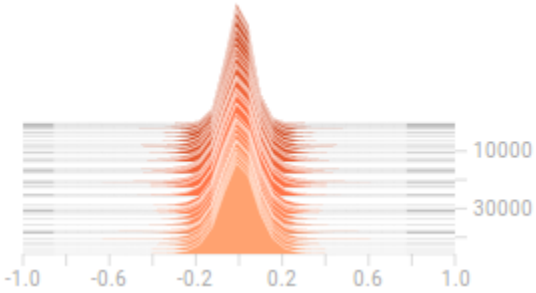

Figure 6.36: Histogram of weights in the third forward hidden layer of the BiLSTM

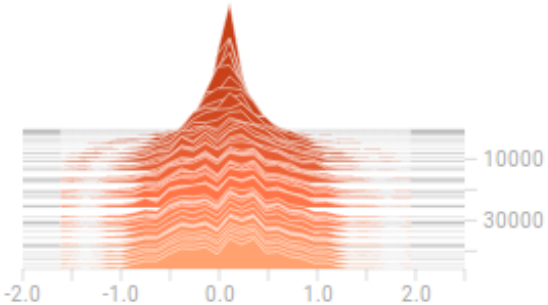

Figure 6.37: Histogram of biases in the fully connected layer at the back end of the network

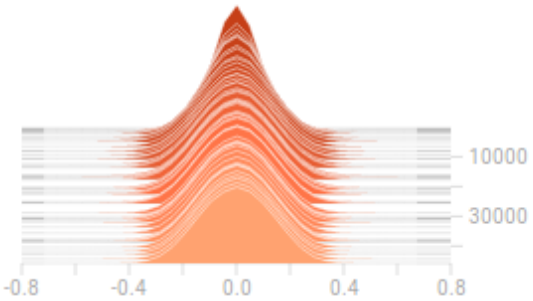

Figure 6.38: Histogram of weights in the fully connected layer at the back end of the network

6.4 Character Decoding Network Training

At this point we have trained a network which is able to perform classification on video samples. The networks maps a video of 29 frames, where a word is spoken, to a word label. However, we could instead map the video to a sequence of characters which make up a word of the dataset. The character decoding network is designed to output a character sequence, given an input sequence of frames. As explained in Chapter 4, this network consists of an encoder and a decoder. The encoding network is the same with the word classification network without the fully connected layer in the back end. Therefore, since we have already a trained encoder, we can use its learned parameters to initialize the variables of the encoder in the character decoding network. The rest trainable variables in the decoder are initialized with the Xavier method. Then, the entire network is trained end-to-end.

For the training process, we initialize the learning rate to $5 \cdot 10^{-3}$ which is decayed every 100 steps with a factor of 0.99. The dropout probability in every hidden layer of the BiLSTM and the LSTM transducer is set to 0.9. Moreover, dropout with $p = 0.95$ is applied to the hidden layer of the MLP in the decoder. Finally, the weight decay term λ is set equal to 10^{-4}. The training procedure is executed for five epochs.

Scheduled sampling, which was inspired from Chung et al. (2016), is used during the training process. When we train the LSTM transducer, the previous time step ground truth could be constantly used as the next time step input. However, during prediction the previous step ground truth is unavailable. This may lead to very poor performance, since the model was not trained to be tolerant to wrong predictions at some time steps. To solve this problem, we randomly use the previously predicted output character, instead of always using the ground truth from the dataset. The probability of using the predicted character of the previous output time step rather than the target character is set to zero for the first epoch. Then, we gradually increment it to 0.2, 0.4, 0.7 and 0.85 in the next four epochs. The effect of this strategy can be seen in the total loss in Figure 6.39. As can be observed from the graph, the overall loss across a batch decreases in the first epoch, reaching a minimum value around 30. In the second epoch, when we start using the predicted character labels from previous time steps in the LSTM transducer, the loss increases and begins to drop again as the training procedure evolves. The same trend is repeated in every subsequent epoch when the probability of selecting the predicted labels is incremented. At the end of the training process, the total loss value on a batch has been decreased to 55.

Finally, Figure 6.40 shows the learning rate value during training, which is exponentially decayed with the number of optimization steps.

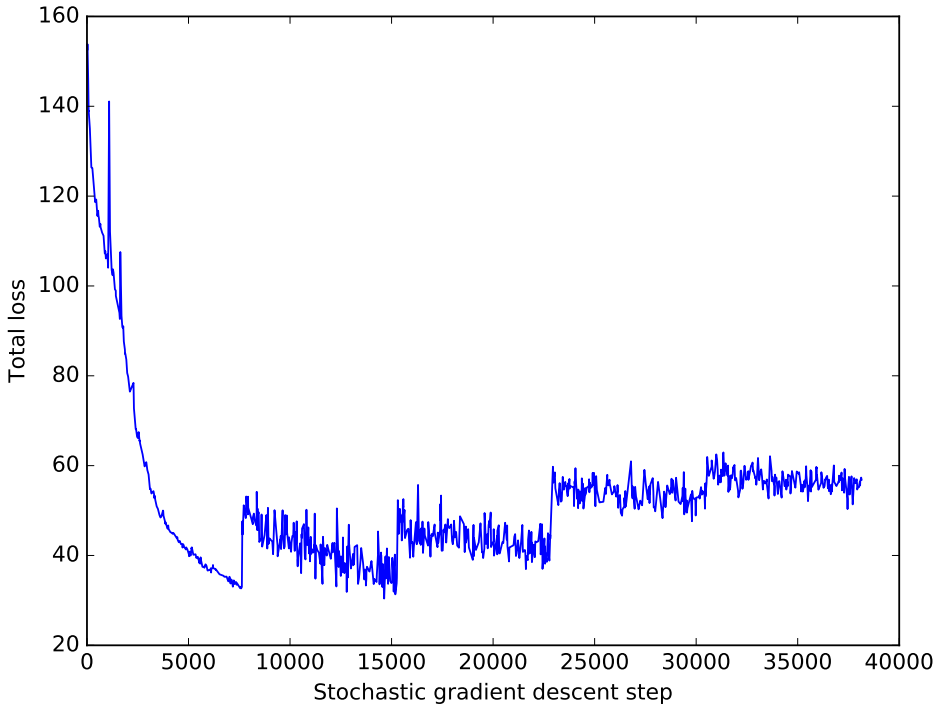

Figure 6.39: The total loss against the number of optimization iterations for five epochs.

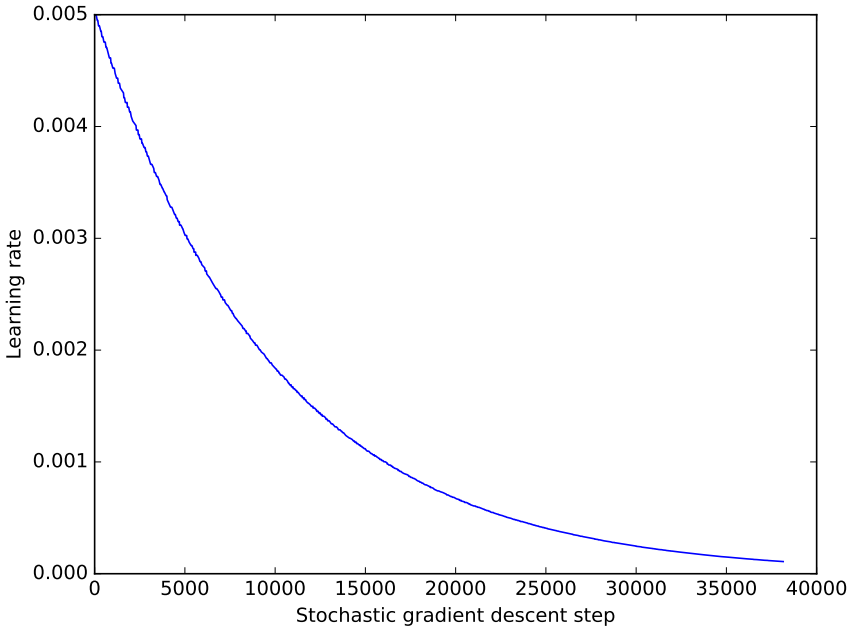

Figure 6.40: The learning rate against the number of optimization iterations for five epochs.

Chapter 7

Evaluation Results

In Chapter 6 we described the training procedure for both proposed lipreading neural networks. In Chapter 7, we focus on the predictive performance of the two networks. We evaluate the neural networks on test samples from the Lip Reading in the Wild (LRW) dataset. The test set of LRW consists of fifty video samples for each one of the five hundred words in the vocabulary. These samples were not used during training, so they can be used to determine if the two models learned to recognize patterns in lip movement and produce correct predictions of the spoken words.

7.1 Word Classification Network Evaluation

First, we present the results for the word classification network. This network produces a word label for each video sample in its input. The evaluation procedure was executed for each one of the three training stages of the word classification network. The results are presented in the Table 7.1 below, in terms of word accuracy.

Number of word classes	Accuracy
5	92.2%
50	71.1%
500	53.2%

Table 7.1: Word accuracies in the LRW test set after each one the tree stages in the word classification network training process.

As can be seen from Table 7.1, the predictive accuracy of the word classification network is 92.2% for five classes and decreases as the number of classes increases. The network trained on the entire dataset can successfully recognize the spoken word in 53.2% of the video samples of the test set.

If we plot the predictive accuracy against the logarithm of the number of words kept from the dataset, we observe that the accuracy decreases almost linearly with the logarithm of words in the vocabulary. This trend is shown in Figure 7.1.

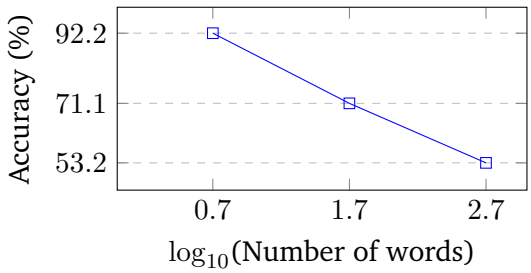

Figure 7.1: The Accuracy against the logarithm of words kept in the vocabulary.

Table 7.2 shows the predictive performance of the proposed lipreading network in comparison with other existing lipreading systems. As can be seen, the current state-of-art network proposed by Stafylakis and Tzimiropoulos (2017) achieves a classification rate of 83.0%, even thought it shares many components with our network. Our word classification network's performance is more close the performance of the lipreading network proposed by Chung and Zisserman (2016a).

Lipreading system	Accuracy
Word classification network (ours)	**53.2** %
Chung and Zisserman (2016a)	61.1 %
Chung et al. (2016)	76.2 %
Stafylakis and Tzimiropoulos (2017)	83.0 %

Table 7.2: Performance compared with other lipreading neural networks on the LRW dataset.

Another interesting aspect of the network's performance would be the words in the vocabulary with the highest and lowest accuracy. As can be seen form Table 7.3, our lipreading network always recognizes successfully the words INFORMATION and WESTMINSTER, which is something we could expect since these words are long and involve a characteristic lip movement. In general, video samples with longer words are classified correctly more often than videos with short words. For instance, words such as YEARS, THINGS, CLEAR and GOING which are short were commonly predicted wrong.

Target word	Accuracy (%)	Target word	Accuracy (%)
WESTMINSTER	100	MINUTES	12
INFORMATION	100	THOUGHT	12
SUNSHINE	96	YEARS	14
ALLEGATIONS	96	THINGS	14
REFERENDUM	96	CLEAR	14
INQUIRY	94	GOING	16
IMMIGRATION	94	UNTIL	18
TEMPERATURES	92	CALLED	18
BEFORE	92	TALKING	18
THEMSELVES	92	STATE	18

Table 7.3: Words with the highest accuracy (left) and words with the lowest accuracy (right).

Next, we present some interesting examples of words from the LRW dataset which were not predicted correctly by the word classification network. These examples are shown in Table 7.4.

Ground truth word	Predicted word
DIFFERENCE	DIFFERENT
UNDERSTAND	STAND
THINGS	SINCE
WINDS	WEEKS
TERMS	TIMES

Table 7.4: Examples of words which were wrongly predicted by the lipreading network.

Finally, the average loss on batches for the training and test (evaluation) sets of the LRW dataset during the three stages of the training process is shown in Figure 7.2. The evaluation average loss is the average loss on all batches of the test set during the evaluation procedure. However, the training average loss refers to the average loss on batches of the training set during the last steps of the training process. As can be observed from the bar chart, in the first stage where the network is trained for five word classes the evaluation loss is ten times larger than the training loss. This behavior suggests that there exists over-fitting. However, the evaluation loss is still small and the predictive performance of the network very good. In the second stage, where the network is trained on fifty words, there is again a large discrepancy between the training and evaluation loss. On the contrary, the two losses are almost the same in the final step, where the network is trained on the entire dataset with five hundred words. This could be explained by the fact that we added dropout to all BiLSTM layers and we used L2 regularization (weight decay) during training.

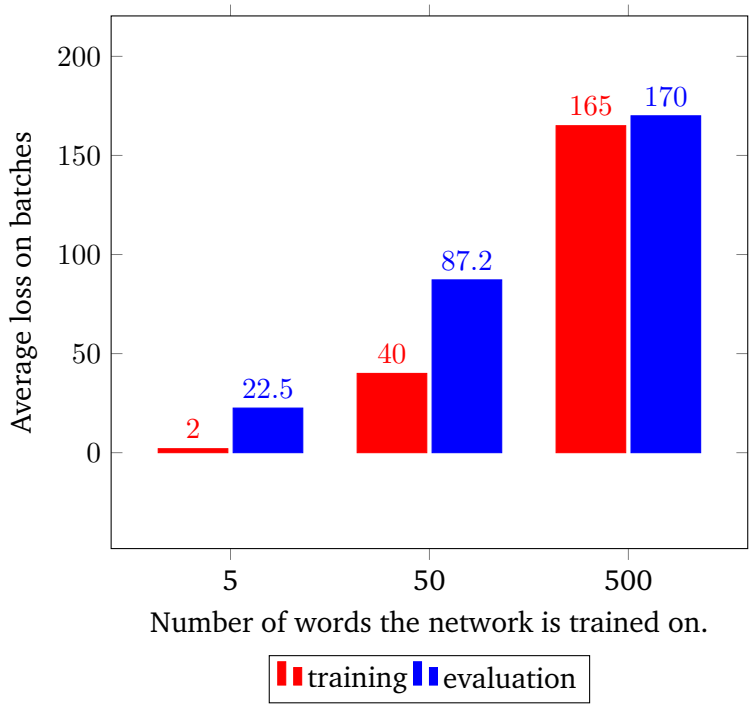

Figure 7.2: The training and evaluation average loss on batches for the training and test (evaluation) sets of the LRW dataset.

7.2 Character Decoding Network Evaluation

The next step would be to evaluate the performance of the second neural network, which receives video samples and produces a sequence of characters as a prediction for each video sample. We assume that a word is predicted correctly if and only if the character sequence produced by the character decoding network coincides with the characters in the ground truth word. As mentioned in Section 6.4, we use schedule sampling during the training procedure. More specifically, we randomly use the predicted character from the previous output time step as input to the LSTM transducer for the prediction of the character in the current time step. Therefore, instead of using the ground truth character of the previous output time step, the predicted character is used with probability p_{pred}. This probability is zero for the first epoch and increases for each one of the next four epochs.

We run the evaluation process on the test set of LRW after the second, third and fifth epoch of training and obtain the accuracy results of the neural network shown in Table 7.5.

Epochs	p_{pred}	Accuracy
2	0.2	39.02 %
3	0.4	41.33 %
5	0.85	**41.34** %

Table 7.5: Word accuracies in the LRW test set, after two, three and five training epochs.

As can be seen from the table, the predictive performance of the character decoding network is improved from the second to the third epoch, but not from the third to the fifth where p_{pred} is increased above 0.4. Another interesting aspect of the evaluation procedure is the average loss on batches, which is demonstrated in Figure 7.3.

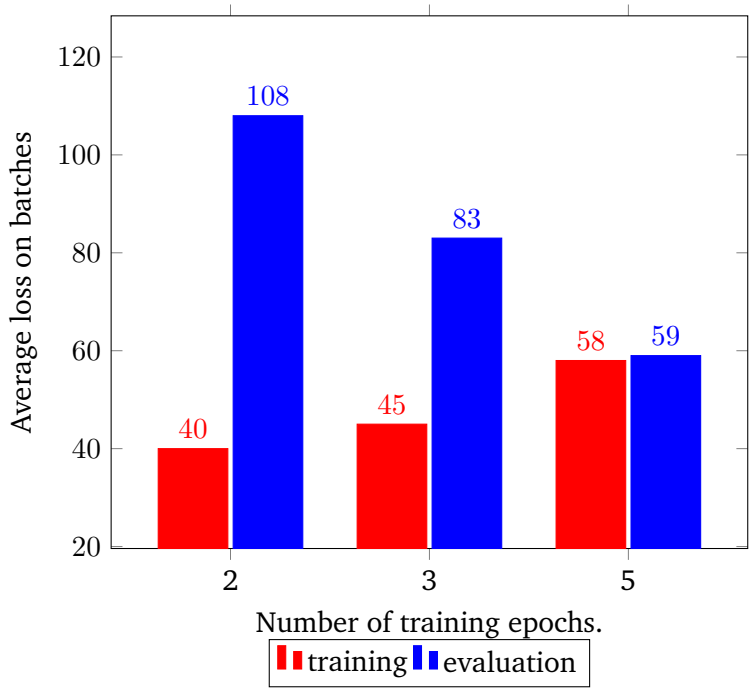

Figure 7.3: The training and evaluation average loss on batches for the training and test (evaluation) sets of the LRW dataset for epochs two, three and five.

The average training loss on batches during the last optimization steps of the second epoch is around 40. However, the average loss on batches during the evaluation of the trained network on the test set is almost three times larger. This behavior can be explained from the fact that p_{pred} is small during the first two epochs (0 and 0.2 respectively) and the network is not tolerant to incorrect predictions of characters from previous output time steps in the decoder. After the fifth epoch, where the network was trained with $p_{pred} = 0.85$, the training and evaluation average loss on batches are almost the same. Nonetheless, the predictive accuracy of the network is not significantly increased between epochs two and five even if the average batch loss during evaluation is reduced in half, from 108 to 59.

Table 7.6 illustrates some examples of wrong character sequence predictions on video samples from the test set.

Ground truth word	Predicted sequence of characters
SCOTLAND	s c o t t i s h
SAYING	s a k i n g
BUSINESS	b i s s i n g
PHONE	f o r c e
LARGE	c h a r g e
THOUGHT	s h o r t

Table 7.6: Examples of words that were wrongly predicted by the character decoding network.

Finally in Table 7.7 , we present the most and least successfully recognized words of the LRW dataset by the character decoding network.

Target word	Accuracy (%)	Target word	Accuracy (%)
ALLEGATIONS	98	NEEDS	0
WESTMINSTER	98	BETTER	0
INFORMATION	96	LATER	0
SUNSHINE	96	STATES	0
REFERENDUM	94	CERTAINLY	0
DESCRIBED	92	PAYING	2
EVERYBODY	90	CLEAR	2
CAMPAIGN	90	SENSE	2
WEAPONS	88	GETTING	2
INQUIRY	88	YEARS	2

Table 7.7: Words with the highest accuracy (left) and words with the lowest accuracy (right).

Comparing the results on words accuracies between the two networks, shown in Table 7.7 and Table 7.3 we can observe that the character decoding network performs poorly for some words by yielding a zero percent accuracy. To the contrary, the word classification network provides at least 12% predictive accuracy for all words in the test set. Another important feature is that both networks classify the words ALLEGATIONS, WESTMINSTER, SUNSHINE, INFORMATION, REFERENDUM and INQUIRY correctly with high accuracy, whereas they both fail to recognize words such as CLEAR and YEARS.

Chapter 8

Conclusions and Future Work

8.1 Lipreading Networks Summary

Lipreading is the ability to recognize what is being said by as speaker, using visual information only. Lipreading a a challenging problem both for human beings and machines, since there is a high level of ambiguity in the lip movements that produce a word or sentence. Two or more words could be produced by the same lip movements sequence, hence one could argue that predicting the spoken word based only on visual data is a hard problem. These ambiguities can be resolved to an extent using the context of neighboring words in a sentence. However, for the purposes of this project, we were focused on recognizing single words and therefore contextual information of full sentences could not be exploited.

In this project, we presented two different lipreading neural networks which perform word prediction on videos where speakers utter a single word. Both neural networks process video samples and produce predictions for the spoken words in the frame sequences. We used the Lip Reading in the Wild (LRW) dataset, described in Chapter 3, in order to train and evaluate the performance of both lipreading neural networks. The LRW dataset was initially developed and presented in Chung and Zisserman (2016a) and has been used by numerous lipreading networks developed since then.

Much research in automated lipreading suggests that deep neural networks can be successfully utilized to tackle the lipreading problem. The main neural networks that have been used so far as building components of deep lipreading systems are the Convolutional neural networks (CNNs) and the Long short-term memories (LSTMs). Moreover, attention-based LSTM transducers, which had been usually employed in speech recognition, were proven by Chung et al. (2016) to perform very well in lipreading, yielding state-of-art results in full sentences prediction. Later, Stafylakis and Tzimiropoulos (2017) suggested that a 3D CNN with a Residual network and a bidirectional LSTM could perform even better in word classification on the LRW dataset. Using as inspiration these recent advancements in lipreading networks, we proposed a word classification and a character decoding network. The first one consists of a 3D CNN followed by multiple parallel CNNs with shared weights and a BiLSTM with three hidden layers. The second network uses the first one as a feature extractor in the front end and employs an attention-based LSTM transducer in the back end to perform classification in the character level.

Both deep lipreading systems were implemented in Python with the Tensorflow library which provides a robust API for designing, training and evaluating the performance of neural networks.

8.2 Challenges of the Implementation

Designing an efficient mechanism to read the dataset and forward it to the model of the neural network was the first challenge presented in the direction of implementing a lipreading network in Tensorflow. The LRW dataset encloses five hundred thousand training video samples and loading the entire training set in DRAM would be impossible. Moreover, the fact that deep networks are trained with mini batches of data suggested that we should implement a mechanism that reads batches from the disk and transfers them to the network. In case data samples are stored in a format such as CSV files or fixed length records, Tensorflow provides reading methods that create batches of samples efficiently. However, video samples in the LRW dataset are stored in the .mp4 format and external Python libraries should be employed to read such files. We used skvideo.io library which is designed to open and read frames from .mp4 files. Nonetheless, reading multiple video files and extracting images sequentially was not efficient in terms of execution time. Considering that the same procedure should be repeated in each step of the optimization process during training, we conclude that the procedure of generating batches of samples must be as fast as possible, so that the overall training time is not dissuasive. For this reason we decided to employ Python Threads in order to read video files and generate data samples for the batches in parallel. Tensorflow provides multithreaded queues, where multiple Threads place items in a queue and another Thread extracts samples to form batches for the training or network evaluation procedure. Nonetheless, in practice this method was not performing as expected in terms of time efficiency. The main reason was that the main Thread that performed the training operation restricted the performance of reading Threads. In order to disengage the batch generation and the training (or evaluation) procedure we chose to implement them in separate Python Processes. The Multiprocessing library was employed to generate two Processes, one responsible for launching multiple Python Threads to read .mp4 files and create data samples and another one responsible for performing optimizations steps during training or evaluating the predictive performance during inference. The architecture of the lipreading application for both proposed neural networks was presented in detail in 5.2.

The second major obstacle emerged during the training procedure of the two lipreading networks. Training the two deep networks end-to-end, without initializing the trainable parameters from other pre-trained networks, turned out to be a challenging task. During the first attempt to execute the training operation in the word classification network the total loss remained in the same levels throughout the optimization process. Applying batch normalization to the hidden layers of the 3D and 2D CNN mitigated the problem, though the loss remained large during training. For this reason, we exercised a training strategy with three stages. The network was initially trained for five classes on a small subset of the LRW dataset. In next stage, the optimization process started with the learned parameters from the previous stage, but for fifty classes of words this time. Finally, during the third and final stage, the full network was optimized on the entire dataset of five hundred classes using the previously learned values of the trainable parameters. The character decoding network, which essentially includes the first network, was trained with the same philosophy, by utilizing the previously learned values of the word classification network.

Another more technical problem that appeared in the process of designing both lipreading networks was related to the variable length of frame sequences in the input. This issue was tackled in different parts of both networks with various methods. We used zero padding in the input level and dynamic LSTM cells in the BiLTSM network, so that only the time steps corresponding to actual frames from the video to be considered during back propagation in time. Additionally, we used a mask of weights so that cross entropy loss would be computed only on valid time steps for each sample in the batch.

8.3 Interpretation of the Results

The results of the evaluation process on both proposed neural networks were presented in Chapter 7. The word classification network achieved a predictive accuracy of 53.2% on the test set of LRW. Although this accuracy is lower than the one achieved by the state-of-art network of Stafylakis and Tzimiropoulos (2017), it is close to the accuracy of the CNN-based networks proposed in Chung and Zisserman (2016a). Moreover, a classification accuracy of 53.2%, in combination with the fact that the training and evaluation loss values on a batch are almost identical, indicates that the network was successfully trained and generalized well on unseen video samples of the test set. The second network, which predicts sequences of character labels instead of word labels, yielded an accuracy of 41.34%. Each word was considered to be predicted correctly if and only if the predicted sequence of characters was exactly the same with the target word. The character decoding network performed poorly on the RLW dataset in comparison with the similar network with an encoder and a decoder proposed by Chung et al. (2016). However, that network was pre-trained on another large dataset with full sentences and was fine-tuned for one epoch on LRW. That huge dataset with full sentences enabled their network to learn better parameters in the attention mechanism, something that may not be possible on a dataset with single word videos.

8.4 Limitations and Future Work

Both lipreading networks proposed in this project were successfully trained and demonstrated strong predictive abilities on the LRW dataset. However, neither of them reached the state-of-art performance of the network presented by Stafylakis and Tzimiropoulos (2017). Apparently, there is a number of limitations in our approach that could be resolved in the future.

8.4.1 2D CNN Depth

One major limitation of both neural networks lies in the core of the encoder. The 2D CNN, which could by viewed as multiple parallel 2D CNNs with shared weights, consists of three convolutional layers each one followed by a max-pooling layer. This is arguably not a very deep CNN and its size could be the bottleneck of the entire encoder,since it may not have enough capacity to extract high quality visual features. Therefore, we could replace the existing CNN with a VGG convolutional network that would be more deep, with an architecture which has been proved to outperform others in image feature extraction.

8.4.2 Employ a Residual Network

The current state-of-art neural network for word classification on the LRW dataset proposed by Stafylakis and Tzimiropoulos (2017) contains a Residual network instead of a 2D CNN. In general, Residual networks have been proven highly efficient in image classification tasks and they often outperform VGG convolutional networks. Their main advantage is their depth that can reach 150 hidden layers, so Residual networks can demonstrate stronger data representation abilities than CNNs. However, a lipreading system with a deep Residual network as a visual feature extractor would require substantially more computational resources (multiple GPUs) to get trained and evaluated, because of its huge size.

8.4.3 Beam Search in Character Prediction

During the evaluation of the character decoding neural network, we use the character predicted in the previous output time step as input to the LSTM transducer for the next step. This character is always chosen to be the one with the maximum probability in the probability vector produced by the Softmax unit. Chung et al. (2016) showed that implementing a beam search strategy could increase the predictive ability of an encoder-decoder network for lipreading. At each output time step, the hypotheses in the beam are expanded with every possible character, and only the $beam\_width$ most probable hypotheses are kept. Therefore, we could use beam search during evaluation and search for character sequences with higher joint probability instead of being greedy in each step and choosing the character with the larger probability which may lead to suboptimal sequences.

8.4.4 Full Sentences Dataset

After incorporating a deeper CNN or a Residual network in the character decoding network and employing a beam search strategy during inference, we could train the character decoding network to recognize full sentences instead of words. We could utilize the already learned parameters in various parts of the network and launch a training procedure in the Lip Reading Sentences (LRS) dataset presented in Chung et al. (2016). In this way the lipreading abilities of the character decoding network could be expanded to recognize full sentences using contextual information as well.

8.4.5 Hyperparameters Optimization

Choosing the correct set of hyperparameters is arguably one of the most challenging tasks. This process includes training a network repeatedly with different combinations of hyperparameters and evaluating its performance on the validation dataset in order to determine the best configuration. For the purposes of this project, the hyperparameters were chosen by studying similar lipreading networks and no optimization took place. Utilization of multiple GPUs could enable a process where several lipreading networks are trained in parallel with a random search on a bounded hyperparameter space. Finally, as can be seen from Figure 6.6 and 6.40 we could use a different learning rate decay strategy with a smaller decay rate and let the leaning rate drop near zero slower in terms of optimization steps.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. pages 3, 25, 35

Anina, I., Zhou, Z., Zhao, G., and Pietikäinen, M. (2015). Ouluvs2: A multi-view audiovisual database for non-rigid mouth motion analysis. In *11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition, FG 2015, Ljubljana, Slovenia, May 4-8, 2015*, pages 1–5. pages 20

Assael, Y. M., Shillingford, B., Whiteson, S., and de Freitas, N. (2016). Lipnet: Sentence-level lipreading. *CoRR*, abs/1611.01599. pages 21

Bahdanau, D., Cho, K., and Bengio, Y. (2014a). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473. pages 5

Bahdanau, D., Cho, K., and Bengio, Y. (2014b). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473. pages 13

Chan, W., Jaitly, N., Le, Q. V., and Vinyals, O. (2015a). Listen, attend and spell. *CoRR*, abs/1508.01211. pages 5

Chan, W., Jaitly, N., Le, Q. V., and Vinyals, O. (2015b). Listen, attend and spell. *CoRR*, abs/1508.01211. pages 13

Chatfield, K., Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Return of the devil in the details: Delving deep into convolutional nets. *CoRR*, abs/1405.3531. pages 20, 21

Chorowski, J., Bahdanau, D., Serdyuk, D., Cho, K., and Bengio, Y. (2015). Attention-based models for speech recognition. *CoRR*, abs/1506.07503. pages 13

Chung, J. S., Senior, A. W., Vinyals, O., and Zisserman, A. (2016). Lip reading sentences in the wild. *CoRR*, abs/1611.05358. pages 3, 5, 13, 21, 22, 31, 54, 65, 68, 72, 74, 75

Chung, J. S. and Zisserman, A. (2016a). Lip reading in the wild. In *Asian Conference on Computer Vision*. pages 2, 4, 20, 21, 22, 23, 24, 68, 72, 74

Chung, J. S. and Zisserman, A. (2016b). Out of time: automated lip sync in the wild. In *Workshop on Multi-view Lip-reading, ACCV*. pages 20

Cooke, M., Barker, J., Cunningham, S., and Shao, X. (2006). An audio-visual corpus for speech perception and automatic speech recognition. *Acoustical Society of America Journal*, 120:2421. pages 21

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`. pages 5

Lucey, P., Potamianos, G., and Sridharan, S. (2007). A unified approach to multi-pose audio-visual asr. In *INTERSPEECH*, pages 650–653. ISCA. pages 17

Lucey, P. J., Sridharan, S., and Dean, D. B. (2008). Continuous pose-invariant lipreading. In *Interspeech 2008*, pages 2679–2682, Brisbane, Australia. Casual Productions. pages 17

Mroueh, Y., Marcheret, E., and Goel, V. (2015). Deep multimodal learning for audio-visual speech recognition. *CoRR*, abs/1501.05396. pages 18

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. `http://www.neuralnetworksanddeeplearning.com/`. pages 7, 9, 10

Noda, K., Yamaguchi, Y., Nakadai, K., Okuno, H., and Ogata, T. (2014). *Lipreading using convolutional neural network*, pages 1149–1153. International Speech and Communication Association. pages 18

Noda, K., Yamaguchi, Y., Nakadai, K., Okuno, H. G., and Ogata, T. (2015). Audio-visual speech recognition using deep learning. *Applied Intelligence*, 42(4):722–737. pages 18

Ong, E. and Bowden, R. (2011). Learning temporal signatures for lip reading. In *Proc. IEEE Int. Conf. Comput. Vis. Workshops (ICCVW)*, pages 958–965. pages 17

Pachoud, S., Gong, S., and Cavallaro, A. (2008). Macro-cuboid based probabilistic matching for lip-reading digits. In *Proc. IEEE Int. Conf. Comput. Vis. Pattern Recognition (CVPR)*, pages 1–8. pages 17

Papandreou, G., Katsamanis, A., Katsamanis, A., Pitsikalis, V., and Maragos, P. (2008). *Adaptive Multimodal Fusion by Uncertainty Compensation with Application to Audio-Visual Speech Recognition*, pages 1–15. Springer US, Boston, MA. pages 18

Petridis, S. and Pantic, M. (2016). *Deep Complementary Bottleneck Features for Visual Speech Recognition*, pages 2304–2308. IEEE International Conference on Acoustics, Speech and Signal Processing. Institute of Electrical and Electronics Engineers. eemcs-eprint-27130. pages 19

Potamianos, G., Neti, C., Iyengar, G., Senior, A. W., and Verma, A. (2001). A cascade visual front end for speaker independent automatic speechreading. *International Journal of Speech Technology*, 4(3):193–208. pages 16, 17

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. S., Berg, A. C., and Li, F. (2014). Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575. pages 21

Stafylakis, T. and Tzimiropoulos, G. (2017). Combining residual networks with lstms for lipreading. *CoRR*, abs/1703.04105. pages 3, 21, 54, 55, 68, 72, 74, 75

Tamura, S., Ninomiya, H., Kitaoka, N., Osuga, S., Iribe, Y., Takeda, K., and Hayamizu, S. (2015). Audio-visual speech recognition using deep bottleneck features and high-performance lipreading. *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, pages 575–582. pages 19

Zhou, Z., Hong, X., Zhao, G., and Pietikäinen, M. (2014a). A compact representation of visual speech data using latent variables. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(1):181–187. pages 16

Zhou, Z., Zhao, G., Hong, X., and Pietikinen, M. (2014b). A review of recent advances in visual speech decoding. *Image and Vision Computing*, 32(9):590 – 605. pages 16