MAIS 202 – Project Deliverable 2

Xinyi Zhang

0. Discussion with other people about the project

In general, I consulted my significant other/roommate, Félix Boulet, during this project. Félix is a recent graduate of Polytechnique who is passionate about programming. I am grateful that he still volunteered his time to help me (from answering admittedly basic but crucial questions to showing me the way to find answers on my own, from listening to me rambling about how something is making no sense to keeping me motivated) even with his full-time job.

1. Problem statement

The goal of the project is to train a convolutional neural network to recognize speech from speakers with dysarthria. Specifically, for the scope of this project, the speech of interest is simple commands (e.g., directions, numbers, yes/no).

2. Data preprocessing

As mentioned in D1, I used the TORGO database [1] , a database developed by University of Toronto. The steps described in D1 were executed to preprocess the raw data. To be specific, by searching through the prompts (orthographic transcription of the recorded speech), the commands were located. Then, using the label of the prompts, the WAV files and the phonetic transcriptions were fetched and matched. After, using the phonetic transcriptions, the silence/noise parts in the recordings were cut out and standardized into 1-second-long waveforms (using zero padding if needed). Then, the waveforms were transformed into spectrograms using a scaled Fast Fourier Transform (SFFT). The SSFT scales the result to make it into a smaller image (using a "step" value). The steps were chosen to give a small, approximately square image (124 x 129), as it is easier to process in a CNN (faster training while keeping a good amount of detail).

[1] http://www.cs.toronto.edu/~complingweb/data/TORGO/torgo.html

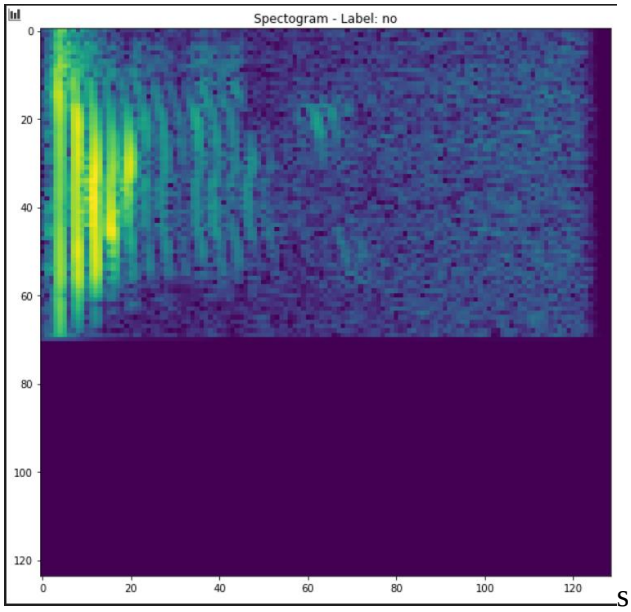

Figure 1: The spectrogram for the word "no", zero-padded to be one second long

The biggest problem was the small size of the dataset. Effectively, after filtering for the words of interest, it was found out that data on the simple commands only were a small proportion. Moreover, as it turned out, there were also many files that were corrupted, mislabeled, and wrongly time-stamped. In the end, there were only **91** usable inputs (a processed WAV file with a matching label) covering **17** different commands from dysarthria patients. Admittedly, the number of data points is just sad (and completely beyond my expectation). Thus, I added some more data from the control groups provided in the TORGO database. However, that only brought the number of data points to 176 (since there were also mislabeled and corrupted files). At first, I thought this could be sufficient to obtain at least minimally satisfying results, but I progressively realized that it is unrealistic. It was especially dreading to witness the lowering amount of valid data as the preprocessing was applied. Debugging also revealed unsatisfactory data that was initially causing issues and then cut out from the usable dataset. These issues will be addressed in the "Next steps" section.

3. Machine learning model

Initially, the TensorFlow framework was chosen to construct the ML dataset and model, based on a tutorial provided by TF[2]. After a lot of struggling and wasted time trying to transform my custom input files into a TF dataset (and reading undecipherable errors from the depths of the library), I gave up and chose to use a combination of SciKit-Learn (often referred to as SKLearn) and Keras. This combination of libraries was inspired by a Kaggle submission[3], where it seemed to be simple and easy to use. SKLearn allowed me to transform my spectrograms and label pairs to inputs with categorically encoded (one hot) outputs. That data was then split into training and validation sets with a ratio of 80-20. This ratio was chosen rather arbitrarily, but wouldn't have a big influence since the amount of data is so low (80% of 176 inputs is just 141). In any case, it is very possible that some categories

[2] https://www.tensorflow.org/tutorials/audio/simple_audio

[3] https://www.kaggle.com/harunshimanto/speech-classification-using-cnn

(commands) will never be seen by the network in the training set, or in very few instances. This seemed to be the case during training, as I will explain below.

The model was implemented using Keras layers Conv2D, MaxPooling2D, Dropout, Flatten and Dense.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 124, 129, 1)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 124, 129, 2)       52
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 62, 64, 2)         0
_________________________________________________________________
dropout (Dropout)            (None, 62, 64, 2)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 62, 64, 4)         76
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 32, 4)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 31, 32, 4)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 31, 32, 8)         40
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 16, 8)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 15, 16, 8)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 16, 16)        144
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 8, 16)          0
_________________________________________________________________
dropout_3 (Dropout)          (None, 7, 8, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 896)               0
_________________________________________________________________
dense (Dense)                (None, 512)               459264
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 17)                2193
=================================================================
```

Figure 2: The network architecture for the CNN

Conv2D layers are the “CNN” part of the network, as they perform 2D convolutions on the input data (here, the spectrograms). They start with small “filters” to first identify small, specific features from each spectrogram, and then get progressively larger to gather bigger features (e.g., the general shape of the amplitudes of frequencies over time represented in the spectrogram). The output of each convolution layer gets “max-pooled” to reduce its size for the next layer, keeping the maximum value in blocks of 2x2 “pixels”. The output of those pooling layers is then processed through a dropout layer with a proportion of 0.1 (10% of values are ignored for the input of the next layer), in order to avoid overfitting (which were tested to happen when dropout layers were not present). Finally, after the convolution-pooling-dropout layers, the output is flattened into a single dimension (it doesn’t make much sense to keep it into a 2D format at that point) that is then fed to two “dense” layers (which are the “typical” fully-connected neural network layers) of 512 and 128 neurons each. The last layer is the

actual classification layer (using a “Softmax” function) that has 17 neurons for our 17 output categories.

Sadly, even though most of this seems to make sense (as it follows general tips for creating a good CNN), the end results were not good at all; see below.

4. Preliminary results

As mentioned above, the results obtained from this first iteration of the dataset and model were pretty terrible.

Typical training accuracy hovered around 20-25%, and validation (test) accuracy was even lower at 10-15%. Changing layer count, hyper-parameters (filter size, pooling size, batch size...) and other network attributes (optimizer, number of epochs...) didn’t change much to that result. Actually, when manually testing predictions with randomly sampled data from the validation set, it seemed that the network only learned a few of the labels (generally, one or two at most). For example, it would give “yes” or “no” as a prediction for any input (even acoustically very different ones such as “seven”). This seems to indicate that the input data might not be evenly distributed (i.e., more occurrences of certain labels such as “yes”). In addition to those issues, during one of the training phases, the network achieved a 100% accuracy on the training set and got progressively worse validation loss and accuracy. This indicates that the network is prone to overfitting, and that is why dropout layers were added afterwards.

The problems were not able to be resolved for this deliverable. I believe that most of them stem from the input dataset itself, and not the model (which should be able to achieve passable accuracy without much tuning).

Here is an example of the evaluation metric (confusion matrix) for a case where the network only learned “no”:

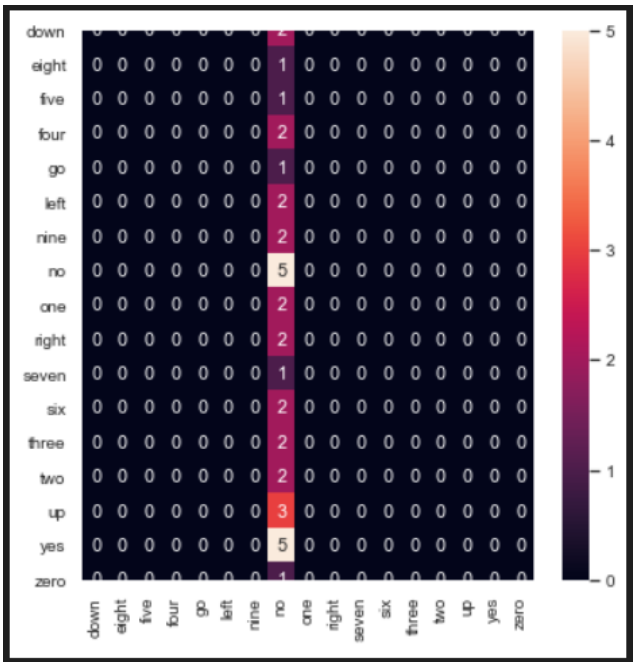

Figure 3: The confusion matrix for the validation set

With the confusion matrix, it is easy to see that the network seemingly learns only a single possible output in this case. In addition, we can observe that the true labels are far from being uniformly distributed in the validation set, which probably indicates that the same kind of issue plagues the training set, as theorized.

At this point, changes must be made to both the dataset and the model. Otherwise, the project is simply not feasible. Possible improvements are discussed below.

5. Next steps

The approach using the SFFT on the waveforms seems to make sense in order to use a convolutional network. It gives more dimensionality than simply analyzing the 1-dimensional waveform. However, some similar implementations use 1D convolutions directly on the waveforms and get pretty reasonable results (such as 80% accuracy on the TensorFlow speech dataset); this could be explored, but as someone who studies linguistics, my intuition tells me that the spectrogram is closer to how humans recognize speech.

The next step would be to try transfer learning, as suggested by my TPM, William. This would give me the opportunity to start from a model that is already trained on a lot of generic speech data, and might already have a better chance at recognizing the patterns present in the speech input from dysarthria patients. Most of the simple commands can still be deduced by a trained human from the spectrograms (i.e., me), so I don't see why a well-trained generic network could not do the same.

Another thing that will need to be done in regards to the dataset is to make the distribution of each label uniform. To do so, uncommon labels will be "augmented" through random noise addition and audio cuts, as suggested in data augmentation tutorials from TensorFlow[4].

[4] https://www.tensorflow.org/io/tutorials/audio