Comparison of Voice-to-Text Neural Networks

DATS 6203: Final Project Report

Alex Cohen • Anamay Agnihotri • Tanaya Kavathekar

Introduction

Languages are an integral part of cultures worldwide. Due to the sheer variety of languages as well as the infinite permutations of accents and dialects, speech recognition is a complex problem that is growing more relevant in our day-to-day lives. With the advent of technologies like Siri, the Amazon Echo, and Google Home, major technology companies are bringing these voice-to-text recognition systems into tasks that users perform every day. Each of these major tech companies has likely spent years on development, with thousands of hours of speech data to arrive at their proprietary neural architectures used to recognize a voice when spoken into a cell phone or Alexa speaker, and translate it to a series of commands. These speech recognition systems are all highly advanced and do not provide many with the ability to look under the hood and see what is actually going on when you ask Siri to pull up directions home. Deep Speech2, an end-to-end Automatic Speech Recognition (ASR) system, allows us to study and get some insight into these highly sophisticated models, and see if domain-specific transfer learning techniques can be used to compare the capabilities of this open-source model with that of Google's Speech-To-Text API. Thus, the objective of this project is to assert whether open-source voice-to-text neural architectures like DeepSpeech2, either through pre-trained models or transfer learning, can compete with the proprietary, broadly developed translation models of tech giants like Google.

The Deep Speech 2 model is a hybrid Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) model that transcribes language by converting spectrograms of audio data to an attempted series of alphabet characters that are believed to be present in the audio file. The original model was developed by Baidu, an internet and artificial intelligence company, and published in their paper *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*. A *Pytorch* implementation of DeepSpeech2 (DS2) that we used for this project came courtesy of GitHub user SeanNaren, whose code and pre-trained models were used as the basis of our project. Our approach for this project was to compare the performance of three pre-trained DS2 implementations against Google's Speech-to-Text API, select the best performing model, and use transfer learning techniques to see how close we can get an open-source model to that of Google.

The rest of this report will have the following structure. The *Description of the Dataset* will contain an overview of the different audio libraries used to develop the training and testing sets for the models. The *Model Overview* section will describe the DS2 model and its architecture in more detail. The *Experimental Setup* section will describe the transfer learning approach undertaken to fine tune the pre-trained models and the *Results* section will provide an analysis of the outcomes of these training rounds. The *Summary and Conclusion* section will summarize the project and provide an outline for future work.

Description of the data set

There were four different audio datasets used throughout this project: the An4 dataset, the LibriSpeech dataset, the TED-LIUM dataset, and the Voxforge dataset. Each can be described as follows:

- **An4**: A directory containing alphanumeric responses to census questions. Each of the unique speakers would spell out responses to census questions, along with randomly generated words as prompted by the project directors. The data was collected by Carnegie Mellon University, and contains 948 training wav files and 130 testing wav files.
- **LibriSpeech**: A corpus containing approximately 1000 hours of English speech derived from read audiobooks assembled by the LibriVox project, in which volunteers read public-domain audiobooks. The audio is divided into three categories: training, dev, and testing. The training collection contains sets of 100 (clean), 360 (clean), and 500 (other) hours of audio; the dev collection contains sets of 5 (clean), and 5 (other) hours of audio, and the test contains sets of 5 (clean) and 5 (other) hours of audio. The difference between the "clean" and "other" audio collections was determined by the reader proficiency, with those reading with fewer speaking hours partitioned into the "clean" branch, and those with a higher number of errors into the "other" branch. The LibriSpeech dataset has also been added as a resource in TensorFlow datasets, accessible under the `tfds.audio.librispeech.Librispeech` package. In total, there are 292,367 individual audio files, however the existing model is trained on just the dev-clean and test-clean components, comprising 2,703 and 2,620 examples, respectively.
- **TED-LIUM**: A corpus containing 1495 transcribed TED talks assembled by researchers from Laboratoire d'Informatique de l'Université du Maine (LIUM), and represent over 118 hours of speech. The TED-LIUM dataset has also been added as a resource in TensorFlow datasets, accessible under the `tfds.audio.tedlium.Tedlium` package. In total, there are 58,863 files contained in the dataset.

- **Voxforge**: An open speech corpus created in 2008 with accented English recordings to aid in speech recognition tasks. It contains over 94,000 individual wav files, from which 1000 files were randomly selected and split to have 500 training files and 500 testing files.

As the data is in audio file format (.wav) and the Deep Speech 2 network first uses convolutional layers (see below section), each audio file must be pre-processed in order to be used for either training or transcribing purposes. The first step is to use the load functionality from the *scipy signal* package to load the files and create an array of audio magnitudes. All audio files were considered to be sampled at a 16000 sample rate. All the signals were normalized by 32767, which is a max value for 16 bit PCM (Pulse Code Modulation). The amplitudes of the sound waves are not very informative as they only signify the loudness of the audio recording. From there, a Fourier Transform is performed to decompose these signals into its component frequencies. An audio wave is composed of many different sounds at many different frequencies that all overlap to produce the sound of someone reading an audiobook, for example. A Fourier Transform converts the composite sound wave (which is in the time domain - measuring the amplitude of the sound wave over time) into individual sound waves of differing frequencies and their respective magnitudes (which is in the frequency domain - measuring the intensity of waves of different frequencies). This process is done using the *stft* command from the *LibROSA* package.

Once the audio file has been decomposed into its component frequencies, it is converted back into the time domain by creating a spectrogram. A spectrogram is a composite image of the different frequency waves (identified using the Fourier Transform) and their amplitude or magnitude over the duration of the sound sample. Essentially, a Fourier Transformation is done within a small window of time (approximately 0.02 seconds) and a slide rate of 0.1, and then the intensity of sounds at each frequency is calculated and combined to create an image. This image shows how the different component frequencies vary in intensity over time as the audio file is played. The spectrograms are developed from the Fourier Transformed data using the *magphase* command, again from the *LibROSA* python package. An example of this audio processing can be seen in Figure 1 below.

The process of creating a spectrogram from the initial audio file is done as part of the data loading pipeline and it is this spectrogram image that is fed to the model for training and transcription purposes.

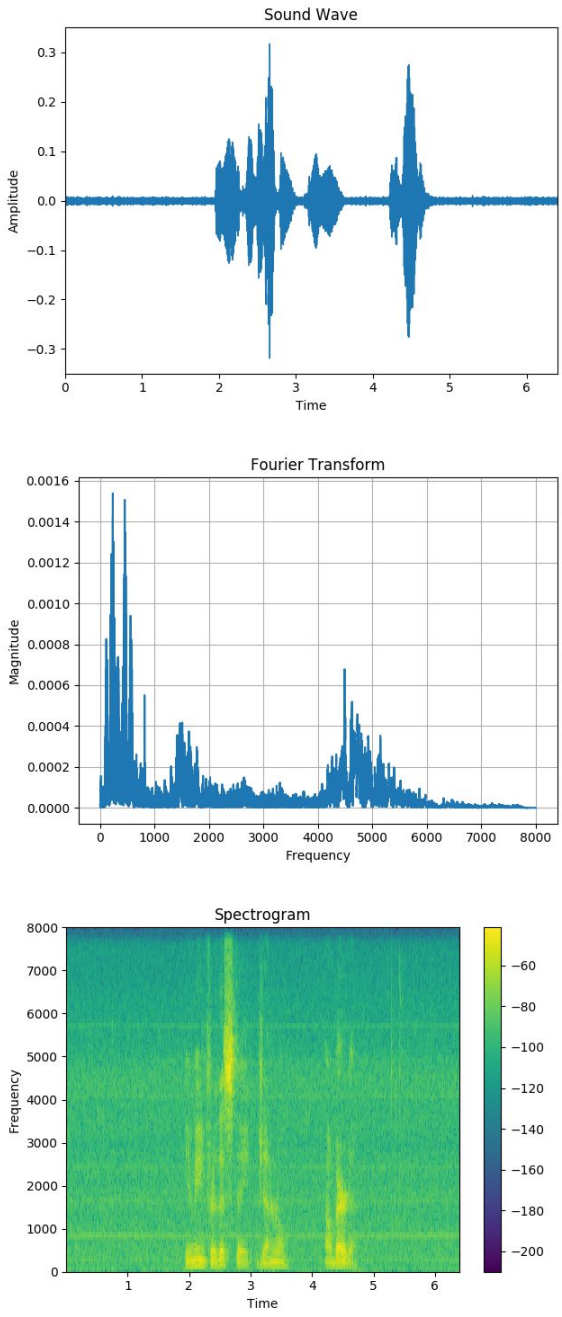


Figure 1 : The sound wave, Fourier transform, and spectrogram for the phrase "YOU MEAN FOR THIS STATE GENERAL ALBERTA"

Model Overview

DeepSpeech2 is an end-to-end Automatic Speech Recognition (ASR) pipeline that addresses the entire range of speech recognition contexts handled by humans. The objective of DS2 is to move beyond the complex ASR pipelines that are often made up of numerous components including complex feature extraction, acoustic models, language and pronunciation models, speaker adaptation, etc. Building and tuning these individual components makes developing a new

speech recognizer very hard, especially for a new language and do not generalize well across environments. The DS2 system simplifies this pipeline and claims to approach or exceed the performance of even human workers on several tests in two very different languages: Mandarin and English.

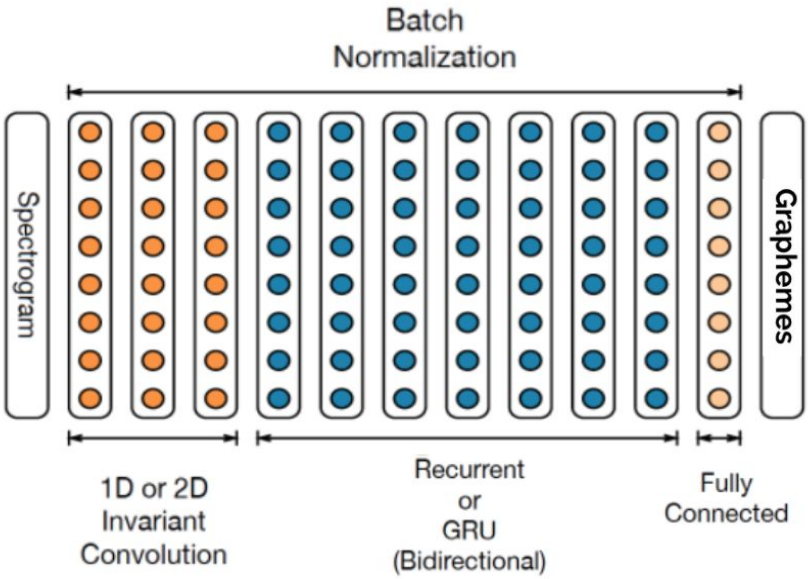


Figure 2 : DeepSpeech 2 architecture

Figure 2 above depicts the general architecture of the DS2 system. The system is made up of one or more convolutional layers, followed by three or more recurrent layers, followed by a final, fully connected layer. Not only is regular batch normalization applied to the convolutional layers, but a modified batch normalization (BatchNorm) is also applied to the recurrent layers to reparameterize the layer outputs of the LSTM. Throughout the network, the clipped ReLU (hardtanh in *Pytorch*) is used as the hidden activation function with the formula: $\sigma(x) = \min\{\max\{x, 0\}, 20\}$ as seen in the plot below.

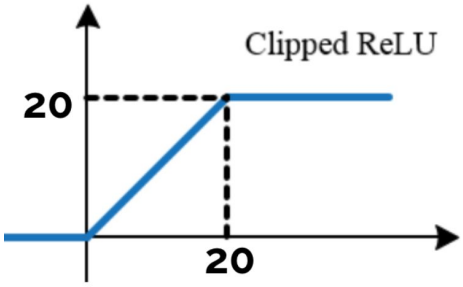


Figure 3 : Clipped ReLU/Hardtanh hidden activation

As mentioned earlier, the network takes audio spectrograms, which are electronic representations of audio waves, and generates text transcriptions by predicting one grapheme (character/unit of language) per time step *t*. Let a single utterance $x^{(i)}$ and label $y^{(i)}$ be sampled from a training set

$X=\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ...\}$. Each utterance, $x^{(i)}$, is a time-series of length T where every time-slice is a vector of audio features, $\{x^{(i)}_0, x^{(i)}_1, ..., x^{(i)}_{T-1}\}$. The normalized audio clips are converted to spectrograms by subsampling the maximum frequencies at a fixed interval *p* such that $x^{(i)}_{t,p}$ denotes the power of the p^{th} frequency bin in the audio file at time *t*. The goal of the network is thus to convert an input sequence $x^{(i)}$ into a final transcription $y^{(i)}$.

Specifically for the Deep Speech 2 English model, the output layer consists of 29 nodes, with each node signifying a grapheme in the English language. These include all alphabets (a through z), apostrophes, spaces, and blanks. The space and apostrophe symbols are used to denote word boundaries whereas the blank (null) symbol is used for cleaning the predicted transcription (discussed below). Since the output of the network at each time step is a character, we use a softmax activation in the last layer of the network such that the probability of some character *k* being the true character label l_t given some input *x* is:

$$p(\ell_t = k|x) = \frac{\exp(w_k^L \cdot h_t^{L-1})}{\sum_j \exp(w_j^L \cdot h_t^{L-1})}$$

where w^L_k is the weight on the character *k* in the last layer *L* and h^{L-1}_t is the output of the previous layer *L-1* at time step *t*.

One of the key insights of DeepSpeech is the usage of a CTC-RNN setup that allows for training from scratch without the need of framewise alignments for pre-training. CTC (Connectionist Temporal Classification) loss is simply a loss function that is used where having aligned data is an issue, like in the case of ASR. CTC eliminates the need of having each time-step annotated with a character label (which is often infeasible) and only requires the full transcript for a given spectral input. Moreover, it proves useful in reducing the temporal character outputs to meaningful words. For example, depending on the speaker's tone and accent, the word "speech" can be predicted by the network as "sssppppeeech" or "sppeeechh". Clearly, the duplicate characters need to be dropped to accurately transcribe. However, not all duplicates can be naively dropped; the two "e"s in "speech" need to be kept!

The CTC loss function proves to be very useful in both these cases. Since there are no character-level labels available to calculate the performance at each time-step, the CTC operator waits for the input-level predictions to be made. It then tries all possible alignments of the actual text in the audio input and takes the sum of all alignment scores. The score of the actual text is high if the sum over the alignment-scores has a high value and thus a higher score indicates better performance. In the case of dropping duplicate (decoding) characters, the blank (null) symbol proves to be instrumental. When predicting a text, we insert arbitrary blanks at any position, which will be removed when decoding. We then define a rule such that if an alphabet

follows a blank which follows the same alphabet, we do not drop the duplicate alphabet. Thus, under the CTC operator, alignments like "-sss-p-e-e-c-h", "s-ppe-e-c-h" and "-s-p-e-ec-hhh" (among many other) are all valid predictions that can be assigned a score. For example, for a particular iteration of the network predicting the word "too" the probability p_t at time t={0,1,2} :

$p_0(t)=0.5, p_0(-)=0.3, p_0(o)=0.2 \quad p_1(t)=0.1, p_1(-)=0.4, p_1(o)=0.5 \quad p_2(t)=0.1, p_2(-)=0.3, p_2(o)=0.6$

Here, the probability of the alignment "t--" is 0.5*0.4*0.3 = 0.06 whereas for "too" this is 0.15. Since the CTC operator uses the sum of all such possible alignments while calculating the error with respect to the actual text, using the blank character as just another output symbol allows for the network to identify the valid duplicates. Moreover, since there is no character-level dependence on the error calculation, CTC loss coupled with the RNN architecture can be used to map variable length audio input to variable length output to model temporal information. The precise CTC loss function $L(x,y,\Theta)$ used by DS2 is:

$$\mathcal{L}(x, y; \theta) = -\log \sum_{\ell \in \text{Align}(x,y)} \prod_{t}^{T} p_{\text{ctc}}(\ell_t | x; \theta).$$

where *Align(x, y)* is the set of all possible alignments of the characters of the actual text y to the spectral input x under the CTC operator. The performance index is defined as the negative log of the sum of these alignment probabilities, which is then backpropagated through the network. Since the CTC loss function integrates over the alignments, it is never explicitly asked to produce an accurate alignment. In principle, CTC could choose to emit all the characters of the transcription after some fixed delay but empirically produces accurate alignments when trained on bi-directional recurrent layers, which this project uses. The DS2 model uses synchronous Stochastic Gradient Descent (SGD) as the training algorithm due to its reproducibility and deterministic results across parallelized GPU computations. For the purposes of this project, the best transcription is created using a greedy method of choosing the most likely character per time-step. The DS2 model also proposes a Beam Search approach for creating a transcription that maximizes *Q(y)*, which is a linear combination of log probabilities from the CTC trained network and language model, along with a word insertion term. This approach can be explored in the future to improve transcription results.

In the CTC loss function, the character products for each utterance alignment across T time steps shrinks with the length of the sequence since more character probability products (each < 1) lead to smaller total values. To account for this, DS2 uses SortaGrad, which sorts the training batch in increasing order of the length of the longest utterance. Sorting the input length-wise lets the loss function use the length of the utterance as a heuristic of difficulty, where longer utterances should have higher costs. While not critical to model performance, this approach proves effective in maintaining numerical stability. This is primarily because long utterances tend to have larger

gradients and are more likely to cause the internal state of the network to explode at an early stage of training.

The DS2 system uses many such performance-enhancing tricks to ensure the ASR pipeline remains simple yet effective. The convolution layers use Conv2d filters which convolve across both the frequency and time domains as a pre-processing measure. Convolutions in frequency model spectral variance due to speaker variability more concisely than larger, fully connected networks. Uniform striding is also used to subsample the input while preserving the number of input features across frequency and time. However, since the number of characters per time step is relatively high (compared to Mandarin) in English language speech, large strides can skip on crucial information. To correct for this, non-overlapping bi-grams can be used as the prediction graphemes. However, model training for this project used unigrams with minimal striding since the training set was much smaller than the 11,940 hours of augmented English speech used by the original DS2 model.

Beyond the architectural tweaks, a key contribution of the DS2 system is the computational optimizations. These include many high performance computing tricks like GPU parallelization and clever, dynamic memory allocations to the GPUs and CPUs. Moreover, data augmentation techniques like noise injections also make the pre-trained models robust to noisy speech. Such optimizations were beyond the scope of this project.

Experimental Setup

For the first experiment, our idea was to test the generalizability of the Deep Speech 2 pre-trained models. The 3 pre-trained models, which were trained on an4, tedlium, and librispeech data sources were tested on a subset of the Voxforge data. We randomly sampled 500 audio files of durations between 1 and 15 seconds. Audio files were run through the pre-trained models to produce a predicted transcription. The pre-trained model has the following architecture:

The spectrogram signal generated after the data preprocessing is passed through the convolution 2D layer of 32 channels. The idea behind using the convolution layer is to capture the important information and high level features from the spectrogram image. The 1st convolution layer which has an input of 1 channel is traced using the kernel of size 41x11. The spectrogram image is padded with zeros of size 20x5. Every layer is batch normalized to optimize the process, and the hard tanh function is used as the hidden activation. The next convolution layer reduces the size of the kernel by half (21,11). The reduced spectrogram image of 32 channels is passed through the bi-directional LSTM RNN network. The 5 layered bi-directional LSTM has an input units of 1312 (32*41 feature space) and 1024 hidden units. The RNN layer is further connected

to a fully connected linear layer of 29 neurons. The output of the neurons is passed through a softmax activation function to get a class probability score. Below is the network architecture diagram of librispeech pre-trained model:

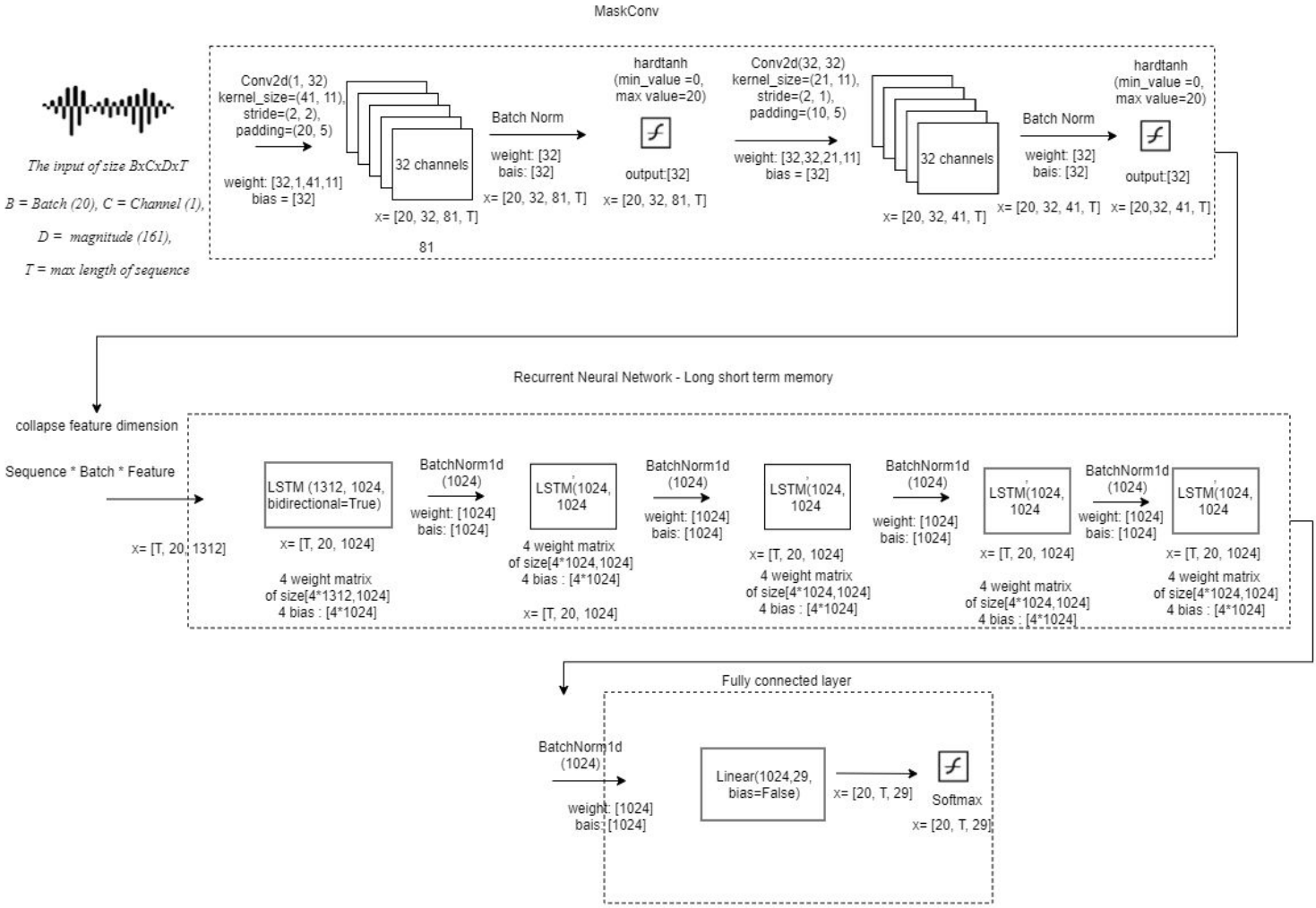


Figure 4 : Network Architecture diagram for Librispeech model

Once the voxforge audio files were passed through the pre-trained models, they were further evaluated using the *Transcribe_and_Compare.py* file. This python script takes a wav files path, the corresponding text files path, and a model path. All the wav files in the provided directory and the corresponding transcriptions will be loaded into the working directory. The model located at the provided directory will also be loaded into memory and onto the GPU (CPU if running the code without a GPU). From there, the function will loop through each input-target (wav and txt) pair, converting the wav file to a spectrogram, and retrieving the output from the model. The output will then be decoded using a Greedy decoder, as described below.

The model will return the predicted graphemes across each 20ms segment of the spectrogram, however, depending on the audio clip, it is possible for a single character to span across multiple 20ms windows. For this reason, a Greedy decoder will be used to interpret the model output and reduce the predicted values down to a logical string. As the model produced a probability for each label ({a..z, comma, space, and blank}), the Greedy decoder will take the label with the maximum predicted probability as the output. This means that, for example, if the letter “a” is predicted with a 25% likelihood, yet the second most likely character is “e” with only 20% likelihood, the decoder would return “a” for that 20ms window. The beam decoder approach discussed earlier would be a future adaptation of this decoding strategy.

Once the entire probability matrix has been reduced to the most likely characters, the string must be simplified. This is done by converting the numeric argument to the character using a label dictionary, and then appending each successive character, spaces included, as the output. If the *remove_repitions* argument is invoked (as it is during the *Transcribe_and_Compare* script), characters will be reduced when the 20ms transcription is equal to the previous 20ms transcription. As explained in the *Model Overview* section, the blank character prevents returning output such as “sssppppeeeeeeeeecchhh” and would ideally return the word “speech”. Once the output of the model has been decoded, the actual transcription will be loaded from the associated text file for evaluating the predicted transcriptions.

All models are evaluated on a set of 500 test voxforge audio files using the word error rate (WER) and character error rate (CER). The WER is a measure of how many words need to be converted to align a test text to the reference text. The equation measures the number of substitutions (S), deletions (D), and insertions (I) needed to make the attempted translation the same as the actual transcription with a total number of words (N). The CER is a similar metric and uses character substitutions, insertions, and deletions in order to align two sets of text by character/letter. Both metrics will be reported when comparing model performance.

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

Figure 5 : Error Rate Calculation

These two metrics, along with lists of the predicted and actual transcriptions, were returned to determine how the Deep Speech 2 models compare with Google's speech-to-text API.

The Google speech-to-text tool enables developers to convert audio to text by applying powerful neural network models. The tool recognizes over 120 languages and can process real-time streaming and prerecorded audio. Beyond naive speech recognition, the tool allows for much

more complex tasks like separating multiple speakers, detecting speaker language as well as multiple channel (audio sources like phone call) transcriptions. This tool can be accessed via the google cloud API wrapper in python known as *speech_v1* which was used for this project. Each API request returns multiple transcriptions which are sorted in decreasing order of transcription confidence. The API is easy to use and reliable and can be accessed by simply enabling the API and obtaining credentials for a service account on google cloud. The same 500 Voxforge test files were used to similarly calculate the WER and CER statistics using the API. The most likely (first) transcription from the received response is used for the evaluation.

We observed that the librispeech model worked relatively well on the voxforge test set but was still inferior to the Google API results. Hence for our transfer learning experiment, we decided to focus on the librispeech model and further tune it to compete with the Google API.

For transfer learning, we used different parameter settings to fine-tune the models. We started with the librispeech model and tuned hyper-parameters such as the hidden size of RNN 500 and RNN type as Gated Recurrent Unit (GRU). GRU is just like LSTM with a forget gate but has fewer parameters than LSTM as it lacks an output gate. Synchronous SGD was used as a training algorithm with a learning rate of 3e-4 and a momentum of 0.9. In order to optimize the learning process and not to miss the global minimum, we adjusted the learning rate by 1.1 after every iteration. This method of starting at a relatively high learning rate and gradually lowering it is called learning rate annealing. The model training starts with loading of the weights and biases of the Librispeech pre-trained model. The data generator script generates 20 audio files in every batch. This number was decided after checking optimal time performance and the CTC loss. The data generator returns input, targets, input_percentages, and target sizes. The input percentage is calculated as the length of the input sequence divided by the maximum length of the sequence in that batch. These inputs are then moved to the GPU to perform model building. The input percentage sizes are further used to mask the maximum length of the sequence in the corresponding batch. The input is passed through 5 layers of 2D Conv, 5 layers of GRU and a fully connected linear layer. The output of the fully connected linear layer is of shape (batch size, max length of the sequence, 29 labels). The probability matrix generated at the end of the model is further decoded using the Greedy decoding technique to generate the sequence. The performance index used is CTC loss for every epoch. The model is saved after epoch and overwritten by the best model across epochs. These models and optimizers are loaded with the Nvidia apex AMP function. AMP is automatic mixed precision which allows users to easily experiment with different pure and mixed precision levels.

In the first iteration in the fine-tuning process, we trained the model over 70 epochs. Every epoch took around 71 secs. As the results were not satisfactory we changed the RNN type to bi-directional LSTM and ran an iteration with the same settings as before. The results were still

not satisfactory hence we increased the hidden size of RNN to 1000 and kept the rest of the settings the same. The results were still not up to the mark so we decided to increase the training dataset to ~7500 audio files from the voxforge data source. An iteration was run with the same hyperparameter setting. But since the data size had substantially increased and the execution time for each epoch was approximately 5 hours, we only ran 3 epochs.

Results

The results of the first experiment, testing the generalizability of the DS2 pre-trained models are summarized in Table 1 below. The pre-trained models were tested with the randomly sampled 500 voxforge audio files, as this dataset was entirely unknown to the models. The models were compared using the WER and CER evaluation metrics. Within the pre-trained models, the An4 dataset had the maximum WER and CER of 10.05% and 30.80% respectively, followed by the Tedlium model with a WER and CER of 4.86% and 9.23%. The LibriSpeech model had the lowest WER and CER of 2.27% and 4.28%, respectively. We hypothesize that this could be due to the similarities between the training and testing datasets, as the Voxforge data and LibriSpeech data are both audiobook corpora. Moreover, the LibriSpeech dataset's size is relatively higher than the An4and Tedlium datasets and thus the pre-trained model has many more hours of audio data. Unsurprisingly Google had the least WER of 1.61% and CER of 3.73% on the test set. The pre-trained models took approx. 4 mins of evaluation time. This evaluation time was doubled (~8 mins) for the google API since each external interface call involved loading the model, backend processing and request communication for each audio file and understandably delayed its response time. Overall as the librispeech pre-trained model had a better performance on the voxforge dataset when compared to the other pre-trained models, we further fine-tuned this model to compete with the Google API error rate.

Model	WER(%)	CER(%)	Eval time
An4 - Pretrained	10.05	30.80	~4 mins
LibriSpeech - Pretrained	2.27	4.28	~4 mins
TED-LIUM - Pretrained	4.86	9.23	~4 mins
Google Speech-to-Text	1.61	3.73	~8 mins

Table 1 : Word Error Rate and Character Error Rate on pre-trained models

Starting with the librispeech pre-trained models, the first iteration was executed using the GRU RNN and hyperparameters of 5 hidden layers and 500 neurons in each recurrent hidden layer.

The model was trained for 70 epochs with a batch size of 20 audio files. The WER and CER were 3.09% and 6.09% respectively, which had increased from the initial baseline of 2.27% and 4.28%. We then switched to the bi-directional LSTM RNN layers as suggested in the literature while keeping the same hyperparameter configuration as before. In this case, the WER and CER fell to 3.15% and 6.35%. Encouraged by this, we decided to increase the hidden layer size to 1000 neurons per layer while keeping the rest of the parameters the same. Though an improvement from the previous model, we decided to further enhance performance by increasing the training size to ~7.5k voxforge audio files. The training with the huge dataset expectedly took over 5 hours per epoch. Due to our limited GPU resources, the model was evaluated after 3 epochs (~15 hours of execution time). This model has substantially better results with a CER of 3%, beating the Google API's CER of 3.73%. Even in the case of WER, there was only a 0.05% difference between our transfer learning model and the Google API. This shows that the pre-trained Librispeech model, when trained with a larger dataset with specific parameter tunings performs at par with the Google API. Table 2 below summarizes the WER, CER, and evaluation time for the transfer-learned models.

Model	WER(%)	CER(%)	Eval time
LibriSpeech - Pretrained	2.27	4.28	~4 mins
LibriSpeech - Pretrained -Transfer Learning (GRU)	3.09	6.09	~4 mins
LibriSpeech - Pretrained -Transfer Learning (LSTM)	3.15	6.35	~4 mins
LibriSpeech - Pretrained -Transfer Learning (LSTM+1000)	3.14	6.31	~4 mins
LibriSpeech - Pretrained -Transfer Learning (LSTM+1000)	1.66	3.00	~4 mins
Google Speech-to-Text	1.61	3.73	~8 mins

Table 2 : Word Error Rate and Character Error Rate on the fine tuned models

Summary and Conclusions

In conclusion, we found that it was possible to create a domain-specific, open-source model that is competitive with Google's proprietary, generalized speech-to-text tool. The Librispeech model generalized the best out of the pre-trained Deep Speech 2 models, likely due to the larger amount of training data available as well as due to its similarities to the Voxforge dataset. However the pre-trained model was still not quite comparable to Google's speech-to-text model. It was not until we trained the LibriSpeech Deep Speech 2 model on the larger 7500 audio files that we achieved a model with similar WER and CER scores to the Google API. However, it is important to note that our model is highly tuned to the input data and is likely to not generalize as well as the Google model. A source of further investigation would be to test the performance of the refined LibriSpeech model and Google API on a new, unseen dataset to test generalizability.

The biggest challenge of our project was to bridge the understanding between the DS2 literature and the github implementation. As seen in the appendix below, the implemented DS2 system has a lot of pieces, each with a specific tuning purpose. Deconstructing the code while relating it back to the original literature proved to be a time-consuming task. Since our project was more geared towards framework exploration and evaluation, our attempt was to strip down the complex code to a readable and well-documented format and thus some of our training approaches were constricted by the lack of time.

Throughout this project, there were a few tasks that could have led to better results, the first being computing power. Due to the fact that we were training these models on instances with a single GPU, the training process was quite time intensive. The five hours per epoch severely limited the number of training rounds that could be achieved in the limited time we had. This could have been corrected by getting access to improved computing resources or implementing parallelized GPU computing as the DS2 paper suggests. Training efficiency is an aspect of this project that could be improved in the future. Another future task would be to additionally train the transfer-learned model on all three datasets merged into a single training audio repository. This would likely increase the robustness and generalizable performance of our final model.

Additional work that could be undertaken in the future would revolve around noise augmentation and improved training monitoring. We hypothesize that adding augmentation to the input wav files, such as louder background noise, speeding up/slowing down of the recordings, and volume alterations could likely improve the performance of our final models. We would be better equipped to track our tuning efforts by building out more visualization capabilities, including packages like TensorBoard or Visdom as they would allow us to more closely monitor the

training process. These improvements would likely define a promising path to the development of a more robust and generalizable speech to text model.

References

Baidu Silicon Valley AI Lab, B. S. (2015). Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. 28.

SeanNaren (2017-2020). https://github.com/SeanNaren/deepspeech.pytorch

Chaudhary, Kartik. "Understanding Audio Data, Fourier Transform, FFT, Spectrogram and Speech Recognition." *Medium*, Towards Data Science, 10 Mar. 2020, towardsdatascience.com/understanding-audio-data-fourier-transform-fft-spectrogram-and-speech-recognition-a4072d228520.

Dossman, Christopher. "A Data Lake's Worth of Audio Datasets." *Medium*, Towards Data Science, 14 Nov. 2018, towardsdatascience.com/a-data-lakes-worth-of-audio-datasets-b45b88cd4ad.

Panayotov, Vassil, et al. "Librispeech: An ASR Corpus Based on Public Domain Audio Books." *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, doi:10.1109/icassp.2015.7178964.

Rousseau, Anthony, et al. "TED-LIUM: an Automatic Speech Recognition Dedicated Corpus." *Laboratoire Informatique De l'Université Du Maine (LIUM)*, pp. 125–129.

Scheidl, Harald. "An Intuitive Explanation of Connectionist Temporal Classification." *Medium*, Towards Data Science, 4 Dec. 2018, towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c.

Documented Computer Listings (code)

The code is divided into three different categories:

Data Loading

`data_loader_stripped.py` - a file used to create data loader objects, parsers, and spectrogram classes

`create_train_val_set.py` - a file used to generate the Voxforge training and validation sets

`utils_stripped.py` - a file housing miscelaneous utility functions, including the `load_data()` function

Model Training

`model.py` - a file storing the Deep Speech 2 model class and associated properties

`train_stripped.py` - a file used to train new models and perform transfer learning on existing models

Transcription

`decoder_stripped.py` - a file used to create the Greedy Decoder class for decoding model output

`Google_API.py` - a file used to transcribe audio files with Google's Speech-to-Text API

`Transcribe_stripped.py` - a file used to house the transcribe functionality

`Transcribe_and_Compare.py` - a file used to transcribe data using a given model and calculate WER and CER measures

One sample code file is below:

Transcribe and Compare Function

```
def transcribe_and_compare(wav_dir, txt_dir, model_dir, n_files=500, verbose=False):
        # set random seed for sampling wav files
```

```
        random.seed(1)

        try:
        # set device as cuda if possible
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        # load list of wav files that we will transcribe
        wav_dir = os.getcwd() + wav_dir
        wav_files = [wav_dir + f for f in os.listdir(wav_dir) if f[-4:] == '.wav']
        print('wav files found 1/3')

        # since the an4 model is built on the smallest dataset (130 test cases), downsample if needed to ensure
        # equivalent sizes
        #           wav_files = random.sample(wav_files, 130)  # 130 is the length of the an4 testing set

        # load list of txt files containing the actual transcriptions
        txt_dir = os.getcwd() + txt_dir
        txt_files = [txt_dir + f[len(wav_dir):][:-4] + '.txt' for f in wav_files]
        print('txt files found 2/3')

        # load the model that will be used to transcribe - look into why half
        model_path = os.getcwd() + model_dir
        model = load_model(device, model_path, use_half=False)
        print('model found 3/3')

        # print error if loading fails
        except:
        print('Model and source not found, returning NaN')
        return np.nan, np.nan

        try:
        # specify decoder
        decoder = GreedyDecoder(model.labels, blank_index=model.labels.index('_'))

        # specify spectrogram parser - turn wav files into spectrograms
        spect_parser = SpectrogramParser(model.audio_conf, normalize=True)

        # set n_files to the max possible, unless the user specifies a number of files to use
        if n_files == 'All':
        n_files = len(wav_files)

        # initialize empty lists to store information
        cer = []
        wer = []
        transcribed = []
        actual = []

        print('Paths specified and model loaded. Transcribing')

        # raise error if parser and decoder have an issue loading
        except:
    print('Parser and decoder issue')
        return np.nan, np.nan
```

```
        try:
        # loop through each file in list of files
        for i in tqdm(range(n_files)):
        decoded_output, decoded_offsets = transcribe(audio_path=wav_files[i],        # name of wav file
                                    spect_parser=spect_parser,         # spectrogram parser
                                    model=model,                # model
                                     decoder=decoder,           # greedy or beam decoder
                                    device=device)              # cuda or cpu

        # open associated txt file and get contents
          f = open(txt_files[i], 'r')
        act = f.read()
        f.close()

        # get the contents of the decoded output
        decode = decoded_output[0][0]

        # METRICS TO UPDATE! Currently, calculate Levenshtein distance between transcription and
predicted
        ### CER ###

        # replace any spaces
        decode_lev, act_lev = decode.replace(' ', ''), act.replace(' ', '')

        # calculate distance without spaces
        ld_cer = Levenshtein.distance(decode_lev, act_lev)

        # append CER to running list
        cer = np.append(cer, ld_cer)

        ### WER ###

        # split output strings by spaces and create set of unique words
        uniquewords = set(decode.split() + act.split())

        # create dictionary of each word and a corresponding index
        word2char = dict(zip(uniquewords, range(len(uniquewords))))

        # map the words to a char array (Levenshtein packages only accepts
        # strings)

        # map words to index in the dictionary word2char
        w1 = [chr(word2char[w]) for w in decode.split()]
        w2 = [chr(word2char[w]) for w in act.split()]

        # calculate distance from word vectors and append to running total
        ld_wer = Levenshtein.distance(''.join(w1), ''.join(w2))
        wer = np.append(wer, ld_wer)

        # option for user to print as the data as it is transcribed
        if verbose:
        print('Predicted: %s' % decoded_output[0][0])
        print('Actual: %s' % act)
        print('Levenshtein Distance (CER): %i' % ld_cer, end='\n\n')
```

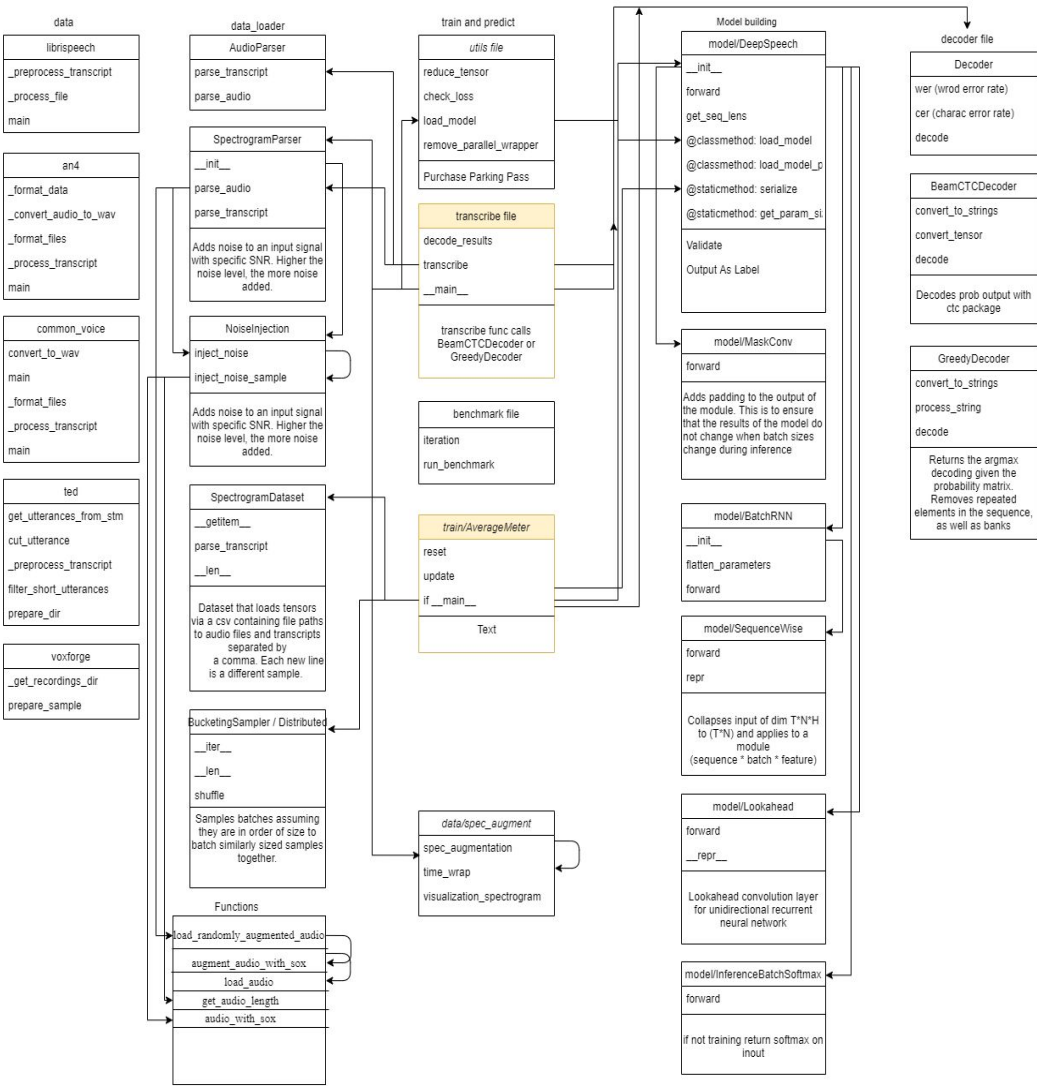
```
        # append pred and actual to respective lists
        transcribed = np.append(transcribed, decoded_output[0][0])
        actual = np.append(actual, act)

        print('Completed Parsing')
        print('Mean Levenshtein distance (CER): %f' % np.mean(cer))
        print('Mean Levenshtein distance (WER): %f' % np.mean(wer))

         # return lists of predicted transcriptions and actual transcriptions
        return transcribed, actual

        except:
        print('Transcription Prediction failed')
        return np.nan, np.nan
```

Appendix



Appendix 1 : The flow diagram of the original deepspeech2 code base.