Lip Reading Model Optimization

Samuel Youssef • Jason Chou • Maleye Diakhate
Senior Design Project CSc 59938
Professor George Wolberg
05/17/2019

Contents

I. Executive Summary

The overall goal of this project was to improve an existing lipreading model LipNet. We accomplish this by implementing an unmaintained method of curriculum training that speeds up convergence as well as modifying hyperparameters of the model so that it can more easily accept variations in datasets. As a byproduct of this work, we also streamlined the interface for the model for training a variety of different video formats and deployed it on the cloud for scalable processing.

Over the course of this project, we've changed directions many times as a result of failed approaches. Some of these include choosing incompatible datasets for training or picking an unsuitable cloud service to execute our model on. In this paper, we explore the different difficulties we have faced as well as how it contributed to our end product. In doing so, we hope to highlight both the achievements and shortcomings of our work.

II. Introduction

1. Background

Lip reading is the technique of transcribing speech by visually interpreting the movements of the lips, face, and tongue.

Typically, traditional human lip reading relies on information provided by the context and knowledge of the language. However, lipreading is commonly not very precise because some sounds and words look very similar which would lead a lipreader to heavily rely on their own background knowledge of language and on the quality of speech of the speaker. This makes lipreading is notoriously difficult task for humans, especially in the absence of context. Consequently, human lipreading performance is poor. Hearing-impaired people achieve an accuracy of only 17±12% even for a limited subset of 30 monosyllabic words and 21±11% for 30 compound words.[1] On top of that, there are some situations that make it even more difficult to lipread in such as lipreading in large, noisy groups and meetings.

Machine lip reading, on the other hand, is difficult because it requires extracting spatiotemporal features from the video since both the position and motion of the lips are important. Traditional automation approaches separate the problem into two categories: (1) designing or (2) learning visual features to predict a sound. In this paper, we attempt to extract these visual features with a novel method using an existing lipreading model.

2. Motivation

The main motivation for this project stems from wanting to approach a problem that is well suited to being solved with an integration of image processing techniques as well as machine learning, as opposed to traditional programmatic methods. We wanted a problem that let us sharpen both our machine learning skills as well as our image processing skills. On top of that, we found the problem of lipreading to be highly beneficial to hearing-impaired individuals. If computers were able to lip read, that may improve the quality of life for these individuals.

[1] https://link.springer.com/article/10.3758/BF03204211

An important goal, therefore, is to automate lipreading. Machine lipreaders have enormous practical potential, with applications not only in what was just mentioned but also in:

- improving hearing aids
- silent dictation in public spaces
- security
- speech recognition in noisy environments
- biometric identification
- silent film processing

3. Metrics

We first define the metrics for how we evaluate our model through these four metrics: (1) character error rate, (2) word error rate, (3) bilingual evaluation understudy score, and (4) loss rate.

A. Character Error Rate

Character Error is a common metric performance of a speech recognition system. It allows us to obtain the error of the models made decrypting characters. Derived from the Levenshtein distance, it is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

B. Word Error Rate

In speech recognition, word error rate (WER) is the most commonly used metric performance of a speech recognition system. WER is obtained by comparing the prediction sentence from our model to the reference sentence. It uses a method that is essentially the Levenshtein distance as mentioned in Section II.3.A applied to a sequence of words rather than a sequence of characters. However, unlike Levenshtein distance, the WER counts the number of deletions, insertion and substitutions done as a whole, instead of just summing up the penalties. Widely used for scoring speech recognition systems, we define WER as:

$$WER = \frac{S+D+I}{S+D+C} \tag{1}$$

In equation 1, S is the number of substitutions, D is the number of deletions, I is the number of insertions, and C is the number of correct words.[2] Here is an example of the calculation:

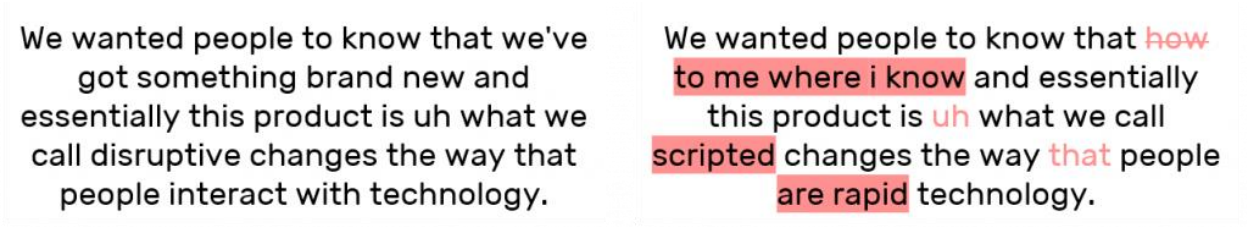


Fig. 1. Example reference sentence on the left and example predicted sentence on the right

[2] https://holianh.github.io/portfolio/Cach-tinh-WER/

The total of substitutions, insertions, and deletions is 11 and then dividing that by the total number of words spoken in the reference sentence, 29, we get a word error rate of about 38 percent. One difficulty or inaccuracy is that WER can't calculate when the entire meaning of the sentence has changed. Another is that in some instances, the recognized word sequence can have different lengths from the reference word sequence.

C. BLEU

The BLEU (bilingual evaluation understudy) score is another metric for evaluating a generated sentence to a reference sentence. BLEU score implementation is used for sentence similarity detection. It is used at the Corpus evaluation since any sentence that does not match at least one 4-gram match will be given a score of 0. Core BLEU calculates the geometric mean of n-gram precisions that is scaled by a brevity penalty to prevent very short sentences with some matching material from being given inappropriately high scores.

Quality is considered to be the correspondence between a machine's output and that of a human: "the closer a machine translation is to a professional human translation, the better it is"

D. Loss Rate

Training a model means learning good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called empirical risk minimization. Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. The goal of training a model is to find a set of weights and biases that have low loss, on average, across all examples.[3]

4. Related Work

In this subsection, we detail the main papers that we have read and followed. They're ordered by chronology as well as complexity: 3D CNNs for Cross AVR Matching was what we started off with, but we transitioned into the LipNet paper as well as the Lip Reading in the Wild and Lip Reading Sentences in the Wild papers.

Of these, the one that we really stuck with was LipNet. Throughout this paper, we'll discuss the reasons for why we stayed with the second paper of the four and some decisions we've made as a result of it.

A. 3D CNNs for Cross AVR Matching

To approach this problem, we first started off with this paper that related audio and visual components. 3D Convolutional Neural Networks (CNNs) for Cross Audio-Visual Recognition (AVR) Matching[4] explains the approach of leveraging the extracted information from one modality to improve the recognition ability of the other modality by complementing the missing information.

By finding the correspondence between the audio and visual streams, which is the goal of this work. Torfi, et al. used coupled 3D Convolutional Neural Network (3D CNN) architecture that can

[3] https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss

[4] https://arxiv.org/pdf/1706.05739.pdf

map both modalities into a representation space to evaluate the correspondence of audio-visual streams using the learned multimodal features.

Essentially, they matched mouth movements with the sound. We determined this to be a little tough to start with although we did spend quite some time learning the fundamentals of 3D CNNs due to this paper.

B. *LipNet: End-to-End Sentence-Level Lipreading*

LipNet is a model that maps a variable-length sequence of video frames to text. They make use of spatiotemporal convolutions, a recurrent network, and the connectionist temporal classification loss, trained entirely end-to-end. We will explain more of this in Section IV. This paper was also cited by the following paper Lip Reading Sentences in the Wild.[5]

C. *Lip Reading Sentences in the Wild*

Lip Reading Sentences in the Wild[6] is an upgraded version of LipNet that has a 'Watch, Listen, Attend and Spell' (WLAS) network. As a part of this WLAS network, there are 5 3D CNNs and a more advanced spelling architecture. At each time step, the decoder outputs a character as well based on two LSTM attention vectors. The attention vectors are used to select the appropriate period of the input visual and audio sequences. The reason for this more advanced architecture is because Lip Reading Sentences in the Wild tackles a much more natural language dataset than the one LipNet deals with.

This paper also details a strategy where they train on single word examples and then let the sequence length grow as the network trains. These short sequences are parts of the longer sentences in the dataset. They observed that the rate of convergence on the training set is several times faster because of this method.

III. Neural Network Structures

Here, we detail the neural network structures that are relevant to our project including ones we've deemed relevant and have considered as well as structures we are currently using in LipNet.

Fig. 2. Key for nodes in figure 3 and 4 diagrams of neural network models[7]

[5] https://arxiv.org/pdf/1611.01599.pdf
[6] https://arxiv.org/pdf/1611.05358.pdf
[7] http://www.asimovinstitute.org/neural-network-zoo/

1. Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are a class of deep neural networks, most commonly applied to analyze visual imagery.[8] Spatiotemporal CNNs (STCNNs) are a subset of these, specialized in extracting features from visual frames with respect to time and space. Generally, STCNNs are three-dimensional (3D): they have a dimension for length, width, and time, each.

CNNs consist of four main layers: convolutional, pooling, fully connected, and normalization layers. A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, RELU layer i.e. activation function, pooling layers, fully connected layers and normalization layers. In the case of LipNet, the normalization layer is the softmax algorithm.

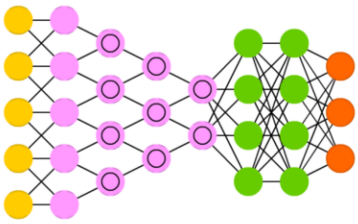


Fig. 3. Example CNN model

LipNet uses 3D STCNNs to analyze video data divided into frames of images. Specifically, LipNet uses three of these convolutional layers. For more information on this, see Section IV.1. Each convolution has the following parameters:

- filters: the dimensionality of the output space or number of filters in the convolution (32 for the first convolutional layer, 64 for second, and 96 for the third)
- kernel size: list of 3 integers specifying the depth, height and width of the 3D convolution window ([3, 5, 5] for the first two convolutional layers, [3, 3, 3] for the third)
- strides: list of 3 integers specifying the strides of the convolution along the depth, height, and width
- activation: activation function
- kernel initializer: an initializer for the kernel function

2. Recurrent Neural Network (RNN)

A Recurrent Neural Network (RNN) is a class of artificial neural network that exhibits temporal dynamic behavior.[9] RNNs use their internal states of "memory" to process sequences of inputs. They are commonly used in speech recognition problems due to its usefulness in recognizing temporal elements.

[8] https://docs.w3cub.com/tensorflow~python/tf/layers/conv3d/

[9] https://arxiv.org/abs/1406.1078

RNNs are designed to recognize a data's sequential characteristics and use patterns to predict the next likely scenario. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. One of the appeals of RNNs is the idea that they may be able to connect previous information to the present task such as using previous video frames might inform the understanding of the present frame. There are two main examples: one where context is obvious and where context is not:

1. Context is obvious: “The clouds are in the [blank].”
2. Context is not obvious: "I grew up in France.” followed by a long pause and then “I speak [blank] fluently."

In the first example, the word is obviously sky. No more context is needed and that is where traditional RNNs succeed in doing. In the second example, the first sentence is far away in the text from the second sentence. It becomes less and less obvious to put “French” in the blank when we are predicting. Hence, the job of traditional RNNs become more significantly difficult and they often fail to predict the word correctly. However, for our purposes, RNNs work perfectly since we don’t need to consider paragraph level contexts.

RNNs have variations such as Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM). As shown in figure 3 below, GRUs and LSTM have different memory cells.

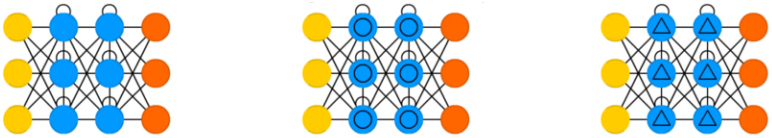


Fig. 4. Example RNN models; From left to right: (1) typical RNN, (2) LSTM, and (3) GRU

A. Gated Recurrent Unit (GRU)

In our project we used Gated Recurrent Units.

Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks.[10] The GRU is like a long short-term memory (LSTM) with forget gate but has fewer parameters than LSTM, as it lacks an output gate. GRU's performance on certain tasks of polyphonic music modeling and speech signal modeling was found to be similar to that of LSTM. GRUs have been shown to exhibit even better performance on certain smaller datasets.

B. Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) are an artificial RNN that has feedback connections that make it a general-purpose computer and can process entire sequences of data such as speech and videos.[11] GRUs are similar to long short term memory with forget gate . However, it has fewer parameters than LSTM.

Long Short-Term Memory (LSTM) networks are a special kind of RNN capable of learning long-term dependencies. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is their default behavior.

[10] https://arxiv.org/abs/1406.1078

[11] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

The input is a sequence of observations, and the outputs are a sequence of labels, which can include blank outputs. The difficulty of training comes from there being many more observations than there are labels. For example, in speech audio, there can be multiple time slices which correspond to a single phoneme. Since we don't know the alignment of the observed sequence with the target labels, we predict a probability distribution at each time step.

Recurrent neural networks (RNNs) such as LSTM networks to tackle sequence problems where the timing is variable. It can be used for tasks like on-line handwriting recognition or recognizing phonemes in speech audio.

However, as shown by Gail Weiss & Yoav Goldberg & Eran Yahav, the LSTM is "strictly stronger" than the GRU as it can easily perform unbounded counting, while the GRU cannot. That's why the GRU fails to learn simple languages that are learnable by the LSTM. Similarly, as shown by Denny Britz & Anna Goldie & Minh-Thang Luong & Quoc Le of Google Brain, LSTM cells consistently outperform GRU cells in "the first large-scale analysis of architecture variations for Neural Machine Translation."

3. Connectionist Temporal Classification (CTC)

Connectionist Temporal Classification (CTC) is a type of neural network output associated with a scoring function. It is commonly used alongside RNNs such as GRUs or LSTM. It is usually used when time is variable as well as recognizing phonemes in speech recognition.[12]

CTC networks have a continuous output (i.e. softmax in the case of LipNet), which is fitted through training to model the probability of a label. This softmax is used in to map the non-normalized output to a probability distribution over predicted output class. CTC does not attempt to learn boundaries and timings. Label sequences are considered equivalent if they differ only in alignment, ignoring blanks.

CTC is widely used in modern speech recognition as it eliminates the need for training data that aligns inputs to target outputs. Given a model that outputs a sequence of discrete distributions over the token classes (vocabulary) augmented with a special “blank” token, CTC computes the probability of a sequence by marginalizing over all sequences that are defined as equivalent to this sequence. This simultaneously removes the need for alignments and addresses variable-length sequences.

4. Beam Search

Beam search is used in systems to maintain large systems as it stores the entire search tree. Beam search is very often used in translation. To select the best translation, each part is processed, and many different ways of translating the words appear. The best translations according to their sentence structures are kept, and the rest are discarded.[13] LipNet adapted this algorithm to decode.

[12] https://en.wikipedia.org/wiki/Connectionist_temporal_classification

[13] https://towardsdatascience.com/beam-search-decoding-in-ctc-trained-neural-networks-5a889a3d85a7

IV. LIPNET MODEL

The LipNet codebase[14] we based our project around was implemented by a contributor separate from the authors of the paper described in Section II.4.B. From here on, the contributor will be referred to as the LipNet maintainer.

LipNet is a supervised learning model that is trained end-to-end to perform word classification rather than sentence-level sequence prediction. It maps a variable-length sequence of video frames to text. LipNet achieves 95.2% accuracy in the sentence-level for GRID corpus, outperforming even experienced human lipreaders.

1. Neural Network Model

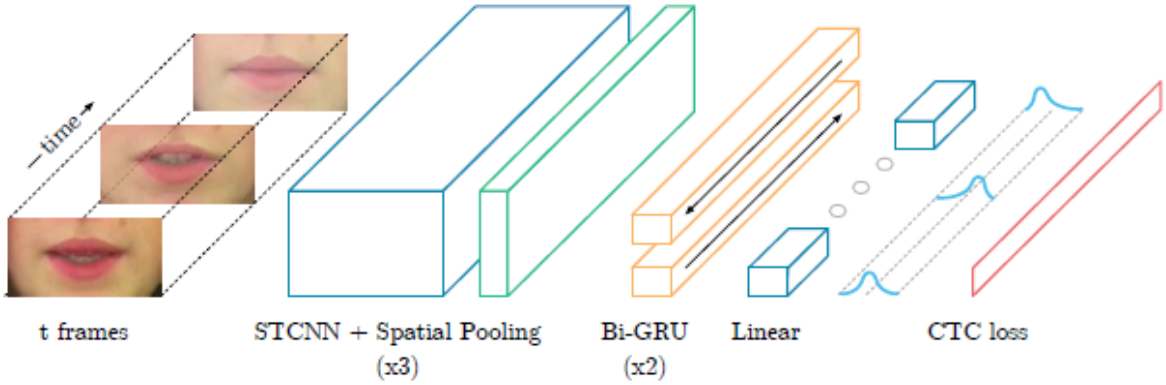


Fig. 5. LipNet model; from left input to right output: t frames → spatiotemporal convolutional neural network → bidirectional gated recurrent unit → linear transformation (along with a softmax not pictured) → connectionist temporal classification loss

Figure 5 illustrates the LipNet architecture, which starts with 3 spatiotemporal convolutions, channel-wise dropout, spatial max-pooling. Subsequently, the features extracted are followed by two Bi-GRUs. The Bi-GRUs are crucial for efficient further aggregation of the STCNN output. Finally, a linear transformation is applied at each time-step, followed by a softmax over the vocabulary augmented with the CTC blank, and then the CTC loss. All layers use rectified linear unit (ReLU) activation functions. Figure 6 below gives more details on the parameters.

[14] https://github.com/rizkiarm/LipNet

Layer	Size / Stride / Pad	Input size	Dimension order
STCNN	$3 \times 5 \times 5 \,/\, 1,2,2 \,/\, 1,2,2$	$75 \times 3 \times 50 \times 100$	$T \times C \times H \times W$
Pool	$1 \times 2 \times 2 \,/\, 1,2,2$	$75 \times 32 \times 25 \times 50$	$T \times C \times H \times W$
STCNN	$3 \times 5 \times 5 \,/\, 1,2,2 \,/\, 1,2,2$	$75 \times 32 \times 12 \times 25$	$T \times C \times H \times W$
Pool	$1 \times 2 \times 2 \,/\, 1,2,2$	$75 \times 64 \times 12 \times 25$	$T \times C \times H \times W$
STCNN	$3 \times 3 \times 3 \,/\, 1,2,2 \,/\, 1,1,1$	$75 \times 64 \times 6 \times 12$	$T \times C \times H \times W$
Pool	$1 \times 2 \times 2 \,/\, 1,2,2$	$75 \times 96 \times 6 \times 12$	$T \times C \times H \times W$
Bi-GRU	256	$75 \times (96 \times 3 \times 6)$	$T \times (C \times H \times W)$
Bi-GRU	256	75×512	$T \times F$
Linear	$27 + \text{blank}$	75×512	$T \times F$
Softmax		75×28	$T \times V$

Fig. 6. LipNet neural network architecture hyperparameters

2. Alignment

```
0     12250 sil
12250 19250 set
19250 27250 white
27250 30500 with
30500 36000 p
36000 43250 two
43250 55250 soon
55250 74500 sil
```

Fig. 7. Sample alignment file

The above shows a sample align file. This describes the time of a given video. On the left, the numbers can be converted into seconds using equation 2 below, where t_a is the time in figure 7 and t_s is the time in seconds.

$$t_s = \frac{t_a}{25000} \tag{2}$$

Using this formula, we can make our own align files.

3. Dictionary

```
set white with p two soon
bin green with y eight please
set green with j six please
place green at y three again
bin green by r seven soon
bin blue with q four again
...
```

Fig. 8. Sample dictionary file

The dictionary file above in figure 8 is a collection of all the possible different sentences from the dataset. Since LipNet uses supervised learning, it matches these sentences up to train.

4. Optimizer

Adaptive Moment Estimation (Adam) is an optimization algorithm that uses network weights iterative based in training data. It is an extension to stochastic gradient descent.

Adam configuration Parameters are the followings:

alpha: also known as the learning rate

beta1: exponential decay for the first moment estimates

beta2: exponential decay rate for the second moment estimates. This value should beset close to 1 on problems with a sparse gradient.

epsilon. This is a very small number to prevent any division by zero in the implementation

Adam parameters are set as follows:

```
Adam = Adam (lr=0.0001, beta_1=0.9, beta_2=0.999, epsilon = 1e-08)
```

Fig. 9. Adam optimizer where α is 0.0001, β_1 is 0.9, β_2 is 0.999, and ε is 10^{-8}

We did not change the Adam optimizer from the original implementation since it already had its learning rate tuned by the implementors. The β_1, β_2, and ε values were kept at the usual default recommended ones. Further work can be done by integrating an automated hyperparameter tuning library to ensure its validity. We chose to stick with Adam since it had faster convergence than Stochastic Gradient Descent (SGD), especially for the GPU we ended up using.[15] We also preferred the tradeoff of overfitting over slow convergence since our datasets were on the homogeneous side.

V. IMPLEMENTATION

This section discusses the implementation details and especially failures we have had when setting up LipNet as well as training it. Each subsection discusses a different aspect of the implementation.

1. Cloud Computing Platforms

This subsection is about the cloud computing platforms we have attempted to deploy on. Keep in mind that our experiences are only with the free tier and not reflective of the rest of each of these services. Originally, we had planned on using a pipeline that combined both Amazon EC2 and Amazon SageMaker but we realized that couldn't work when we discovered issues with training on SageMaker.

Here are the main libraries that we needed for training and predicting:

Table 1. Libraries Used

Library	*Usage*
keras	neural network running on top of tensorflow
tensorflow	tensorflow

[15] https://medium.com/bigdatarepublic/cost-comparison-of-deep-learning-hardware-google-tpuv2-vs-nvidia-tesla-v100-3c63fe56c20f

ffmpeg	video decoding
dlib	facial feature extraction (requires CMake and Boost to compile)

A. *Amazon EC2*

Amazon Elastic Compute Cloud (EC2) provides scalable computing capacity in the Amazon Web Services (AWS) cloud. Using Amazon EC2 eliminates the need to invest in hardware up front, so we can develop and deploy applications faster. At the earlier stages of this project, we attempted to use the free tier services of EC2 by initializing a `t2.micro` virtual machine. Amazon provides around 750 hours for free each month for 12 month, but only few instances types are eligible for that free tier. However the computing capabilities of such instance were not matching the requirements we needed to deploy our model and start training the data we had in hand. The various types of instances offered under the free tier were not equipped with enough memory. The specs for the `t2.micro` instance were the following:

- Memory : 1 GiB
- vCPUs: 1
- Network performance: low to moderate.

To train our data, we were in need of a GPU so that we can speed up the processing of data through various layers of the neural network. No GPUs were offered on amazon EC2 for free, so we decided to move forward and try to use Amazon SageMaker to deploy and train our model.

Amazon EC2 Key Pairs. Amazon EC2 uses public–key cryptography to encrypt and decrypt login information. Public–key cryptography uses a public key to encrypt a piece of data, such as a password, then the recipient uses the private key to decrypt the data. The public and private keys are known as a key pair. When we were setting up the instances, having to figure out the cryptography and setting up the user credentials slowed us down a bit.

B. *Amazon SageMaker*

Amazon SageMaker is a fully managed machine learning service. It provides an integrated Jupyter authoring notebook instance for easy access to your data sources for exploration and analysis, so you don't have to manage servers. It enables data scientists and developers to quickly and easily build, train, and deploy machine learning models at any scale.[16] The free services offered by SageMaker were the following:

- 250 hours per month of `t2.medium` notebook usage for the first two months
- 50 hours per month of `m4.xlarge` for training for the first two months
- 125 hours per month of `m4.xlarge` for hosting for the first two months

None of these instance types were a good fit for our model. The reasons behind most difficulties using the above instance types were of memory and CPU capabilities. The `t2.medium` instance provides only 4GB of RAM which was not enough as we were in need of at least 16 GB of memory. The other two instance types above provided very limited usage time under the free tier which diminished the possibility of successfully deploying our model and training our data.

[16] https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html

Another difficulty we faced on SageMaker platform is that once we initialized any of the above aforementioned instances, we had to keep it always running. The reason behind this is once the instance is turned off, all installed Python dependencies are completely removed and only the dependencies which came with the instance are kept. To avoid this, we had to write a bash script and add the script to the configuration of the instance so that we do not lose the Python environment upon reinitializing the instance.

Overall, the difficulties we faced on SageMaker platform including the lack of free and accessible GPU instances led us to deploy and train our model on Google Cloud platform on which we had access to powerful memory and GPU capabilities. This was the point when we tried it out on Sam's laptop with 16GB of RAM and realized that we needed a lot of memory to train.

C. Google Compute Engine (GCE)

Google Compute Engine (GCE) offers virtual machines running in Google's data centers connected to its worldwide fiber network.[17] The tooling and workflow offered enables scaling from single instances to global, load-balanced cloud computing. We first looked into GPUs because that was what each of the cloud computing platforms before lacked. Table 1 shows the options we had considered.

Table 2. NVIDIA Tesla GPUs[18] [19]

Tesla GPU	*Available*	*VRAM*
T4	✓	16GB
P4	✓	8GB
V100		16GB
P100	✓	16GB
K80		24GB

Neither the NVIDIA Tesla V100 nor NVIDIA Tesla K80 were not available in `us-east-1b`, where we set up our GCE instance. So, we decided to go with the NVIDIA Tesla P100 which edged the T4 out in terms of computation power. Getting access to the GPU was easy, we requested a quota increase via the GCP website and they granted us the increase within one working day. Here's how we set up GCP:

```
gcloud compute firewall-rules create admin \
    --direction=INGRESS \
    --priority=1000 \
    --network=default \
    --action=ALLOW \
    --rules=tcp:3000,tcp:10000 \
    --source-ranges=0.0.0.0/0 \
    --target-tags=admin
```

Fig. 10. Starting up firewall rules to download the data

First, we set up the firewall rules as shown in figure 13. Then, we set up the actual instance below in figure 14:

[17] https://cloud.google.com/compute/

[18] https://developer.nvidia.com/deep-learning-performance-training-inference

[19] https://cloud.google.com/blog/products/gcp/introducing-improved-pricing-for-preemptible-gpus

```
gcloud compute instances create pipeline \
    --custom-cpu 6 --custom-memory 16 \
    --zone us-east-1b \
    --accelerator type=nvidia-tesla-p100,count=1 \
    --image-family ubuntu-1604-lts \
    --image-project gce-uefi-images \
    --maintenance-policy TERMINATE
    --restart-on-failure \
```

Fig. 11. Gcloud command to create an instance called pipeline with 6 vCPUs, 16GB RAM, 1 NVIDIA Tesla P100, and Ubuntu 16.04[20]

With GCE, the VMs boot quickly, come with persistent disk storage, and deliver consistent performance. The machines were available in many configurations including predefined sizes and can also be created with Custom Machine Types (CMTs) optimized for our specific needs. In this case, we used a CMT with 6 vCPUs and 16GB of RAM. We came to this number after determining exactly how much training we needed.

Entering the instance is as simple as the command `gcloud compute ssh`. From there, we installed the GPU driver and then set up the environment. See the appendix for more details on how we set up the GPU driver.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.40.04    Driver Version: 418.40.04    CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  On   | 00000000:00:04.0 Off |                    0 |
| N/A   36C    P0    26W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Fig. 12. `nvidia smi` command to show NVIDIA Tesla P100 was installed and working correctly

By having 16GB of VRAM and 16GB of RAM, we no longer ran into the memory issues we were having before and were able to train. Looking back, we're not exactly sure which was a bigger issue—not having enough computing power or not having enough memory. And yet, we still ran into outstanding issues in the form of the training either stalling or stopping after ≥ 50 epochs. This problem was a lot more approachable since the model had begun training. By using the saved h5 weights from the epoch before as predefined weights for a new run, we were able to resume our run as normal.

[20] https://cloud.google.com/compute/docs/instances/creating-instance-with-custom-machine-type

2. Datasets

A. GRID

The GRID corpus has audio and video recordings of 34 speakers who speak 1000 sentences each, for a total of 28 hours across 34000 sentences. Although the GRID corpus contains entire sentences, we consider only the simpler case of predicting isolated words. LipNet predicts sequences and hence can exploit the temporal context to attain much higher accuracy. Phrase-level approaches were treated as plain classifications.

We use the GRID corpus to evaluate LipNet because it is sentence-level and has the most data. The sentences are drawn from the following simple grammar:

```
command(4) + color(4) + preposition(4) + letter(25) + digit(10) + adverb(4)
```

Fig. 13. Sentence structure of GRID corpus; The number denotes how many word choices there are for each of the 6 word categories

The categories consist of, respectively:

1. {bin, lay, place, set}
2. {blue, green, red, white}
3. {at, by, in, with}
4. {A, . . . , Z}
5. {zero, . . . , nine}
6. {again, now, please, soon}

This sentence structure with these options yield 64000 possible sentences. For example, two sentences in the data are “set blue by A four please” and “place red at C zero again”.[21]

Since this was the dataset we started off with,

B. The Oxford-BBC Lip Reading in the Wild (LRW) Dataset

The Lip Reading in the Wild (LRW) dataset consists of BBC newscasts. We obtained this data through Rob Cooper at BBC Research & Development.[22]

Each of these BBC newscasts contains at least a single word spoken within the span of 1.16 seconds. This parameter fit our needs pretty well but the problem dataset with this lay in the fact that the videos were not properly documented. The captioning for the videos only included at most one word when at times, multiple words could have been spoken. To avoid training a model that unnecessarily made incorrect associations with the wrong words, we decided to avoid this dataset as we couldn’t generate proper align files for it.

C. Lip Reading Sentences 3 (LRS3) Dataset

The Lip Reading Sentences 3 dataset contained TED/TEDx talks.[23] Compared to the LRW dataset, the captioning was much better: each of the words were properly documented and it came

[21] https://spandh.dcs.shef.ac.uk/gridcorpus/

[22] http://www.robots.ox.ac.uk/~vgg/data/lip_reading/lrw1.html

[23] http://www.robots.ox.ac.uk/~vgg/data/lip_reading/lrs3.html

with an accurate timestamp that we could easily convert to an align file. Unfortunately, the problem for LRS3 came in the form of videos that were too long. Each of the videos were easily over 5 seconds in length. Our solution for this was to split each of the words up and train each word individually and seeing if anything stuck. We implemented a method so that we can train on only a part of a sentence as a solution this problem.

When we began training, however, we realized we had another problem: being that LRS3 consisted of people giving a speech, they naturally moved their heads around when they spoke. This caused problems: sometimes the mouth couldn't be seen perfectly from the front and other times the mouth extraction script wouldn't even detect the mouth. Even if we were to try to attempt it on the frames that have been extracted, however, we weren't able to get it to converge well. Even the LRS3 paper detailed in Section II.4.C struggled with this: they couldn't obtain a BLEU score of higher than 65%. From this failed attempt, we decided to go back to GRID.

3. Data Processing

A. Image Normalization

Our image normalizing consists of two major parts: (1) face detection, (2) extracting frames. When preparing our datasets to be processed by the model, they need to be 100×50 and have up to 75 frames. To extract the mouth, `shape_predictor_68_face_landmarks.dat` was used to get the face shape. The videos were processed with the DLib face detector and the iBug face shape predictor. From there, we used ffmpeg to extract the video frames. See the Appendix for details on this code.

We briefly considered the idea of normalizing the images with each channel using standard deviations, but we abandoned that idea after determining that we weren't going to use multiple different datasets.[24]

B. Processing Alignments

To process the alignments, we obtained each of the individual frames and synced it with aligns. We also made a dictionary from compiling all the words from the align folder. See the Appendix for the code.

4. Training Schema

```
python scripts/extract_mouth_batch.py ~/GRID/s10/ *.mpg ~/LipNet/output_vids/
      ~/LipNet/common/predictors/shape_predictor_68_face_landmarks.dat
python make_dictionary.py ~/LipNet/training/random_split/var/s10/ dictionary.txt
./train
```

Fig. 14. Sample training commands; (1) extracts and normalizes training videos; (2) compiles a dictionary file using the selected speakers; (3) begin training

A. Overlapped Speaker

By training on each speaker separately, the overlapped speaker script developed by the LipNet maintainer has a learning curriculum validates against a whole sentence. For each batch of learning, it uses the same speaker. Essentially, it is the most basic way of learning with this dataset.

[24] https://becominghuman.ai/image-data-pre-processing-for-neural-networks-498289068258

LipNet developers have found that it takes 368 epochs to achieve a model with 96.93% BLEU score, 1.56% CER, and 3.38% WER. While our work on the random-split script involved many different aspects, which were taken into consideration, the random-split and overlapped speaker scripts do have a lot in common.

B. Random Split

The name "random split" comes from splitting the dataset randomly and then training in batches with multiple different speakers. This differs from the method in overlapped speaker where batches were trained with the same speaker each time. Derived from its name, the script splits the dataset into two groups: 80% for training and 20% for validation. We avoided the 60/20/20 split for our training because we trained on a smaller subset of the entire GRID Corpus; We have a huge amount of test data.

The main problem we encountered when adapting random split using the overlapped speakers learning curriculum was that overlapped speakers originally validated on the whole sentence from the beginning, at epoch 0, which we determined was the cause for a lack of convergence. We discovered that using this method lead to generating successive zero BLEU scores. Hence, the CERs and WERs are affected as well. We approached this problem by changing the learning curriculum so that it validated on increasingly lengthier sentences as detailed in the paper Lip Reading Sentences in the Wild as detailed in Section II.4.C. We did this since we have already modified the code for accepting variable length sentences as a result of attempting to work with the Lip Reading Sentences dataset itself as detailed in Section V.2.C. For epoch 1, we only validate on a single word. With each subsequent epoch, we validate on more words until we turn the learning curriculum back to its original training of whole sentences consisting of 6 words each.

```
def curriculum_rules(epoch):
    if epoch < 1:
        return { 'sentence_length': 1 }
    elif 1 <= epoch < 2:
        return { 'sentence_length': 2 }
    elif 2 <= epoch < 3:
        return { 'sentence_length': 2, 'flip_probability': 0.5 }
    elif 3 <= epoch < 4:
        return { 'sentence_length': 3, 'flip_probability': 0.5,
                 'jitter_probability': 0.05 }
    elif 4 <= epoch < 5:
        return { 'sentence_length': -1 }
    elif 5 <= epoch < 6:
        return { 'sentence_length': -1, 'flip_probability': 0.5 }
    return { 'sentence_length': -1, 'flip_probability': 0.5,
             'jitter_probability': 0.05 }
```

Fig. 15. Code that determines the sentence length for random split; `flip_probability` and `jitter_probability` are for generating randomness to the images; See the Appendix for more details behind the code.

In addition to that problem, the original number of epochs for overlapped speaker was 20 which will serve no purpose if the script is used on a large dataset of various speakers. We attempted to train by using different number of epochs with different dataset sizes and different datasets by

shuffling around various speakers and trying to elicit optimal hyperparameters that would lead to an increasing BLEU score, decreasing CER and WER, and decreasing loss rate. The difficult part about training random split script is the aspect to train on multiple speakers at the same time. This makes it hard to converge easily in the beginning like in the case of overlapped speaker script. Training and validating on multiple speakers at the same time makes the process of maintaining the optimal weights file much harder.

C. File Hierarchy

The following is the file hierarchy of the LipNet directory and how to execute python scripts for:

- random split
- overlapped speaker

For the overlapped speaker directory, the tree is described below:

```
/home/LipNet

-> LipNet
--> training
---> overlapped_speaker
----> prepare.py / train.py / s[i]     # s[i] are folders with identical hierarchy,
                                         train.py and prepare.py are python scripts.
-----> datasets                       # (inside of s[i] folder)
------> | align / train / val          # align (folder contains .align files for the
                                         speaker)
# train (folder contains another folder named 's[i]';
         i is respective index for the speaker.)
# val  (folder contains another folder named 's[i]';
         i is respective index for the speaker.)

-------> | s[i]     # (inside of train directory) and contains processed videos
       folders for the speaker (for training purposes); each video folder contains 75
       frames.
-------> | s[i]     # (inside of val directory) and processed videos folders for the
       speaker (for validation purposes); each video folder contains 75 frames. i is
       the respective number of the speaker dataset.
```

Fig. 16. Overlapped speaker file hierarchy

To train a specific speaker using `train.py` with overlapped speaker, you must do the following: Go to the directory that contains `train.py` and input the following `python train.py s[i]`. If you need to add more speakers for training, you need to add their folders at the `---->| prepare.py / train.py / s[i]` level with the appropriate index of the speaker. The tree branches down from the specified level must maintain an identical hierarchy to what was described above.

For random split, the tree is described below:

```
/home/LipNet:

-> | LipNet
--> | training
---> | random-split
----> | train.py / datasets      # datasets folder is a folder and train.py is a
```

```
                                    python script
-----> | align / video           # (inside of datasets folder) align folder contains
                                    all the .align files for videos inside the video
                                    folder
                                  # video folder contains s[i] folders; i is
                                    respective index for the speaker

------> | s[i]     # (inside of video folder) and contains processed videos folders
      for the speaker (for both training and validation purposes); each video folder
      inside s[i] folder contains 75 frames
```

Fig. 17. Random split file hierarchy

To train using random-split script `train.py`, you must do the following: Go to the directory that contains `train.py` and type the following `python train.py`. If you need to add more processed videos of a specific speaker, add speaker's align files to the "align" folder at the `-----> | align / video` level. Then, create `s[i]` folder inside video folder at the `------> | s[i]` level and add processed video folders inside the `s[i]` folder you just created. The tree branches down from the specified levels must maintain an identical hierarchy to what was described above.

The structure for each of the training video files processed using `extract_mouth_batch.py` script

- each video must be between 2 and 3 seconds
- frames extracted from each video should be 100×50
- each video must produce 75 frames of the above aforementioned dimensions

Each speaker in the GRID dataset has exactly 1000 videos with the above specifications.

Upon attempting to process the speakers respectively, the script, for each speaker, produces a folder that contains 1000 video folders. Each video folder contains 75 `.png` frames; each of which are 100×50 in dimensions. GRID dataset has around 33 speakers. For the purpose of maintaining our training and validation steps with respect to the amount of resources we have, we have decided to process only a subset of the total number of 33 speakers and proceeded accordingly.

VI. RESULTS

1. Training Progress

So far, we have four different training attempts (3 failed attempts and 1 successful attempt); each of which is explained below:

By changing the number of epochs to 2000, we attempted to train 6 different speakers with a total of 5150 videos for 249 epochs. However, we did not change the content of the dictionary, i.e., the dictionary included all sentences of the 33 speakers in GRID dataset. The following are the results for the BLEU score, CER, WER, and loss rate during the training and validation process:

A. *First Attempt (Failed)*

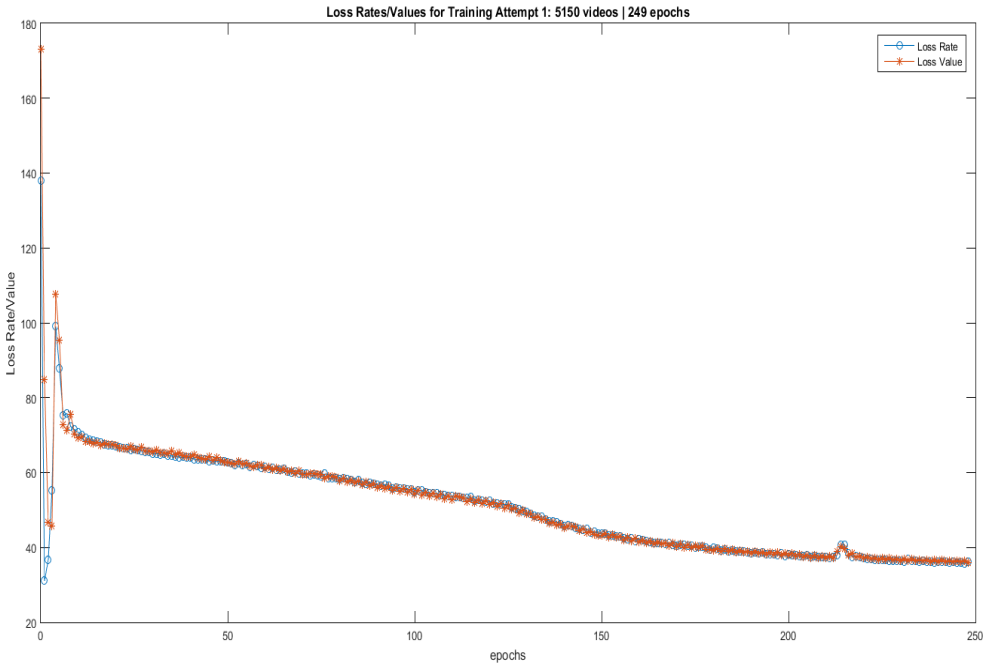


Fig. 18. Loss rate graph

As we can see from the graph the loss rate and loss values are both decreasing as the number of epochs increases.

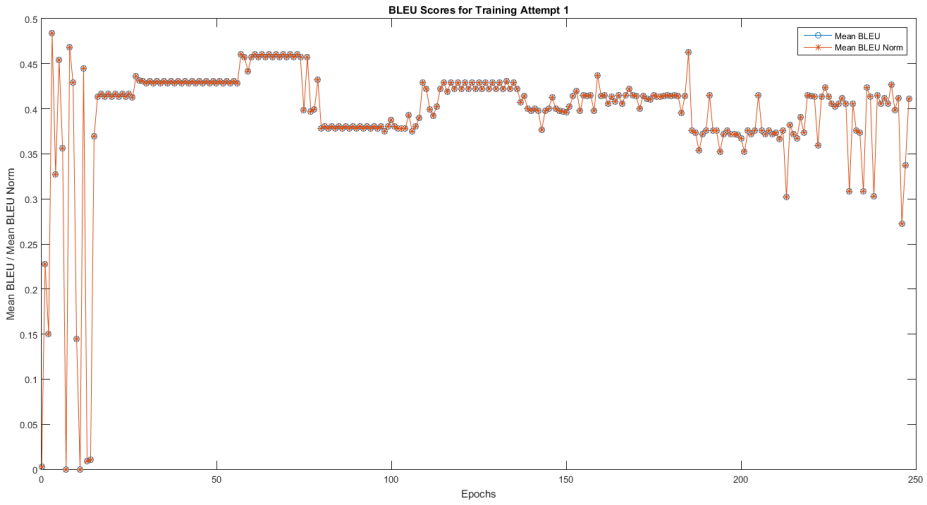


Fig. 19. BLEU score graph

While it is not clear from the picture the trend that the BLEU score tends to take throughout the training cycle. We expect the BLEU score to increase as the number of epochs significantly increase. Due to technical difficulties in deploying our code on Google Cloud (aka GCP) during

this training attempt, the random-split script in this attempt was run for only 249 epochs. If the script ran for a few hundred more epochs, we would be able to clearly see that the BLEU score is actually increasing.

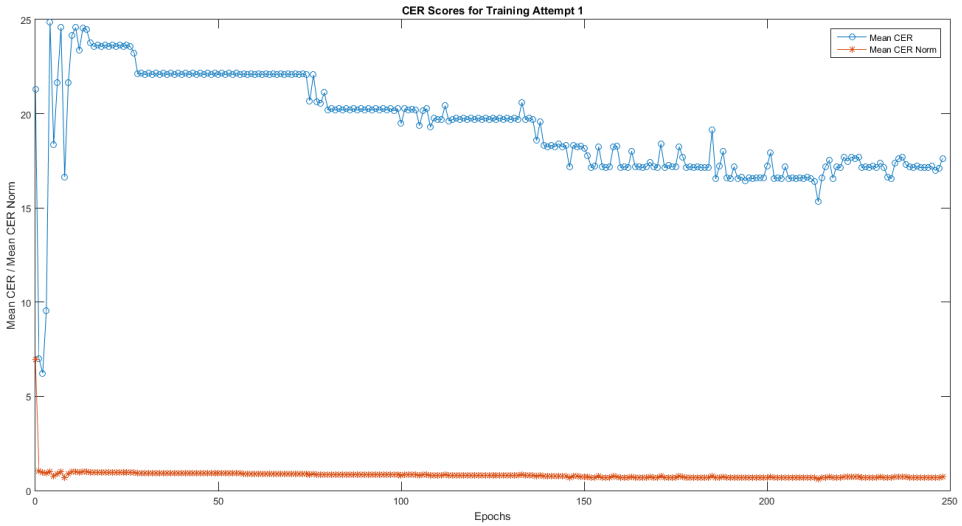


Fig. 20. CER graph

As we can clearly see from the above graph, the CER is dropping as more and more epochs are completed which is the desired pattern.

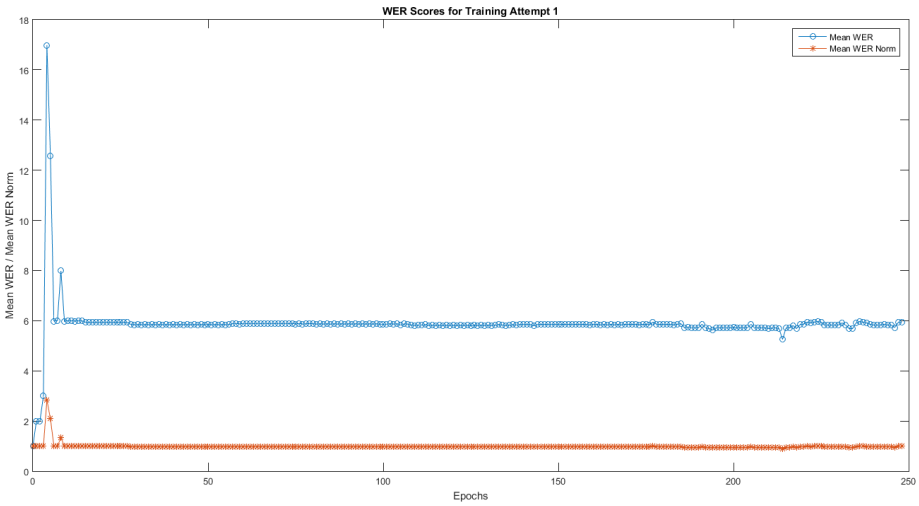


Fig. 21. WER graph

The trend that the WER takes during the training and validation cycle is not clear from the above graph. Again, we believe that if the random-split script was able to achieve a few hundred more epoch, we would have been able to see that the WER is actually dropping as more epochs are completed.

B. *Second Attempt (Failed)*

By changing the number of epochs to 2000, we attempted to train videos of 3 different speakers with a total of 2180 videos for 143 epochs. However, we did not change the content of the dictionary, i.e., the dictionary included all sentences of the 33 speakers in GRID dataset The following are the results for the BLEU score, CER, WER, and loss rate during the training and validation process:

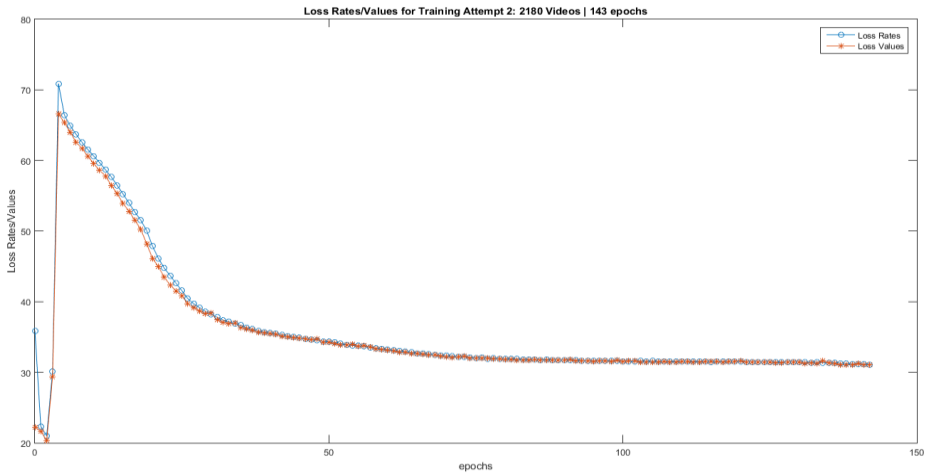


Fig. 22. Loss rate graph

As we can see from the graph the loss rate and loss values are both decreasing as the number of epochs increases.

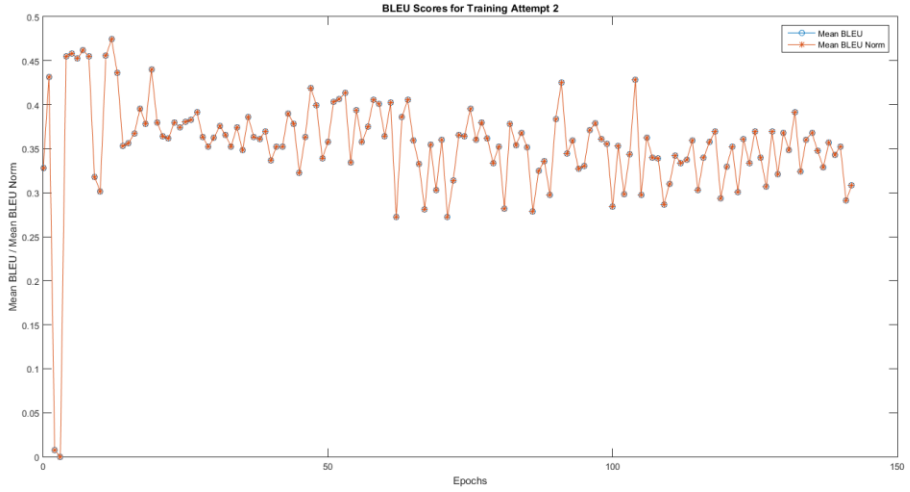


Fig. 23. BLEU score graph

While it is not clear from the picture the trend that the BLEU score tends to take throughout the training cycle. We expect the BLEU score to increase as the number of epochs significantly increase. Due to technical difficulties in deploying our code on Google Cloud (aka GCP) during this training attempt, the random-split script in this attempt was fun for only 143 epochs. If the

script ran for a few hundred more epochs, we would be able to clearly see that the BLEU score is actually increasing.

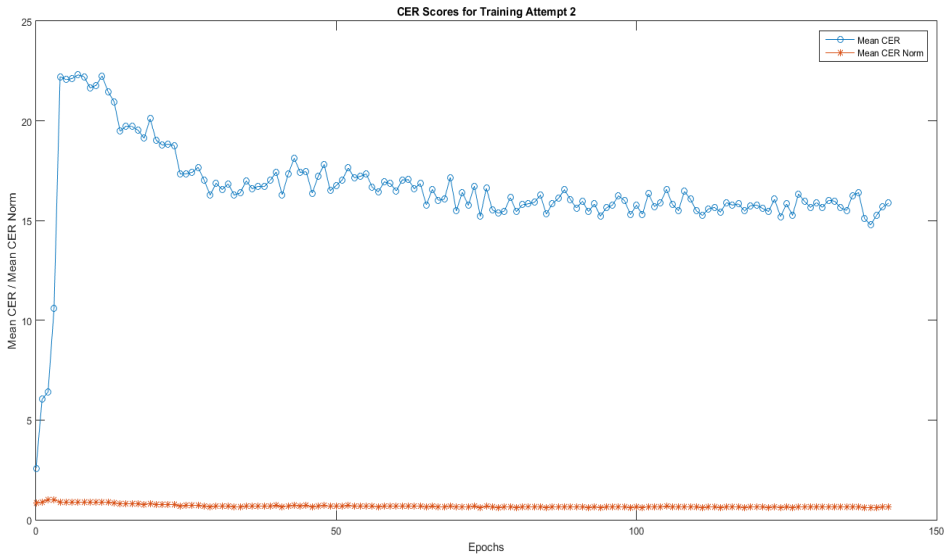


Fig. 24. CER graph

As we can clearly see from the above graph, the CER is dropping as more and more epochs are completed which is the desired pattern.

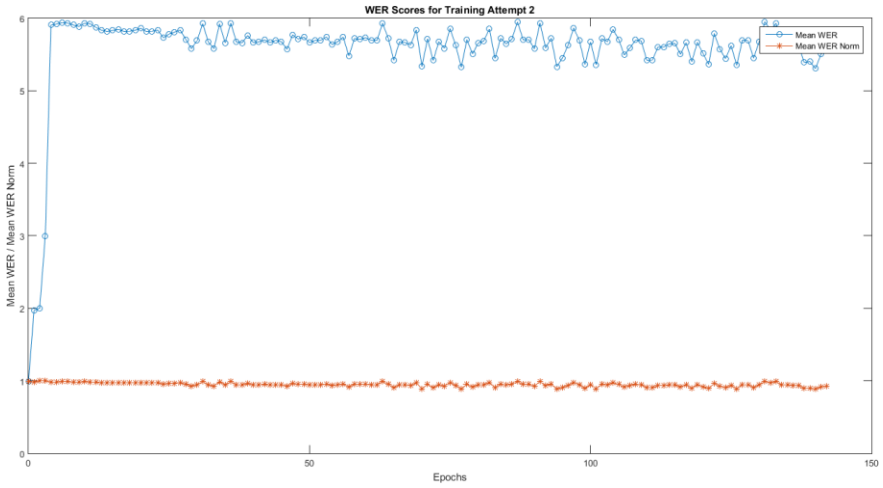


Fig. 25. WER graph

The trend that the WER takes during the training and validation cycle is not clear from the above graph. Again, we believe that if the random-split script was able to achieve a few hundred more epochs, we would have been able to see that the WER is actually dropping as more epochs are completed.

C. *Third Attempt (Failed)*

By changing the number of epochs to 2000, we attempted to train 6 different speakers with a total of 5150 videos for 78 epochs [the speakers used in this training attempt were different that those used in training attempt 1 mentioned above]. However we did not change the content of the dictionary, i.e., the dictionary included all sentences of the 33 speakers in GRID dataset The following are the results for the BLEU score, CER, WER, and loss rate during the training and validation process:

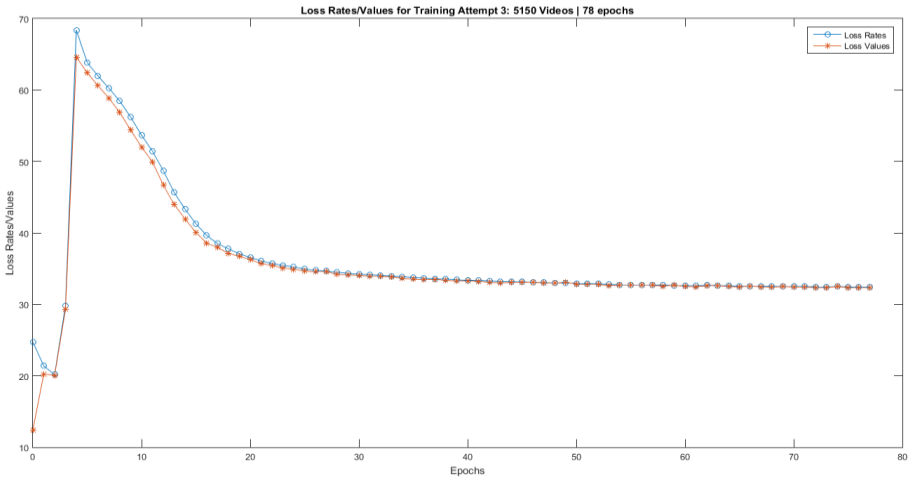


Fig. 26. Loss rate graph

As we can see from the graph the loss rate and loss values are both decreasing as the number of epochs increases.

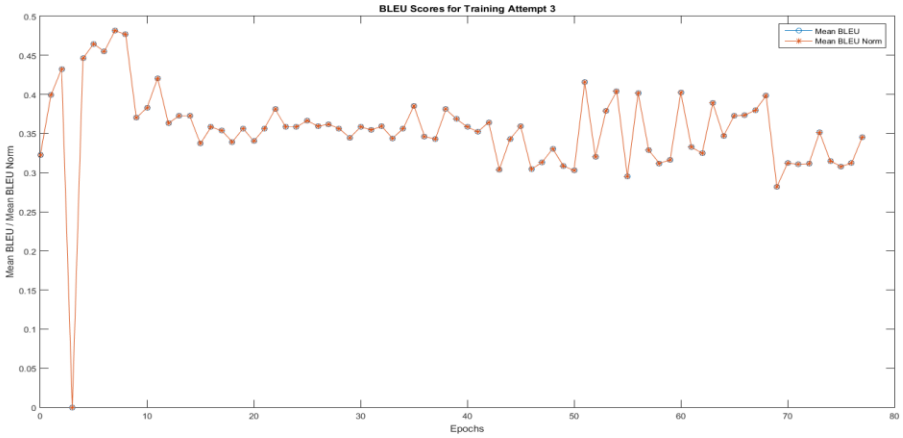


Fig. 27. BLEU score graph

While it is not clear from the picture the trend that the BLEU score tends to take throughout the training cycle. We expect the BLEU score to increase as the number of epochs significantly increase. Due to technical difficulties in deploying our code on Google Cloud (aka GCP) during this training attempt, the random-split script in this attempt was fun for only 78 epochs. If the script

ran for a few hundred more epochs, we would be able to clearly see that the BLEU score is actually increasing.

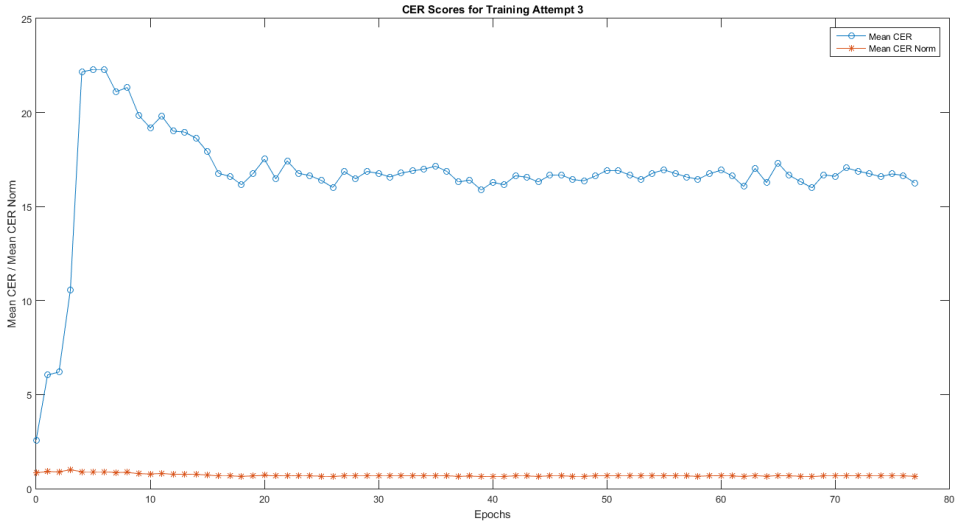


Fig. 28. CER graph

As we can clearly see from the above graph, the CER is dropping as more and more epochs are completed which is the desired pattern.

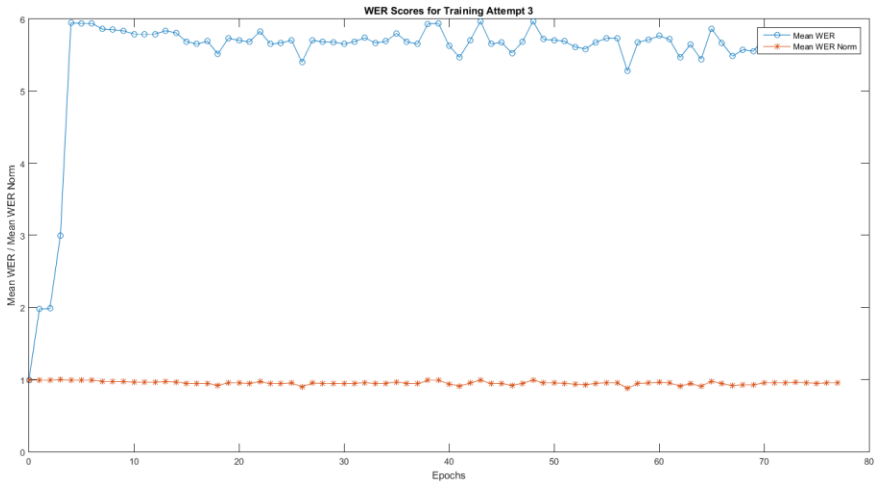


Fig. 29. WER graph

The trend that the WER takes during the training and validation cycle is not clear from the above graph. Again, we believe that if the random-split script was able to achieve a few hundred more epochs, we would have been able to see that the WER is actually dropping as more epochs are completed.

D. Fourth Attempt (Successful)

By changing the number of epochs to 2000, we attempted to train 3 different speakers with a total of 3000 videos for 805 epochs. This time, we did change the content of the dictionary, i.e., the dictionary included only the data for 3 speakers in GRID dataset. During this training attempt, the script was resumed multiple times by loading the weights of the previous epoch at which the script stopped. The following are the results for the BLEU score, CER, WER, and loss rate during the training and validation

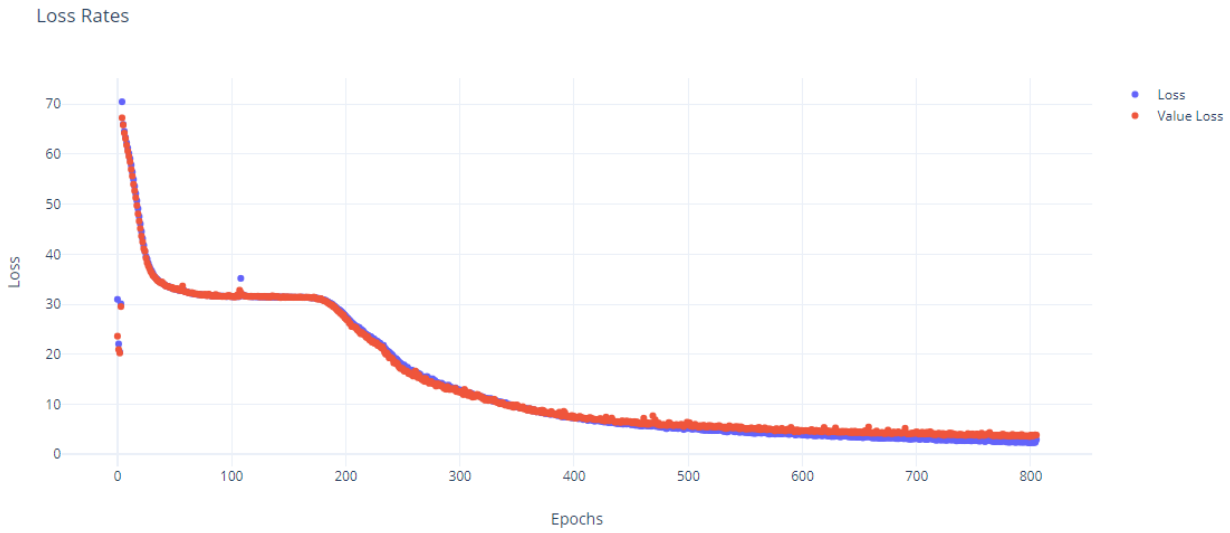


Fig. 30. Loss rate graph

As we can clearly see, the model converged with less than 2 percent error at epoch 805.

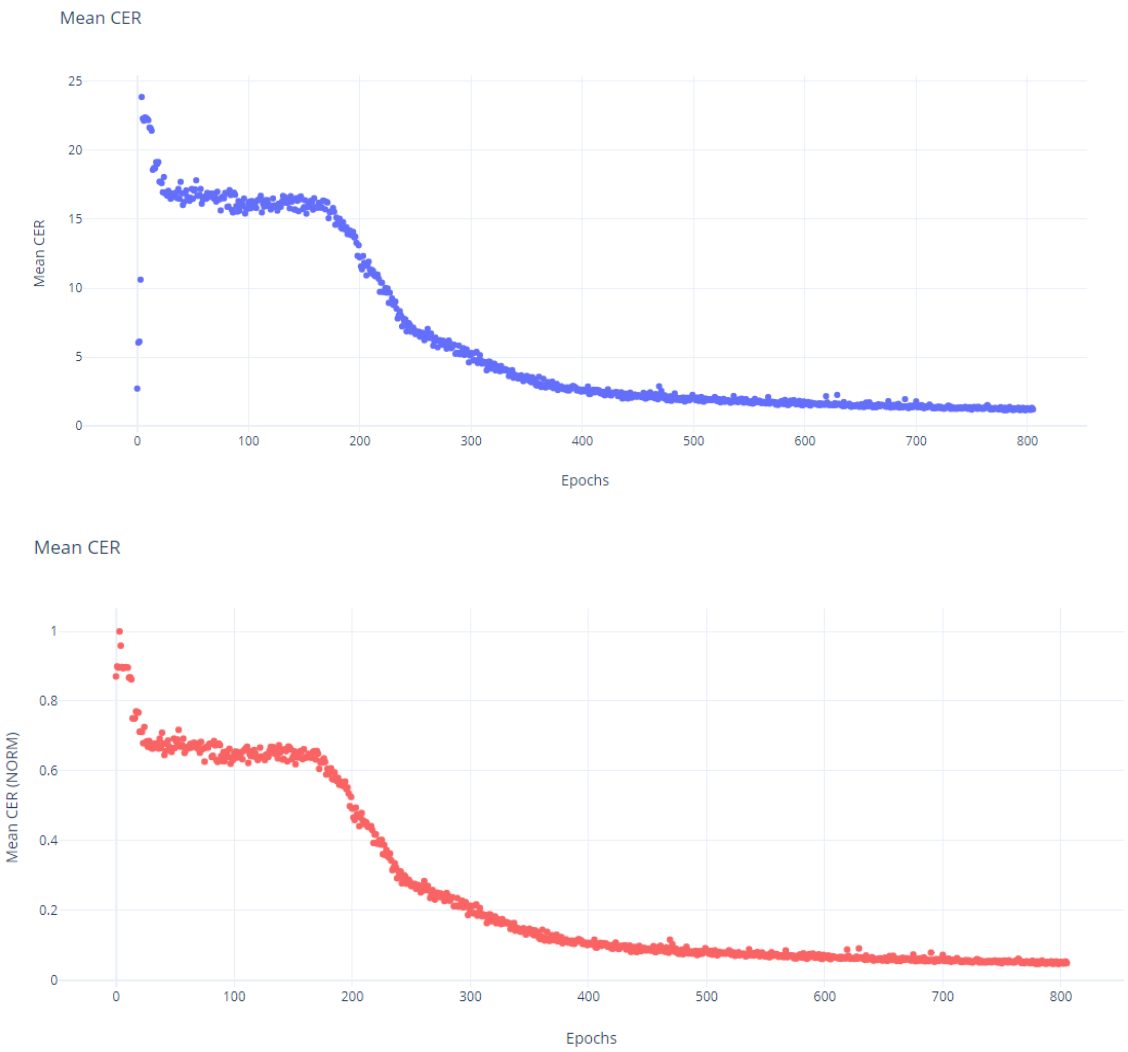


Fig. 31. CER and CER norm (CER norm is the normalization of the CER between 0 and 1)

As we can clearly see, the CER of this training attempt dropped to around 4 percent at epoch 805.

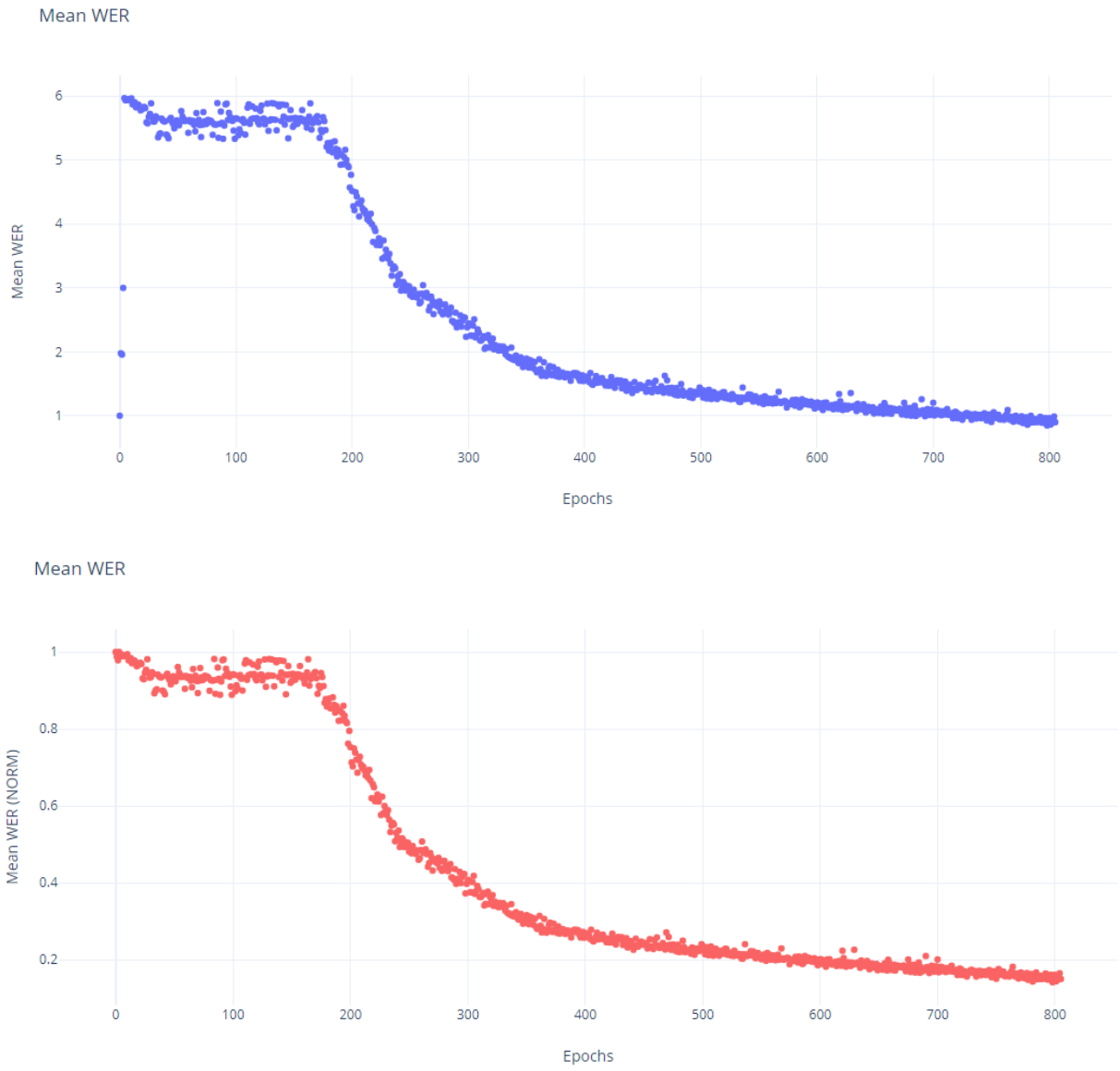


Fig. 32. WER and WER norm (WER norm is the normalization of the CER between 0 and 1)

As we can clearly see, the WER dropped to around 14 percent at epoch 805.

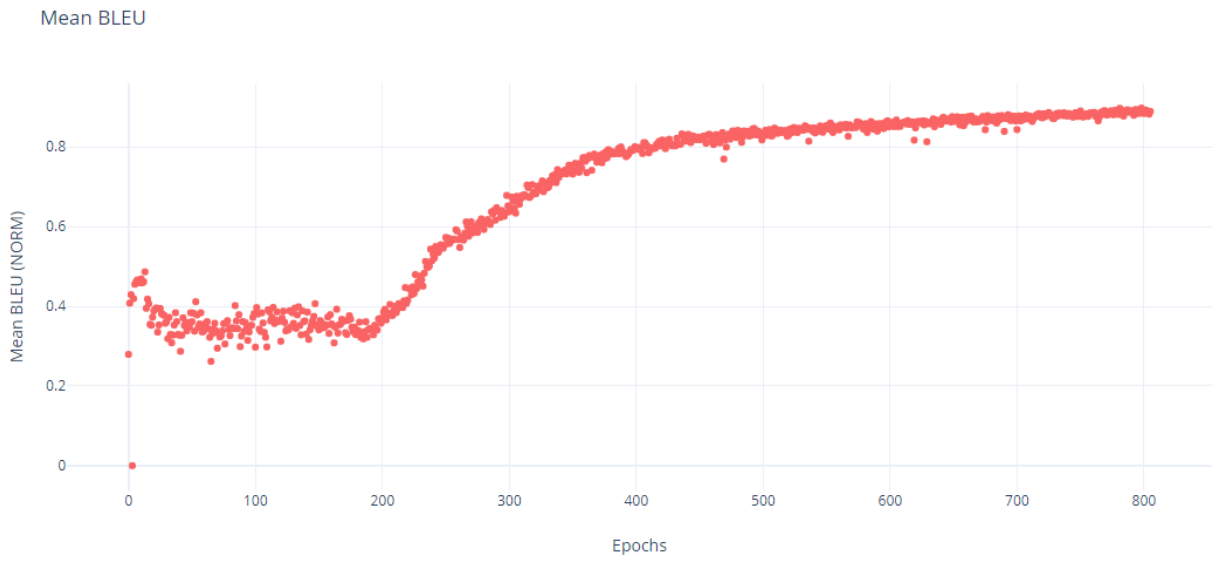


Fig. 33. Mean BLEU score

As we can clearly see from the above picture, the BLEU score has jumped up to more than 89 percent at epoch 805.

As a result of this successful training attempt, we were able to pick the weights file that generated the maximum BLEU score the minimum CER and WER. The weights file came to be of epoch 781. The BLEU score was about 89.15%. The WER came to be 15.43%. The CER came to be 4.608%. The previous scores were the best scores this training attempt has achieved.

2. Prediction

```
./predict ~/LipNet/training/random_split/results/resumrun2/weights781.h5
      ~/LipNet/test_vid/video.mpg
```

Fig. 34. Invoke prediction on `video.mpg` using `weights781.h5`

The weights781.h5 files we obtained during the fourth successful training attempt did give us some decent results. As mentioned in the training progress section that each speaker has 1000 videos; each produces 75 frames when properly processed. As long as the training proceeded on only three speakers, namely speaker 2 (s2), speaker 3 (s3), and speaker 4 (s4), we had around 30 more different speakers as our test data set [the reason we did not increase the training dataset size is because of time limits, i.e., the script would take a huge amount of time to finish a few hundred epochs]. The following are the results of predicting on 37 different videos from 9 different speaker. The first 3 speakers' predictions were of speaker 2, 3, and 4, i.e., those predictions were made using data that the NN has already seen (we are just showing this as a proof of concept that we maintained an 89.15% BLEU score, 15.43% WER, and 4.608% CER). The other 6 speakers' predictions were of speaker 7, 8, 10, 12, 13, 14, i.e., those predictions were made using data the NN has NOT seen before and those predictions, in fact, give us how well our generated weights do when seeing data different than that the model has trained on. Our results were compared with results that LipNet developers have attained.

- The two files that belong to LipNet developers are called “weights368.h5” and “weights 178.h5”, and the weights file that belong to our best training attempt is called “weights781.h5”
- “weights368.h5” was acquired by LipNet developers upon using the overlapped speaker script for training and validation.
- “weights178.h5” was acquired by LipNet developers upon using unseen speaker script for training and validation.

The following two figures show the results of predicting on test data using our weights file (weights781.h5), and the results of predicting on test data using the weights files that LipNet developers have attained (weights178.h5 and weights368.h5).

Note: The results belonging to LipNet developers were attained using different curriculum learning than our own training attempt. Furthermore, LipNet developers used different scripts to generate those weights files (namely, overlapped speaker, and unseen speaker scripts). Our weights file (weights781.h5) was generated as a result of maintaining the random split script that essentially trains on different speakers at the same time.

Table 3. Predictions Using `weights781.h5`

Speaker	*original sentence*	*predicted sentence (various speaker faces)*	*weights file*
s2	bin blue at l five now	bin blue at t five now	weights781.h5
s2	bin green at t seven now	bin green at c seven now	weights781.h5
s3	bin blue by f five soon	bin blue by 5 five soon	weights781.h5
s3	bin white in g zero please	bin white in r zero please	weights781.h5
s4	lay green in f zero again	lay green in f zero again	weights781.h5
s4	lay white by f two again	lay white by r two again	weights781.h5
s7	lay red in x six please	lay red by f five please	weights781.h5
s7	set blue by a two now	set blue by k three now	weights781.h5
s7	place blue at b seven again	place blue by t five now	weights781.h5
s7	bin blue at r six please	bin blue at v five please	weights781.h5
s7	place white at c six now	place white in j nine now	weights781.h5
s8	lay red in x six again	lay red at z six again	weights781.h5
s8	set blue with a six soon	set blue with s e soon	weights781.h5
s8	place blue in u four again	place blue at r four now	weights781.h5
s8	bin blue at e five now	bin blue at v five now	weights781.h5
s8	place white at i nine now	place white at c set now	weights781.h5
s12	lay red in p five now	lay red bin f five now	weights781.h5
s12	set blue in f six again	set blue with s eight again	weights781.h5
s12	place blue in n five please	place blue with f five please	weights781.h5
s12	bin blue at k seven please	bin blue with c seven please	weights781.h5
s12	place white at v four soon	place white at s four soon	weights781.h5
s13	lay red with y one soon	lay r with m four soon	weights781.h5
s13	set blue with g five soon	bin blue with m e soon	weights781.h5
s13	place blue with i three again	bin blue with m four now	weights781.h5

s13	bin blue at e one soon	bin blue with u four soon	weights781.h5
s13	place white at i four now	bin white with v four now	weights781.h5
s14	lay red at c nine now	lay green at h six now	weights781.h5
s14	set blue with g four soon	set green with b five soon	weights781.h5
s14	place blue in n four again	place blue at g four again	weights781.h5
s14	bin blue at k four soon	bin blue at j four soon	weights781.h5
s14	place white in i two again	place white at h two again	weights781.h5
s10	lay red at h four please	lay red in f five please	weights781.h5
s10	set blue in q zero please	lay blue at g zero please	weights781.h5
s10	place blue in r seven again	place blue by m eight again	weights781.h5
s10	bin blue at i three soon	bin blue at y six please	weights781.h5
s10	place white at g two now	place white in r two now	weights781.h5

video (dir)	actual sentence	predicted sentece	weights file used (dir)
test_vid/test_dataset/s2	lay blue in x five please	lay blue in x five please	weights368.h5
test_vid/test_dataset/s7	lay red in d two now	lay red in d two now	weights368.h5
test_vid/test_dataset/s10	lay green at v three soon	lay green at v three soon	weights368.h5
test_vid/test_dataset/s12	lay green at y eight again	lay green at y eight again	weights368.h5
test_vid/test_dataset/s13	lay white at k one soon	lay white at k one soon	weights368.h5
test_vid/test_dataset/s14	lay red by j eight soon	lay red by j eight soon	weights368.h5
test_vid/test_dataset/s2	lay blue in x five please	lay blue in x five please	weights178.h5
test_vid/test_dataset/s7	lay red in d two now	lay red in d two now	weights178.h5
test_vid/test_dataset/s10	lay green at v three soon	lay green at v three soon	weights178.h5
test_vid/test_dataset/s12	lay green at y eight again	lay green at y eight again	weights178.h5
test_vid/test_dataset/s13	lay white at k one soon	lay white at k one soon	weights178.h5
test_vid/test_dataset/s14	lay red by j eight soon	lay red by j eight soon	weights178.h5

Fig. 35. Predicted using `weights368.h5` and `weights178.h5`

As we can see from the above figures, the weights files that belong to authors of LipNet are slightly better than that produced by our best training attempt to maintain random-split script. However, we believe that if the training and validation dataset size is increased (as in trying to train and validate on more than 3 speakers), we can further decrease the WER and CER, and also significantly increase the BLEU score.

VII. Discussion

1. Comparison

The training curriculum implemented by the LipNet maintainer trains on each speaker separately and uses a whole sentence to validate. They have found that it takes 368 epochs while training on 33 speakers to achieve a 96.93% BLEU score, 1.56% character-error rate, and 3.38% word-error rate.

The training curriculum that we have implemented trains on multiple speakers at the same time and trains and validates on one word in the beginning and then adds words on with each subsequent epoch. We trained using 3 speakers and The BLEU score was about 89.15%, the WER came to be 15.43%. the CER came to be 4.608% in 781 epochs. This is the approximate BLEU score of the model that LipNet was trying to succeed.

Note that the amount of time we took to train was much shorter than the amount of time it took the authors of LipNet to train although we have double the number of epochs. Since we are training on 3 speakers only, the total amount of time spent training on the successful attempt was approximately 2 days as opposed to their 7-10 days.

2. Timeline

Week	*Description*
1/7/19	basic research
1/14/19	basic research
1/21/19	basic research
1/28/19	decided to meet regularly on Mondays
2/4/19	first presentation
2/11/19	started using LipNet
2/18/19	EC2/access to LRS
2/25/19	EC2/toy example of machine learning
3/4/19	EC2/SageMaker/got access to LRW
3/11/19	SageMaker
3/18/19	SageMaker
3/25/19	SageMaker/Sam tries to train on his laptop
4/1/19	GCE/first successful training
4/8/19	modifying hyperparameters
4/15/19	fine tuning model
4/22/19	fine tuning model
4/29/19	finishing up draft paper
5/6/19	finishing up final presentation/presenting
5/13/19	writing final paper

Fig. 36. Brief timeline of events; the week is the Monday of each week; description details the main thing all 3 of us worked on

VIII. CONCLUSION

Over the course of training on Google Cloud Platform, we've spent approximately 400 dollars in free trial credits. Part of this is having those multiple failed attempts.

One thing we could have done better was having better direction. We spent the first half of the semester wondering what to do and spent too much time on the 3D AVR paper as well as on trying to get AWS to work without considering other options. A bit of this is attributed to a lack of thorough research. Rather than coming up with things as we went, it would've been better if we did all the research that we needed to at the beginning without having to backtrack.

If this lead us to more time, we could have implemented hyperparameter tuning to get even better results.

Overall, we learned the most on how to approach choosing datasets, working with Keras, and deploying to the cloud. Of these, we have shown similar results to the original implementation with our method that changes up:

- the size of the dataset
- the duration of training
- curriculum learning

We implemented tools to better adjust the processing of input video files in the form of:

- image normalization
- processing alignment and dictionary files

We also demonstrated changing up convergence through the use of:

- training on variable sentence length
- training on random batches of speakers at once

IX. Appendix

Because copying over the entire program may be out of scope for this appendix, the sections below will be code we have written or modified taken out of the context of the rest of LipNet. This is due to the fact that we wrote our program directly on top of the already existing model. Please let us know if you have any questions about anything undocumented.

```
from lipnet.lipreading.videos import Video
import os, fnmatch, sys, errno
from skimage import io

SOURCE_PATH = sys.argv[1]
SOURCE_EXTS = sys.argv[2]
TARGET_PATH = sys.argv[3]

FACE_PREDICTOR_PATH = sys.argv[4]

def find_files(directory, pattern):
    for root, dirs, files in os.walk(directory):
        for basename in files:
            if fnmatch.fnmatch(basename, pattern):
                filename = os.path.join(root, basename)
                yield filename

for filepath in find_files(SOURCE_PATH, SOURCE_EXTS):
    print "Processing: {}".format(filepath)
    video = Video(vtype='face',
face_predictor_path=FACE_PREDICTOR_PATH).from_video(filepath)
```

```
    filepath_wo_ext = os.path.splitext(filepath)[0]
    target_dir = os.path.join(TARGET_PATH, filepath_wo_ext)
    os.makedirs(path)

    i = 0
    for frame in video.mouth:
        io.imsave(os.path.join(target_dir, "mouth_{0:03d}.png".format(i)), frame)
        i += 1
```

Fig. 1. `extract_mouth_batch.py`; d

```
def get_frames_mouth(self, detector, predictor, frames, **kwargs):
    normalize_ratio = None
    mouth_frames = []
    for frame in frames:
        dets = detector(frame, 1)
        shape = None
        for k, d in enumerate(dets):
            shape = predictor(frame, d)
            i = -1

        # Extract mouth
        mouth_points = []
        for part in shape.parts():
            i += 1
            if i < 48:  # Only take mouth region
                continue
            mouth_points.append((part.x,part.y))
        np_mouth_points = np.array(mouth_points)

        mouth_centroid = np.mean(np_mouth_points[:, -2:], axis=0)

        # Normalize mouth
        if normalize_ratio is None:
            mouth_left = np.min(np_mouth_points[:, :-1]) * (1.0 - H_PAD)
            mouth_right = np.max(np_mouth_points[:, :-1]) * (1.0 + H_PAD)

            normalize_ratio = MOUTH_WIDTH / float(mouth_right - mouth_left)

        new_img_shape = (int(f.shape[0]*n_ratio), int(f.shape[1] * n_ratio))
        resized_img = imresize(frame, new_img_shape)
        mouth_centroid_norm = mouth_centroid * normalize_ratio

        mouth_l = int(mouth_centroid_norm[0] - MOUTH_WIDTH / 2)
```

```
        mouth_r = int(mouth_centroid_norm[0] + MOUTH_WIDTH / 2)
        mouth_t = int(mouth_centroid_norm[1] - MOUTH_HEIGHT / 2)
        mouth_b = int(mouth_centroid_norm[1] + MOUTH_HEIGHT / 2)

        mouth_crop_image = resized_img[mouth_t:mouth_b, mouth_l:mouth_r]

        mouth_frames.append(mouth_crop_image)
    return mouth_frames
```

Fig. 2. `videos.py`

```
from keras.optimizers import Adam
from keras.callbacks import TensorBoard, CSVLogger, ModelCheckpoint
from lipnet.lipreading.generators import RandomSplitGenerator
from lipnet.lipreading.callbacks import Statistics, Visualize
from lipnet.lipreading.curriculums import Curriculum
from lipnet.core.decoders import Decoder
from lipnet.lipreading.helpers import labels_to_text
from lipnet.utils.spell import Spell
from lipnet.model2 import LipNet
import numpy as np
import datetime
import os
import sys

np.random.seed(55)

CURRENT_PATH = os.path.dirname(os.path.abspath(__file__))
DATASET_DIR  = os.path.join(CURRENT_PATH, 'datasets')
OUTPUT_DIR   = os.path.join(CURRENT_PATH, 'results')
LOG_DIR      = os.path.join(CURRENT_PATH, 'logs')

PREDICT_GREEDY      = False
PREDICT_BEAM_WIDTH  = 200
PREDICT_DICTIONARY  =
os.path.join(CURRENT_PATH,'..','..','common','dictionaries','grid.txt')

def curriculum_rules(epoch):
    if epoch < 1:
        return { 'sentence_length': 1 }
    elif 1 <= epoch < 2:
        return { 'sentence_length': 2 }
    elif 2 <= epoch < 3:
        return { 'sentence_length': 2, 'flip_probability': 0.5 }
    elif 3 <= epoch < 4:
```

```
        return { 'sentence_length': 3, 'flip_probability': 0.5,
'jitter_probability': 0.05 }
    elif 4 <= epoch < 5:
        return { 'sentence_length': -1 }
    elif 5 <= epoch < 6:
        return { 'sentence_length': -1, 'flip_probability': 0.5 }
    return { 'sentence_length': -1, 'flip_probability': 0.5,
'jitter_probability': 0.05 }

def train(run_name, start_epoch, stop_epoch, img_c, img_w, img_h, frames_n,
absolute_max_string_len, minibatch_size):
    curriculum = Curriculum(curriculum_rules)
    lip_gen = RandomSplitGenerator(dataset_path=DATASET_DIR,
                                   minibatch_size=minibatch_size,
                                   img_c=img_c, img_w=img_w, img_h=img_h,
frames_n=frames_n,
                                   absolute_max_string_len=absolute_max_string_len,
                                   curriculum=curriculum,
start_epoch=start_epoch).build(val_split=0.2)

    lipnet = LipNet(img_c=img_c, img_w=img_w, img_h=img_h, frames_n=frames_n,
                               absolute_max_string_len=absolute_max_string_len,
output_size=lip_gen.get_output_size())
    lipnet.summary()

    adam = Adam(lr=0.0001, beta_1=0.9, beta_2=0.999, epsilon=1e-08)

    # the loss calc occurs elsewhere, so use a dummy lambda func for the loss
    lipnet.model.compile(loss={'ctc': lambda y_true, y_pred: y_pred},
optimizer=adam)

    # load weight if necessary
    if start_epoch > 0:
        weight_file = os.path.join(OUTPUT_DIR, os.path.join(run_name,
'weights%03d.h5' % (start_epoch - 1)))
        lipnet.model.load_weights(weight_file)

    spell = Spell(path=PREDICT_DICTIONARY)
    decoder = Decoder(greedy=PREDICT_GREEDY, beam_width=PREDICT_BEAM_WIDTH,
                      postprocessors=[labels_to_text, spell.sentence])

    # define callbacks
    statistics  = Statistics(lipnet, lip_gen.next_val(), decoder, 256,
output_dir=os.path.join(OUTPUT_DIR, run_name))
```

```
    visualize   = Visualize(os.path.join(OUTPUT_DIR, run_name), lipnet,
lip_gen.next_val(), decoder, num_display_sentences=minibatch_size)
    tensorboard = TensorBoard(log_dir=os.path.join(LOG_DIR, run_name))
    csv_logger  = CSVLogger(os.path.join(LOG_DIR, "{}-
{}.csv".format('training',run_name)), separator=',', append=True)
    checkpoint  = ModelCheckpoint(os.path.join(OUTPUT_DIR, run_name,
"weights{epoch:02d}.h5"), monitor='val_loss', save_weights_only=True,
mode='auto', period=1)

    lipnet.model.fit_generator(generator=lip_gen.next_train(),
                           steps_per_epoch=lip_gen.default_training_steps,
epochs=stop_epoch,
                           validation_data=lip_gen.next_val(),
validation_steps=lip_gen.default_validation_steps,
                           callbacks=[checkpoint, statistics, visualize, lip_gen,
tensorboard, csv_logger],
                           initial_epoch=start_epoch,
                           verbose=1,
                           max_q_size=5,
                           workers=2,
                           pickle_safe=True)

if __name__ == '__main__':
    run_name = datetime.datetime.now().strftime('%Y:%m:%d:%H:%M:%S')
    train(run_name, 0, 2000, 3, 100, 50, 75, 32, 50)
```

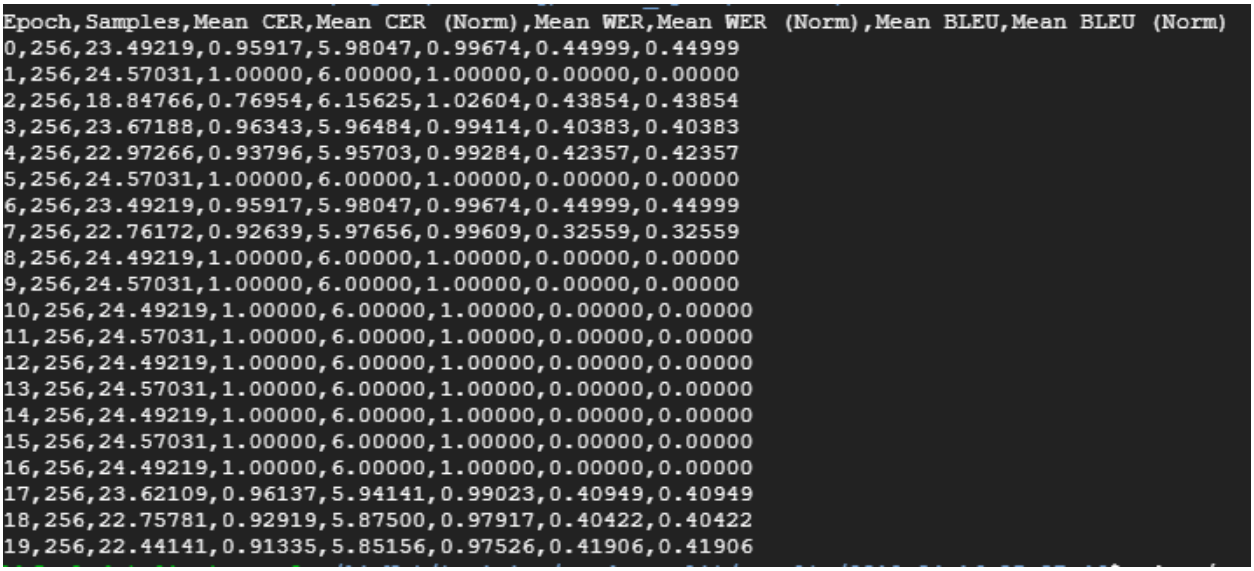
Fig. 3. random_split.py

In `random_split.py` shown in figure 3, the method to train on words of increasing length per epoch is found in the following lines:

```
def curriculum_rules(epoch):
    if epoch < 1:
        return { 'sentence_length': 1 }
    elif 1 <= epoch < 2:
        return { 'sentence_length': 2 }
    elif 2 <= epoch < 3:
        return { 'sentence_length': 2, 'flip_probability': 0.5 }
    elif 3 <= epoch < 4:
        return { 'sentence_length': 3, 'flip_probability': 0.5,
                 'jitter_probability': 0.05 }
    elif 4 <= epoch < 5:
        return { 'sentence_length': -1 }
    elif 5 <= epoch < 6:
```

```
        return { 'sentence_length': -1, 'flip_probability': 0.5 }
    return { 'sentence_length': -1, 'flip_probability': 0.5,
             'jitter_probability': 0.05 }
```

The above method validates on a single word in the first epoch, on two words on the second epoch, three words on the third epoch, and on with the full lenth of the sentence starting from the fourth epoch. This method helped us to speed up the convergence of the model. Without the previous method, the model validates on full sentences starting from the first epoch which leads to nearly 100% WER and CER and BLEU score of 0 as depicted in the following figure:



```
Epoch,Samples,Mean CER,Mean CER (Norm),Mean WER,Mean WER (Norm),Mean BLEU,Mean BLEU (Norm)
0,256,23.49219,0.95917,5.98047,0.99674,0.44999,0.44999
1,256,24.57031,1.00000,6.00000,1.00000,0.00000,0.00000
2,256,18.84766,0.76954,6.15625,1.02604,0.43854,0.43854
3,256,23.67188,0.96343,5.96484,0.99414,0.40383,0.40383
4,256,22.97266,0.93796,5.95703,0.99284,0.42357,0.42357
5,256,24.57031,1.00000,6.00000,1.00000,0.00000,0.00000
6,256,23.49219,0.95917,5.98047,0.99674,0.44999,0.44999
7,256,22.76172,0.92639,5.97656,0.99609,0.32559,0.32559
8,256,24.49219,1.00000,6.00000,1.00000,0.00000,0.00000
9,256,24.57031,1.00000,6.00000,1.00000,0.00000,0.00000
10,256,24.49219,1.00000,6.00000,1.00000,0.00000,0.00000
11,256,24.57031,1.00000,6.00000,1.00000,0.00000,0.00000
12,256,24.49219,1.00000,6.00000,1.00000,0.00000,0.00000
13,256,24.57031,1.00000,6.00000,1.00000,0.00000,0.00000
14,256,24.49219,1.00000,6.00000,1.00000,0.00000,0.00000
15,256,24.57031,1.00000,6.00000,1.00000,0.00000,0.00000
16,256,24.49219,1.00000,6.00000,1.00000,0.00000,0.00000
17,256,23.62109,0.96137,5.94141,0.99023,0.40949,0.40949
18,256,22.75781,0.92919,5.87500,0.97917,0.40422,0.40422
19,256,22.44141,0.91335,5.85156,0.97526,0.41906,0.41906
```

Fig. 4. Statistics of training and validation on full sentence starting from the first epoch which leads to 100% WER and CER, and BLEU scores of 0 through consecutive epochs

```
from lipnet.lipreading.helpers import text_to_labels
from lipnet.lipreading.videos import Video
from lipnet.lipreading.aligns import Align
from lipnet.helpers.threadsafe import threadsafe_generator
from lipnet.helpers.list import get_list_safe
from keras import backend as K
import numpy as np
import keras
import pickle
import os
import glob
import multiprocessing


# datasets/[train|val]/<sid>/<id>/<image>.png
# or datasets/[train|val]/<sid>/<id>.mpg
# datasets/align/<id>.align
class BasicGenerator(keras.callbacks.Callback):
```

```
    def __init__(self, dataset_path, minibatch_size, img_c, img_w, img_h,
frames_n, absolute_max_string_len=30, **kwargs):
        self.dataset_path   = dataset_path
        self.minibatch_size = minibatch_size
        self.blank_label    = self.get_output_size() - 1
        self.img_c          = img_c
        self.img_w          = img_w
        self.img_h          = img_h
        self.frames_n       = frames_n
        self.absolute_max_string_len = absolute_max_string_len
        self.cur_train_index = multiprocessing.Value('i', 0)
        self.cur_val_index   = multiprocessing.Value('i', 0)
        self.curriculum      = kwargs.get('curriculum', None)
        self.random_seed     = kwargs.get('random_seed', 13)
        self.vtype               = kwargs.get('vtype', 'mouth')
        self.face_predictor_path = kwargs.get('face_predictor_path', None)
        self.steps_per_epoch     = kwargs.get('steps_per_epoch', None)
        self.validation_steps    = kwargs.get('validation_steps', None)
        # Process epoch is used by non-training generator (e.g: validation)
        # because each epoch uses different validation data enqueuer
        # Will be updated on epoch_begin
        self.process_epoch       = -1
        # Maintain separate process train epoch because fit_generator only use
        # one enqueuer for the entire training, training enqueuer can contain
        # max_q_size batch data ahead than the current batch data which might be
        # in the epoch different with current actual epoch
        # Will be updated on next_train()
        self.shared_train_epoch  = multiprocessing.Value('i', -1)
        self.process_train_epoch = -1
        self.process_train_index = -1
        self.process_val_index   = -1

    def build(self, **kwargs):
        self.train_path     = os.path.join(self.dataset_path, 'train')
        self.val_path       = os.path.join(self.dataset_path, 'val')
        self.align_path     = os.path.join(self.dataset_path, 'align')
        self.build_dataset()
        # Set steps to dataset size if not set
        self.steps_per_epoch  = self.default_training_steps if
self.steps_per_epoch is None else self.steps_per_epoch
        self.validation_steps = self.default_validation_steps if
self.validation_steps is None else self.validation_steps
        return self
@property
    def training_size(self):
```

```
        return len(self.train_list)

    @property
    def default_training_steps(self):
        return self.training_size / self.minibatch_size

    @property
    def validation_size(self):
        return len(self.val_list)

    @property
    def default_validation_steps(self):
        return self.validation_size / self.minibatch_size

    def get_output_size(self):
        return 28

    def get_cache_path(self):
        return self.dataset_path.rstrip('/') + '.cache'

    def enumerate_videos(self, path):
        video_list = []
        for video_path in glob.glob(path):
            try:
                if os.path.isfile(video_path):
                    video = Video(self.vtype,
self.face_predictor_path).from_video(video_path)
                else:
        video = Video(self.vtype,
self.face_predictor_path).from_frames(video_path)
            except AttributeError as err:
                raise err
            except:
                print "Error loading video: "+video_path
                continue
            if K.image_data_format() == 'channels_first' and video.data.shape !=
(self.img_c,self.frames_n,self.img_w,self.img_h):
                print "Video "+video_path+" has incorrect shape
"+str(video.data.shape)+", must be
"+str((self.img_c,self.frames_n,self.img_w,self.img_h))+""
                continue
            if K.image_data_format() != 'channels_first' and video.data.shape !=
(self.frames_n,self.img_w,self.img_h,self.img_c):
```

```
                print "Video "+video_path+" has incorrect shape
"+str(video.data.shape)+", must be
"+str((self.frames_n,self.img_w,self.img_h,self.img_c))+""
                continue
            video_list.append(video_path)
        return video_list

    def enumerate_align_hash(self, video_list):
        align_hash = {}
        for video_path in video_list:
            video_id = os.path.splitext(video_path)[0].split('/')[-1]
            align_path = os.path.join(self.align_path, video_id)+".align"
            align_hash[video_id] = Align(self.absolute_max_string_len,
text_to_labels).from_file(align_path)
        return align_hash

    def build_dataset(self):
        if os.path.isfile(self.get_cache_path()):
            print "\nLoading dataset list from cache..."
            with open (self.get_cache_path(), 'rb') as fp:
                self.train_list, self.val_list, self.align_hash = pickle.load(fp)
        else:
            print "\nEnumerating dataset list from disk..."
    self.train_list = self.enumerate_videos(os.path.join(self.train_path, '*',
'*'))
            self.val_list   = self.enumerate_videos(os.path.join(self.val_path,
'*', '*'))
            self.align_hash = self.enumerate_align_hash(self.train_list +
self.val_list)
            with open(self.get_cache_path(), 'wb') as fp:
                pickle.dump((self.train_list, self.val_list, self.align_hash),
fp)

        print "Found {} videos for training.".format(self.training_size)
        print "Found {} videos for validation.".format(self.validation_size)
        print ""

        np.random.shuffle(self.train_list)

    def get_align(self, _id):
        return self.align_hash[_id]

    def get_batch(self, index, size, train):
        if train:
            video_list = self.train_list
```

```
        else:
            video_list = self.val_list

        X_data_path = get_list_safe(video_list, index, size)
        X_data = []
        Y_data = []
        label_length = []
        input_length = []
        source_str = []
        for path in X_data_path:
            video = Video().from_frames(path)
            align = self.get_align(path.split('/')[-1])
    video_unpadded_length = video.length
            if self.curriculum is not None:
                video, align, video_unpadded_length =
self.curriculum.apply(video, align)
            X_data.append(video.data)
            Y_data.append(align.padded_label)
            label_length.append(align.label_length) # CHANGED [A] -> A, CHECK!
            # input_length.append([video_unpadded_length - 2]) # 2 first frame
discarded
            input_length.append(video.length) # Just use the video padded length
to avoid CTC No path found error (v_len < a_len)
            source_str.append(align.sentence) # CHANGED [A] -> A, CHECK!

        source_str = np.array(source_str)
        label_length = np.array(label_length)
        input_length = np.array(input_length)
        Y_data = np.array(Y_data)
        X_data = np.array(X_data).astype(np.float32) / 255  #Normalize image data
to [0,1],TODO: mean normalization over training data

        inputs = {'the_input': X_data,
                  'the_labels': Y_data,
                  'input_length': input_length,
                  'label_length': label_length,
                  'source_str': source_str  # used for visualization only
                  }
        outputs = {'ctc': np.zeros([size])}  # dummy data for dummy loss function

        return (inputs, outputs)

    @threadsafe_generator
    def next_train(self):
        r = np.random.RandomState(self.random_seed)
```

```
        while 1:
    print "SI: {}, SE: {}".format(self.cur_train_index.value,
self.shared_train_epoch.value)
            with self.cur_train_index.get_lock(),
self.shared_train_epoch.get_lock():
                cur_train_index = self.cur_train_index.value
                self.cur_train_index.value += self.minibatch_size
                # Shared epoch increment on start or index >= training in epoch
                if cur_train_index >= self.steps_per_epoch * self.minibatch_size:
                    cur_train_index = 0
                    self.shared_train_epoch.value += 1
                    self.cur_train_index.value = self.minibatch_size
                if self.shared_train_epoch.value < 0:
                    self.shared_train_epoch.value += 1
                # Shared index overflow
                if self.cur_train_index.value >= self.training_size:
                    self.cur_train_index.value = self.cur_train_index.value %
self.minibatch_size
                # Calculate differences between process and shared epoch
                epoch_differences = self.shared_train_epoch.value -
self.process_train_epoch
            if epoch_differences > 0:
                self.process_train_epoch += epoch_differences
                for i in range(epoch_differences):
                    r.shuffle(self.train_list) # Catch up
                print "GENERATOR EPOCH {}".format(self.process_train_epoch)
                print self.train_list[0]
            print "PI: {}, SI: {}, SE: {}".format(cur_train_index,
self.cur_train_index.value, self.shared_train_epoch.value)
            if self.curriculum is not None and self.curriculum.epoch !=
self.process_train_epoch:
                self.update_curriculum(self.process_train_epoch, train=True)
            print "Train [{},{}] {}:{}".format(self.process_train_epoch,
epoch_differences, cur_train_index,cur_train_index+self.minibatch_size)
            ret = self.get_batch(cur_train_index, self.minibatch_size,
train=True)
            if epoch_differences > 0:
                 print "GENERATOR EPOCH {} -
{}:{}".format(self.process_train_epoch, cur_train_index, cur_train_index +
self.minibatch_size)
                 print ret[0]['source_str']
                 print "--------------------"
            yield ret

    @threadsafe_generator
```

```
    def next_val(self):
        while 1:
            with self.cur_val_index.get_lock():
                cur_val_index = self.cur_val_index.value
                self.cur_val_index.value += self.minibatch_size
                if self.cur_val_index.value >= self.validation_size:
                    self.cur_val_index.value = self.cur_val_index.value %
self.minibatch_size
            if self.curriculum is not None and self.curriculum.epoch !=
self.process_epoch:
                self.update_curriculum(self.process_epoch, train=False)
            print "Val [{}] {}:{}".format(self.process_epoch,
cur_val_index,cur_val_index+self.minibatch_size)
            ret = self.get_batch(cur_val_index, self.minibatch_size, train=False)
            yield ret

    def on_train_begin(self, logs={}):
        with self.cur_train_index.get_lock():
            self.cur_train_index.value = 0
        with self.cur_val_index.get_lock():
            self.cur_val_index.value = 0

    def on_epoch_begin(self, epoch, logs={}):
        self.process_epoch = epoch

    def update_curriculum(self, epoch, train=True):
        self.curriculum.update(epoch, train=train)
        print "Epoch {}: {}".format(epoch, self.curriculum)
# datasets/video/<sid>/<id>/<image>.png
# or datasets/[train|val]/<sid>/<id>.mpg
# datasets/align/<id>.align
class RandomSplitGenerator(BasicGenerator):
    def build(self, **kwargs):
        self.video_path = os.path.join(self.dataset_path, 'video')
        self.align_path = os.path.join(self.dataset_path, 'align')
        self.val_split = kwargs.get('val_split', 0.2)
        self.build_dataset()
        # Set steps to dataset size if not set
        self.steps_per_epoch  = self.default_training_steps if
self.steps_per_epoch is None else self.steps_per_epoch
        self.validation_steps = self.default_validation_steps if
self.validation_steps is None else self.validation_steps
        return self

    def build_dataset(self):
```

```
        if os.path.isfile(self.get_cache_path()):
            print "\nLoading dataset list from cache..."
            with open (self.get_cache_path(), 'rb') as fp:
                self.train_list, self.val_list, self.align_hash = pickle.load(fp)
        else:
            print "\nEnumerating dataset list from disk..."
            video_list = self.enumerate_videos(os.path.join(self.video_path, '*',
'*'))
            np.random.shuffle(video_list) # Random the video list before
splitting
            if(self.val_split > 1): # If val_split is not a probability
                training_size = len(video_list) - self.val_split
            else: # If val_split is a probability
                training_size = len(video_list) - int(self.val_split *
len(video_list))
            self.train_list = video_list[0:training_size]
            self.val_list   = video_list[training_size:]
            self.align_hash = self.enumerate_align_hash(self.train_list +
self.val_list)
            with open(self.get_cache_path(), 'wb') as fp:
                pickle.dump((self.train_list, self.val_list, self.align_hash),
fp)

        print "Found {} videos for training.".format(self.training_size)
        print "Found {} videos for validation.".format(self.validation_size)
        print ""
```

Fig. 5. generators.py

Because our data size was small, i.e., we only used 3 speakers to train the model out of 33 available speaker of the GRID dataset, we had to reduce the content of the dictionary used in `generators.py` to only include data of the speakers we used during the training process. Hence, we had to develop a script to process the align files of various speaker: `make_dic.py`.

```
import io
import os
import sys
# python make_dic.py s10 s10_dictionary.txt

align_folder = sys.argv[1]
dictionary = sys.argv[2]


def strip(align):
        data = [line.split(' ')[-1] for line in align]
        return ' '.join([line for line in data if line != 'sil']) + '\n'
```

```
sentences = []

for file in os.listdir(align_folder):
        if file.split('.')[-1] == 'align':
                with open(align_folder + file, 'rb') as f:
                        sentences.append(strip(f.read().splitlines()))

print sentences

with open(dictionary, 'w') as f:
        f.writelines(sentences)
```

Fig. 6. `make_dic.py`