# Voice Identification on the VoxCeleb Dataset

**Nicholas Guo** nguo17@mit.edu
**Sanjeev Murty** smurty@mit.edu
**Tony Wang** twang6@mit.edu

**Abstract**

We train a recurrent neural network for voice identification of celebrities on the VoxCeleb dataset (Nagrani et al., 2017). We then compare the performance of this network to the convolutional neural network that Nagrani et al. trained in their original paper.

## 1. Introduction

### 1.1. Dataset and Objective

The goal of this project was to use a Recurrent Neural Network to handle the task of classifying the speaker of audio clips of many different celebrities speaking in different environments. The motivation behind using an RNN was that we believed an RNN would perform better on time series data (which sound files are) than the convolutional neural networks (Nagrani et al., 2017) trained in their original paper. To make this comparison, we used their VoxCeleb dataset for our evaluation.

The VoxCeleb dataset contains around 150,000 sound clips from 1251 different celebrity voices. To ensure that our model could distinguish many different types of voices, the sound clips for our dataset are evenly distributed from male and female speakers and range from a wide variety of different ethnicities, accents, professions and ages. The clips themselves are chunks of sound ranging primarily from 3 to 10 seconds taken from many YouTube videos of speakers talking in different environments such as red carpet, large speeches, quiet interviews, and many others. Note that the dataset is affected by noise with factors such as background sounds, other interrupting voices, and potentially poor recording equipment.

For each sound clip, we aim to perform speaker classification among the 1251 speakers in the dataset. We evaluate the performance of our model by comparing to the performance of the network in (Nagrani et al., 2017).

## 2. Data Preprocessing

The audio files in the dataset are given as single channel WAV files sampled at 16 kHz.

### 2.1. Spectrograms

Instead of directly using the raw WAV files, we transformed our data into spectrograms. Spectrograms give a two dimensional representation of audio files, where one dimension is time, and the other is frequency. The spectrograms are computed by performing Fourier transforms on short windows of the audio file (we used a window size of around 8ms). The windows can be overlapping and there can be a weightage over the window. We employ a Mel frequency spectrogram which gives a more useful scaling of frequencies for human voice identification. We use window sizes of 1024 sam-

*Masschusetts Institute of Technology Fall 2017* Machine Learning (6.867) Final Project.

ples (our audio files are sampled at 16 kHz), and subsequent windows are shifted by 128 samples.

Figure 1 gives some examples of Mel spectrograms we generated. Unlike in Figure 1, where the entire spectrum was squashed to the range $[0, 1]$, the spectrograms fed into our network were had each of their rows normalized to zero mean and unit variance.

#### 2.1.1. Motivation for using spectrograms

Fourier transforms on short time scales are far more useful for speaker identification than raw audio data, since a section of speech is composed of a sequence of utterances. The Fourier transform of a short window breaks down such primitive utterances into component sinusoids. In this sense, a spectrogram gives a more natural representation of audio.

## 3. Architecture

### 3.1. LSTMs

In the first iteration of our RNN architecture, we used LSTM units. While the original model seemed to work fairly well, we switched to GRUs since empirically they elicited equal or better performance, are more computationally efficient, and are easier to modify.

### 3.2. GRUs

A gated recurrent unit (GRU), introduced by Cho et al. (2014), is a hidden unit for recurrent neural networks designed as an alternative to a LSTM unit.

Our architecture uses a variant the original GRU unit. Denoting its previous hidden state as $\boldsymbol{h}_{t-1}$, and its current input as $\boldsymbol{x}_t$, our GRU

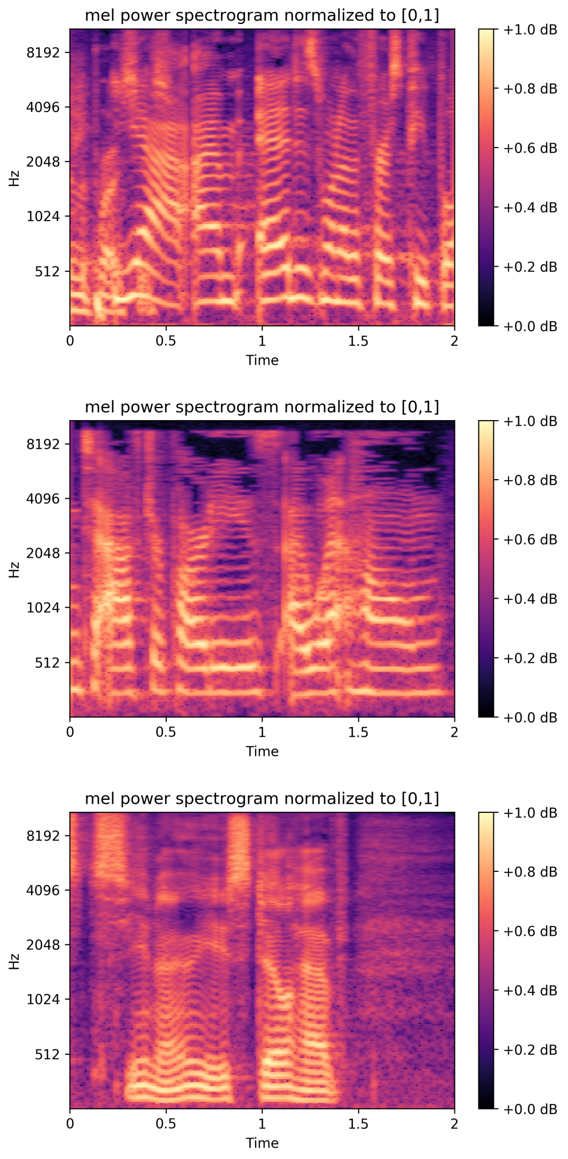


*Figure 1.* Mel Frequency Spectrograms: The top two spectrograms are of Taylor Swift, the bottom spectrogram is of Alan Rickman. Differences in the spectrograms of the two speakers are visible even to an untrained human. Obvious ones include the heights of subtler bands and their spacing. The goal is for our neural network to learn these differences and perhaps pick up on subtler ones.

unit computes

$$\begin{bmatrix} \boldsymbol{r} \\ \boldsymbol{z} \end{bmatrix} = \sigma\left(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{U}_1\boldsymbol{h}_{t-1}\right),$$

$$\hat{\boldsymbol{h}} = \phi\left(\boldsymbol{W}_2\boldsymbol{x} + r \odot \boldsymbol{U_2}\boldsymbol{h}_{t-1}\right),$$

$$\boldsymbol{h}_t = \boldsymbol{z} \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}) \odot \hat{\boldsymbol{h}},$$

where $\sigma$ is the logistic sigmoid function and $\phi$ is the hyperbolic tangent function. The network's output for this timestep is $\hat{\boldsymbol{h}}$.

## 3.3. Components/Ideas

### 3.3.1. Layer Normalization

Layer normalization is a technique introduced by Lei Ba et al. (2016) designed to speed up training akin to batchnorm (Ioffe & Szegedy, 2015). Unlike batchnorm however, layer normalization normalizes each batch item independently. For our GRU unit, we layer normalized every matrix multiplication with added trainable scale and shift parameters.

If $\boldsymbol{x} \in \mathbb{R}^n$ denotes a single batch item, then our layer normalization function $L$ acts as:

$$L : \boldsymbol{x} \rightsquigarrow \frac{a(\boldsymbol{x} - \overline{\boldsymbol{x}})}{\|\boldsymbol{x} - \overline{\boldsymbol{x}}\|_2} + b,$$

where $\overline{\boldsymbol{x}} \in \mathbb{R}$ is the mean of the coordinates of $\boldsymbol{x}$, and $a, b \in \mathbb{R}$ are trainable parameters.

With layer normalization, the equations in 3.2 become

$$\begin{bmatrix} \boldsymbol{r} \\ \boldsymbol{z} \end{bmatrix} = \sigma\left(L_1(\boldsymbol{W}_1\boldsymbol{x}) + L_2(\boldsymbol{U}_1\boldsymbol{h}_{t-1})\right),$$

$$\hat{\boldsymbol{h}} = \phi\left(L_3(\boldsymbol{W}_2\boldsymbol{x}) + r \odot L_4(\boldsymbol{U_2}\boldsymbol{h}_{t-1})\right),$$

$$\boldsymbol{h}_t = \boldsymbol{z} \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}) \odot \hat{\boldsymbol{h}},$$

where the $L$s with different subscripts denote layer normalizations with different trainable constants. The same layer normalizations *are* used across all time steps though.

### 3.3.2. Dropout

We applied dropout, introduced by Srivastava et al. (2014), to the input and output connections to our GRU units. For the recurrent connections of our GRUs, we applied recurrent dropout as detailed in Semeniuta et al. (2016). Letting $D$ denote the dropout function, our dropout scheme modifies the equations in 3.2 to be

$$\begin{bmatrix} \boldsymbol{r} \\ \boldsymbol{z} \end{bmatrix} = \sigma\left(\boldsymbol{W}_1 D_1(\boldsymbol{x}) + \boldsymbol{U}_1\boldsymbol{h}_{t-1}\right),$$

$$\hat{\boldsymbol{h}} = \phi\left(\boldsymbol{W}_2 D_1(\boldsymbol{x}) + r \odot \boldsymbol{U_2}\boldsymbol{h}_{t-1}\right),$$

$$\boldsymbol{h}_t = \boldsymbol{z} \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}) \odot D_2(\hat{\boldsymbol{h}}),$$

where the subscripts on $D$ are meant to clarify if two dropouts are the same. Our dropout functions were the same for all time steps.

Figures 2 and 3, demonstrate the ability of dropout to mitigate overfitting.

### 3.3.3. Averaging RNN Output

Most RNN architectures, including LSTMs and GRUs, are sequence to sequence architectures – that is they produce a sequence of outputs from a sequence of inputs, usually one output per input.

In our earliest experiments, our network architecture consisted passing our RNN's last time step output into a fully-connected output layer to produce class scores. This model yielded very poor performance, achieving only 52% training accuracy on a 10 class subset of our data.

In the second iteration of our network, instead of taking the last time step output, we instead averaged the outputs of all time steps. Our new network architecture is given in 4. This modification raised our training accuracy on the 10 class subset to 93.64%.

We believed that the reason this change induced such a large performance boost is that using only the last time step output of an RNN makes the network only learn from the last part of its input. The reason for this is that RNN gradients are known to become unstable the longer back in time they are back-

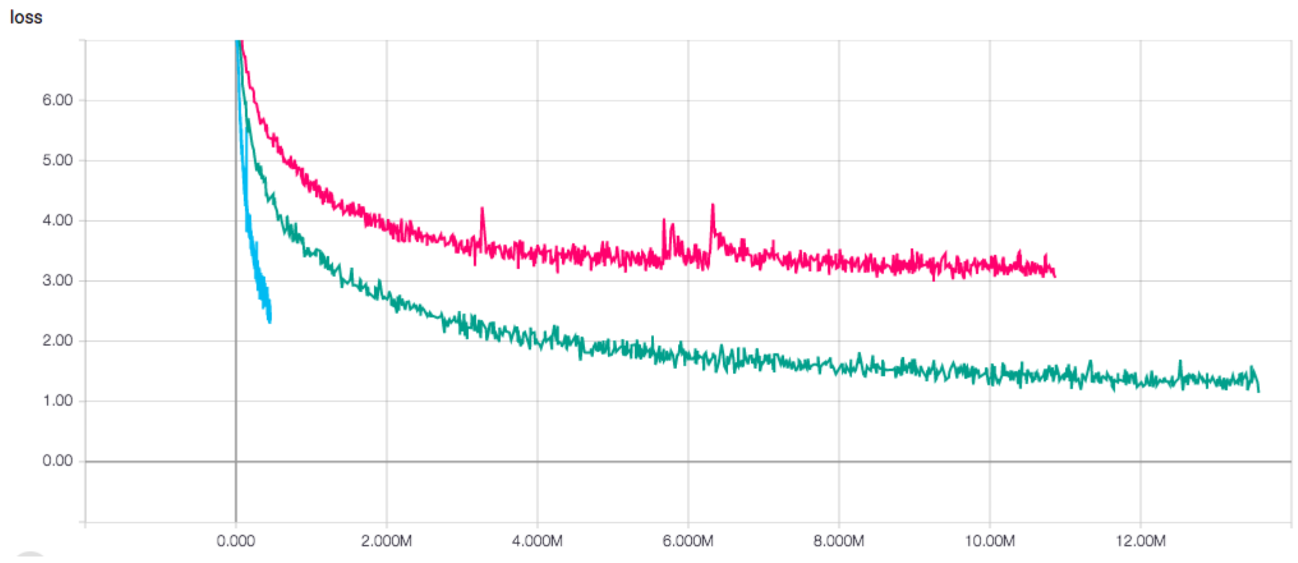


*Figure 2.* Training loss over time for different dropout rates: 0% (blue), 20% (green), 50% (red). As is evident, higher dropout rates lead to higher loss on the training data.

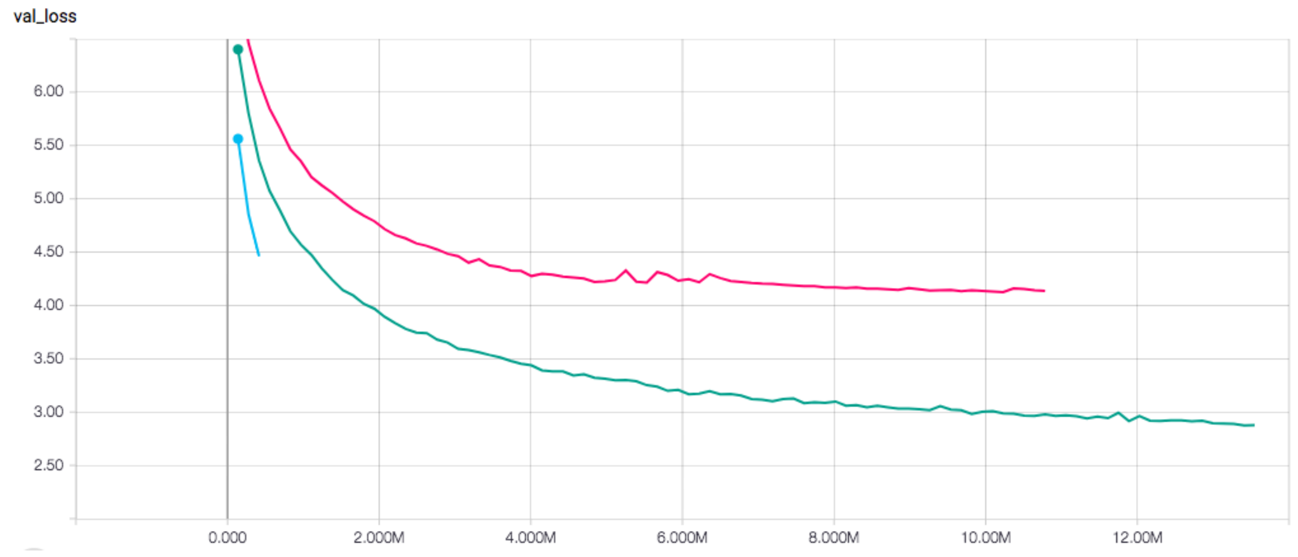
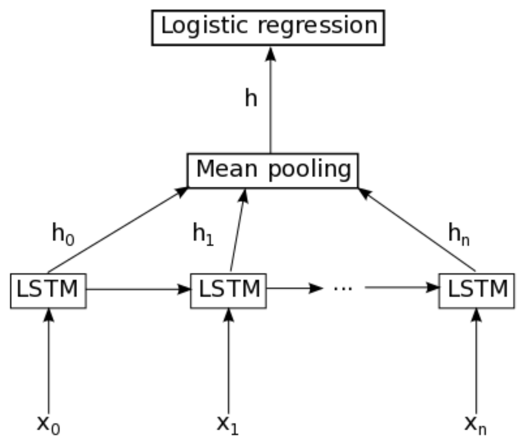


*Figure 3.* Validation loss over time for different dropout rates, as above. Comparing this plot to Figure 2, we see that dropout reduces the effects of overfitting and makes the training loss more representative of the validation loss.

propogated. Thus, the inputs at the earliest timesteps will be associated to the most unstable gradients, and thus the network will have a hard time updating itself correctly with respect to the earliest inputs. Average pooling solves this issue by providing a direct path for gradients to backpropogating to early timesteps.

### 3.3.4. Residual Stacked RNNs

We found that stacking RNNs on top of each other made our models converge faster, as evidenced by Figure 5. To make stacked networks easier to train, we made them residual, adapting the architecture given in He et al. (2015).



[t]

*Figure 4.* An RNN architecture with mean pooling. Image taken from a deeplearning.net tutorial. Our network architecture was heavily inspired after reading the linked guide.

## 3.4. Final Network Architecture

Our final model consisted of three stacked layer normalized GRU layers with residual connections, mean pooled, and then fed followed by a fully connected softmax output layer. See Figure 6.

## 3.5. Software Details

Our networks were written in Keras 2.1.2 using a TensorFlow backend. Our source code is available in our GitHub repository.

### 3.5.1. GPU-based Spectrogram Generation

We originally generated spectrograms with Librosa, a Python package for music and audio analysis. While this package generated the spectrograms with our desired parameters, it was too memory consuming to store spectrograms for all of our data points and took too long to reload them on the fly since they were only compatible with CPU's. So instead, we used Kapre, a package that made the spectrogram generation as another layer in our Keras model which allowed for GPU-based spectrogram generation (Choi et al., 2017). This was much faster for us and yielded the same spectrograms as Librosa.

### 3.5.2. GPU-Kernel for Layer Normalization

We used a custom written GPU-kernel for fast layer normalization. The GPU-kernel was around 4 to 5 times faster than our original implementation built by composing basic TensorFlow functions. The source of the GPU-Kernel is available on GitHub.

## 4. Training

The audio clips were first designated for training, validation, and testing by specifications provided by Nagrani et al. (2017). Every epoch of training, a random 2 second interval from each clip in our train set was taken and fed into our network. To clarify, the crops from each clip were different for each epoch. This cropping was done for both efficiency and because RNNs are easier to train on shorter sequences. The validation data was also cropped randomly during this stage for speed purposes (although the crop was the same for every epoch). Since we only used validation statistics to monitor overfitting, it was acceptable to not to have the true validation statstics over full clips.

We trained our models using Adam optimizer provided by Keras with default settings. We decayed the learning rate using Keras's `ReduceLROnPlateau` with decay factor of 0.2 and a patience of 5 epochs.

Most of our training (not including early prototyping) occurred on AWS p3.2xlarge instances using CUDA 9. Each epoch of our final model took around 6 minutes to train (for reference, this was on 138k training examples).

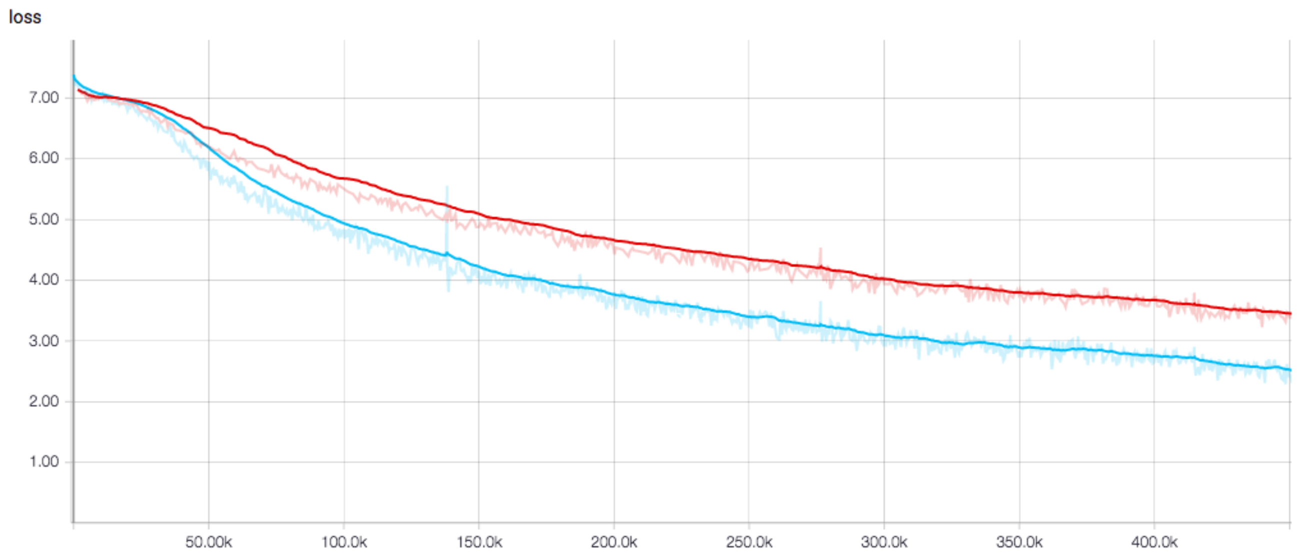


*Figure 5.* Training loss for the residual stacked architecture (blue) and the single GRU layer (red). $x$-axis is number of examples. One epoch is 138k examples.

### 4.1. Overfitting

The main challenge during training was dealing with overfitting. Our model had the capability to learn the training data, with our final architecture achieving 80% top-1 training accuracy and 93% top-5 training accuracy in less than 15 epochs. However, this came at the cost of severe overfitting, with the aforementioned model getting 34.5% top-1 validation accuracy and 47.3% top-5 validation accuracy.

We combated over-fitting through recurrent dropout. While this improved model generalization, it resulted in a model that underfitted training data and thus underfitted as a whole.

## 5. Performance of GRU Architecture

After training our RNN model with our final residual stacked GRU architecture, we obtained a training accuracy of 72.5% with top 5 accuracy of 89.3%. On our validation data, our model had an accuracy of 59.8% and 77.1% accuracy for top 5 classification. On the test data, our model had an accuracy of 58.1% and 75.9% accuracy for top 5 classification.

## 6. Comparison of GRU Architecture and Nagrani et al. (2017) CNN Architecture

### 6.1. Architectural Differences

The paper by Nagrani et al. (2017) used a CNN architecture on spectrograms of the input, which allows the model to exploit the structure of the data in both the frequency and time domains. Their architecture is slightly more difficult to generalize to audio clips of arbitrary lengths, compared to our RNN which is well-suited to time series data. Additionally, their architecture was extremely deep and contains many more parameters.

### 6.2. Comparison of Results

We ran each data point through both networks and recorded the rank of the data point's true class within the resulting prediction vectors for each network. We took these metrics and separated them into categories. Top 1 meant that the network predicted exactly the correct class. Top 10 meant that the network predicted the actual class to be within the top 10 choices, but did not predict the speaker cor-

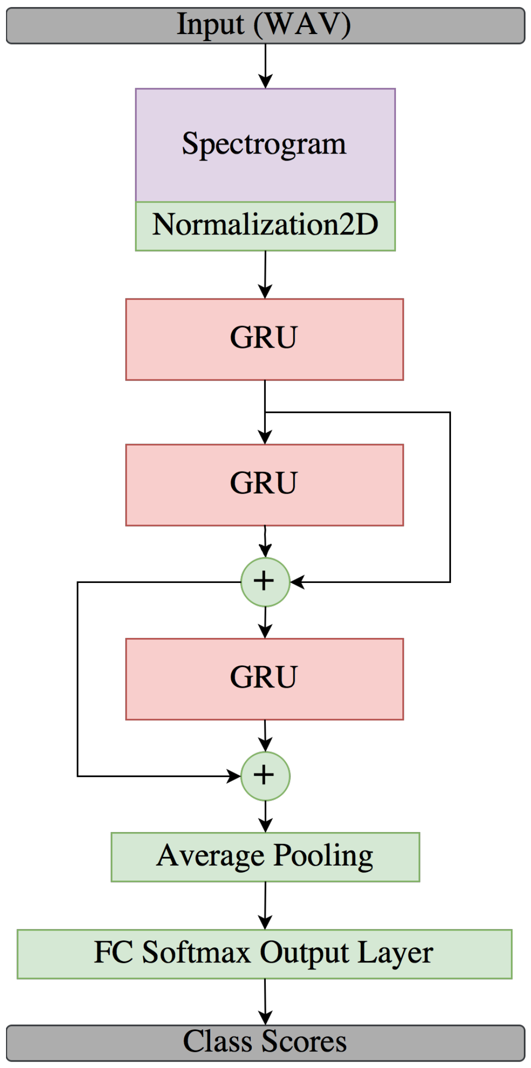


*Figure 6.* The stacked residual GRU network.

rectly. Not top 10 meant that the model was quite off. We then compared our GRU to their CNN model on the test data of 8251 examples. The breakdown is shown in table below.

| | | RNN rank | | | |
|---|---|---|---|---|---|
| | | 1 | 2-10 | >10 | Total |
| CNN Rank | 1 | 4281 | 1352 | 533 | 6166 |
| | 2-10 | 312 | 504 | 492 | 1308 |
| | >10 | 201 | 150 | 426 | 777 |
| | Total | 4794 | 2006 | 1451 | 8251 |

The rows in the table describe metrics achieved by the CNN model and the columns describes metrics achieved by our RNN.

Clearly, our model does not perform as well as the CNN in these rank metrics, but the table shows that there are still many examples that our model classifies correctly are completely fumbled by the CNN. This suggests that our model may be capturing some properties of the desired function that the CNN is not.

Overall our model failed to achieve the benchmarks obtained in Nagrani et al. (2017). They claimed to have achieved an 80.5% accuracy for predicting exact classes whereas our RNN model only had a accuracy of 58.1%. As for top 5 classification, our model only had an accuracy of 75.9% whereas their paper reported an accuracy of 92.1%. However, when we ran their architecture on the test data, it didn't match their reported results, obtaining a 74.7% for top-1 classification and 87.3% for top-5.

One particular case that the CNN outperformed the RNN on was the two speakers Allison Williams and Kate Mara. Our RNN evaluated all clips from the test set spoken by Allison Williams having highest probability of belonging to Kate Mara. Listening to the individual clips, the two actresses sound extremely similar; it was difficult to tell the difference.

The major difference between our RNN and their CNN was that their CNN was much more complex than our model, making the CNN more capable of observing subtleties. More specifically, their network had around 67 million trainable parameters whereas our network only had 1.7 million trainable parameters. This means that their model was able to fit much more complex functions than ours. Our number of parameters may have been on the low side. However, we did have limitations in terms of computing power. To increase our network complexity and thus the number of

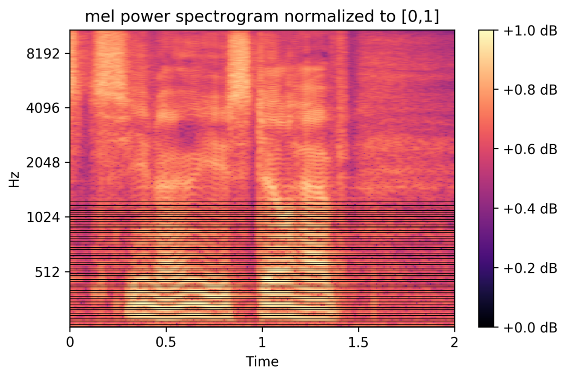


*Figure 7.* A corrupted spectrogram.

parameters, training would have taken much more time and GPU usage – resources we have limited access to (we were bounded by our $100 in free AWS credits).

We also had many hyper-parameters that may not have been tuned optimally. First, we could have tuned the parameters of the spectrogram such as the number of frequency bins or the window length for the FFT. Next we could have experimented more for the dropout rates for the input and output connections of our GRU units. We only tried dropout values of 0.2 and 0.5 for our network and didn't have the resources to try other values or combinations.

Finally, we may have had some issues with our spectrogram generation that might have corrupted our data a bit. When we were investigating the actual images that were generated we observed black bands that corresponded to values of 0 in particular frequency bins. This can be seen in Figure 7. We think that this was because the Fourier transform degenerates when the number of points is less than the number of frequency bins.

## Acknowledgements

We wish to thank Yuan Wang of Google Research for his suggestions on how to improve our network architecture. We also wish to thank Jeff Bezos for AWS.

## Division of Labor

### Nicholas Guo (`nguo17`)

Wrote RNN network in Keras, integrated spectrograms with GPU/network, wrote evaluator functions for both RNN and CNN models and other miscellaneous code for evaluating metrics, helped with network architecture design.

### Sanjeev Murty (`smurty`)

Worked on generating spectrograms, training a CNN model, running the VoxCeleb CNN, writing code to evaluate networks, researching network designs.

### Tony Wang (`twang6`)

Set up AWS instances and installed necessary software. Developed utilities to load data. Wrote code to monitor training and to checkpoint and load models. Conducted research into improving RNN performance. Modified Keras's `GRUCell` to support layer normalization and recurrent dropout. Experimented with training different architectures with respect to performance.

## References

Cho, Kyunghyun, van Merrienboer, Bart, Gülçehre, Çaglar, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

Choi, Keunwoo, Joo, Deokjin, and Kim, Juho. Kapre: On-gpu audio preprocessing layers for a quick implementation of deep neural network models with keras. In *Machine Learning for Music Discovery Workshop at 34th International Conference on Machine Learning*. ICML, 2017.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456, 2015.

Lei Ba, J., Kiros, J. R., and Hinton, G. E. Layer Normalization. *ArXiv e-prints*, July 2016.

Nagrani, A., Chung, J. S., and Zisserman, A. Voxceleb: a large-scale speaker identification dataset. In *INTERSPEECH*, 2017.

Semeniuta, S., Severyn, A., and Barth, E. Recurrent Dropout without Memory Loss. *ArXiv e-prints*, March 2016.

Srivastava, Nitish, Hinton, Geoffrey E, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.