A neural attention model for speech command recognition.

E4040.2020Fall.TPIA.report

Huiyan Xing hx2302, Haikang Tan ht2545, Jiawen Yan jy3088

Columbia University

Abstract

Speech command recognition is widely used in multiple novel technologies and online applications. This paper is aiming to reproduce a convolutional recurrent neural network with attention for speech command recognition [1]. By adding data augmentation and utilizing various parameter adjusting functions, the best accuracy we can achieve is around 95.37%, which is 1.47% higher than the origin results [1].

1. Introduction

Speech command recognition has always been a fascinating field, which is notably utilized in companies like Google and Baidu. In our project, we realize a convolutional recurrent network with attention for speech command recognition that was proposed in [1], along with some modifications and improvements, including data augmentation before converting the audio data, four models with different architectures, and a self-defined mel spectrogram algorithm. The goal of this project is trying to reproduce the results in [1], and hopefully achieving a better accuracy model. The dataset we used is google speech command dataset version 2 with task '35 words'.

This model is an attention-based model, meaning that it can detect the part of the audio which needs to be considered into the model. Moreover, it is a light weight that could be loaded into local devices.

2. Summary of the Original Paper

2.1 Methodology of the Original Paper

The paper introduces a convolutional recurrent network with attention for speech command recognition. Attention models are powerful tools to improve performance on natural language, image captioning and speech tasks.

The database they used is Google Speech Commands dataset which includes version 1 and version 2. The inputs to the models are raw WAV files and then were converted to numpy arrays for efficient load with Keras generators which is no longer available now.

The network had 17 hidden layers (input layer not included) as well as one layer whose input was extracted from one of the hidden layers.

Before the training model, there were two layers which started by computing the mel-scale spectrogram of the audio using non trainable layers implemented in the kapre library. After mel-scale spectrogram computation, a set of convolutions is applied to the mel-scale spectrogram in the time dimension to extract local relations in the audio file. A set of two bidirectional LSTM units is used to capture the long term dependencies in the audio file. one of the output vectors of the last LSTM layer is extracted, projected using a dense layer and used as a query vector to identify what part of the audio is the most relevant. To achieve this goal, a dot layer was used between the query vector and the output of the last LSTM layer. After a pre-process of the output from the dot layer, another dot layer was conducted together with the same output from the last LSTM layer to get the weighted average. Then it was fed into 3 fully connected layers for classification. The number of neurons of convolutional layers, LSTM layers and dense layers are [10, 1, 128, 128, 128, 64, 32, number of classes] in sequence. The paper used convolutional kernels of size [5×1] in each of their convolutional layers and the parameter "padding" equals to "same". The activation of the last Dense layer was softmax. The activation of the 64 and 32 dense classification layers was ReLU.

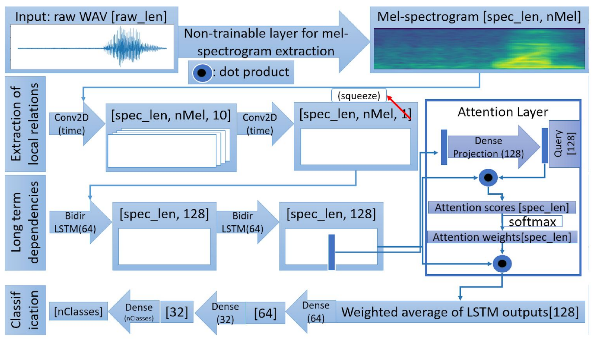


Figure 1 [1] Proposed architecture: recurrent neural network with attention mechanism. Numbers between [brackets] are tensor dimensions, raw_len is WAV audio length (16000 in this case), spec_len is the sequence length of the generated mel-scale spectrogram, nMel is the number of mel bands, nClasses is the number of desired classes.

Three optimized operations were also added into the model. The first one was changing the learning rate. Training was done with an initial learning rate of 0.001 and decay of 0.4 every 15 epochs. Early stopping and Check point were added in order to optimize the training process. What's more, attention plots and confusion matrices were also presented to allow for better comparison of the results.

2.2 Key Results of the Original Paper

The model had achieved an accuracy of 94.1% on Google Speech Commands dataset V1 and 94.5% on V2 (for the 20-commands recognition task) – significantly better than the 20-cmd baseline of 88.2% from Warden (2018), while still keeping a small footprint of only 202K trainable parameters. The paper demonstrated that the proposed attention mechanism not only improves performance but also allows inspecting what regions of the audio were taken into consideration by the network when outputting a given category.

To summarize, when applied to the tasks: 20-cmd, 12-cmd, 35-word and left-right, the accuracies are respectively 94.1%, 95.6%, 93.9% and 99.2% on the V1 dataset and 94.5%, 96.9%, 93.9% and 99.4% on the V2 dataset.

3. Methodology

Objectives and challenges were introduced in 3.1. Then in 3.2, approaches to processing the dataset and model construction were included in different sub-parts

3.1. Objectives and Technical Challenges

The objective was divided into a few parts:

1. Implementing all functions needed, which included data downloading, data processing, model constructing, parameters adjusting and results analyzing.

2. Testing models with different parameters to figure out the influence and trying to achieve greater accuracy.

3.2.Problem Formulation and Design Description

3.2.1 Model Construction

Since the existing model was complex, it would be hard to change the model in order to achieve better performance. With this concern, we chose to modify the parameters of the model to figure out their influences.

The biggest challenges would be correctly building the model with appropriate parameters, figuring out changeable parameters and adjusting them using curract functions.

4. Implementation

Detailed data pre-processing methodology is given in section 4.1. Four implemented deep learning structures explanations and comparison are given in section 4.2.

4.1 Data Pre-processing

Before processing data into the convolution recurrent network, a couple of tasks should be done. Firstly, audio data should be downloaded from the google speech command dataset with corresponding categories. These naive audio files should be then categorized into four different sets: training set, validation set, testing set, and real testing set. The part above is done in *SpeechDownloader*. Secondly, audio data should be converted into numpy arrays, along with selective data augmentation. This part above is achieved in *AudioGen*, which is a generator that successively generates audio numpy data within each epoch during the training. The generator has the following functions:

- __len__: calculate the overall batch numbers within each epoch
- on_epoch_end: update the index after each epoch due to shuffling.
- __getitem__: generate a batch of data by using the __data_generation function below
- __data_generation: generate data containing samples of batch_size numbers.

Data augmentation is a commonly used tool to help increase model accuracy in visual tasks. However, it is not a common method for researchers to deal with audio data. We implemented four different ways of data augmentation and explored whether it could increase the predicted accuracy or enhanced the model's generalization ability.
The four methods we used in data augmentation are:

- Noise Injection: Add random value into data
- Time Shifting: Shift the audio to right/left by certain seconds
- Changing Pitch
- Changing Speed

Each of them targets the original audio data in the form of 2D array instead of the Mel-spectrogram form that we used for later training. The main reason is that we could more easily observe what changes had been made to the original data. And it is important to not manipulate the data “too much” and backfire.

The last data pre-processing, which is actually embedded with the network, is mel-spectrogram calculation of the audio signal. The mel-spectrogram is a tool to visualize how mel-scale frequency changes over time. It is useful to choose the attention of the signal. In our implementation, we simply use the *spela* package which uses tensorflow as a basis to provide layers that calculate the mel-spectrogram of given audio arrays. We also wrote a python function which uses only numpy functions to calculate the mel-spectrogram. However, we are not able to find the method to embed the python function into model layers. As a result, we are listing it as future work.

4.2. Deep Learning Network

4.2.1 Dataset

As mentioned in section 2.2, the dataset original paper used included a relatively huge amount of different parts. It would cost an immense amount of time to train the models with all datasets it provided. Thus, version 2 with 35 classes was chosen because the accuracy it reached was slightly higher than version 1 and it has the biggest number of classes which means the possibility of the application of the trained model would be the greatest.

4.2.2 Model Construction

The flowchart of the model part is shown in figure 2. In order to build the model, two packages were used. The first one is spela which was used to build two untrainable layers to pre-process the data. Then, trainable hidden layers were added using the API provided by keras. The architecture of the model is shown in figure 1.

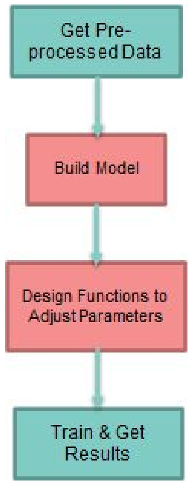


Figure 2 Flowchart of Model Construction & Training

4.3. Software Design

4.3.1 Model Construction

In order to build the model, a python file (speech_model.py) was created which contained one function named att_rnn_model. The input parameters needed were:

- number of classes
- sampling rate (provided by google speech command dataset)

- input length (equaled to sampling rate)
- keras model LSTM (needed for Bidirectional layer)

The design of the model is shown below:

```
create input layer
reshape input layer
add Melspectrogram layer, define it as untrainable
add Normalization 2D layer
add Permute layer, switch the output dimension
add Conv2D layer and BatchNormalization layer × 2
add Lambda layer to squeeze the last dimension
add Bidirectional layer (uses LSTM layer) × 2 → BL
add Lambda layer to squeeze the first dimension (in order to extract the output from previous layer) → x_part
add Dense layer to x_part → query
add Dot layer to calculate query dot x_part
add Softmax layer to get the weights → att_weights
add Dot layer to calculate att_weightsdot BL
add Dense layer × 3
```

4.3.2 Adjust Parameters

In order to get the influences of different parameters, adjusting training parameters would be a good approach. The python file named *adjust_utils.py* was implemented to realize this design which includes four functions:

- power_decay_1:
```
learning rate = 0.001 * 0.4^((1 + epoch) / 15)
if learning rate < 4e-5:
    reset learning rate = 4e-5
```
- power_decay_2:
```
learning rate = 0.001 * (1 - epoch / 15)^0.9
if learning rate < 4e-5:
    reset learning rate = 4e-5
```
- exp_decay:
```
learning rate = (0.001^0.9)^(epoch + 1)
if learning rate < 4e-5:
    reset learning rate = 4e-5
```
- multi_decay:
```
uses ReduceLROnPlateau
if val_sparse_categorical_accuracy does not improve in 10 epochs:
    learning rate * 0.1
```

Beside the functions, four APIs from keras.callbacks were also imported to optimize the process of training:

- EarlyStopping: stops training when a specified parameter does not changing any more
- ModelCheckpoint: stores the model after one epoch of training
- LearningRateScheduler: makes learning rate visible
- ReduceLROnPlateau: reduces learning rate when the specific evaluation stop changing

First Model:

For the first Model whose model was saved as "*model_attRNN_p1.h5*", function power_decay_1 was used to reduce learning rate after each 15 epochs. At the same time, EarlyStopping and ModelCheckpoint were also added so that the valid sparse categorical accuracy would not decay during the process of training.

For EarlyStopping, the monitor was set to be 'val_sparse_categorical_accuracy' and the model would stop training if the monitor did not increase in 10 epochs.

Second Model:

For the second model whose model was saved as "*model_attRNN_p2.h5*", function power_decay_2 was used, as well as EarlyStopping and ModelCheckpoint. In this model, learning rate decayed faster than the first model. The parameters of EarlyStopping and ModelCheckpoint were the same as the first model.

Third Model:

For the third model whose model was saved as "*model_attRNN_e.h5*", function exp_decay was used, EarlyStopping and ModelCheckpoint still had the same parameters.

Fourth Model:

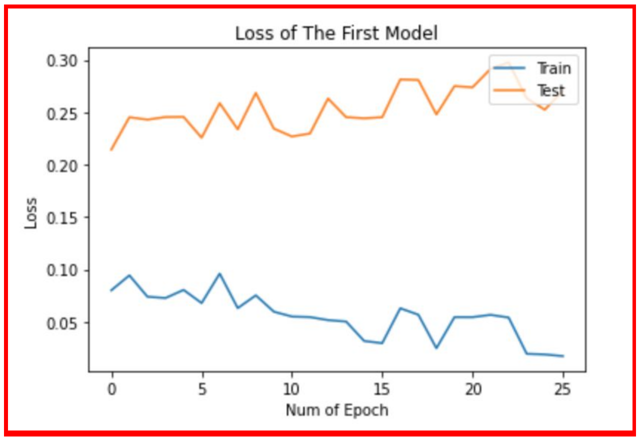
For the fourth model whose model was saved as "*model_attRNN_m.h5*", function multi_decay was used, as well as EarlyStopping and ModelCheckpoint. For EarlyStopping, the parameter "patience" equaled 5. Because during the process to previous training, if the learning rate would change, the duration would not exceed 4 epochs. For ReduceLROnPlateau,

5. Results

5.1. Project Results

Our final accuracies are based on the models performance on the test dataset. We tested four different models (denoted as **model_p1**, **model_p2**, **model_e** and **model_m**) with different learning rate decaying methods.

We reached the highest accuracy of **95.37%** from the [**model_p1**], which use the power_decay_1 method when updating the learning rate. Our second best performance model [**model_e**] achieves an accuracy of 94.0%. Figure 3 show the result of [model_p1].



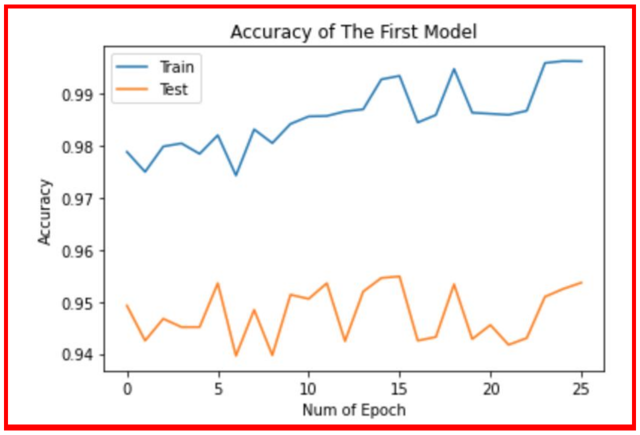


Figure 3,4 Training and Validation Loss/Accuracy History of [**model_p1**]

The training time of four of our models vary from 3.5 hours to 9 hours. We used the TensorFlow 2.0 package and GCP with an NVIDIA Tesla K80.

5.2. Comparison of the Results Between the Original Paper and Students' Project

The paper achieved an accuracy of 93.9% using Google Speech Command Dataset V2 in the 35 word task.

Table 1: Accuracy results on testing dataset, table shows the testing accuracy of Attention RNN (model in original paper) and four of our models.

Model	**Testing Accuracy (%)**
Attention RNN (Original paper)	93.9
model_p1 (Our)	**95.37**
model_p2 (Our)	93.7
model_e (Our)	94.0
model_m (Our)	93.1

Table 2: Training time/Training accuracy/Validation results of Attention RNN (model in original paper) and four of our models.

Model	**Training Time (Each epoch)**	**Training Accuracy (%)**	**Validation Accuracy (%)**
Attention RNN	180s	/	/
model_p1 (Our)	510s	99.2	95.5
model_p2 (Our)	1200s	98.4	93.6
model_e (Our)	900s	98.0	95.0
model_m (Our)	1000s	96.5	94.0

5.3. Discussion of Insights Gained

In our work, we first implemented Data Augmentation using different four methods mentioned in part 3. We obtained approximately 20% more data than the original dataset which may help improve our model's generalization ability. We also tested four different models with different learning rate decaying methods. We achieved our best test accuracy of 95.37% using the power-decay method when updating the learning rate. Our test accuracy is 1.47% higher than the accuracy in the original paper! We believe that both the use of Data Augmentation and different learning rate decaying methods contributes to our better predicting accuracy.

Due to the limit of computing power, we only augmented 20% of original data for our training. We think it might improve the performance more if we augmented a larger proportion of the datasets.

6. Conclusion

Speech Command recognition is widely used worldwide. In our project, we implement four convolutional recurrent neural networks with different decay functions. The dataset we are using is Google speech command dataset version, task 35 words. Although a python implementation of mel-spectrogram calculation is included and it functions well in the python environment, we are not able to embed it into the model layer of Tensorflow. As a substitution, we are using the mel-spectrogram layer in spela package to calculate the mel-spectrogram of audio arrays. By using data augmentation and various learning decay algorithms, we managed to achieve a better result than the original paper did. The best model we can get is an accuracy of 95.37%, which is 1.47% higher than the best accuracy achieved in [1]. This model also comes with an attention plot, in which the significance of the signal is considered through time. Confusion matrices are also shown for completeness.

7. Acknowledgement

We draw on the instructions on the repository of A Keras Implementation of neural attention model for speech command recognition to complete this project.

8. References

[1] Douglas C.A., Sabato L., Martin L., et al., A neural attention model for speech command recognition[J], 2018.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, https://www.deeplearningbook.org/

9. Appendix

9.1 Individual Student Contributions in Fractions

	ht2545	hx2302	jy3088
Last Name	Tan	Xing	Yan
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	Mel-spectrogram python implementation	Model Construction	Data Augmentation implementation
What I did 2	data-preprocessing	Parameter Adjusting	Data Preparation
What I did 3	Report abstract, 1,part of 4 and 6	Report (Section 2, Section 3, Section 4.2 & 4.3)	Report (Part of 4.1 and Part 5.Result)
What I did 4	Model Testing (model_p1)	Model Testing (model_p2, moedl_m)	Model Testing (model_e)