Speaker Diarization Project

ECE/CS 6930 Neural network

Zekun Zhang A01768067

Vishal Sharma A01789836

Zac Neubert

# Overview

**Speaker diarisation** is the process of partitioning an input audio stream into homogeneous segments according to the speaker identity.  It is used to answer the question "who spoke when?"[Wiki] In this project, we implemented speaker diarization by using different types of neural networks. Specifically, multilayer perceptron (MLP), recurrent neural network (RNN) and convolutional neural network (CNN) are all used to solve this task. Among all the networks we tried, RNN achieves the highest accuracy which is ~%92. CNN achieves similar accuracy.

Introduction

In this section, we introduce what the original problem is, how we formulate it into a neural network problem, and what criteria we use to evaluate the performance of neural network. The benchmark is also provided given our evaluation criteria.

# Original Problem

The provided data set is a collection of audio files (37 “.wav” files) along with the corresponding labels (“.csv” files) indicating the time period of each speech from each speaker. The task is to train a neural network which can generate the diarization results automatically given an audio file.

The provided audio files are “.wav” files with sample rate 44100. The content of each audio file is a conversation between speakers. Each audio file has two channels, one for each speaker. More specifically, in each audio file, channel\_1 is recorded by the microphone in front of speaker 1 and channel\_2 is recorded by the microphone in front of speaker 2.

The provided labels are “.csv” files. There are multiple types of “.csv” files provided, the one we used throughout this project is like below

tmin tier text tmax

0 CH2 N 1.361079

0 CH1 N 4.550097

1.361079 CH2 S 4.996529

4.550097 CH1 S 5.541915

4.996529 CH2 N 5.547973

5.541915 CH1 N 8.183008

# Formulated Problem

We formulate this task to a supervised classification problem. Generally, we divide each audio file into small segments on time domain with fixed segment size and use trained neural network to classify whether a segment belongs to the speech of a speaker. The segment size is a parameter and can be changed to any number in our code. If the segment size is too small, then the samples within a segment may be not enough to do the classification and hence the performance is poor. If the segment size is too big, then the precision of our final output is low compared to the provided labels. **Throughout this project, we set segment size as 0.1 second to achieve a good performance as well as high precision.** The original audio file is segmented as the graph shown below. For example, the length of an audio file is 10 seconds and we set the segment size as 0.1 second. The length of each channel is also 10 s. Then each channel can be divided into 100 segments. As the sample rate of provided audio file is 44100, each single segment contains 4410 samples.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Channel\_1 | | | | |
| Segment\_1 | Segment\_2 | Segment\_3 | Segment\_4 | … |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Channel\_1 | | | | |
| Segment\_1 | Segment\_2 | Segment\_3 | Segment\_4 | … |

Given a segment, we predict the class of this segment. There are two types of classifications we tried in this project as below

1. 2 classes:

0: Non-speech (including noise, speech of another speaker, et al)

1: Speech of the speaker of the current channel

Detail: this type of classification can only be used when treating each channel of an audio file separately. For example, we extract two channels, channel\_1 and channel\_2, from an audio file. Then for a segment of channel\_1, it belongs to class 1 if this segment is part of the speech of speaker 1, and belongs to class 0 otherwise. For a segment of channel\_2, class 1 means this segment is part of the speech of speaker 2 the class 0 means others.

1. 4 classes:

0: Non-speech (including noise, speech of another speaker, et al)

1: Speech of speaker 1

2: Speech of speaker 2

3: Overlap of two speakers

Detail: This classification can be used for either treating two channels separately or considering them together. However, in practice this kind of classification does not perform as well as the 2 classes method. **In this project, we mainly use 2 classes method and will briefly introduce how to implement 4 classes method.**

# Evaluation

The performance of the network is evaluated by the percentage of correctly classified segments. For example, an audio file in the test set has 100 segments and out trained network can correctly classify 90 out of them, then the accuracy of our classification on this file is %90. We use the average accuracy of all the audio files in the test set as the final accuracy of our network. So if there are 3 files in the test set, and the accuracy provided by one network on each file is %88, %90 and %92, then we say the accuracy of this network is %90.

# Benchmark

By analyzing the provided label files (“.csv” files), we find that basically the speech time of a speaker is about %40. So if we always guess the majority (non-speech), the benchmark of accuracy is about %60 for 2 classes method. For 4 classes method, the benchmark is about %30.

Methodology

In this section we first descripted how we pre-processed data and labels. Then we elaborate all the networks we tried to solve this classification problem.

# Pre-processing

Data Normalization

Normalized audio files are generated by normalizing all original audio files to the same average volume (-20 dBFS). Normalization is implemented by code in “Normalize\_Audio.py”. At the beginning of this project, we didn’t implement normalization cause we thought all files are at similar level. Later we found that the accuracy can achieve %90 on file 1-file 30, however, can only get %60 on file 31-file 37. By listening them, we found file 31-file 37 are not as loud as other files. After normalization this problem is solved and all files can achieve approximately the same accuracy.

Load Audio Data

“.wav” file is loaded by using the “get\_data” function written in “Load\_Audio\_Data.py” which is attached at the end of the report. Specifically “get\_data” function read the “.wav” file located at certain directory and return a tuple (channel\_1\_matrix, channel\_2\_matrix) where each element is a matrix contains the samples of a channel. Each row of a matrix is the samples within a segment. The shape of each matrix is

(number\_of\_segments\_each\_channel, number\_of\_samples\_each\_segment).

For example, if the length of a “.wav” file is 10 seconds, then two matrixes in the returned tuple is as below. Recall that the segment size is 0.1 second, so we have 100 segments hence there are 100 rows in each matrix. Sample rate is 44100 which means there are 44100 samples in one second and 4410 samples in each segment. In the graph below we use index 1.

|  |  |
| --- | --- |
| channel\_1\_matrix | channel\_2\_matrix |
| Segment 1 (sample 1-4410) | Segment 1 (sample 1-4410) |
| Segment 2 (sample 4411-8820) | Segment 2 (sample 4411-8820) |
| Segment 3 (sample 8821-13230) | Segment 3 (sample 8821-13230) |
| … | … |
| Segment 100 (sample 436591-441000) | Segment 100 (sample 436591-441000) |

“get\_data” function can also down sample the original data. For example, if set argument “down\_sample = true” and “down\_sample\_rate =4”, then it only retain the 1st sample of every 4 samples (discarding the residue). For the example above, after down sampling the number of samples in each segment is 1102. **We use “down\_sample\_rate=4” throughout this project**

Clean Labels

When loading the “.csv” files, we encountered some exceptions. By checking the “.csv” files, we find the following problems:

1. In some files, speakers are named “CH1” and “CH2”. However in other files, speakers are named as “HEALTHY1” and “HEALTHY2”. This problem is solved by modifying our code.
2. In some files, values and attributes are misaligned. This problem is solved by cleaning the data manually.
3. Some files contained Unicode characters.

Load Labels

“.csv” file is loaded by using the “get\_labels” function in “Labels.py”. Specifically “get\_labels” function read the “.csv” file located at certain directory and return a tuple (label\_1, label\_2) where each element is a matrix (actually it is a binary vector) contains the labels for each channel. The shape of each matrix is

(number\_of\_segments\_each\_channel,1)

For example, the returned label matrixes corresponding to the data matrixes above are

|  |  |
| --- | --- |
| Label\_1 | Label\_2 |
| 0 or 1 (label for segment 1) | 0 or 1 (label for segment 1) |
| 0 or 1 (label for segment 2) | 0 or 1 (label for segment 2) |
| 0 or 1 (label for segment 3) | 0 or 1 (label for segment 3) |
| … | … |
| 0 or 1 (label for segment 100) | 0 or 1 (label for segment 100) |

“get\_labels” and “get\_data” can be override to modify the format of input and labels for different neural networks

Now we depict how to convert the time period in “.csv” file to the matrix above. Given the “.csv” file as below

tmin tier text tmax

0 CH2 N 1.361079

0 CH1 N 4.550097

1.361079 CH2 S 4.996529

4.550097 CH1 S 5.541915

4.996529 CH2 N 5.547973

5.541915 CH1 N 8.183008

We at first truncate the time by discarding numbers after 0.1, and divide the truncated numbers by segment size 0.1, then the original time is transferred into index as below

tmin tier text tmax

0 CH2 N 13

0 CH1 N 45

13 CH2 S 49

45 CH1 S 55

49 CH2 N 55

55 CH1 N 81

After that we create an all zeros vector for each channel and change the positions where is the speech of corresponding speaker to 1. For example, if channel one has 100 segment as in previous examples, then we create an all zeros vector “label\_1”. As indicated by the row 4 of the table above, index 45-55 is the speech of speaker 1, so we set label\_1[45:55] =1. Labels are generated in this way by reading the whole table.

Grouping Data

There are 37 pairs of “.wav” and “.csv” files in total. We randomly divide them into training set, cross-validation set and testing set with the ratio %70/%15/%15

# Neural Networks

MLP

We first tried a simple multilayer perceptron. The code corresponding to this part of work is in “Alg1\_MLP\_1channel\_2classes”. Each time we feed one row of the loaded data matrix to the network to do classification. Each channel is treated as a single audio file. So the number of input neurons equals to the number of samples in each segment (each row of data matrix). The number of hidden layers and number of neurons there in can be changed to any number. The number of output neurons is just 1 to predict 0 or 1. A typical graph for a 2 layer MLP is shown below

Output

Layer

1 neuron

Hidden

Layer\_2

50 neurons

Hidden

Layer\_1

200 neurons

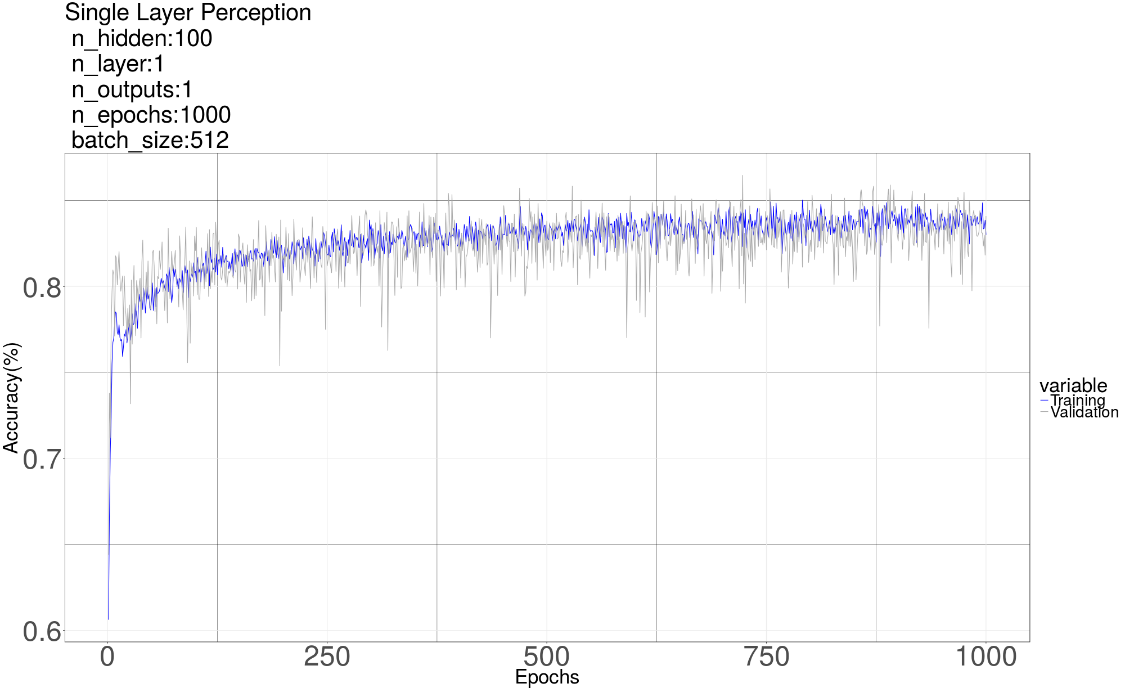
Input

Layer

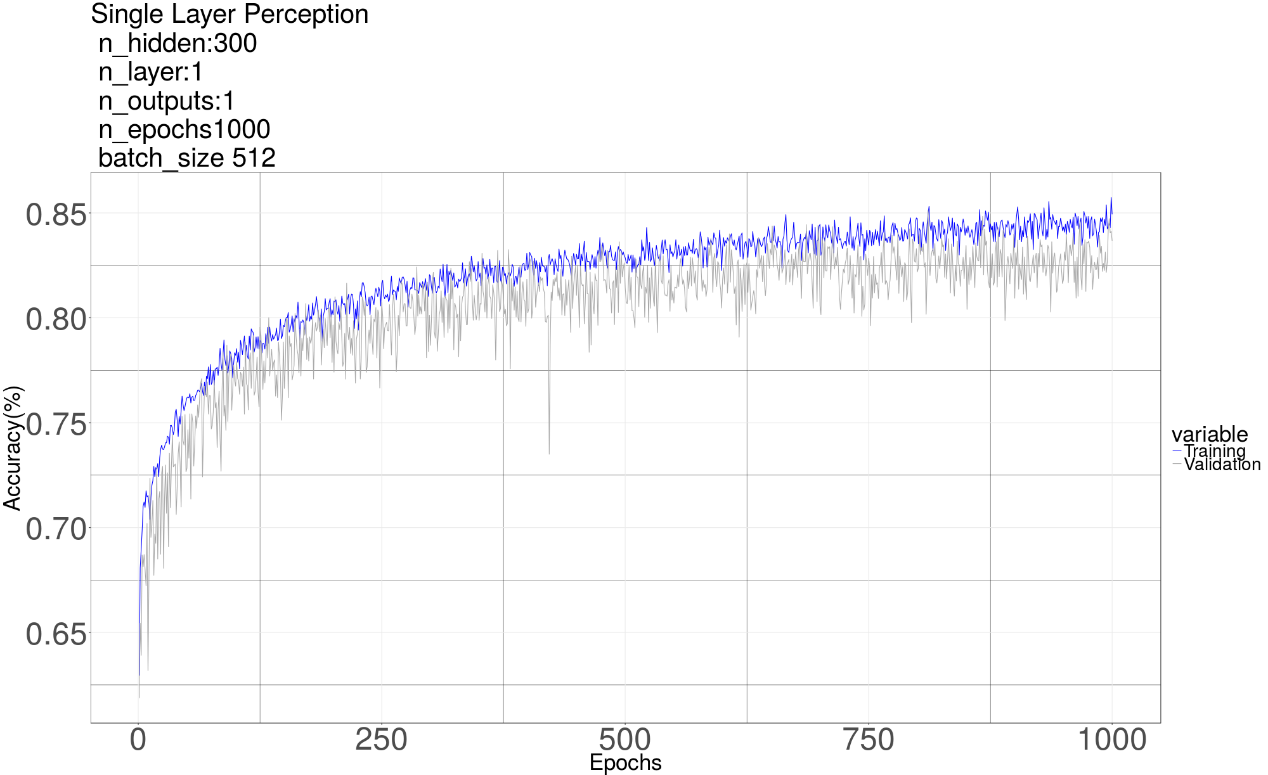
1102 neurons

**For all the networks used in this project, the hidden neurons are ReLu and the output neuron is sigmoid (for 4 classes method, there are 4 output neurons and the output neuron is softmax). If the value after applying sigmoid is above 0.5, we classify it as 1, otherwise classify it as 0 (in the code we judge whether the logit before sigmoid greater than 0 or not). The cost function is cross entropy. Mini-batch gradient descent with Adam optimization is used to train network. Learning rate is 0.001 (default learning rate) throughout this project. Then number of weights depends on the number of layers and neurons therein. Basically the number of weights is the multiply of number of neurons in every two consecutive layers.**

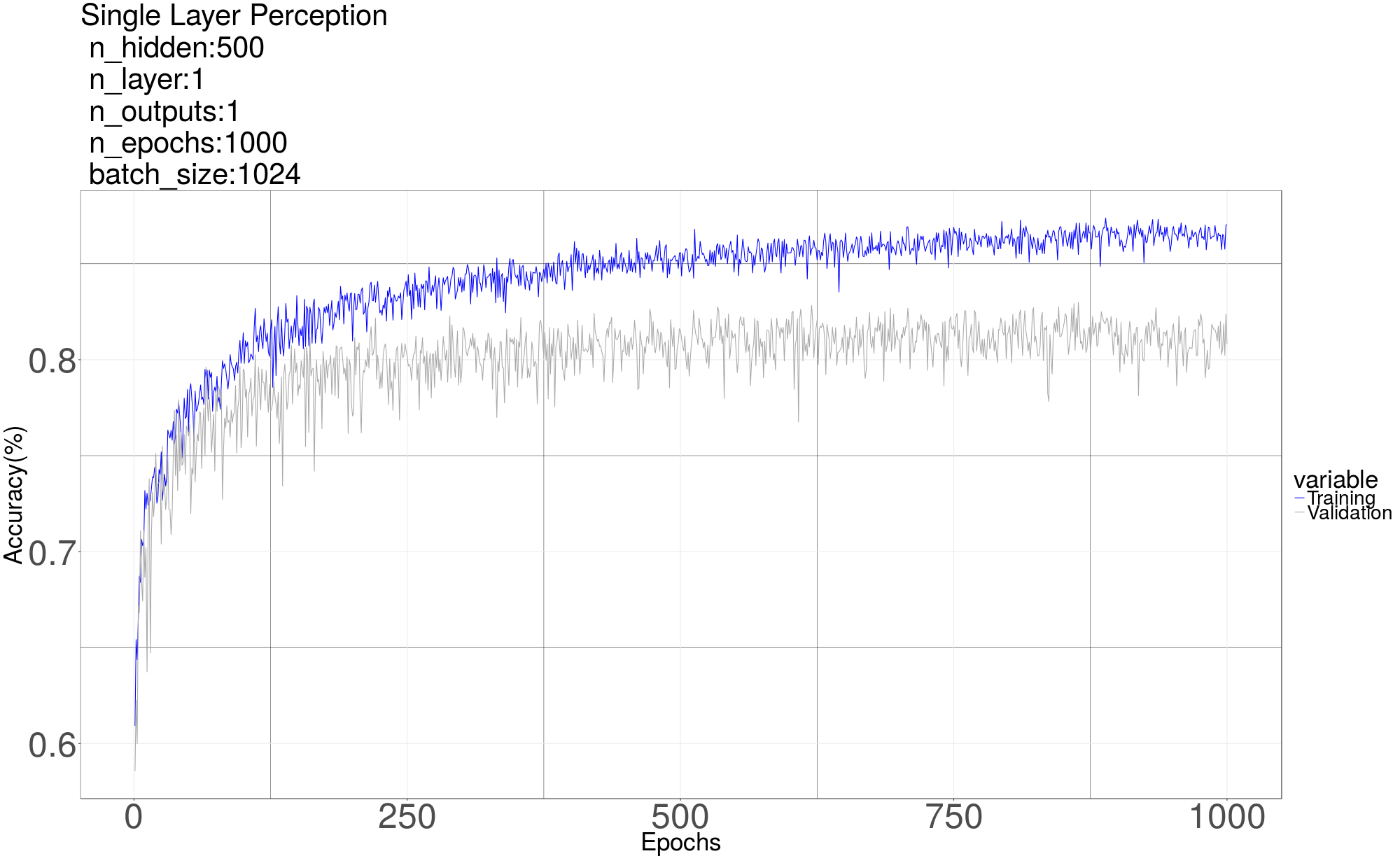
Some results are plotted below. The parameters used to generate each result are included.



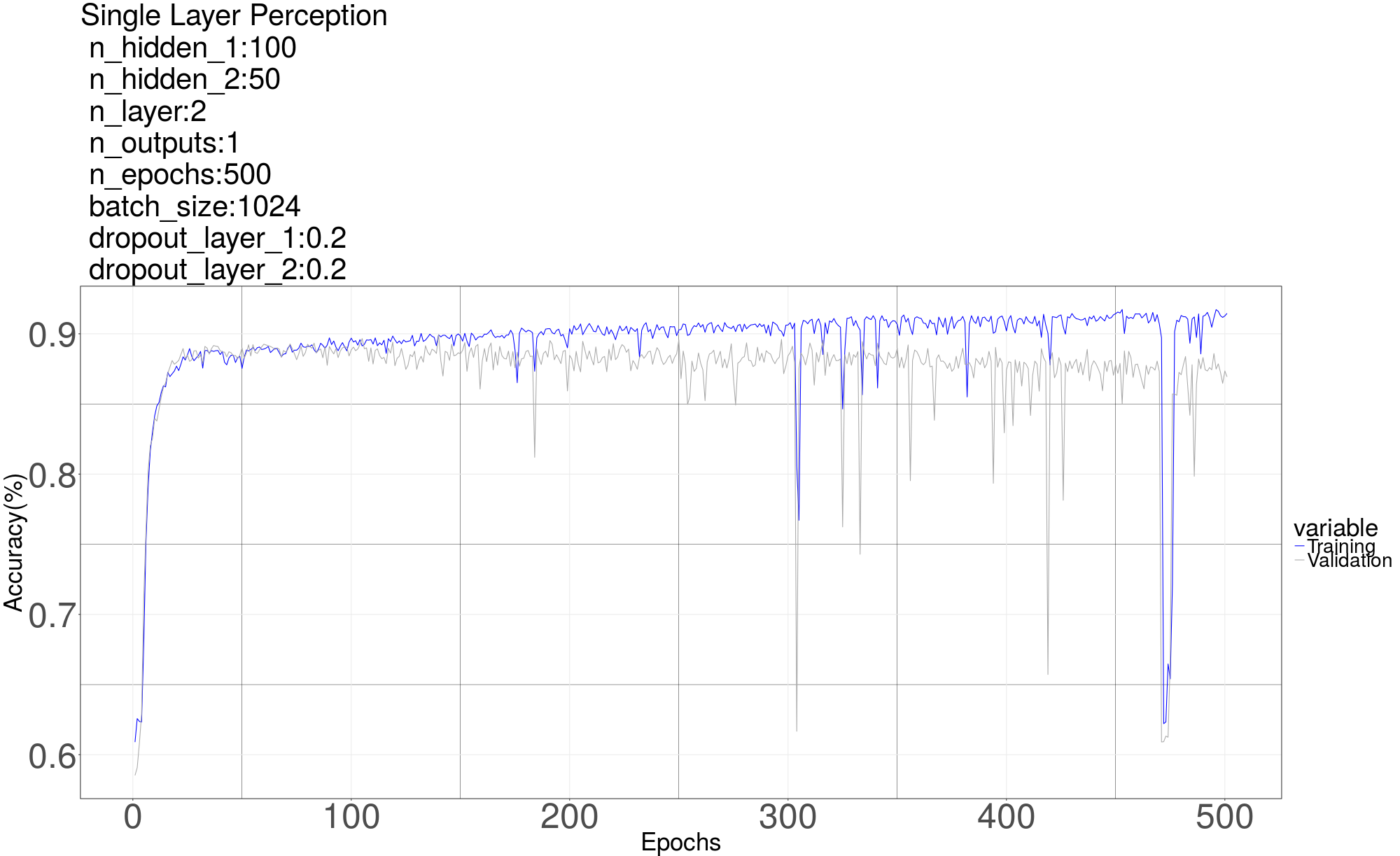
In this plot above, we used one layer with 100 neurons in it. We can see the accuracy is less than %85. We stopped training after 1000 epochs as it almost converged. **In each epoch, all training data are training once. In each epoch, the order of files are shuffled.**



In this plot above, we still use one layer but increased the number of neurons to 300. We can see the training accuracy is improved a little bit, however, the over-fitting problem is more obvious in this plot. The training accuracy looks still increasing slowly. We stopped training as the improvement is marginal and we want to move forward.



In this plot, we can see as the number of neurons is further increased to 500, the training keep improving, however, the validating accuracy is not improved. The over-fitting problem is more obvious in this plot which means we need to use some regularization. **Throughout this project, we use dropout as our regularization.**



Then we tried a 2-layer MLP with other parameters included in the graph as shown above. At first we can see the training accuracy can over %90 now which is obviously higher than a single layer network. This observation indicates that more layers can increase the capacity of the network and hence improve the training accuracy. Another interesting observation is that sometimes the training accuracy may suddenly drop significantly. The possible reason is that the gradient descent may be trapped at a local minimal or a saddle point. Dropout is applied to mitigate the gap between training accuracy and validating accuracy. The dropout rate is 0.2 for each layer in this graph. The gap is not completely eliminated, which means we should increase the dropout rate. However, as MLP is not the focus of this project, we decide to move forward.

Extension to 4 classes method

We tried to extend the work so far to 4 classes method. To do so, we only need to do some modification on the code as below

1. Change the output neurons to 4 and change the output unit to softmax
2. Override the “get\_labels” function as below

def get\_label(filename):

#get label matrixes by using original “get\_label” function

temp1,temp2=Labels.get\_labels(filename,chunk\_time=chunk\_time)

temp1, temp2 =temp1.reshape(-1), temp2.reshape(-1)

#compute the new labels

return temp1+temp2\*2,temp2+temp1\*2

This 4 classes method can be used either to train each single channel independently or to train two channels together. When using on one channel, we just change the 2 classes label to 4 classes label. The relationship between 2 classes label and 4 classes label is as below

|  |  |
| --- | --- |
| Label of channel 1 | |
| 2 classes method | 4classes method |
| 0 | 0: not speech |
| 2:speech of speaker2 |
| 1 | 1: speech of speaker 1 |
| 3: overlap of two speakers |

|  |  |
| --- | --- |
| Label of channel 2 | |
| 2 classes method | 4classes method |
| 0 | 0: not speech |
| 2:speech of speaker1 |
| 1 | 1: speech of speaker 2 |
| 3: overlap of two speakers |

When training two channels together, we concatenate two segments from two channels together (these two segments have the same index number in each channel) and train them together. In this case we need to double the number of input neurons.

**For 4 classes methods, the accuracy is only about %50 after trained several hours.** Maybe the 4 classes method need more time to train, or need more complicated network. As the result here is not promising, we didn’t spend so much effort on the 4 classes method. The codes here are in “Alg2\_MLP\_1channel\_4classes” and “Alg3\_MLP\_2channel\_4classes”.

Summary for MLP

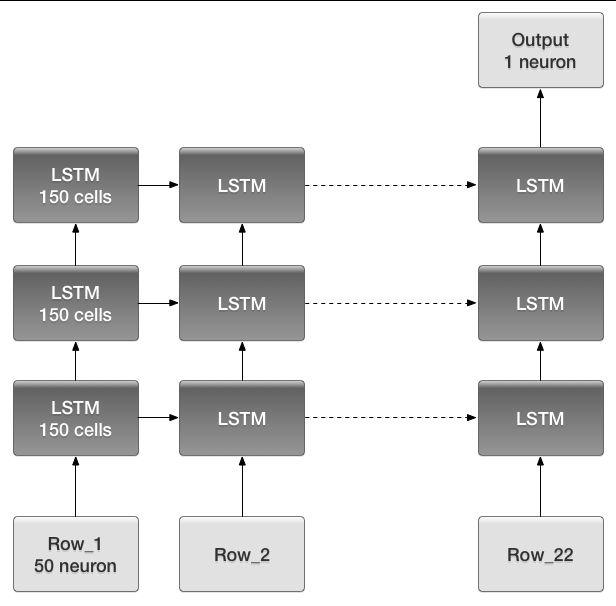
By looking at the results, we can see basically MLP with one or two layers can achieve about %85 accuracy. Also we find that local minimal and saddle point can be a severe problem for deep MLP (even with only 2 layers).

RNN

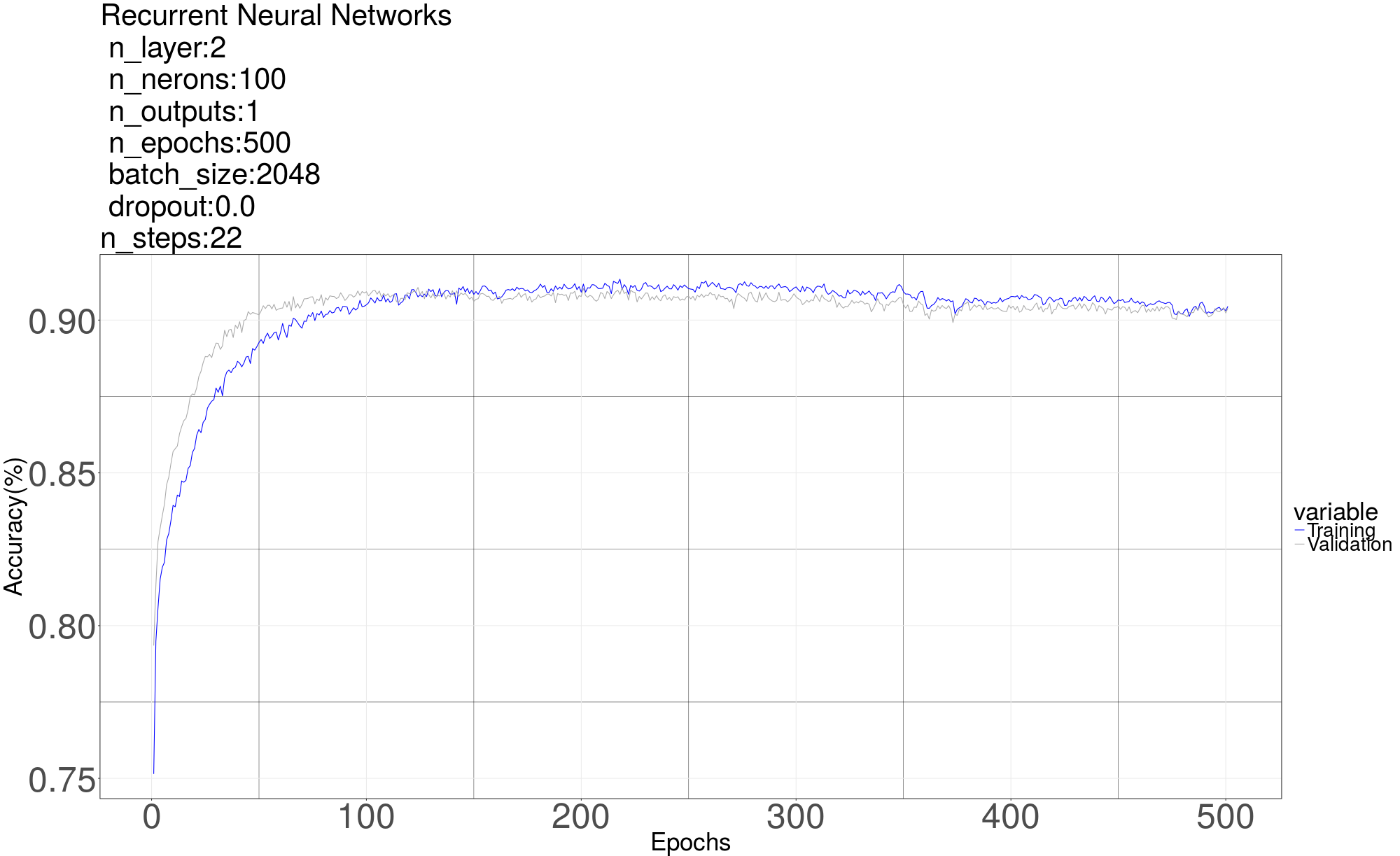
After MLP, we tried RNN. The code here is in “Alg4\_RNN\_1channel\_2classes”For input, we reshape a segment (one row of data matrix) into a matrix whose shape is

(number\_of\_steps, number\_of\_samples\_each\_segment// number\_of\_steps)

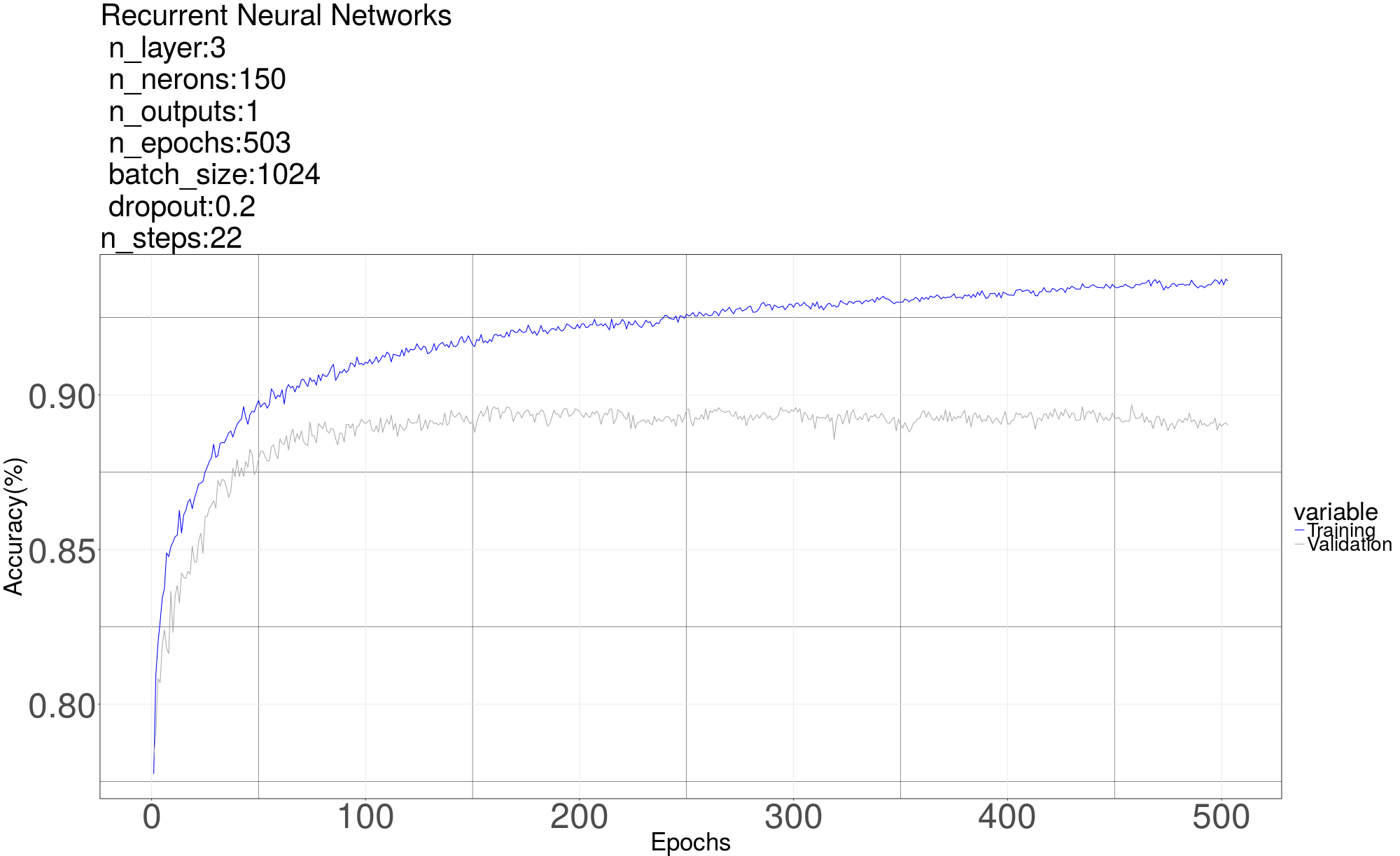
Recall that the number\_of\_samples\_each\_segment after down sampled is 1102, the number\_of\_steps we used is 22, so the shape of input is (22, 50), residue samples are discarded. The number of input neurons is 50, in each time step, one row of reshaped matrix is feed in. After 22 steps, a single output is used to do classification. We tried different parameters (number of layers, number of cells in each layer, number of steps, dropout rate, et al). The RNN giving us the best result has 3 layers with 150 Long short-term memory (LSTM) cells in each layer. The graph of RNN is as below. The LSTM in the graph means a LSTM layer which consists of 150 LSTM cells. The output only has one neuron with sigmoid to predict 0 or 1. We didn’t try 4 classes method on RNN cause the result by using 4 classes method seems not promising as shown in MLP.



Some results for RNN are provided below

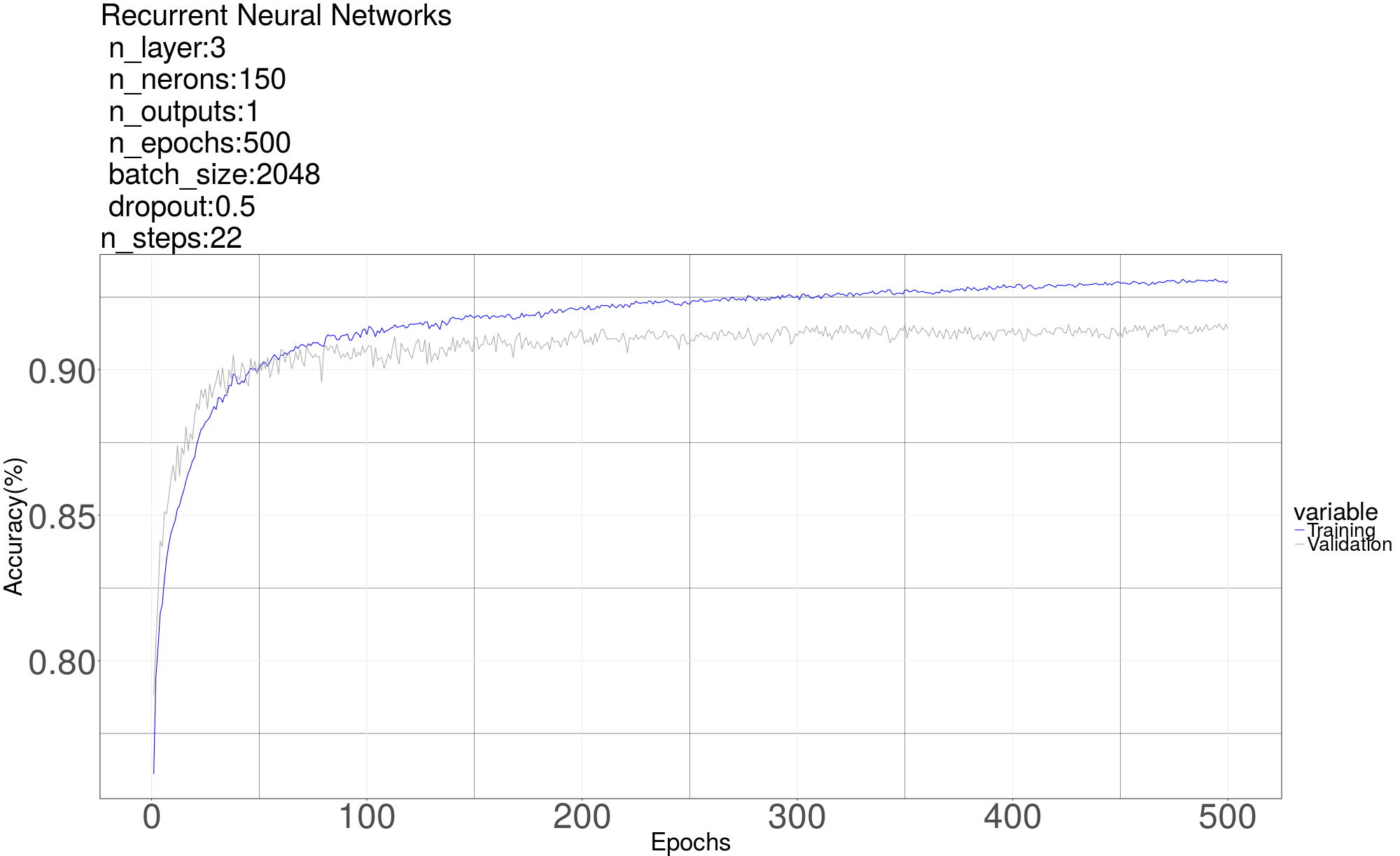


We first tried a 2 layer RNN. It achieves about %90 accuracy for both training and validation without dropout, which is much better than the MLP we tried before. Dropout is not used here as the validation is very close to the training accuracy. We stopped training since it converges.



We then tried a 3 layer RNN. We can see the training accuracy can go over %95 now. However the gap between training and validation is obvious even though we have set dropout rate to 0.2. The accuracy of validation is below %90. This observation shows that deeper network tends to be over-fitting. To solve this problem, we need to increase the dropout rate. We stopped training here cause it is already over-fitting.

The dropout is applied at the fully connected part of each recurrent layer. All layers have the same dropout rate.



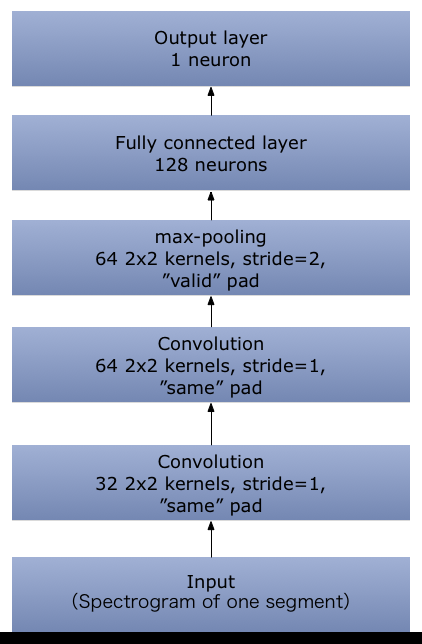
We then increased dropout rate to 0.5. We can see the discrepancy between training and validation is mitigated. The accuracy of validation is over %90 now (about %92-%93). **The accuracy on testing set is %91.47**, which is very close to the accuracy of validation set. The training accuracy is still above %96 in this plot, which means if we keep increasing the dropout rate, the accuracy on validation and testing set can be further improved. We stopped training as the result is already good and we are running out of the time.

Summary for RNN

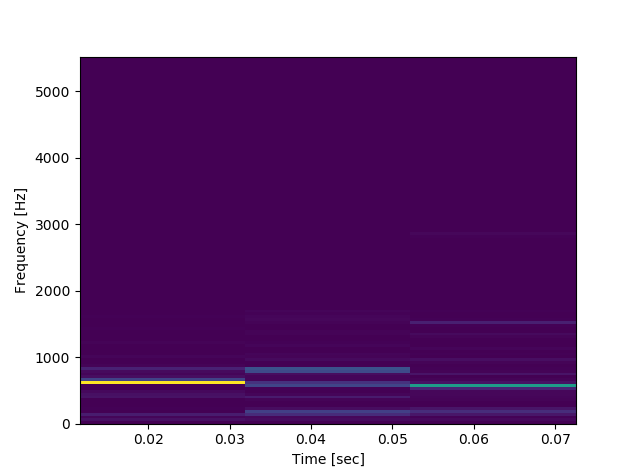
The accuracy achieved by using RNN is much better than MLP. And we didn’t encountered saddle point problem. Deep RNN is easier to train thanks to LSTM. **We choose the RNN with 3 layers as depicted above as our best network which achieves %91.47 on test set**

CNN

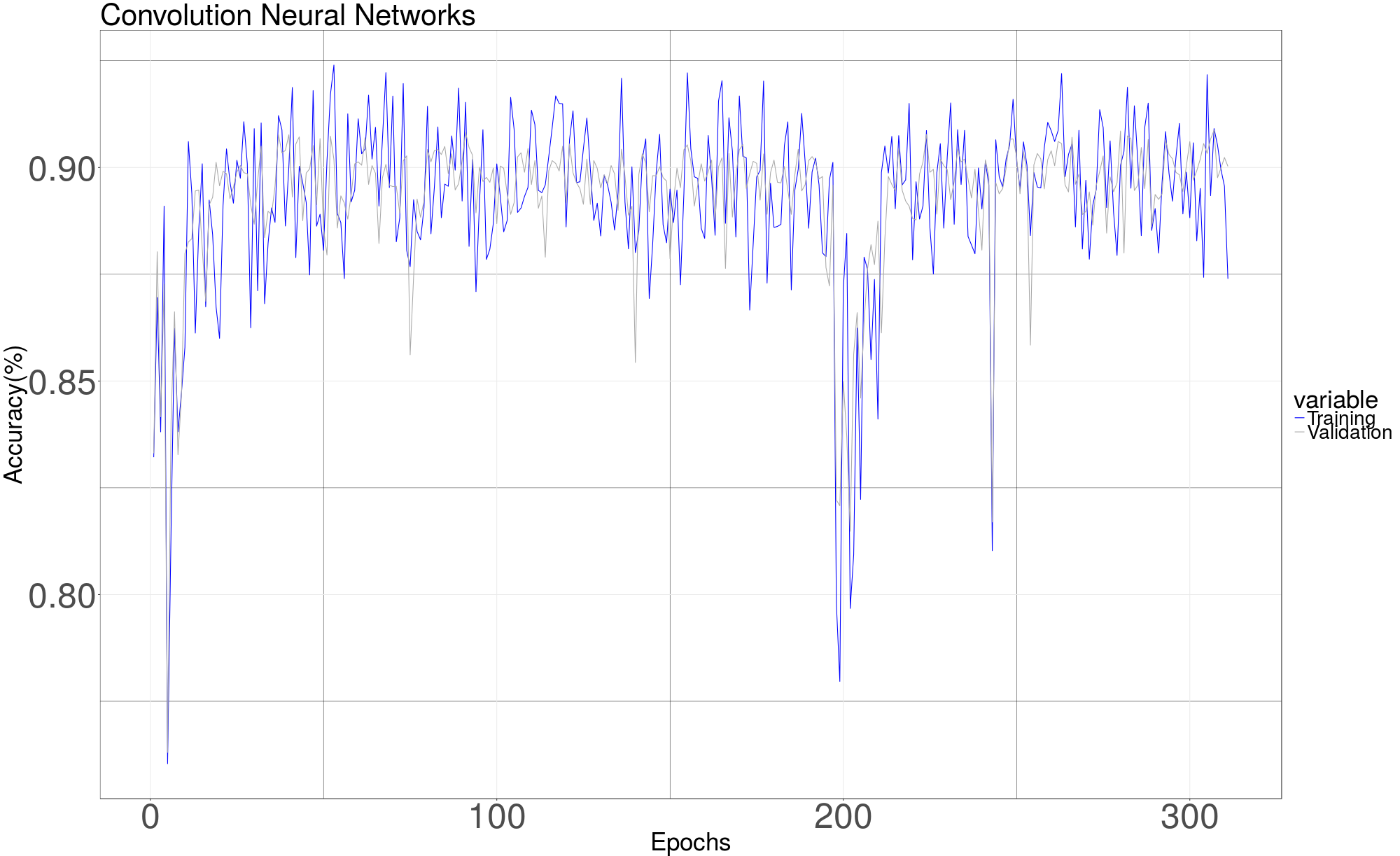
To apply CNN, we at first compute the spectrogram for each row of the data matrix returned by “get\_data”, then store them into a new file by using pickle. In this way we don’t need to compute spectrogram online and hence can save a lot of training time. Function “scipy. signal.spectrogram” is used to compute the spectrogram for each segment. The recomputed spectrogram of each segment then is organized to a 3 dimension matrix with shape (number\_of\_segments, height, width). For example, the down sampled data matrix of a channel returned by “get\_data” has the shape (100, 1102) for a channel with 100 segments, then the shape of recomputed spectrogram matrix is (100,129,4). The number of segments remains the same. The height 129 and width 4 come from using the default parameters of function “scipy. signal.spectrogram” . Spectrogram matrixes are computed and stored by using code in “Spectrogram\_Generator”.

After get spectrogram matrix, we can use it as an image with a single channel and use it to train a CNN. Again we consider a single channel as the input (later we will discuss how to consider two channels together). The code of CNN is in “CNN\_1channel\_2classes”. The architecture of the network is as below

The spectrogram of a segment looks like below



One result by using CNN is as below



This plot is generated with batch\_size=256 and without dropout. The other parameters are as shown in the architecture of the network. We can see the training and validation accuracy go to %90 quickly. We believe by training it with more time or increasing the capacity of the network (increase convolutional layer or fully connected layer, increase the kernels, try different mini-batch size, et al), the performance of CNN can be improved since many published paper indicate that using spectrogram instead of the raw audio file can improve the accuracy of speaker diarization task. We only trained it with 300 epochs due to the limit of time.

Extension of CNN

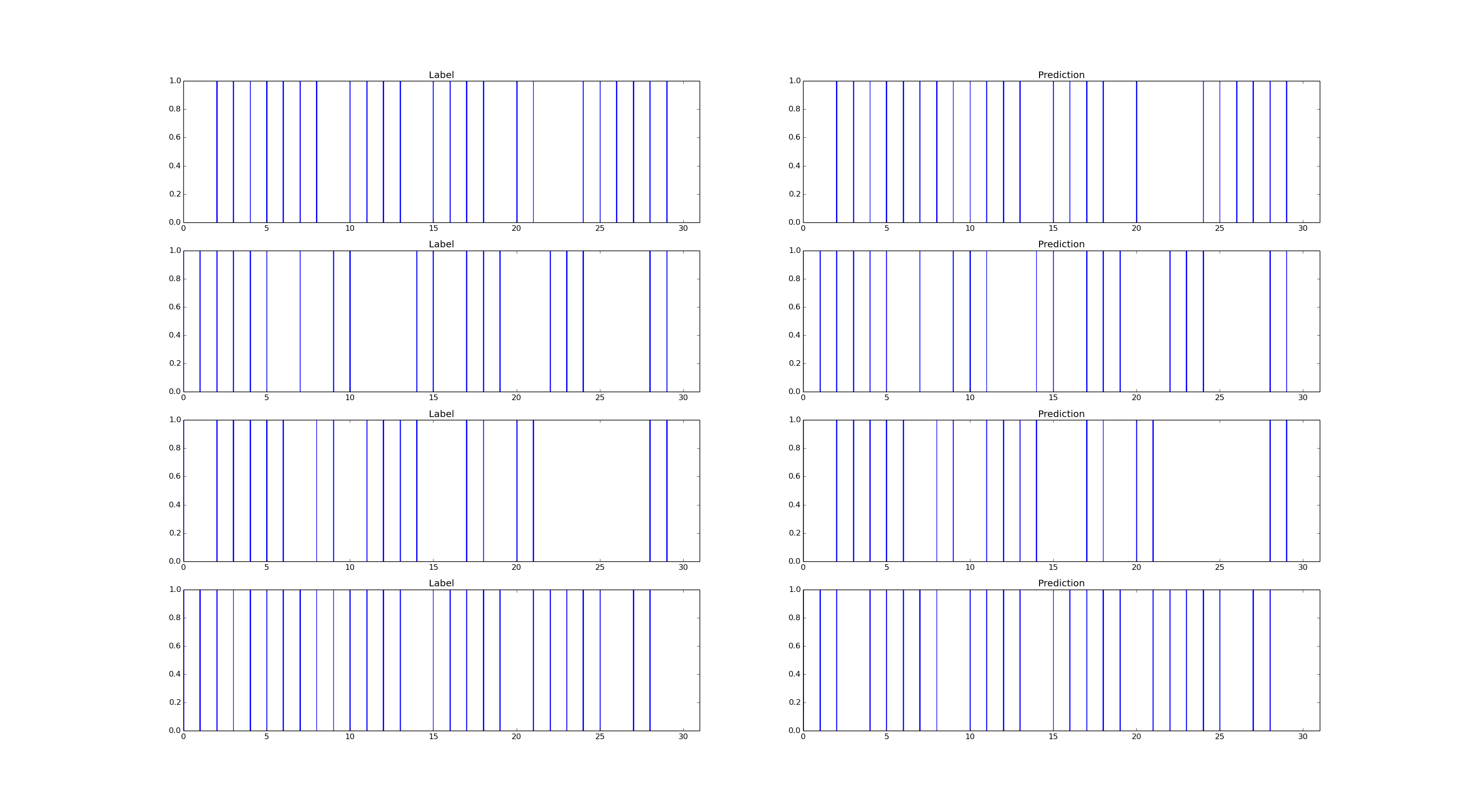
The segments of two channels with the same index (i.e. segment\_1 of channel 1 and segment\_1 of channel 2) can be feed in together just as an image with 2 channels. And then we can use 4 classes method on it. The code to do so is in “CNN\_2channels\_4classes”. However, after trained couple of hours, the accuracy is only about %50. It may need more time to train, or we need to increase the capacity of the network.

Summary for CNN

CNN can get a good result quickly. Due to the limit of time, we didn’t tune it hardly. To improve the performance of CNN, the possible options are

1. Training with more time
2. Increase the capacity of network (number of layers, neurons, kernels)
3. Tune the parameters generating the spectrogram instead of using the default parameters

Testing

Our best result is generated by RNN which achieves %91.47 on test set. We feed the test data to the network and visualized the result. The graph below compared our prediction with truth label for a certain period of time of one test file. Left side is the label, right side is our prediction. Vertical line indicates the prediction/label is 1. Horizontal axis is the index of segment, which can be mapped back to time axis. We can see our prediction looks very similar to the labels.

Summary

In this project, we tried 3 types of neural networks to do speaker diarization. The original task is formulated into a supervised classification problem. Based on the results we get so far, the accuracy (on test set) for each type of network is

1. 2 layer MLP: about %87
2. 3 layer RNN: about %92
3. 2 convolutional layer CNN: about %90

The performance for each network can be further improved if keep tuning parameters and training with more time. What we learned from this project can be summarized as below

1. We get more familiar with different types of neural networks
2. We get more familiar with tensorflow.
3. We learned how important it is to normalize the data before training
4. We learned how to increase the training accuracy by increasing the capacity of network (increase the number of layers, neurons)
5. We learned how to decrease the gap between training accuracy and validation accuracy by using dropout
6. We learned how to use spectrogram to process audio files
7. Based on our observation, we realized that saddle point can be a severe problem when training deep neural network (even with just 2 layers).
8. We learnt to use GPU with TensorFlow to perform parallel computation. Using GPU required setting up CUDA and cuDNN.

Code

1. “Load\_Audio\_Data.py”

import scipy.io.wavfile

import numpy as np

import pydub

from pydub import AudioSegment

def get\_data(filename, chunk\_time=0.1, down\_sample=False,down\_sample\_rate=4):

# Get sample rate and read data from both channel

sampleRate, data = scipy.io.wavfile.read(filename)

num\_of\_frames\_per\_chunk=int(sampleRate\*chunk\_time)

if num\_of\_frames\_per\_chunk!=sampleRate\*chunk\_time:

raise ValueError('inappropriate chunk\_time')

# Two channels

channel\_1=data[:,0]

channel\_2=data[:,1]

#abandon residue

data\_length = int(len(channel\_1) / num\_of\_frames\_per\_chunk) \* num\_of\_frames\_per\_chunk

channel\_1 = channel\_1[:data\_length].reshape(1, -1)

channel\_2 = channel\_2[:data\_length].reshape(1, -1)

no\_of\_blocks = data\_length // num\_of\_frames\_per\_chunk

# reshape the data to a vector where each row is the data of a chunk

data\_matrix\_1 = channel\_1.reshape(-1, num\_of\_frames\_per\_chunk)

data\_matrix\_2 = channel\_2.reshape(-1, num\_of\_frames\_per\_chunk)

if down\_sample:

data\_matrix\_1 = data\_matrix\_1[:, ::down\_sample\_rate]

data\_matrix\_2 = data\_matrix\_2[:, ::down\_sample\_rate]

# return a tuple of two matrixes, one for each channel

return (data\_matrix\_1, data\_matrix\_2)

1. “Labels.py”

import pandas as pd

import numpy as np

def get\_labels(filename,chunk\_time=0.1):

# Step 1: Read CSV file

# Read file in utf-8 format

data = pd.read\_csv(filename, encoding ='utf-8')

dataHealthy1 = data[data['tier'].str.contains(u"1") & data['text'].str.contains(u'S')]

dataHealthy2 = data[data['tier'].str.contains(u"2") & data['text'].str.contains(u'S')]

healthy1Time = dataHealthy1[['tmin','tmax']]

healthy2Time = dataHealthy2[['tmin','tmax']]

# We slice the float to and convert it to 0.1sec label

healthy1Time = healthy1Time.astype(float) / chunk\_time

healthy1Time = healthy1Time.astype(int)

healthy2Time = healthy2Time / chunk\_time

healthy2Time = healthy2Time.astype(int)

maxFrame = (data['tmax'].max()/chunk\_time).astype(int)

label\_1 = np.zeros((maxFrame), dtype=np.int)

label\_2 = np.zeros((maxFrame), dtype=np.int)

# Assign values to each frame for each time duration of speech

for index, row in healthy1Time.iterrows():

label\_1[row['tmin']: row['tmax']] = 1

for index, row in healthy2Time.iterrows():

label\_2[row['tmin']: row['tmax']] = 1

#return the labels for each channel

return (label\_1.reshape(-1,1),label\_2.reshape(-1,1))

1. “Alg1\_MLP\_1channel\_2classes.py”

# 3 main functions in this code, uncommon each function to use

#train\_and\_save\_network() train the network and save resuelt with early stopping

#test\_with\_restore\_data() test results on test data set

#compare(data\_file,label\_file) com predicted result with true labels

import Audio\_data

import Labels

import numpy as np

import tensorflow as tf

import os

import random

import sys

import matplotlib.pyplot as plt

import datetime

#dir of data and labels

dir\_data="./data/Datas/"

dir\_label="./data/Labels/"

dir\_cross\_data="./data/Cross\_datas/"

dir\_cross\_label="./data/Cross\_labels/"

dir\_test\_data="./data/Test\_datas/"

dir\_test\_label="./data/Test\_labels/"

save\_path="./Alg4/ALg4\_best\_acc.ckpt"

os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

save\_path\_latest="./Alg4/ALg4\_latest\_acc.ckpt"

os.makedirs(os.path.dirname(save\_path\_latest), exist\_ok=True)

chunk\_time=0.1

down\_sample=True

down\_sample\_rate=4

#overload Audio\_data.get\_data() and Label.get\_labels() to make sure same chunk size and downsampling rate to all data

def get\_data(filename):

return Audio\_data.get\_data(filename,chunk\_time,down\_sample,down\_sample\_rate)

def get\_label(filename):

return Labels.get\_labels(filename,chunk\_time=chunk\_time)

#get the data matrix

(data\_1,data\_2) = get\_data("./Convo\_Sample.wav")

test\_data=np.concatenate((data\_1, data\_2), axis=0)

#check dimension

print(test\_data.shape)

#reset graph

def reset\_graph(seed=1):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

#shuffle data along with label

def shuffle\_data(data,label):

#number of samples

num=data.shape[0]

seq=np.random.permutation(num)

return data[seq],label[seq]

n\_inputs = test\_data.shape[1]

n\_steps=n\_inputs//50

n\_inputs=n\_inputs//n\_steps

n\_data=n\_steps\*n\_inputs

n\_layers = 1

n\_neurons = 200

n\_outputs = 1

n\_epochs = 1000

batch\_size = 256

rnn\_dropout=0#dropout rate, 0 means no dropout

best\_acc=0

#start seting network

reset\_graph()

y = tf.placeholder(tf.float32, shape=(None, n\_outputs))

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

training = tf.placeholder\_with\_default(1.0, shape=[], name='training')

keep\_rate=1-rnn\_dropout# this is the value used to feed into training, 1 menas no dropout

#recurrent network

lstm\_cells = [tf.contrib.rnn.DropoutWrapper(tf.contrib.rnn.BasicLSTMCell(num\_units=n\_neurons),output\_keep\_prob=training)

for layer in range(n\_layers)]

multi\_cell = tf.contrib.rnn.MultiRNNCell(lstm\_cells)

outputs, states = tf.nn.dynamic\_rnn(multi\_cell, X, dtype=tf.float32)

top\_layer\_h\_state = states[-1][1]

logits = tf.contrib.layers.fully\_connected(top\_layer\_h\_state, n\_outputs, activation\_fn=None)

xentropy=tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits = logits, labels = y)

loss = tf.reduce\_mean(xentropy)

optimizer = tf.train.AdamOptimizer()

training\_op = optimizer.minimize(loss)

pred=tf.cast(logits>0,"int32")

# Test model

correct\_prediction = tf.equal(tf.cast(logits>0,"float32"), y)

# Calculate accuracy

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

#check dir before use

# after training, use this function observe predict and result

def compare(data\_file,label\_file):

with tf.Session() as sess:

saver.restore(sess, save\_path) # or better, use save\_path

(data\_1, data\_2) = get\_data(data\_file)

Test\_datas = np.concatenate((data\_1, data\_2), axis=0)

Test\_datas = Test\_datas[:, :n\_data]

Test\_datas = Test\_datas.reshape(-1, n\_steps, n\_inputs)

(label\_1, label\_2) = get\_label(label\_file)

Test\_labels = np.concatenate((label\_1, label\_2), axis=0)

out = pred.eval({X: Test\_datas})

print(Test\_labels[0:300])

#starting time point in unit of chunck size

start\_point=0

for x,y,z in zip((421,423,425,427),(422,424,426,428),(start\_point,start\_point+300,start\_point+600,start\_point+900)):

plt.subplot(x)

plt.title("Label")

j=-1

for i in Test\_labels[z:z+300]:

j+=1

if i:

plt.axvline(j/10)

plt.xlim((0,31))

plt.subplot(y)

plt.title("Prediction")

j=-1

for i in out[z:z+300]:

j+=1

if i:

plt.axvline(j/10)

plt.xlim((0,31))

plt.show()

#compare("./data/Test\_datas/HS\_D30.wav","./data/Test\_labels/HS\_D30.csv")

#check dir before use

#after trining, use this function to test

def test\_with\_restore\_data():

with tf.Session() as sess:

saver.restore(sess, save\_path) # or better, use save\_path

for (root\_d, dirs\_d, files\_d), (root\_l, dirs\_l, files\_l) in zip(os.walk(dir\_test\_data), os.walk(dir\_test\_label)):

testing\_acc = {}

for d, l in zip(files\_d, files\_l):

(data\_1, data\_2) = get\_data(root\_d + d)

Test\_datas = np.concatenate((data\_1, data\_2), axis=0)

Test\_datas = Test\_datas[:, :n\_data]

Test\_datas = Test\_datas.reshape(-1, n\_steps, n\_inputs)

(label\_1, label\_2) = get\_label(root\_l + l)

Test\_labels = np.concatenate((label\_1, label\_2), axis=0)

testing\_acc[d] = (accuracy.eval({X: Test\_datas, y: Test\_labels}))

avg\_acc = sum(testing\_acc.values()) / len(testing\_acc)

print( "testing Accuracy:", testing\_acc, "\n average is :", avg\_acc)

print("Testing end")

sys.exit()

#test\_with\_restore\_data()

def train\_and\_save\_network():

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

#train on all training data and get training accuracy

for (root\_d, dirs\_d, files\_d), (root\_l, dirs\_l, files\_l) in zip(os.walk(dir\_data), os.walk(dir\_label)):

training\_acc = {}

cross\_acc = {}

pairs=list(zip(files\_d,files\_l))

random.shuffle(pairs)

for d, l in pairs:

(data\_1, data\_2) = get\_data(root\_d+d)

datas = np.concatenate((data\_1, data\_2), axis=0)

datas=datas[:,:n\_data]

datas=datas.reshape(-1,n\_steps,n\_inputs)

(label\_1, label\_2) = get\_label(root\_l+l)

labels = np.concatenate((label\_1, label\_2), axis=0)

training\_acc[d]=(accuracy.eval({X: datas, y: labels}))

epoch\_data,epoch\_label=shuffle\_data(datas,labels)

for i in range(datas.shape[0]// batch\_size):

X\_batch = epoch\_data[i:i+batch\_size]

y\_batch = epoch\_label[i:i+batch\_size]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch,training :keep\_rate})

print("loss:", loss.eval(feed\_dict={X: X\_batch, y: y\_batch}))

avg\_training\_acc=sum(training\_acc.values())/len(training\_acc)

print("epoch:", epoch, "training Accuracy:", training\_acc,"\n average is :",avg\_training\_acc)

#max\_training\_acc=max(max\_training\_acc,avg\_training\_acc)

#testing

for (root\_d, dirs\_d, files\_d), (root\_l, dirs\_l, files\_l) in zip(os.walk(dir\_cross\_data), os.walk(dir\_cross\_label)):

for d, l in zip(files\_d, files\_l):

(data\_1, data\_2) = get\_data(root\_d + d)

cross\_datas = np.concatenate((data\_1, data\_2), axis=0)

cross\_datas = cross\_datas[:,:n\_data]

cross\_datas = cross\_datas.reshape(-1, n\_steps, n\_inputs)

(label\_1, label\_2) = get\_label(root\_l + l)

cross\_labels = np.concatenate((label\_1, label\_2), axis=0)

cross\_acc[d] = (accuracy.eval({X: cross\_datas, y: cross\_labels}))

avg\_acc=sum(cross\_acc.values())/len(cross\_acc)

print("epoch:",epoch,"Cross Accuracy:", cross\_acc,"\n average is :",avg\_acc)

if avg\_acc>best\_acc:

best\_acc=avg\_acc

saver.save(sess, save\_path)

print("Network saved")

saver.save(sess, save\_path\_latest)

print("max training acc is",best\_acc)

#out=pred.eval({X: datas})[100:200]

#print(np.concatenate((out,labels[100:200]),1))

#train\_and\_save\_network()

“Alg2\_MLP\_1channel\_4classes.py” and “Alg3\_MLP\_2channel\_4classes.py” are very similar to “Alg1\_MLP\_1channel\_2classes.py” and hence not presented here to save space. I can sent them through email if needed.

1. “Alg4\_RNN\_1channel\_2classes”

# 3 main functions in this code, uncommon each function to use

#train\_and\_save\_network() train the network and save resuelt with early stopping

#test\_with\_restore\_data() test results on test data set

#compare(data\_file,label\_file) com predicted result with true labels

import Load\_Audio\_Data

import Labels

import numpy as np

import tensorflow as tf

import os

import random

import sys

import matplotlib.pyplot as plt

import datetime

#settings, pathes

np.random.seed(2)

chunk\_time=0.1

down\_sample=True

down\_sample\_rate=4

rate=44100

fs=rate/down\_sample\_rate

dir\_raw\_data="./data/Sound\_Files/"

dir\_normalized\_data="./data/Normalized\_Sound\_Files/"

dir\_raw\_data=dir\_normalized\_data#Use normalized data instead of raw data

dir\_label="./data/Cleaned\_Labels/"

#save pathes

save\_path="./Alg4\_1channel/ALg4\_best\_acc.ckpt"

os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

save\_path\_latest="./Alg4\_1channel/ALg4\_latest\_acc.ckpt"

os.makedirs(os.path.dirname(save\_path\_latest), exist\_ok=True)

#overload Audio\_data.get\_data() and Label.get\_labels() to make sure same chunk size and downsampling rate to all data

def get\_data(filename):

d1,d2=Load\_Audio\_Data.get\_data(filename,chunk\_time,down\_sample,down\_sample\_rate)

return np.concatenate((d1,d2),axis=0)

def get\_label(filename):

l1,l2=Labels.get\_labels(filename, chunk\_time=chunk\_time)

return np.concatenate((l1,l2),axis=0)

#get the data matrix

example\_data = get\_data("./Convo\_Sample.wav")

#check dimension

print(example\_data.shape)

def divide\_data(dir\_data,dir\_label):

for (root\_d, dirs\_d, files\_d), (root\_l, dirs\_l, files\_l) in zip(os.walk(dir\_data), os.walk(dir\_label)):

full\_files\_d=[root\_d+x for x in files\_d]

full\_files\_l = [root\_l + x for x in files\_l]

pairs=list(zip(full\_files\_d,full\_files\_l))

np.random.shuffle(pairs)

num=len(pairs)

train=pairs[:int(num\*0.7)]

cross=pairs[int(num\*0.7):int(num\*0.85)]

test=pairs[int(num\*0.85):]

return train,cross,test

#reset graph

def reset\_graph(seed=1):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

#shuffle data along with label

def shuffle\_data(data,label):

#number of samples

num=data.shape[0]

seq=np.random.permutation(num)

return data[seq],label[seq]

train\_set,cross\_set,test\_set=divide\_data(dir\_raw\_data,dir\_label)

n\_data = example\_data.shape[1]

n\_steps=22

n\_inputs=n\_data//n\_steps

n\_data=n\_steps\*n\_inputs

n\_layers = 3

n\_neurons = 150

n\_outputs = 1

n\_epochs = 1000

batch\_size = 512

rnn\_dropout=0.5#dropout rate, 0 means no dropout

#start seting network

reset\_graph()

y = tf.placeholder(tf.float32, shape=(None, n\_outputs))

X = tf.placeholder(tf.float32, [None, n\_steps, n\_inputs])

training = tf.placeholder\_with\_default(1.0, shape=[], name='training')

keep\_rate=1-rnn\_dropout# this is the value used to feed into training, 1 menas no dropout

#recurrent network

lstm\_cells = [tf.contrib.rnn.DropoutWrapper(tf.contrib.rnn.BasicLSTMCell(num\_units=n\_neurons),output\_keep\_prob=training)

for layer in range(n\_layers)]

multi\_cell = tf.contrib.rnn.MultiRNNCell(lstm\_cells)

outputs, states = tf.nn.dynamic\_rnn(multi\_cell, 0, dtype=tf.float32)

top\_layer\_h\_state = states[-1][1]

logits = tf.contrib.layers.fully\_connected(top\_layer\_h\_state, n\_outputs, activation\_fn=None)

xentropy=tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits = logits, labels = y)

loss = tf.reduce\_mean(xentropy)

optimizer = tf.train.AdamOptimizer()

training\_op = optimizer.minimize(loss)

pred=tf.cast(logits>0,"int32")

# Test model

correct\_prediction = tf.equal(tf.cast(logits>0,"float32"), y)

# Calculate accuracy

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

#test\_with\_restore\_data()

def train\_and\_save\_network():

best\_acc = 0

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

#train on all training data and get training accuracy

np.random.shuffle(train\_set)

print("training on ", [d.split("/")[-1] for d, l in train\_set[:5]])

whole\_data = np.concatenate([get\_data(x[0]) for x in train\_set[:5]], axis=0)

whole\_data = whole\_data[:, :n\_data]

whole\_data = whole\_data.reshape(-1, n\_steps, n\_inputs)

whole\_label = np.concatenate([get\_label(x[1]) for x in train\_set[:5]], axis=0)

training\_acc = {}

cross\_acc = {}

epoch\_data, epoch\_label = shuffle\_data(whole\_data, whole\_label)

for i in range(epoch\_data.shape[0] // batch\_size):

X\_batch = epoch\_data[i:i + batch\_size]

y\_batch = epoch\_label[i:i + batch\_size]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch, training: keep\_rate})

if i%100==0:print(i," batches trained")

print("loss:", loss.eval(feed\_dict={X: X\_batch, y: y\_batch}))

for d,l in train\_set[:5]:

data=get\_data(d)[:, :n\_data]

data=data.reshape(-1, n\_steps, n\_inputs)

training\_acc[d.split("/")[-1]]=accuracy.eval({X: data, y: get\_label(l)})

avg\_training\_acc = sum(training\_acc.values()) / len(training\_acc)

print("epoch:", epoch, "training Accuracy:", training\_acc, "\n average is :", avg\_training\_acc)

#testing

for d, l in cross\_set:

cross\_data = get\_data(d)

cross\_data=cross\_data[:, :n\_data]

cross\_data = cross\_data.reshape(-1, n\_steps, n\_inputs)

cross\_label = get\_label(l)

cross\_acc[d.split("/")[-1]] = (accuracy.eval({X: cross\_data, y: cross\_label}))

avg\_acc = sum(cross\_acc.values()) / len(cross\_acc)

print("epoch:", epoch, "Cross Accuracy:", cross\_acc, "\n average is :", avg\_acc)

if avg\_acc > best\_acc:

best\_acc = avg\_acc

saver.save(sess, save\_path)

print("Network saved")

saver.save(sess, save\_path\_latest)

print("max training acc is", best\_acc)

train\_and\_save\_network()

1. “Normalize\_Audio.py”

import scipy.io.wavfile

import numpy as np

import pydub

import os

from pydub import AudioSegment

def match\_target\_amplitude(sound, target\_dBFS):

change\_in\_dBFS = target\_dBFS - sound.dBFS

return sound.apply\_gain(change\_in\_dBFS)

dir\_raw\_data="./data/Sound\_Files/"

dir\_normalized\_data="./data/Normalized\_Sound\_Files/"

os.makedirs(os.path.dirname(dir\_normalized\_data), exist\_ok=True)

for (root\_d, dirs\_d, files\_d) in os.walk(dir\_raw\_data):

for d in files\_d:

a, b = d.split(".")

sound = AudioSegment.from\_file(root\_d+d)

normalized\_sound = match\_target\_amplitude(sound, -20.0)

normalized\_sound.export(dir\_normalized\_data+"Normalized\_"+a+".wav", format="wav")

1. “Spectrogram\_Generator.py”

from scipy import signal

import numpy as np

import pickle

import os

import Load\_Audio\_Data

#compute spectomgram

def \_compute\_spectrungram(data,fs):

sg = []

for i in range(data.shape[0]):

f, t, Sxx = signal.spectrogram(data[i], fs)

sg.append(Sxx)

return np.array(sg)

#precompute spectrumgram for the training data set

#the demision of returned matrix is (num\_of\_chuncks,height, width,channels)

def convert\_to\_spectrogram(dir\_source,dir\_out,fs,output\_channel,chunk\_time=0.1, down\_sample=False,down\_sample\_rate=4):

os.makedirs(os.path.dirname(dir\_out), exist\_ok=True)

for (root\_d, dirs\_d, files\_d) in os.walk(dir\_source):

for d in files\_d:

a,b=d.split(".")

(data\_1, data\_2) = Load\_Audio\_Data.get\_data(root\_d + d,chunk\_time=chunk\_time, down\_sample=down\_sample,down\_sample\_rate=down\_sample\_rate)

spectrogram\_1 = \_compute\_spectrungram(data\_1, fs)

spectrogram\_2 = \_compute\_spectrungram(data\_2, fs)

if output\_channel==1:

pickle.dump(np.concatenate((spectrogram\_1,spectrogram\_2),axis=0), open(dir\_out + a + ".p", "wb"))

else:

spectrogram\_mix=np.array((spectrogram\_1,spectrogram\_2))

#now the dimision of spectrogram\_mix is

#(channels, num\_of\_chuncks,height,width)

spectrogram\_mix=np.swapaxes(spectrogram\_mix,0,3)

# now the dimision of spectrogram\_mix is

# (width, num\_of\_chuncks,height,channels)

spectrogram\_mix = np.swapaxes(spectrogram\_mix, 0, 2)

# now the dimision of spectrogram\_mix is

# (height, num\_of\_chuncks,width,channels)

spectrogram\_mix = np.swapaxes(spectrogram\_mix, 0, 1)

# now the dimision of spectrogram\_mix is

# (num\_of\_chuncks,height, width,channels), this is desired

pickle.dump(spectrogram\_mix, open(dir\_out+a+".p", "wb"))

1. “CNN\_1channel\_2classes.py”

#just put all data and labes in below, then run this file

#dir\_raw\_data="./data/Sound\_Files/"

#dir\_label="./data/Cleaned\_Labels/"

import Spectrogram\_Generator

import Labels

import numpy as np

import os

import pickle

import tensorflow as tf

#settings, pathes

np.random.seed(2)

chunk\_time=0.1

down\_sample=True

down\_sample\_rate=4

rate=44100

fs=rate/down\_sample\_rate

dir\_raw\_data="./data/Sound\_Files/"

dir\_normalized\_data="./data/Normalized\_Sound\_Files/"

dir\_raw\_data=dir\_normalized\_data#Use normalized data instead of raw data

dir\_spec\_data="./data/Spectrograms\_1channel-fs-"+str(int(fs))+"/"

dir\_label="./data/Cleaned\_Labels/"

#save pathes

save\_path="./Alg6\_1channel/ALg6\_best\_acc.ckpt"

os.makedirs(os.path.dirname(save\_path), exist\_ok=True)

save\_path\_latest="./Alg6\_1channel/ALg6\_latest\_acc.ckpt"

os.makedirs(os.path.dirname(save\_path\_latest), exist\_ok=True)

#load spectrogram

#the demision of returned matrix is (num\_of\_chuncks,height, width,channels)

def get\_data(filename):

return pickle.load( open(filename, "rb" ) )

#overload Labels.get\_labels for dimension problem, and expand the labels to 4 classes

#0: no speech 1: speaker\_1(speaker of channel 1) 2:speaker\_2 3: overlap of two speaker

def get\_label(filename):

l1,l2=Labels.get\_labels(filename, chunk\_time=chunk\_time)

return np.concatenate((l1,l2),axis=0)

#divide whole data set into train, cross and test

#return 3 lists of file path for each group, each element in a list is (data, label)

def divide\_data(dir\_data,dir\_label):

for (root\_d, dirs\_d, files\_d), (root\_l, dirs\_l, files\_l) in zip(os.walk(dir\_data), os.walk(dir\_label)):

full\_files\_d=[root\_d+x for x in files\_d]

full\_files\_l = [root\_l + x for x in files\_l]

pairs=list(zip(full\_files\_d,full\_files\_l))

np.random.shuffle(pairs)

num=len(pairs)

train=pairs[:int(num\*0.5)]

cross=pairs[int(num\*0.5):int(num\*0.75)]

test=pairs[int(num\*0.75):]

return train,cross,test

# to shuffle data in each eapoch

def shuffle\_data(data,label):

#number of samples

num=data.shape[0]

seq=np.random.permutation(num)

return data[seq],label[seq]

# to reset graph

def reset\_graph(seed=1):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

#generate spectrogram if not exist

if not os.path.isdir(dir\_spec\_data):

Spectrogram\_Generator.convert\_to\_spectrogram(dir\_raw\_data,dir\_spec\_data,

fs,output\_channel=1,chunk\_time=chunk\_time, down\_sample=down\_sample,down\_sample\_rate=down\_sample\_rate)

#make train, cross, test sets, each of them is a list of (data,label)

train\_set,cross\_set,test\_set=divide\_data(dir\_spec\_data,dir\_label)

#pick one data file to get demisions

example\_data=get\_data(train\_set[0][0])

print(example\_data.shape,get\_label(train\_set[0][1]).shape)

height=example\_data.shape[1]

width=example\_data.shape[2]

channels=1

print("hight is ",height,"width is ",width)

#set CNN network

conv1\_fmaps = 32

conv1\_ksize = 2

conv1\_stride = 1

conv1\_pad = "SAME"

conv2\_fmaps = 64

conv2\_ksize = 2

conv2\_stride = 1

conv2\_pad = "SAME"

conv2\_dropout\_rate = 0.2

pool3\_fmaps = conv2\_fmaps

n\_fc1 = 128

fc1\_dropout\_rate = 0 # dropout rate to tune

n\_outputs = 1

reset\_graph()

with tf.name\_scope("inputs"):

X = tf.placeholder(tf.float32, shape=[None, height, width], name="X")

X\_reshaped = tf.reshape(X, shape=[-1, height, width, channels])

y = tf.placeholder(tf.float32, shape=(None))

training = tf.placeholder\_with\_default(False, shape=[], name='training')

conv1 = tf.layers.conv2d(X\_reshaped, filters=conv1\_fmaps, kernel\_size=conv1\_ksize,

strides=conv1\_stride, padding=conv1\_pad,

activation=tf.nn.relu, name="conv1")

conv2 = tf.layers.conv2d(conv1, filters=conv2\_fmaps, kernel\_size=conv2\_ksize,

strides=conv2\_stride, padding=conv2\_pad,

activation=tf.nn.relu, name="conv2")

with tf.name\_scope("pool3"):

pool3 = tf.nn.max\_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="VALID")

pool3\_flat = tf.reshape(pool3, shape=[-1, pool3\_fmaps \* (height//2) \* (width//2)])

pool3\_flat\_drop = tf.layers.dropout(pool3\_flat, conv2\_dropout\_rate, training=training)

with tf.name\_scope("fc1"):

fc1 = tf.layers.dense(pool3\_flat\_drop, n\_fc1, activation=tf.nn.relu, name="fc1")

fc1\_drop = tf.layers.dropout(fc1, fc1\_dropout\_rate, training=training)

with tf.name\_scope("output"):

logits = tf.contrib.layers.fully\_connected(fc1\_drop, n\_outputs, activation\_fn=None)

with tf.name\_scope("train"):

xentropy = tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=logits, labels=y)

loss = tf.reduce\_mean(xentropy)

optimizer = tf.train.AdamOptimizer(learning\_rate=0.01)

training\_op = optimizer.minimize(loss)

with tf.name\_scope("eval"):

correct\_prediction =tf.equal(tf.cast(logits > 0, "float32"), y)

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

pred = tf.cast(logits > 0, "int32")

with tf.name\_scope("init\_and\_save"):

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

n\_epochs = 1000

batch\_size = 256

def train\_and\_save\_network():

best\_acc = 0

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

#train on training set

#in each epoch, randomly read 5 files

np.random.shuffle(train\_set)

#print("training on ",[d.split("/")[-1] for d,l in train\_set[:5]])

whole\_data = np.concatenate([get\_data(x[0]) for x in train\_set[:5]], axis=0)

whole\_label = np.concatenate([get\_label(x[1]) for x in train\_set[:5]], axis=0)

training\_acc = {}

cross\_acc = {}

epoch\_data,epoch\_label = shuffle\_data(whole\_data,whole\_label)

#epoch\_data, epoch\_label=whole\_data,whole\_label

for i in range(epoch\_data.shape[0] // batch\_size):

X\_batch = epoch\_data[i:i + batch\_size]

y\_batch = epoch\_label[i:i + batch\_size]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch, training: True})

#if i%100==0:print(i," batches trained")

current\_loss=loss.eval(feed\_dict={X: X\_batch, y: y\_batch})

#print("loss:", current\_loss)

for d,l in train\_set[:5]:

training\_acc[d.split("/")[-1]]=accuracy.eval({X: get\_data(d), y: get\_label(l)})

avg\_training\_acc = sum(training\_acc.values()) / len(training\_acc)

#print("epoch:", epoch, "training Accuracy:", training\_acc, "\n average is :", avg\_training\_acc)

#check cross accuracy

for d,l in cross\_set:

cross\_data = get\_data(d)

cross\_label = get\_label(l)

cross\_acc[d.split("/")[-1]] = (accuracy.eval({X: cross\_data, y: cross\_label}))

avg\_cross\_acc = sum(cross\_acc.values()) / len(cross\_acc)

#print("epoch:", epoch, "Cross Accuracy:", cross\_acc, "\n average is :", avg\_acc)

print("epoch:", epoch, "training Accuracy:",avg\_training\_acc,"Cross Accuracy:", avg\_cross\_acc,"Loss:", current\_loss)

if avg\_cross\_acc > best\_acc:

best\_acc = avg\_cross\_acc

saver.save(sess, save\_path)

#print("Network saved")

saver.save(sess, save\_path\_latest)

print("max training acc is", best\_acc)

train\_and\_save\_network()

Again “CNN\_2channels\_4classes.py” is very similar to “CNN\_1channels\_2classes.py” and doesn’t provide a good result so far. Can be provided through email if needed.