Digital Speech Processing

Final Term Project

Word Embedding in Speech

Author :
Ehsan Mahmoudi
97543012

Instructor:
Dr.Yasser Shekofteh

February 6, 2019

Contents

List of Figures

List of Tables

1 Abstract

This is the final project for the Digital Signal Processing course in Shahid Beheshti university. This project is indented to find a word embedding method in order to map the words to lower dimension spaces. In a way that the words that sound like each other, will be close. This embedding will help to resolve many ambiguities that will exist in normal speech recognition task. The idea is to convert each word to letter-grams that is the combination of letters in the word and finally take the words that sound similar in real world and find an embedding that have the sound similarity as a distance metric in the target space. This metric could be *Cosine Distance* or *Euclidian Distance*. We will be exploring this approach in this project. In this project because of technical challenges we did not followed the step by step method that is used in the main article. In the other hand we have came up with a novel approach for embedding words into speech perception space.

2 Introduction

The project initially was defined to be implementation of following article [1]. The article in general has following approach:

- Build an end-to-end neural network to perform speech to text
- Create a feature scheme called letter-n-grams to convert a word into a fixed sized vectors (long vectors)
- Train a neural network to map the letter-n-gram to the embedding layer of the main speech to text network

The idea is quiet straight forward we want to have a function from letter-n-grams to a dense representation of the sound of the word. This looks very interesting as it can help us to group the word that are sounding similar together. The is a useful function for performing search operation. Another interesting gain from this mapping is the fact that we can estimate on the words that are not in our speech corpus. This can give us a good advantage to work with larger vocabularies compared to our original speech corpus.

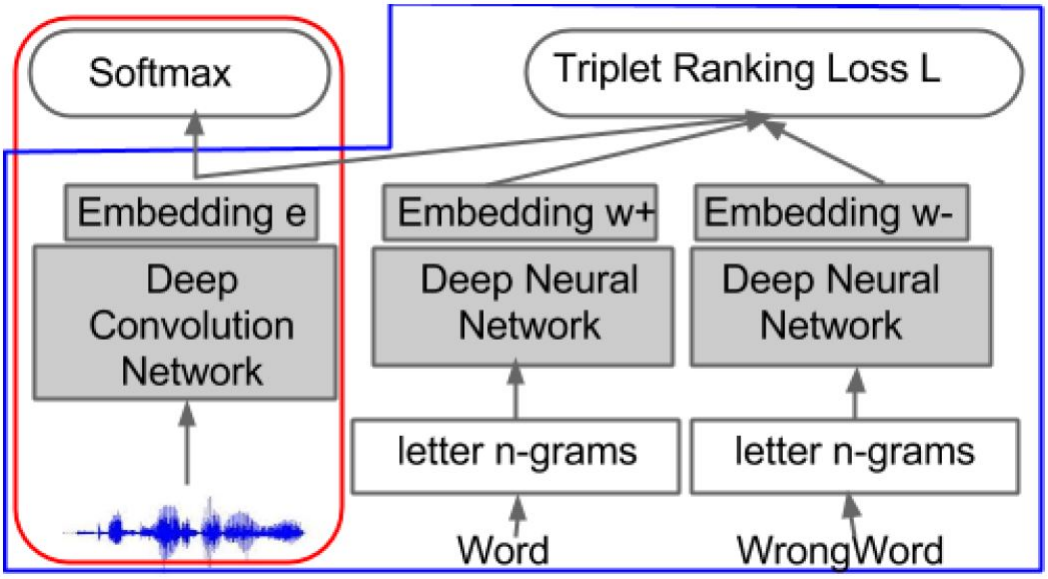

Figure 1: Main Approach in [1] - The image is from the original article

2.1 Implementation Challenge

Although the logic of the article was clear but there were some challenges that made it impossible to directly implement what was in the article. We will going through these challenges.

- Dataset Challenge
- Computational Challenge
- Some unspecified architectural parameters

We will go through all of these challenges one by one.

2.1.1 Dataset Challenge

The dataset that is mentioned in the article is not publicly available. This gives us a big challenge on the getting corpus that can be used. It is stated in the article that some sort of private speech corpus been used. Se we decided to use one of the open speech corpora available publicly.

We used the LibriSpeech [3]. This corpus is based on public domain audio books and is available publicly. We will provide detailed information about this corpus in later sections.

2.1.2 Computational Challenge

In the article the authors try to build an end-to-end neural network for speech processing. Training such model is computationally very expensive. It has 8 fully connected layers which make it very hard to train. From what paper talks about a week of training for such network on GPU heavy machine. Whereas for baseline models it requires 100 machines with a week of time. This was not something that is possible for us to handle within the time frame that we had. To overcome this challenge we were looking for a pre-trained model that already has done the complexity of training for us. We landed to use DeepSpeech model [2] which is an open model based on tensor flow. Both code and pretrained model is available publicly. The plan became to tap into the code and do what we want on it.

2.1.3 Unclear Architecture

The article is quiet clear about the architecture of the text-to-speech network. But unfortunately skips the important details about the layers, activation functions and connectivity of the layers. It only talks about the loss function of this network. This information is crucial for us to implement the model successfully.

2.2 Main Decision

After dealing with above challenges, I decided to take a new but similar approach to what [1] has proposed. The approach it keep the general idea of the article but implement it in a different way. In next section the proposed approach is being discussed.

3 Methodology

3.1 Main Objectives

Considering the challenges we mentioned on the previous section. We came up to the conclusion that in order to have a clear output we have to come up with a new approach while keeping the main points and ideas that are presented in the main article. So lets review what are the main values delivered in the article:

- Use letter-n-grams in order to build a feature set to predict the sound of a word
- Find a mapping from letter-n-gram feature space into a dense embedded space. In such space our assumption is that words with similar sound will stay close to each other
- Ability to map words that are not in speech corpus to embedding space

If we target all of above objectives, with slightly different approach then we can say we have delivered the project. The point is that this is the new potential for publication as we are incorporating a novel approach.

3.2 Overcoming Challenges

The hardest challenge to address in the project was to access the end-to-end speech recognition neural network that is described in the article. Two main challenges there, no data set and also the processing time for such big network is huge. So I made an executive decision to find alternative pre-trained models to enable us to focus on the embedding part of the task.

After looking at available systems I concluded to use Deep Speech system.

3.2.1 Deep Speech

Project DeepSpeech is an open source Speech-To-Text engine, using a model trained by machine learning techniques, based on [2]. Project DeepSpeech uses Google's TensorFlow project to make the implementation easier.

This network is a bidirectional RNN. The core of the system is a recurrent neural network (RNN) trained to ingest speech spectrograms and generate English text transcriptions. The goal of the RNN is to convert an input

sequence x into a sequence of character probabilities for the transcription y. The RNN model is composed of 5 layers of hidden units. Deep Speech has been implemented by Mozilla team as open package. It is available on Github.

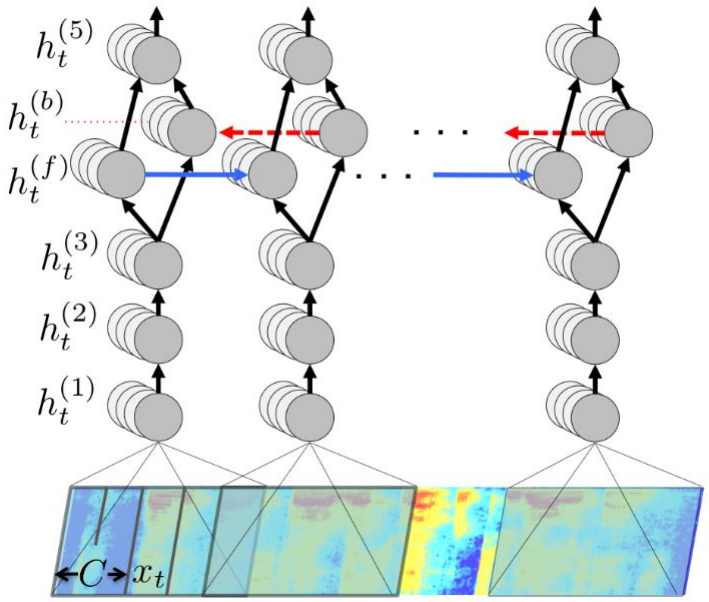

Figure 2: RNN Architecture

We are going to use this model to create the embedding layer of the project. This network is producing a series of probabilities of the alphabet. This includes space character and apostrophe. we are going to process the words and concatenate the probability vectors of words as the embedded version of the word that is spoken.

For each word then the concatenation of the probability vectors is going to make the embedding vector. Based on the size of the word in the input file we decided to make $N = 10$ as the number of concatenated vectors. So for each word N alphabet probability vectors are concatenated. If the length of the word was less than N then empty padding vectors were added to it to make all inputs having the same shape.

Figure 3: The simplistic view of the RNN

3.2.2 Dataset

I decided to use Librispeech dataset. This dataset contains over 1000 hours of speech corpus. It is organized in to multiple collections:

- Dev Clean
- Test Clean
- test Clean
- Train Clean 100
- Train Clean 360
- Train Clean 500

In this project we only used the Dev clean dataset. This data set contains 2800 audio files that each file is one or more sentence. Each file is transcripted. But transcription is in sentence level so the file is not tagged on word level.

Audio files were in FLAC format. So for being used by our library, we had to convert them into wav in addition to that the folder structure was a bit deep make it a little bit hard to work with. For this reason I write a conversion code to convert the file into WAV format and also put the information into one master excel-sheet for easier processing.

3.2.3 Creating Word Level Data

As we said the transcriptions of the files are on sentence level. From the other hand the RNN is producing the probability vectors on letter level. For purpose of this project we needed the probability vectors on word level. So we needed to write a code to give us the data we want.

A code was written to use RNN pretrained model available on Mozilla DeepSpeech project to process all audio files and produce the character level probability vectors. Then using the probability vector to generate the raw transcription of the files. As we only had access to sentence level transcription of the files to build the dataset we had to match between them. As the output of the RNN had lots of errors matching was challenging. After processing file, it seems that using Longest Common Sequence method is going to give us a good result. So LCS method was implemented to break down the word level data. After processing the files the data set resulted in 57000 word level data instances which were stored in CSV file format for further use.

3.3 Preparing the Word Letter-N-Grams

Lat piece of the input puzzle was the letter-n-gram input vectors. A code was written to create the index file for letter-n-grams to be used for both training and testing. One of the hyper parameters here was the maximum n to build n-grams. In this experiment we used $N = 4$ which resulted in 4737 features. This became the dimension of input for the embedding neural network.

3.3.1 Embedding Network

After data was prepared, we are ready to implement our embedding neural network. The embedding neural network was implemented usnig Tensorflow and Keras. The architecture is an auto-encoder with one hidden layer. The actiovation function of the thin hidden layer is RelU function. For test purpose we set the size of hidden layer unit to be $h = 300$. The output layer is actually a set of independent output Softmax layers for each character. The network trained with *AdamOptimizer* with 20 epochs.

4 Results

The embedding network training was straight forward and the network looked to have a normal errors drop rate. One of the interesting points about training was that the first output layer was the hardest one to train. The other $N - 1$ output layers error was very low initially but the first output layer error level dropped very late.

Original Input	Embedding Network Output	RNN Output
arcturus	arrctoris	arrctoris
steadfast	steaddastt	ssteeadfastt
in	in	an
the	the	a
skies	ssskiees	ssskkiees
with	witth	with
tardy	tarrt	tarrty
sense	sensse	sensse
guidest	guuiidest	guuiidest
thy	thi	by
kingdom	cingddoo	kinggddom
fair	fairr	farr
bearing	bearing	bearing
alone	alone	alone
the	the	theyh
load	lloadd	lloadd
of	of	of
liberty	llleerrtty	libeerrtty

Table 1: Sample outputs from RNN and Embedding Network

4.1 Auto-Encoder Performance

One of the aspects of the auto-encoder is that it supposed to generate the sound probabilities from the letter-n-grams. here are some sample output of the network. We are comparing the output of the RNN, Original word and output of embedding network.

In general it looks that embedding network output is something between what RNN produces and the original word. To assess this we have calculated the sum of Levenstein distance of the words. Here you can see the result:

RNN Output	Original Word	72029
RNN Output	Embedding Output	58986
Original Word	Embedding Output	59535

Table 2: Levenstein distance between outputs

Word	Neighbour 1	Neighbour 2	Neighbour 3	Neighbour 4	Neighbour 5
wheel	wheel	heel	wheels	while	when the
buy	but	by	but it	bye	be
heart	heart	hearth	hear	heart's	her at
please	please	pleases	pleased	pleasure	release
plug	plot	plumb	plan	play	plague
chareety	charity	charge	chat	cheatham	share

Table 3: Similar words in embedding space

4.2 Word Level Similarities

After embedding model is built, now it's the time to explore the similarities from this model. For this purpose, Cosine similarity measure is used.

5 Discussions and Conclusion

As mentioned before we are standing on a point that we just built end-to-end pipeline for voice word embedding. This could be the start of a research project. Following could be the next steps for the project:

1. Use this method instead of beam search after RNN output
2. Explore different hyper parameters on the embedding network and it's impacts: Dimension, Activation Functions, Number of Layers, ...
3. Explore voice analogy tasks with embedding
4. Use a larger data set from LibriSpeech (We used the DEV dataset)
5. Properly separate the train and test datasets

References

[1] Samy Bengio and Georg Heigold. Word embeddings for speech recognition. In *Proceedings of the 15th Conference of the International Speech Communication Association, Interspeech*, 2014.

[2] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[3] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *ICASSP*, pages 5206–5210. IEEE, 2015.

A Appendix

A.1 Convert Audio Files

```
#bin/import_librivox
for root, dirnames, filenames in os.walk(source_dir):
    for filename in fnmatch.filter(filenames, '*.trans.txt'):
        trans_filename = os.path.join(root, filename)
        with codecs.open(trans_filename, "r", "utf-8") as fin:
            for line in fin:
                # Parse each segment line
                first_space = line.find(" ")
                seqid, transcript = line[:first_space], line[
    first_space + 1:]

                # We need to do the encode-decode dance here
    because encode
                # returns a bytes() object on Python 3, and
    text_to_char_array
                # expects a string.
                transcript = unicodedata.normalize("NFKD",
    transcript) \
                    .encode("ascii", "ignore") \
                    .decode("ascii", "ignore")

                transcript = transcript.lower().strip()

                # Convert corresponding FLAC to a WAV
                flac_file = os.path.join(root, seqid + ".flac")
                wav_file = os.path.join(target_dir, seqid + ".
    wav")
                if not os.path.exists(wav_file):
                    Transformer().build(flac_file, wav_file)
                wav_filesize = os.path.getsize(wav_file)

                files.append((os.path.abspath(wav_file),
    wav_filesize, transcript))

return pandas.DataFrame(data=files, columns=["wav_filename", "
    wav_filesize", "transcript"])
```

Table 4: Audio Convert

```
#util/lcs.py
m = len(s1)
n = len(s2)
c = [[0] * (n + 1) for i in range(m + 1)]
bt = [[0] * (n + 1) for i in range(m + 1)]

for i in range(1, m):
    c[i][0] = 0
for j in range(1, n):
    c[0][j] = 0
for i in range(1, m + 1):
    for j in range(1, n + 1):
        c[i][j] = c[i][j - 1]
        bt[i][j] = -1
        if (c[i][j] < c[i - 1][j]):
            c[i][j] = c[i - 1][j]
            bt[i][j] = 1
        elif s1[i - 1] == s2[j - 1] and c[i][j] < c[i - 1][j -
    1] + 1:
            c[i][j] = c[i - 1][j - 1] + 1
            bt[i][j] = 0
matched_tuples = []
ind_i = m
ind_j = n

while ind_i > 0 and ind_j > 0:
    if (bt[ind_i][ind_j] == 0):
        matched_tuples.append((ind_i - 1, ind_j - 1))
        ind_i = ind_i - 1
        ind_j = ind_j - 1
    elif bt[ind_i][ind_j] == 1:
        ind_i = ind_i - 1
    else:
        ind_j = ind_j - 1

return list(reversed(matched_tuples))
```

Table 5: Longest Common Subsequence

```
#util/audio.py
def audiofile_to_input_vector(audio_filename, numcep, numcontext):

    # Load wav files
    fs, audio = wav.read(audio_filename)

    # Get mfcc coefficients
    features = mfcc(audio, samplerate=fs, numcep=numcep, winlen=0.032, winstep=0.02, winfunc=np.hamming)

    # Add empty initial and final contexts
    empty_context = np.zeros((numcontext, numcep), dtype=features.dtype)
    features = np.concatenate((empty_context, features, empty_context))

    return features
```

Table 6: Extracting MFCC features

```
def create_embedding_graph(input_vec_size, output_count, output_size, embedding_dim=300):
    inputs = tf.keras.Input(shape=(input_vec_size,))
    embedding = tf.keras.layers.Dense(embedding_dim, activation='relu')(inputs)
    outputs = []
    for i in range(0, output_count):
        outputs.append(tf.layers.Dense(output_size, activation='softmax')(embedding))
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer=tf.train.AdamOptimizer(0.01),
                  loss='mse',
                  metrics=['mse'])

    return model
```

Table 7: Build the Tensor Flow Embedding Graph