Deep Speech 2 Individual Report

DATS 6203
Alex Cohen

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 pretrained 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 Individual Work section the specific tasks that I completed during the course of the project. The Project Contributions section describes how the specific tasks I completed fit in to the larger project and helped answer our original research question. The Results and Summary and Conclusions sections will describe

how my contributions helped enable the eventual conclusions reached during the project, and the Code Comparison will break down just how much of the developed code was adapted from the existing codebase.

Individual Work

I developed the following scripts found in the GitHub repository: *Transcribe_and_Compare.py*, *Create_train_val_set.py*, *data_loader_stripped.py*, *decoder_stripped.py*, *transcribe_stripped.py*, *test_stripped.py*, and *utils_stripped.py*, as well as collaborated with the rest of the team to develop the ReadMe files located on the main repository landing and in the *code* folder of the repository. In terms of the figures used in the final presentation, I created the audio wave, Fourier Transform, and spectrogram example images along with the accompanying description of the audio transformation process, as well as generated the baseline WER and CER for the pretrained models in the results table.

Project Contributions

The main contribution of our project was simplifying the existing Deep Speech 2 codebase to make it more user-friendly for someone without a software engineering background, as well as training new models using transfer learning techniques. This required an extensive amount of rewriting existing python scripts to inject comments, reduce extraneous lines of code that would not be used during our project, and attempt to consolidate the existing codebase. One of my main contributions to this project was the simplification and extensive commenting of the existing codebase, especially as it pertained to the audio loading and conversion to a spectrogram, as well as model output decoding and transcription capabilities already developed by SeanNaren.

```
def parse_audio(self, audio_path):

    # specific function to load audio with volume perturb, otherwise load audio
    y = load_audio(audio_path)

    # number of fft points (sample rate * window size = total points) per second
    # eg. voxforgetest[1] = 320
    n_fft = int(self.sample_rate * self.window_size)

    # set window length (320 hz)
    win_length = n_fft

    # size to hop through spectrogram window
    # eg. 160 for voxforgetest[1] and an4
    hop_length = int(self.sample_rate * self.window_stride)

    # STFT = computes discrete fourier transform, see
    # https://librosa.github.io/librosa/generated/librosa.core.stft.html
    # create an nxm sized array from y where n is the n_fft/2 - 1, is such that
    # (n-1)*(m-1) = len(y); n = 161 = hop_length + 1
```

```
    # 161 x 641
    D = librosa.stft(y, n_fft=n_fft, hop_length=hop_length,
                     win_length=win_length, window=self.window)

    # magphase = separate a spectorgram into magnitude and phase, see more at
    # https://librosa.github.io/librosa/generated/librosa.core.magphase.html
    # spect is n x m (161x641)
    # phase is n x m (161x641)
    spect, phase = librosa.magphase(D)
```

Figure 1 - Example of Code Commenting

Another contribution, besides helping to better understand exactly how the data loader turned the wav files into spectrograms that would serve as input to the model, was generating the training and testing set from the Voxforge data. I wrote a quick script that would sample 1000 files from the available dataset, create the required directories, and move the txt and wav files into training and testing folders that mimic the structure needed to run the transcription code.

```
wav_str = str(os.getcwd() + '/data/voxforge_dataset/wav/')
txt_str = str(os.getcwd() + '/data/voxforge_dataset/txt/')

for i in n:
    wav_name = vox_data.iloc[i, 0]
    wav_name_stripped = wav_name.replace(wav_str, '')

    txt_name = vox_data.iloc[i, 1]
    txt_name_stripped = txt_name.replace(txt_str, '')

    if len(lens) == int(len(n)/2):
        print('Training Files Created')

    if len(lens) < int(len(n)/2):

        copy(wav_name, train_dir + 'wav/' + wav_name_stripped)
        copy(txt_name, train_dir + 'txt/' + txt_name_stripped)

    else:
        copy(wav_name, test_dir + 'wav/' + wav_name_stripped)
        copy(txt_name, test_dir + 'txt/' + txt_name_stripped)

    lens.append(1)
```

Figure 2 – excerpt of code used to create training and testing sets

In addition to commenting and making the codebase more user-friendly, I developed code to allow us to quickly transcribe the test 500 audio files, calculate the WER and CER, and return the transcriptions and actual text to better understand the transcriptions the models are making. This was done using the my *transcribe_and_compare* function from the similarly named script, which would load all the audio and text files, load and put the model onto the GPU, use the model to predict the transcription, and use the decoder to decode the output. This was all done within a single function to reduce the amount of user-input needed and could be run in *verbose* mode to print the results of each file as the code progressed. All that is required of the user is the designation of a path to the wav files, path to the txt files, and path to the model to be used for decoding.

```
def transcribe_and_compare(wav_dir, txt_dir, model_dir, n_files=500, verbose=False):

        …

        # 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)

        …
```

Figure 3 - excerpt from transcribe_and_compare()

Additionally, I translated many of the utility scripts for easier use, including the *evaluate* function from the *test_stripped.py* file, all of the data loaders and spectrogram parsers from the

data_loader_stripped.py file, the *load_model* function from the *utils_stripped.py* file, and the *greedy_decoder* class from the *decoder_stripped.py* file, among others. This (hopefully) made the process of understanding how the Deep Speech 2 model class better works, as well as increased insight into the *train_stripped.py* file, which Tanaya primarily worked on.

Results

As I was not the one that directly trained the transfer-learning based LibriSpeech models, I will be discussing the cumulative results from the entire project, expanding on the results where appropriate.

The first experiment required evaluating the generalizability of the pretrained An4, LibriSpeech, and TED-LIUM DS2 models, and the results are summarized in the below table. The pre-trained models were evaluated on 500 randomly sampled Voxforge audio files (selected randomly from the entire audio collection), as this dataset was not seen by any of the models during the training process. Each of the models were compared using the Word Error Rate (WER) and Character Error Rate (CER) as the two evaluation criteria, as well as tracking the time to transcribe all 500 files.

Among the pre-trained models, the An4 data set had the maximum WER and CER of 10.05% and 30.80%, followed by the TED-LUIUM 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 TED-LIUM datasets by hundreds of hours, meaning the pretrained model is likely more robust due to the amount of data available during training epochs. Unsurprisingly the Google speech-to-text model had the smallest WER and CER of 1.61% and 3.73% (respectively). Of note, the pre-trained models took ~4 mins of evaluation time as these files were evaluated on a GPU; as the Google API is an external interface call, the backend processing and communication time may lead to the slower response time. Overall, as the LibriSpeech pretrained 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. The table below summarized the WER, CER, and evaluation time for these four models.

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

Figure 4 - Word Error Rate and Character Error Rate on pre-trained models

Starting with the LibriSpeech pretrained model, the first training iteration was executed using GRU neurons in the hidden layers, with a total of 5 hidden layers and 500 neurons in each recurrent 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%, which were an increase from the initial baseline of 2.27% and 4.28%.

We ran the second training iteration by changing the RNN neuron type to bi-directional LSTM and keeping the hidden layer, hidden size, number of epochs, and batch size the same as the previous training iteration. With these parameters, the WER and CER increased to 3.15% and 6.35%. Given this again was not performing as well as the initial pretrained model, we decided to increase the hidden layer size to 1000 neurons, keeping the rest of the parameters the same. Although the WER and CER were reduced in comparison to the previous training iteration, the results were still higher than the initial pretrained model score.

To further improve model performance, we increased the training size to ~7.5k Voxforge audio files. The training with the larger dataset took exceptionally long (~ 5 hours per epoch), so the results were evaluated after 3 epochs (~15 hours of training time). The CER (3%) was better than the Google API (3.73%), and there was only a 0.05% WER difference between our highly tuned model and Google API. This shows that the pre-trained model, when trained with more data that is similar to our evaluation set performs at par with the Google speech-to-text API. The below table summarizes the WER, CER, and evaluation time for the fine-tuned 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 - Word Error Rate and Character Error Rate on the fine tuned models

Summary and Conclusions

Based on the experiment outline, we first needed to understand the existing codebase and simplify it for ease of use. This was done by reviewing the existing code documentation, mapping dependencies between scripts, and rewriting classes and functions in order to make them more “user-friendly” (removing arguments, removing unused methods, commenting code with example output, etc.) Once the codebase was better understood, we, as a team, began analyzing the Deep Speech 2 model architecture, understanding the input-output structure, and writing our code for training and transcription. We additionally had to create our training and testing data sets for model evaluation.

Upon comparison of the three pretrained models, we found that the model trained on the LibriSpeech corpus performed the other two pretrained models, likely due to the larger volume of data available for training. This led us to select the LibriSpeech model for our transfer learning approach, where we tried to develop a model with comparable WERs and CERs to the Google speech-to-text API model. When first training the LibriSpeech model on the sample of 500 Voxforge audio files, we found this sample size to be too small to use to train a sufficiently well-performing model, and thus the number of files available for training was increased 15x.

This proved to be a much better approach, as the model with 1000 bi-directional LSTM neurons in each of the five hidden recurrent layers performed at a comparable rate to the Google speech-to-text API when trained for 15 hours on over 7500 Voxforge audio files.

From this project, I learned that (unsurprisingly) most of the modern neural network architectures are extremely complex, often blending differing types of network layers to produce the final model. I had little experience with working with audio data, as well as Recurrent Neural Networks, so to be able to analyze and adapt from some of the state-of-the-art models in the field today was a valuable experience, which absolutely could not have been done without the help of Anamay and Tanaya. However, our ultimate model is one that is likely highly tuned to the Voxforge dataset, so it would improve the project to train the model on different types of audio files and create a combined training set from multiple data sources. Additionally, acquiring further knowledge of parallel training techniques would be valuable in reducing the model training time, allowing us to try different architectures and experiment with different hyperparameters. Finally, getting more familiar with Beam-Decoding would be helpful, as that could potentially further improve our results by taking a more probabilistic transcription instead of simply the maximum value across each independent time window.

Code Comparison

Given that the basis of this project was adapting and expanding on an existing codebase, it is difficult to determine the exact quantity of open-source or internet-based code. It would be easy to say that 50% of the new code is original simply from comments alone, however the new *transcribe_and_compare.py* script will account for the majority of the "new" code I developed, as well as minor tweaks to the existing files like *transcribe_stripped.py* and *decoder_stripped.py*. If I had to make an estimate, I would guess that 50-60% of the code in this repository is from internet-sources if you include the original repository compiled by SeanNaren, however this is just an estimate and not likely to be the exact percentage.

References

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

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.

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