Anamay Agnihotri

Comparison of Voice-to-Text Neural Networks

DATS 6203: Individual Final Report

Introduction

With the increasing market of voice-to-text systems across industries, many tech giants are continuing to develop their own proprietary neural models that accomplish the complex task of speech recognition. Due to the confidential nature of many of these models, the objective of this final project was to explore and evaluate publicly available, open source speech-to-text neural architectures and use our deep learning knowledge to improve upon them for a specific domain. My group and I decided to use the DeepSpeech2 (DS2) speech-to-text system developed by Baidu, an AI organization as our open-source architecture, with the pytorch implementation of github user Sean Naren as a reference.

When my team and I began this project, I decided to take up the task of understanding the DS2 architecture while my teammates began with unpacking the github repository. As I delved deeper into the deep learning components of speech recognition, my focus throughout the project remained on bridging the gap between the original research paper and the pytorch implementation. Sinc the objective of our project was to evaluate model performances across the open-source and Google's proprietary implementations, I also took over developing the Google speech-to-text API code with a focus on ease of implementation, evaluation and code readability.

Description of individual work

Majority of my work focused on studying the DeepSpeech2 system and understanding its specific use cases for the scope of our project. The key contributions of the DS2 system (as explained in further detail in the group report) were:

1. **Model architecture:** Streamlined neural pipeline that eliminated the need for multi-component speech recognition models which made developing a new speech recognizer very hard, especially for a new language and did not generalize well across environments. The DS2 system, with its CTC-RNN structure and a unique approach to LSTM batch normalization proved to be very effective even in comparison to Amazon's Mechanical Turk human evaluations.
2. **Large training data:** Data augmentation techniques along with bootstrapping methods to increase the training data corpus. The large labeled training dataset with over 11,000

hours of audio data really sets the DS2 system apart in terms of robustness and evaluation performance.

3. **Computational scale:** The high performance computing tricks like parallelizing computations on GPU and dynamic memory allocation between GPUs and CPUs to ramp up training speeds.

Figure 1 below depicts the general architecture of the DS2 system. The system is made up of three 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.

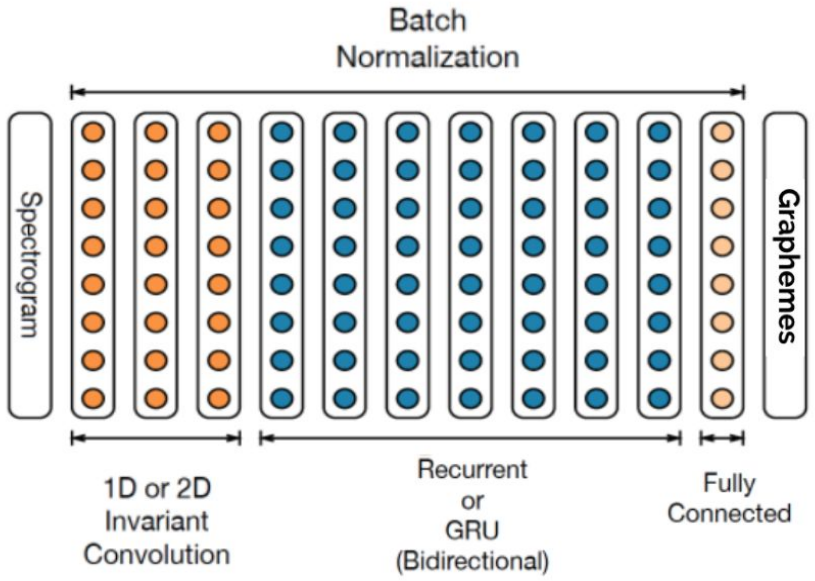


Figure 1 : DeepSpeech 2 architecture

The input to the network is a spectrogram, which is an image representation of the digital audio signals processed from the original analog audio files. Throughout the network, the clipped ReLU (hardtanh in *Pytorch*) is used as the hidden activation function with the formula: $\sigma(x) = \min\{\max\{x, 0\}, 20\}$. For the DS2 English model, the output layer consists of 29 nodes, with each node signifying a character (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 when used in conjunction with the CTC loss function. The fully connected layer uses a softmax activation function that outputs a probability distribution for the character output per time-step.

As mentioned earlier, DeepSpeech2 uses a CTC-RNN setup that allows for training it from scratch without the need of framewise alignments of the labels. CTC (Connectionist Temporal Classification) loss is the performance index used for the weight and bias updates of the DS2 system. 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. 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 negative log of this sum of alignments is defined as the CTC loss as encapsulated in the loss equation below:

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

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.

The DS2 network uses synchronous Stochastic Gradient Descent as the training algorithm for optimizing the CTC loss. The network also uses 2D convolutions to extract features along both the frequency and time axes of the spectrogram input. There are many other such tricks and nuances to the network engineering that I have covered in more depth in the group project report.

Beyond the DS2 framework, I also explored the Google Speech-to-Text API and built a load, request and evaluate script to fit in with the other evaluation scripts of the pre-trained and transfer-learned models.

Description of project contribution

The majority of my contribution to the project was to understand and communicate the technical aspects of the DeepSpeech2 architecture to the rest of the group. As mentioned earlier, bridging the gap between understanding the key components of the DS2 system and the github implementation to fit our project needs was challenging while getting started. As detailed in the final group report, the github code was rather complex with a lot of optional yet significant code components and my task was to sift through the code and identify the key components and think of design choices to improve upon the pre-trained models. For example:

1. Using bi-directional LSTMs instead of the GRUs based on the DS2 literature proved to be an effective approach in the transfer learning models.
2. Turning on the SortaGrad functionality to maintain numerical stability helped us get consistent results.
3. Maintaining the low striding in the convolutional layers for subsampling was inspired by the literature hypothesis on the speed of the English language utterance. Though our exploration into using bigrams in order to use larger strides did not quite work as well, such experiments and tunings helped us understand the architecture much better.

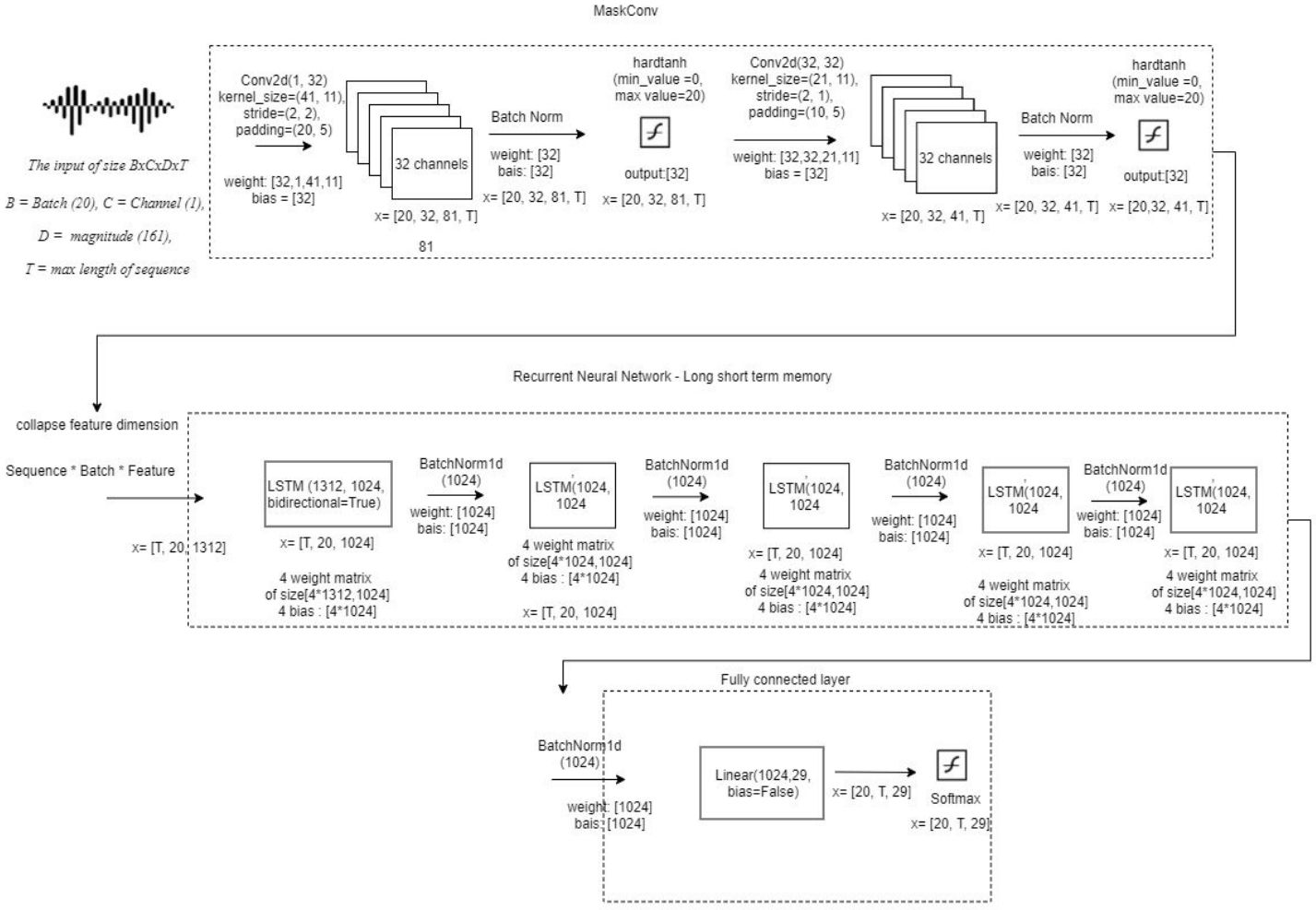
On the model evaluation side of the project, I enabled the speech-to-text API from the Google cloud console and obtained access credentials for my service account to make requests to the Google REST API. Beyond normal speech recognition, the speech-to-text API allows for a lot more complex tasks like multi-speaker identification, multi-channel transcriptions and automatic language detection. After configuring the API parameters according to the project requirements, I wrapped the API call inside a function. The experiment entailed transcribing and evaluating 500 test files from the Voxforge dataset and my Google API script (seen below) did exactly that. I initially wrote my own WER and CER scorers but since the team was using the github evaluation methods that used the Levinstein distance package, I reverted back to the package to maintain ease of readability and consistency throughout the project.

```
# Import packages
import io
import os
import numpy as np

os.system("sudo -H pip install --upgrade python-Levenshtein")
import Levenshtein
from tqdm import tqdm
from google.cloud import speech_v1  # python wrapper for google cloud API calls

# Assign google cloud API credentials variable to personal credentials file
os.environ[
    'GOOGLE_APPLICATION_CREDENTIALS'] = os.getcwd() + '/Machine-Learning-2-bdc5dbdf312d.json'  # add own credentials

# dummy API call to verify authorization
client = speech_v1.SpeechClient()


def sample_recognize(local_file_path):
    """
    Transcribe a short audio file using synchronous speech recognition

    :param
    local_file_path : Path to local audio file (works for flac and wav files)

    :returns
    Most probable transcription for given audio file
    """
    client = speech_v1.SpeechClient()

    language_code = "en-US"  # The language of the supplied audio

    sample_rate_hertz = 16000  # Sample rate in Hertz of the audio data sent
```

```
    # encoding = enums.RecognitionConfig.AudioEncoding.FLAC # add encoding type if not using flac or wav files

    config = {"language_code": language_code,
              "sample_rate_hertz": sample_rate_hertz,
              # "encoding" : encoding,
              "audio_channel_count": 1,
              }

    with io.open(local_file_path, "rb") as f:
        content = f.read()

    audio = {"content": content}
    response = client.recognize(config, audio)  # doing the transcription through Google Speech-to-Text API

    # some API responses return no result/transcription
    try:
        for result in response.results:
            # First alternative is the most probable result (highest confidence)
            alternative = result.alternatives[0]
        return alternative.transcript.upper()  # converting to upper for uniformity in evaluation
    except:
        return " "  # returns empty string if the API returns no transcription


path_txt = "./voxforge_sample_files/test/txt/"
path_wav = "./voxforge_sample_files/test/wav/"

texts = []
trans = []
WER = []
CER = []

for each in tqdm(os.listdir(path_txt)):
    # Read the original/gold-standard transcription
    with open(path_txt + each, "r") as file:
        text = file.read()
    texts.append(text)

    # Function call to obtain decoded transcription from API
    tran = sample_recognize(path_wav + each[:-4] + ".wav")
    trans.append(tran)

    # Calculate Character Error Rate (CER)
    cer = Levenshtein.distance(tran.replace(' ', ''), text.replace(' ', ''))
    CER = np.append(CER, cer)

    # Calculate Word Error Rate (WER)
    uniquewords = set(tran.split() + text.split())  # split output strings by spaces and create set of unique words
    word2char = dict(zip(uniquewords, range(len(uniquewords))))  # create dictionary of unique words

# map words to the word2char dictionary which is converted to character array (Levenshtein package only accepts strings)
    w1 = [chr(word2char[w]) for w in tran.split()]
    w2 = [chr(word2char[w]) for w in text.split()]

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

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

Results

As explained earlier, my personal contribution remained more on the theoretical and research side due to the nature of our project. Due to the circumstances surrounding the project, it was difficult for the whole team to sit together and work on generating results and doing the code-related experiments. Having taken charge of understanding the technical aspects of the project such as the input spectrogram creation, model architecture and its performance enhancing tricks and CTC loss implementation, my role was geared more towards guiding the implementation decisions than actually implementing them. As an example, below is the architecture diagram with input-output changes for the Librispeech pre-trained model that I assisted in understanding and deciphering. The details of the shape changes are outlined in the group report.



As outlined in the earlier section, there were many other educated guesses we made during our parameter tuning that were based on my understanding of the DS2 literature. Moreover, below is the model evaluations I created while making API calls to the Google Speech-to-text tool for the 500 Voxforge test audio files.

Model	WER(%)	CER(%)	Eval time
Google Speech-to-Text	1.61	3.73	~8 mins

On the test set, the API had a WER of 1.61% and a CER of 3.73%, which then became the target to chase for the remainder of our transfer learning experiments.

Summary and conclusions

In conclusion, I believe our project was able to accomplish the objective of creating a domain-specific automatic speech recognition pipeline from open-source architectures that is comparable to the proprietary framework of Google. Though not nearly as generalizable as the Google speech-to-text system, I used a lot of available literature on the subject to make educated decisions towards building/tuning a reasonable neural model.

I believe I was successfully able to decode, apply and communicate my technical understanding of the DeepSpeech2 system with my group and validated their explorations to obtain a well functioning and effective speech recognition model. I have a much deeper understanding of audio pre-processing for neural networks, recurrent neural network designs and implementations (GRU, bi/unidirectional LSTM) as well as dealing with aligned data problems with temporal loss functions like CTC.

In the future, I would like to implement some of the high performance computing optimization proposed in the original literature to efficiently train the tuned models over a larger number of epochs. I would also have liked to experiment further with the unidirectional LSTM approach in conjunction with the row convolution layer proposed in the original paper, as well as the bi-gram grapheme predictions in order to ramp up training through subsampling by increasing the stride lengths. I really enjoyed working on this project and not only learned a lot about interpreting and implementing an academic research paper, but also learned a lot about utilizing object oriented programming for complex problems by sifting through the production-level code of Sean Naren.

Percentage of code : 30% { [(40-10)/(40+60)]*100 }
To reiterate, I chose a more theory-driven role given the scope of this project and believe my contributions are not reflected in the code but in my research and related insights, as stated above and in the group report.

References

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

2. Scheidl, Harald. “An Intuitive Explanation of Connectionist Temporal Classification.” *Medium*, Towards Data Science, 4 Dec. 2018, towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c.
3. 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.