Audio Tagging Challenge

Individual Final Report

Jingjing Xu - April 27, 2019

Introduction

The dataset we obtained from Zenodo is called the “Medley-solos” dataset, which is about musical instrument recognition. There are 8 classes in this dataset. Each contained the audio of a single instrument playing, and was 3 seconds long.

We wrote the python code using the Pycharm framework, and employed NumPy and SciPy.io to extract and process audio information. We decided to use the LSTM network, since the audio was based on time series. The most difficult part was preprocessing the data.

Outline of the Shared Work:

1. Split csv data into test, train, validation sets;
2. Load audio file and extract information linked to the corresponding csv cell;
3. Establish the model;
4. Train the model with train set;
5. Test the model on the validation set and do adjustments on the optimizers, layer size, number of neurons, loss functions, etc;
6. Determine the best combination, and test it finally on the test set.

Individual Work

I split the .csv data into test, train, validation sets, and extracted the audio features in both time and frequency domain using wavfile and librosa. I modified the model using LSTMs, and wrote the epoch loop to train the model, as well as the validation set test algorithm by making adjustments to the code in exam 1, and the visualization by confusion matrix. Also, I created and co-edited the github repo page, and wrote the README file.

The main reason why I decided to use librosa is because it extracts very useful information, for instance, MFCCs, which applies the Fourier Transform and different filters on the raw audio data. Since the audio data involves time series, I suggested using LSTMs network after we encountered it in class. Hence I wrote it in the nn module using Pytorch framework.

Description of Individual Work

The only .csv file we had came with all the test, train, validation sets together, with parts of the name of the corresponding file in different column. We weren't sure if we should extract all the information from the audio file, and then add them into the csv file, or to design a class to load the corresponding audio file when we needed them.

subset	instrument	instrument_id	track_id	uuid4
test	clarinet	0	000	0e4371ac-1c6a-51ab-fdb7-f8abd5fbf1a3
test	clarinet	0	000	33383119-fd64-59c1-f596-d1a23e8a0eff
test	clarinet	0	000	b2b7a288-e169-5642-fced-b509c06b11fc
test	clarinet	0	000	151b6ee4-313a-58d9-fbcb-bab73e0d426b
test	clarinet	0	000	b43999d1-9b5e-557f-f9bc-1b3759659858
test	clarinet	0	000	0e5669a9-9f3c-5fa3-fc80-44dc905d16f2
test	clarinet	0	000	3befab3f-a98f-54f1-fb82-9b42d8db3b9b

Figure 1. Mega CSV

We tried both ways. Madison started putting together separate csv files for each set with the audio information extracted with wavfile, while I tried to get all the audio data into a matrix with each row corresponding to one clip, and each column corresponding to one entry of the audio array. But the matrix took a very long time to load, so I tried to approach it the other way:

```
# ----------------------------------------------------------------------
#Process Dataset

class MedleyDataset(Dataset):
    def __init__(self, dataset_csv, transform=None):
        self.dataset_frame = dataset_csv
        self.transform = transform

    def __len__(self):
        return len(self.dataset_frame)

    def __getitem__(self, index):
        uuid4 = self.dataset_frame.iloc[index, 4]
        instrument_list = self.dataset_frame.iloc[index, 2]
        instrument_id = int(instrument_list)
        link = load_file(uuid4)

        audio = audio.astype('float')

        sample = {'audio': audio, 'label': instrument_id}

        if self.transform:
            sample = self.transform(sample)

        return sample

audio_dataset_train = MedleyDataset(dataset_csv= train)
audio_dataset_validation = MedleyDataset(dataset_csv= validation)
audio_dataset_test = MedleyDataset(dataset_csv= test)
```

Figure 2. Dataset Processing

My "MedleyDataset" class reads train/test/validation subset of the original csv file in __init__, and extract the corresponding audio clip array with __getitem__. The class returns the sample value with audio array and label of instrument in the form {'audio': audio, 'label': label}. While doing that, I rewrote some lines of Madison's full_name definition to get the file name, and added a find_file definition to open the corresponding clip I previously uploaded to my GCP. I also wrote a class_loader definition to get the instrument id given a known index in the csv file, but it seemed unnecessary in the

end.

```
#-----------------------------------------------------------------------------------------
#Load audio file linked to the uuid
def full_name(file):
    correspding_row = Medley.loc[Medley['uuid4'] == file].iloc[0]
    subset = str(correspding_row.loc['subset'])
    instrument_id = str(correspding_row.loc['instrument_id'])
    parts = ['Medley-solos-DB_', str(subset), '-', str(instrument_id), '_', file, '.wav.wav']
    s = ''
    file_name = s.join(parts)
    return file_name

def find_file(file):
    file_name = full_name(file)
    path = '/home/ubuntu/Machine-Learning/Medley-solos-DB/'
    parts = [path, file_name]
    s = ''
    link = s.join(parts)
    return link
```

Figure 3. Getting Fullname and Finding the File (Madison co-wrote it)

Then I used the Pytorch built in DataLoader to divide the dataset loaded to mini batches:

```
train_loader = torch.utils.data.DataLoader(dataset=audio_dataset_train, batch_size=batch_size, shuffle=True)
validation_loader = torch.utils.data.DataLoader(dataset=audio_dataset_validation, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset = audio_dataset_test, batch_size = batch_size, shuffle = True)
```

Figure 4. DataLoader

I modified the exam 1 training loop to incorporate the current dataset, but I keep getting errors of the data type is different than expected, so I changed then before and after putting them to model in order to get the loop running. I also added .cuda() to make it run on GPU (I keep getting complaints from the compiler asking me to buy more RAM while using CPU, perhaps the storage space is not enough).

```
for epoch in range(num_epochs):
    for i, data in enumerate(train_loader):

        model.zero_grad()

        audios = data['audio']
        labels = data['label']

        audios = audios.type(torch.FloatTensor)
        audios = Variable(audios.cuda())

        output = model(audios)

        labels = labels.type(torch.LongTensor)
        labels = Variable(labels.cuda())

        loss = criterion(output, labels).cuda()
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()


        if (i + 1) % 10 == 0:
            print('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'
                  % (epoch + 1, num_epochs, i + 1, len(train) // batch_size, loss.data[0]))
```

Figure 5. Training Loop

For the extraction of the audio files, Madison originally used wavfile, while I used librosa to extract the frequency domain information. I switched to hers since wavfile seemed to take less time. While designing the LSTM model, we used the nn.module as in exam 1, Madison found the base code, but we each designed one separately. I used the audio and label data I got from my “MedleyDataset” to train the model. In order to get the dimensions right, I used the Pytorch built in batch DataLoader, and started to run the program with one layer of nn.Linear first, as LSTM requires one more dimension of input. Once it trained smoothly, I started to modify the model to fit in the LSTM layer.

Since my input has the dimension of batch_size x input_size, I first tried to divide the input_size in to a product of 256 x 512 as I read in an article that LSTM behaved better if the input size is around 200-400. I was wrong. The program didn't seem to be training, as I only achieved around 27% overall accuracy. As I printed in my prediction, it seemed like the program always thought that everything was a class 7-violin.

So I re-examined my model. It occured to me that since I had to adjust the 3-dimensional output of LSTM layer back to 2-dimensions, could I simply drop one dimension, however and it seemed that I dropped too much information in that step. So I redivided the input_size to 1 x 131072, and this time it seemed to be learning much better. I also set the layer number to 2 in the LSTMs to apply the dropout, since it doesn't work on the last layer. 1 epoch gave me around 37% overall accuracy.

```
# ------------------------------------------------------------------------------------------
# LSTM Model

class Net(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, num_classes):
        super(Net, self).__init__()
        self.LSTM1 = nn.LSTM(input_size, hidden_size1, batch_first=True, num_layers=2, dropout=0.1).cuda()
        # self.LSTM2 = nn.LSTM(hidden_size1, hidden_size2, batch_first = True).cuda()
        self.lstm2tag = nn.Linear(hidden_size1, num_classes).cuda()

    def forward(self, x):
        #print(x.shape)
        s = x.shape[0]
        x = x.reshape(s, 1, input_size)
        out, states = self.LSTM1(x)
        # out, states = self.LSTM2(out)
        out = out[:, 0, :]
        out = self.lstm2tag(out)
        return out
```

Figure 6. LSTMs Model

To achieve better results, I added another LSTM layer, but 100 epochs gave only 28% overall accuracy. Something didn't seem right, so I reexamined the algorithm, and think it's that either the size of the training set is too small (we have 5841 out of 21571 total), or that it might be because the data we were using is not that useful at all, or it could be possible that the algorithm is overfitting. I tried batch sizes of 30, 50, and 100, and adjusted the dropout rate of the LSTMs model, however it did not seem to make a significant difference, so I tried to add more data into training set, yet it still didn't work that well.

At this point that I turned back to the frequency domain data extracted with librosa. Theoretically it should contain more useful information. I picked MFCCs, spectral centroid and spectral contrast data because judging by the classes (clarinet, violin, saxophone, electric guitar, etc), they should have different frequencies, "brightness" and "pitch". I revised my algorithm to extract these information, and ran 1 epoch. It gave me 62% overall accuracy.

```
y, sr = librosa.load(link)

mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
spectral_center = librosa.feature.spectral_centroid(y=y, sr=sr)
spectral_contrast = librosa.feature.spectral_contrast(y=y, sr=sr)
```

Figure 7. Frequency Data Extraction

But the librosa runs very slowly. Therefore, I tried to boost up the speed by saving all the frequency domain data in to an ndarray, so that in the epoch period, we only need to load the ndarray, which would increase the speed of each epoch drastically.

My music_feature_loader.py seemed to work well, it takes a while to generate the ndarrays, but then boost up the speed of the program a lot, since all the extracted information is all stored in an .npy file, and the training loop no longer needs to extract data every iteration.

```
#-------------------------------------------------------------------------------------------
#Generating music data ndarray in frequency domain using librosa

def get_music_features(dataset):
    timeseries_length = 3
    audio = np.zeros((len(dataset), timeseries_length, 21), dtype=np.float64)

    for i in range(len(dataset)):
        row = dataset.loc[i]
        uuid4_name = str(row.loc['uuid4'])
        link = find_file(uuid4_name)

        y, sr = librosa.load(link)

        mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
        spectral_center = librosa.feature.spectral_centroid(y=y, sr=sr)
        spectral_contrast = librosa.feature.spectral_contrast(y=y, sr=sr)

        audio[i, :, 0:13] = mfcc.T[0:timeseries_length, :]
        audio[i, :, 13:14] = spectral_center.T[0:timeseries_length, :]
        audio[i, :, 14:21] = spectral_contrast.T[0:timeseries_length, :]


        if ((i + 1) % 100 == 0):
            print("Extracted features audio clip %i of %i." % (i + 1, len(dataset)))

    return audio

train_audio_data = get_music_features(train)
validation_audio_data = get_music_features(validation)
test_audio_data = get_music_features(test)

np.save('train_audio_data.npy', train_audio_data)
np.save('validation_audio_data.npy', validation_audio_data)
np.save('test_audio_data.npy', test_audio_data)
```

Figure 8. Generate and Save Music Feature Ndarray

To achieve better visualization, Madison wrote a “performance index over epoch” matlibplot code. But the figure doesn’t show on my Pycharm, so I rewrite it in order to save the figure, still no luck. Moreover, I wrote the code the confusion matrix information from the validation set testing.

```
    leng = predicted.shape[0]

    a = predicted.cpu().detach().numpy()
    b = labels.cpu().detach().numpy()


    for j in range(leng):
        k1 = a[j]
        k2 = b[j]
        confusion_m[k1,k2] += 1

#confusion matrix
confusion_m.astype(int)
ax = sns.heatmap(confusion_m, annot=True)
print(confusion_m)
```

Figure 9. Confusion Matrix

I have uploaded all the code I’ve written in my personal code folder.

Results

Time Domain

Madison majorly did the test in time Domain. She discovered that while training the network, the loss decreases from 1.5 to 0.01 over the number of epochs. Yet the overall accuracy is around 24%. Here is a figure we acquired:

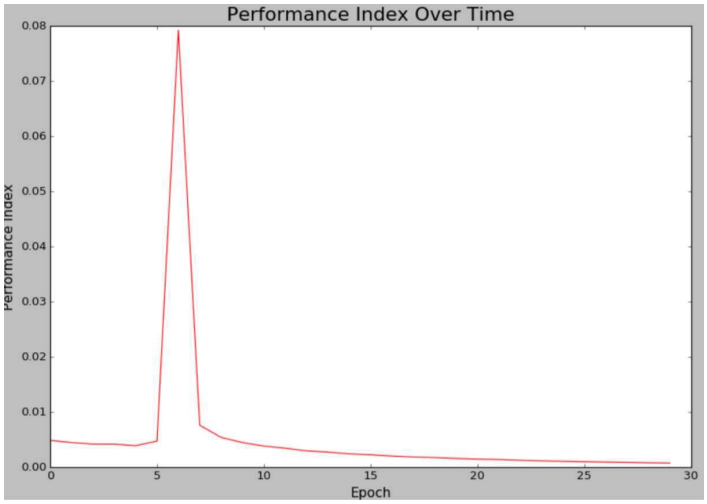

Figure 10. Performance Index over Epoch

The loss decreases steadily overall, but has a sudden boost around the 5-7 epochs, while the reason of the boost is unclear. The confusion matrix with each row and column represent an instrument id gives:

```
[0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0.]]
```

Figure 11. Confusion Matrix

This agrees the low accuracy result, yet it conflicts with the extremely low loss. The low accuracy could due to the fact that our train set is not big enough, or the data we extract is not that useful. We tried to eliminate the over and underfitting by adjusting the batch size and dropout rate, and get very similar results.

Frequency Domain

My frequency domain analysis achieved greatly improved outputs. This in part, confirmed that the low accuracy in time domain analysis was because we weren't extracting enough useful data for the network to train on. During the process of training the network, the loss started at 1.39, and decreased to 0.19 over 50 epochs and batch number 100. This resulted in an overall accuracy of 78% . Here is the loss of training time image, which appears to have a clear trend of decreasing:

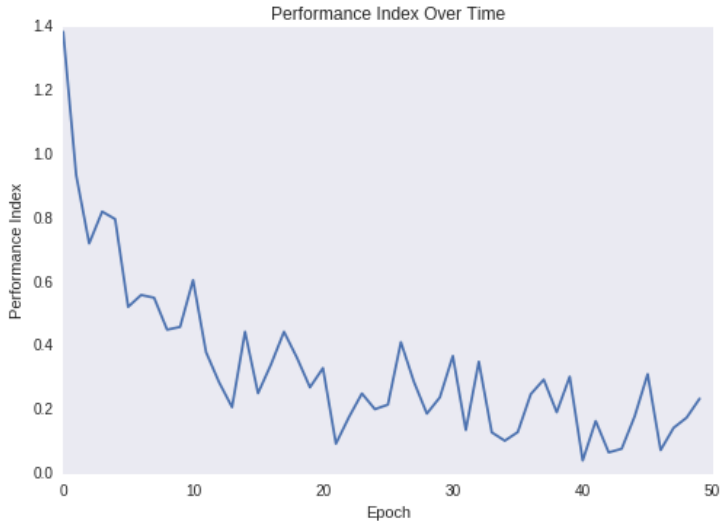

Figure 12-1. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2

The confusion matrix (with rows representing the prediction, columns representing the actual) is as follows:

```
Accuracy of the network on the 3494 validation audio clips: 78 %
Accuracy of clarinet : 48 %
Accuracy of distorted electric guitar : 83 %
Accuracy of female singer : 75 %
Accuracy of flute : 30 %
Accuracy of piano : 89 %
Accuracy of tenor saxophone : 24 %
Accuracy of trumpet : 84 %
Accuracy of violin : 86 %
```

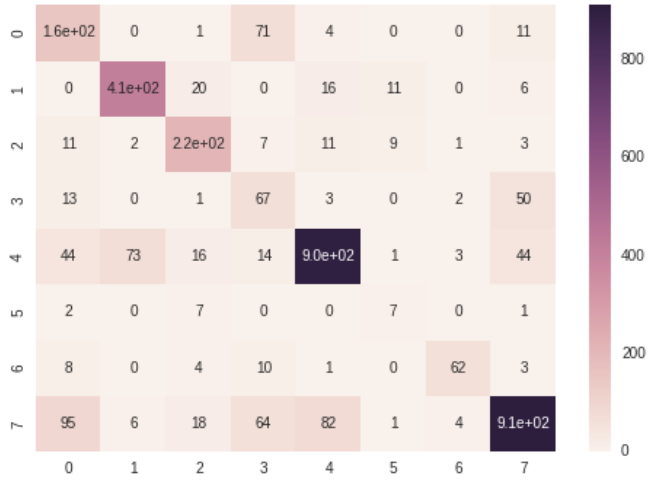

Figure 12-2. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2

From this it can be seen that the confusion matrix confirms the lower category accuracy on both the flute, and the saxophone. The network appears to be confusing flute with clarinet, and the saxophone with the guitar or female singer. This could result from either the frequency of those pairs being quite close, or it could possibly be the result of the fact that the number of clips within those categories are

much smaller than, say, the piano and violin clips. For better results, more data may be needed. To get the optimal training results from my network, we are using the following set of training parameters as my basis of comparison (epoch: 50, batch size: 100, learning : 0.0001, drop out: 0.1, LSTMs layer: 2), we then adjusted the different parameters to see how the code performed.

Batchsize:

Using the same epochs, I adjusted the batch size to 50, and got the following result:

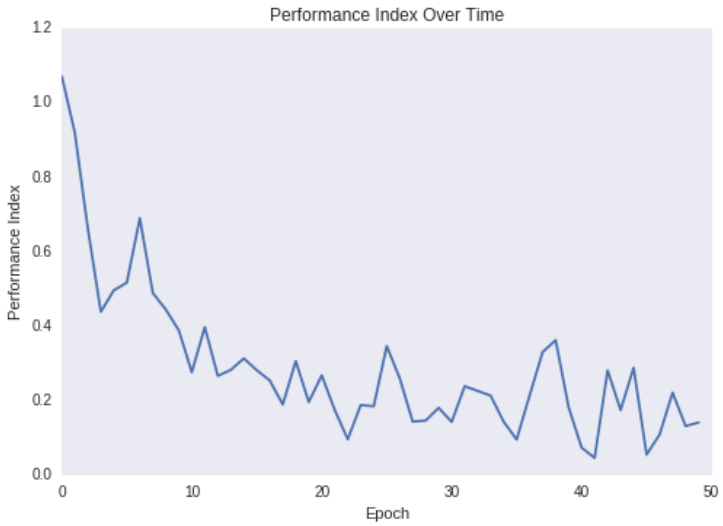

Figure 13-1. Epoch 50, Batch Size 50, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 80 %
Accuracy of clarinet : 52 %
Accuracy of distorted electric guitar : 84 %
Accuracy of female singer : 73 %
Accuracy of flute : 33 %
Accuracy of piano : 88 %
Accuracy of tenor saxophone : 24 %
Accuracy of trumpet : 84 %
Accuracy of violin : 90 %
```

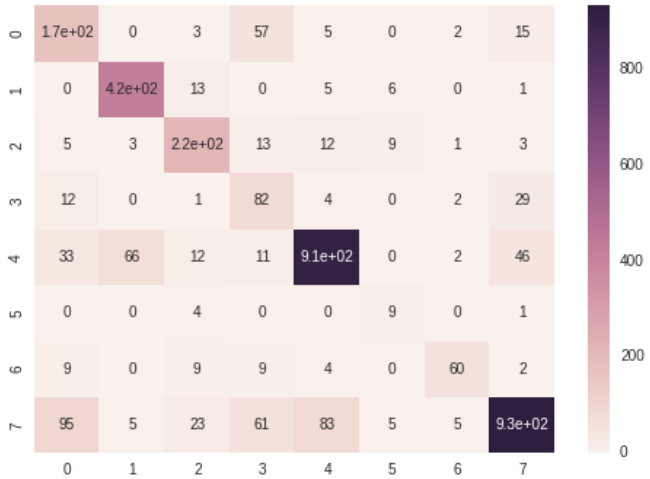

Figure 13-2. Epoch 50, Batch Size 50, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2

Then I changed the batch size to 20 and got:

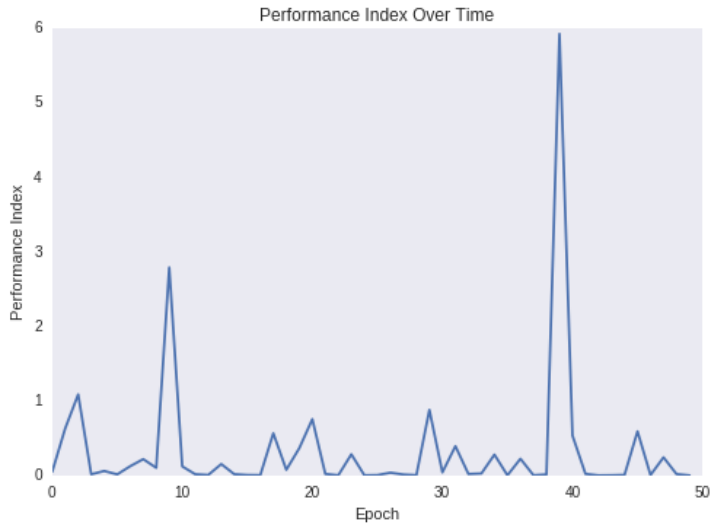

Figure 14-1. Epoch 50, Batch Size 20, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 78 %
Accuracy of clarinet : 44 %
Accuracy of distorted electric guitar : 71 %
Accuracy of female singer : 75 %
Accuracy of flute : 29 %
Accuracy of piano : 89 %
Accuracy of tenor saxophone : 17 %
Accuracy of trumpet : 77 %
Accuracy of violin : 92 %
```

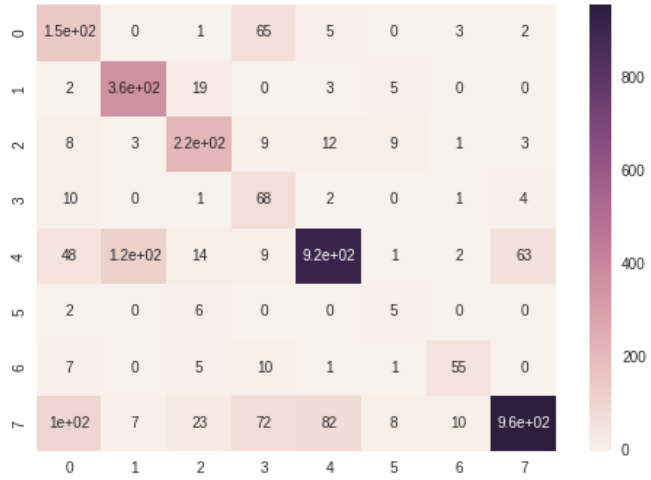

Figure 14-2. Epoch 50, Batch Size 20, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2

As can be seen, the performance index behaved very strangely when batch size was 20, it seemed to be constantly very low, however, there were a few spikes occurring, with the largest ones occurring around epochs 10 and 40. This could due to the fact that the batch size was too small, and therefore it affected

the training of the network, as the calculation of the gradient is closer to the individual file when batch size is smaller. The performance index when the batch size is 50 is quite similar to when it is 100, with a downward trend. It seems that the accuracy peaks in the middle, when the batch size equals 50, although the difference is not that significant. I believe this could be due to the fact that while using mini-batch, we are compromising between stochastic and gradient descent, and there would be a optimal point in the middle. I suppose 50 is closer to that optimal point.

Learning rate:

I now changed the batch size back to 100, and tried to adjust the learning rate to 0.001. We can observe that training loss changes from 0.88 to 0.40 over 50 epochs. We can see that the loss hovers over that number, this suggests that the learning rate is too high, and as a result, every step the weight and bias getting updated, they change too much.

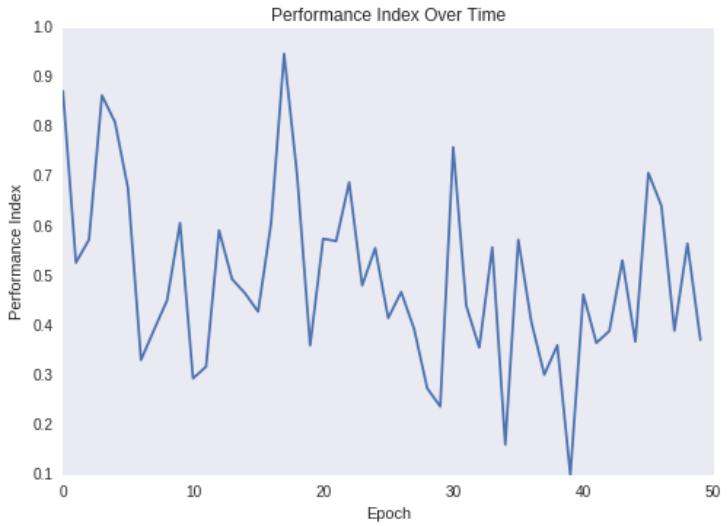

Figure 15-1. Epoch 50, Batch Size 100, Learning Rate 0.001, Dropout 0.1, Adam, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 75 %
Accuracy of clarinet : 35 %
Accuracy of distorted electric guitar : 79 %
Accuracy of female singer : 77 %
Accuracy of flute : 14 %
Accuracy of piano : 86 %
Accuracy of tenor saxophone : 17 %
Accuracy of trumpet : 69 %
Accuracy of violin : 93 %
```

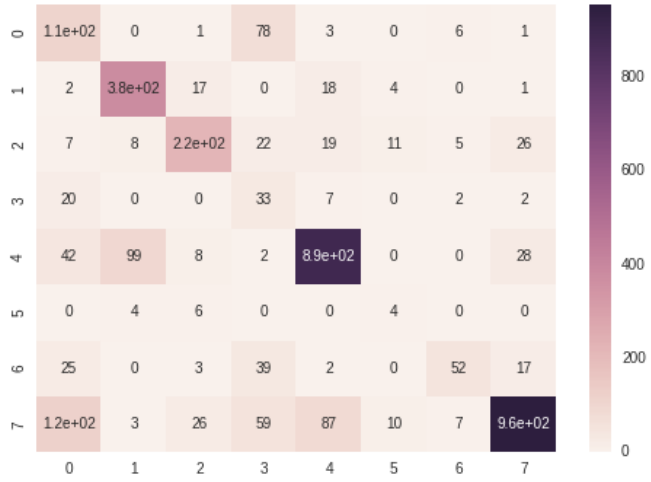

Figure 15-2. Epoch 50, Batch Size 100, Learning Rate 0.001, Dropout 0.1, Adam, LSTM Layer 2

The overall accuracy is at 75%, but it this could be random chance. Therefore I take the step of adjusting the learning rate to 0.00001. The following result suggests that the overall accuracy is at 67%, this is a little bit worse than the result we acquired from the learning rate of 0.001, and certainly not as good as the learning rate of 0.0001. The performance index seem to enter into a rapid oscillation after epoch 10 compare to the result of 0.001.

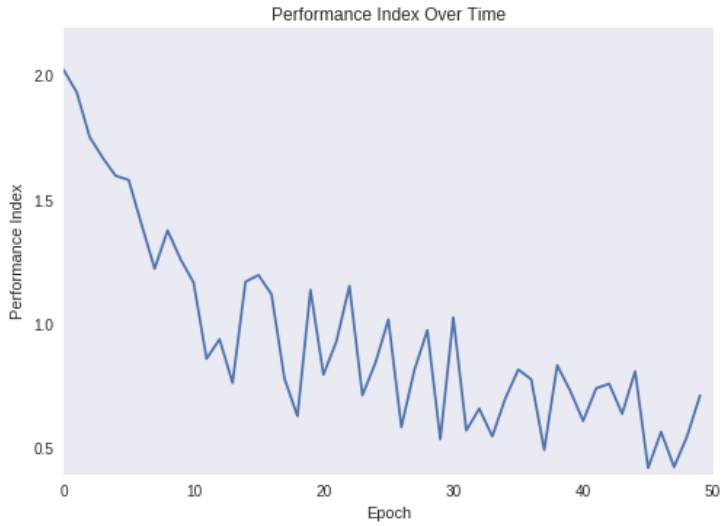

Figure 16-2. Epoch 50, Batch Size 100, Learning Rate 0.00001, Dropout 0.1, Adam, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 67 %
Accuracy of clarinet : 11 %
Accuracy of distorted electric guitar : 58 %
Accuracy of female singer : 45 %
Accuracy of flute :  0 %
Accuracy of piano : 88 %
Accuracy of tenor saxophone :  0 %
Accuracy of trumpet : 13 %
Accuracy of violin : 95 %
```

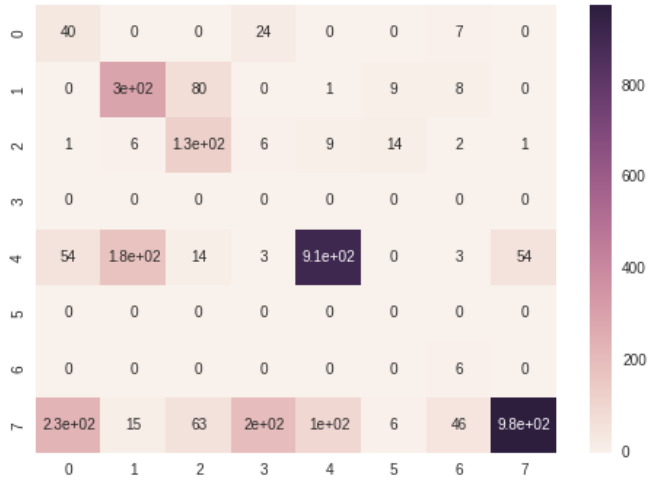

Figure 16-2. Epoch 50, Batch Size 100, Learning Rate 0.00001, Dropout 0.1, Adam, LSTM Layer 2

From this we can see that both learning too fast or too slow has an impact on our result; learning too fast we observe that there is a problem with over fitting, the loss hover around a range, and trouble decreasing; but with learning too slow, it doesn't reach the result as fast.

Dropout:

Till now I have been using the dropout rate 0.1, which I now change it to 0.05. The gives us the following result:

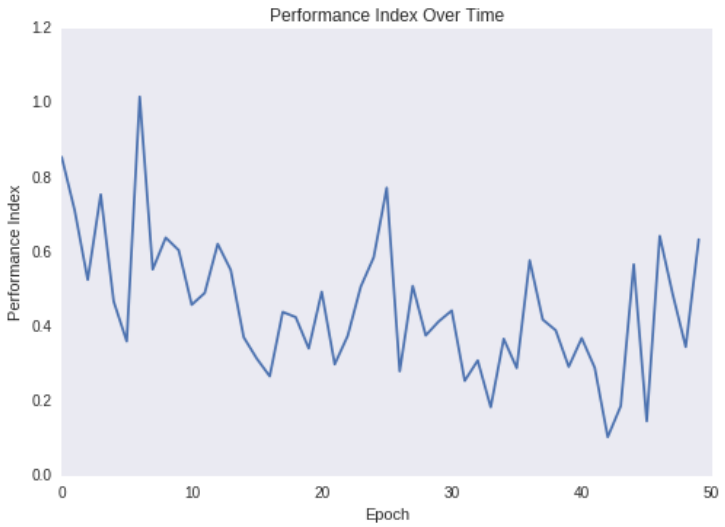

Figure 17-1. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.05, Adam, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 75 %
Accuracy of clarinet : 39 %
Accuracy of distorted electric guitar : 76 %
Accuracy of female singer : 61 %
Accuracy of flute : 27 %
Accuracy of piano : 85 %
Accuracy of tenor saxophone : 17 %
Accuracy of trumpet : 76 %
Accuracy of violin : 94 %
```

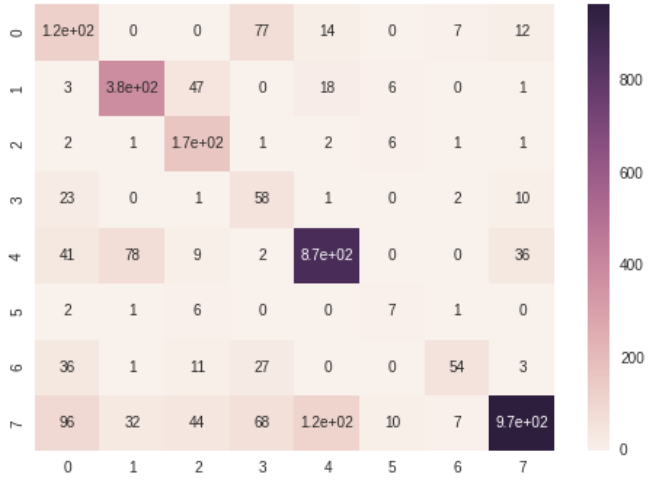

Figure 17-2. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.05, Adam, LSTM Layer 2

We can observe from the confusion matrix that the code seems to mistake the pair of flute and clarinet, saxophone and guitar/female singer, and that the network sometimes confuses clarinet with piano . This agrees with the lack of a significant trend in the performance index. But the overall accuracy is quite good, at 75%. Now have the results from changing the dropout rate to 0.5:

Figure 18-1. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.5, Adam, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 70 %
Accuracy of clarinet : 23 %
Accuracy of distorted electric guitar : 68 %
Accuracy of female singer : 50 %
Accuracy of flute : 15 %
Accuracy of piano : 86 %
Accuracy of tenor saxophone :  3 %
Accuracy of trumpet : 59 %
Accuracy of violin : 88 %
```

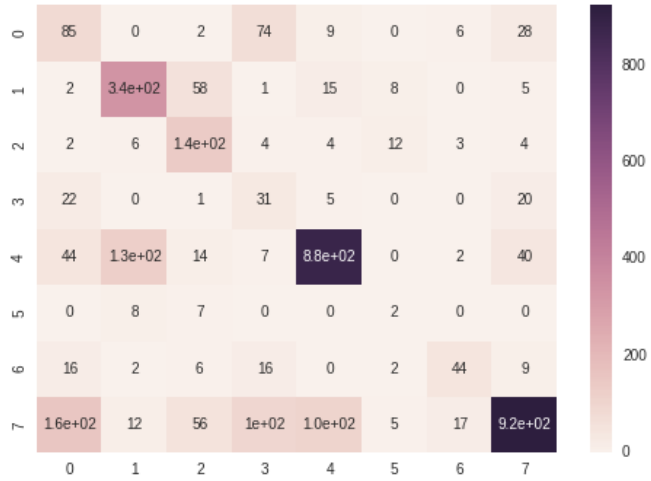

Figure 18-2. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.5, Adam, LSTM Layer 2

We can observe a slight decrease in the overall accuracy, at 70%, but the network behaves badly not only on flute and saxophone, but also several other classes. The only 2 classes providing good results are the piano and violin. Both of the dropout rates work worse than our dropout at 0.1. And if we take a look at the performance index images, we can see that there is no significant trend of loss throughout time. This might due to over or underfitting.

Optimizer:

I'm currently using Adam as my optimizer. Switching to SGD provides this result:

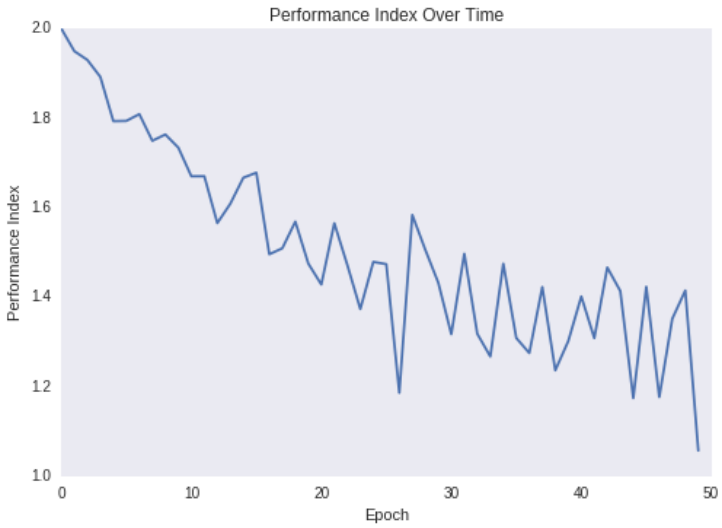

Figure 19-1. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, SGD, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 52 %
Accuracy of clarinet :  0 %
Accuracy of distorted electric guitar :  0 %
Accuracy of female singer :  0 %
Accuracy of flute :  0 %
Accuracy of piano : 87 %
Accuracy of tenor saxophone :  0 %
Accuracy of trumpet :  0 %
Accuracy of violin : 90 %
```

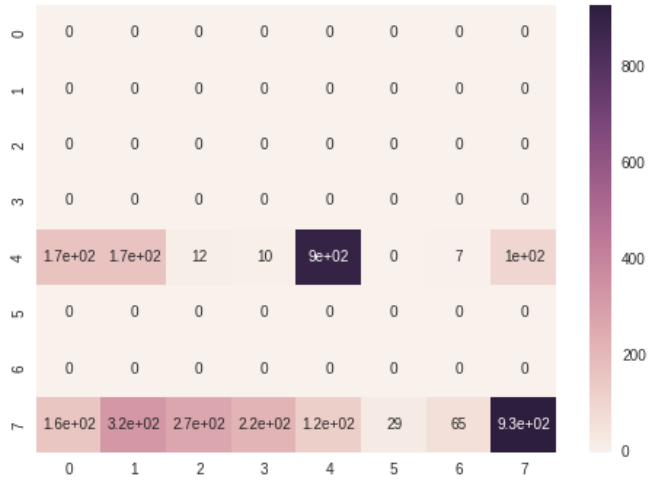

Figure 19-2. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, SGD, LSTM Layer 2

The overall accuracy is approximately 52%, and the confusion matrix suggests that the category of piano and violin are the most likely predicted. While the prediction result seems bad at first glance, we can however note there is a clear trend on the loss over time, it just vibrates at the end of the epochs. Adadelta optimizer with Rhode = 0.8 and eps = 1e-6 give:

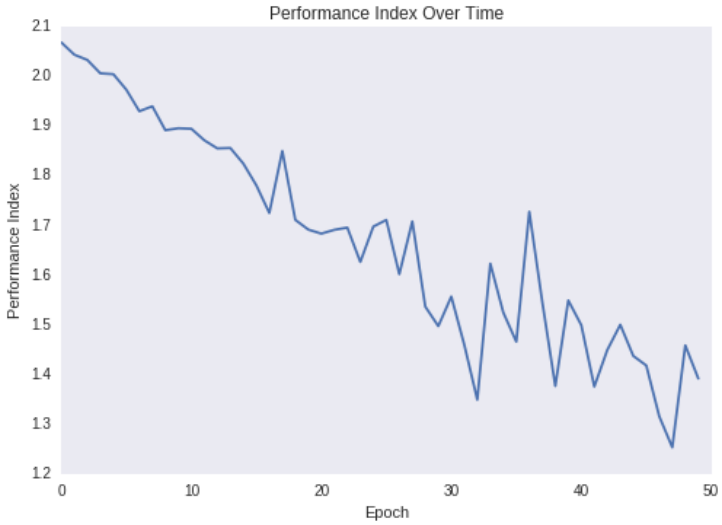

Figure 20-1. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, Adadelta, LSTM Layer 2

```
Accuracy of the network on the 3494 validation audio clips: 49 %
Accuracy of clarinet :  0 %
Accuracy of distorted electric guitar :  0 %
Accuracy of female singer :  0 %
Accuracy of flute :  0 %
Accuracy of piano : 93 %
Accuracy of tenor saxophone :  0 %
Accuracy of trumpet :  0 %
Accuracy of violin : 75 %
```

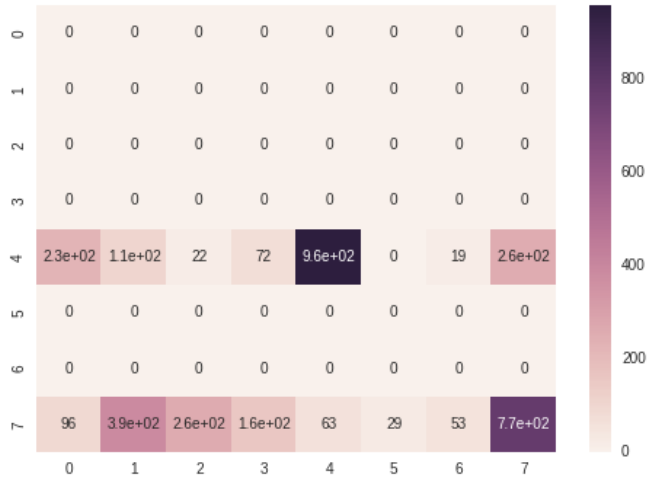

Figure 20-2. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, Adadelta, LSTM Layer 2

This produces 49% of overall accuracy, which is quite close to SGD optimizer. Still the network seems to think everything is either a piano or violin, although we have a clear trend of loss over time. We can see from the results that Adam outperforms SGD and Adadelta.

Extra Layer of LSTMs:

Now I'm adding another layer of LSTMs in the nn module. The result is as follows:

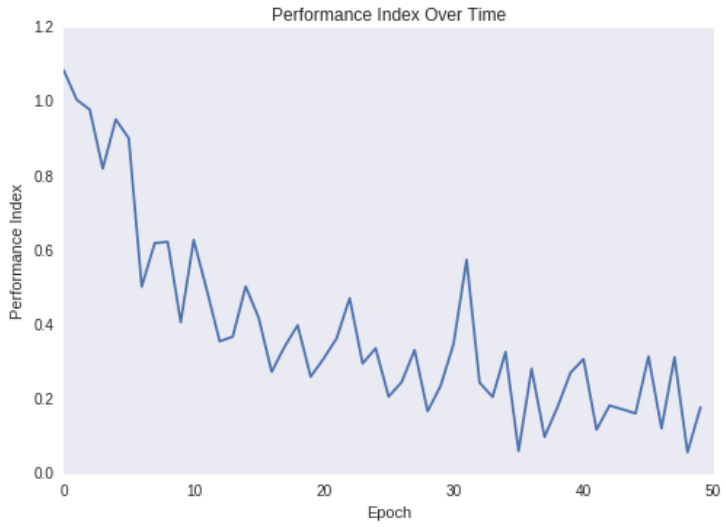

Figure 21-1. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2+1

```
Accuracy of the network on the 3494 validation audio clips: 79 %
Accuracy of clarinet : 45 %
Accuracy of distorted electric guitar : 92 %
Accuracy of female singer : 74 %
Accuracy of flute : 30 %
Accuracy of piano : 88 %
Accuracy of tenor saxophone : 55 %
Accuracy of trumpet : 79 %
Accuracy of violin : 91 %
```

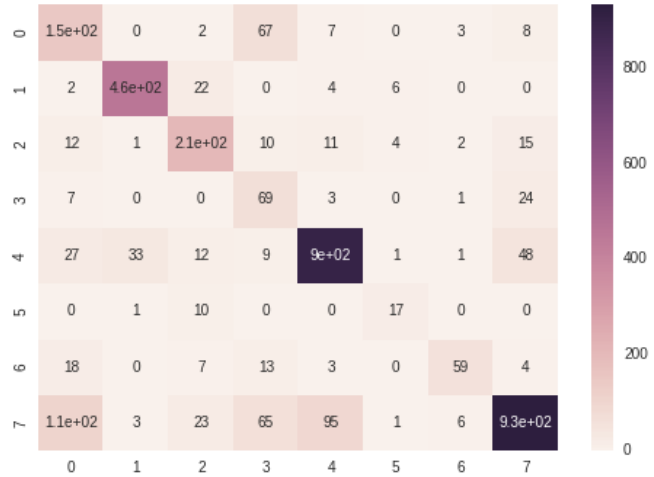

Figure 21-2. Epoch 50, Batch Size 100, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2+1

From the result we can see that the extra layer of LSTMs has an overall accuracy of 79%. The

performance index oscillates similarly to our comparison parameters. The addition of the LSTMs layer does not change the result much. I now use the previously tested optimal training parameters: batch size =50, training rate = 0.0001, dropout = 0.1, optimizer = Adam, to predict the test set, and get the following result:

```
Accuracy of the network on the 3494 validation audio clips: 53 %
Accuracy of clarinet : 29 %
Accuracy of distorted electric guitar : 83 %
Accuracy of female singer : 69 %
Accuracy of flute :  6 %
Accuracy of piano : 98 %
Accuracy of tenor saxophone :  6 %
Accuracy of trumpet : 50 %
Accuracy of violin : 56 %
```

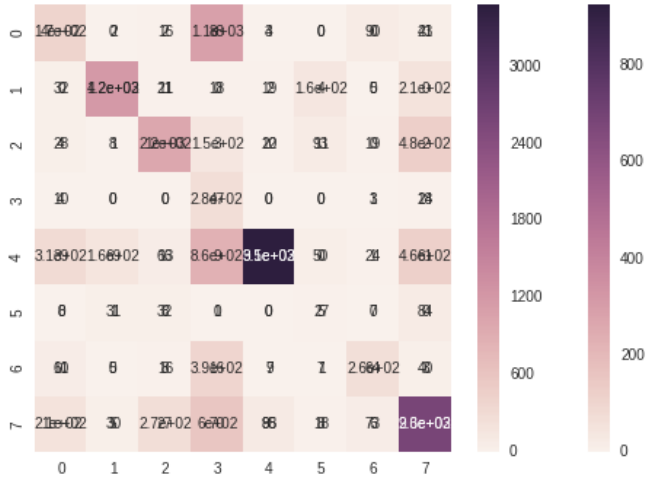

Figure 22. Epoch 50, Batch Size 50, Learning Rate 0.0001, Dropout 0.1, Adam, LSTM Layer 2; Test Set

We see that the overall accuracy for the test set is 53%, which is not as high as the validation set. Again, from the confusion matrix (which should be noted that it's in the scientific mode) we know the major problem is that the network has trouble distinguishing the flute and clarinet pair, and the saxophone and guitar/female singer pair.

Summary and Conclusions

From the result, we can see that the change of the training parameters changes the performance

of the network drastically. We need to find the optimal set of parameters to prevent issues like over or underfitting, learning too fast or slow, and find compromise between accuracy and running time.

I didn't get great results on the test set. This is majorly due to the fact that the network keeps predict flute as clarinet, and saxophone asguitar or female singer. Since this behavior appears in all of the results of validation set as well. I think a better set of useful data extraction, and also a class wise more balanced distributed train set is needed.

Both of the time domain data extraction and analysis algorithm runs very fast, but gives very low accuracy, around 30%, give or take 5% no matter how we adjust the train set size, batch size, optimizer, and dropout rate.

I suppose this is due to the lack of the useful data. The frequency domain data extraction is significantly slower, but the overall accuracy is much higher. I think in the future, we can make improvements by boosting up the train set size to about 60-70% of the entire dataset for better learning. We could also extract more useful information from the audio clips with librosa for instance, chrome, to increase the performance. Furthermore, we could also try to predict music using LSTMs by writing a loop of time in the nn module.

Percentage of Copied Code

The percentage of the code we found from the internet is (92-57)/(92+75) *100% = 20.9%.

References

1. Time domain and frequency domain. Retrieved from https://m.youtube.com/watch?v=tMPDe7z7ERE;
2. Nair, Pratheeksha. (2018). The dummy's guide to MFCC. Retrieved from https://medium.com/prathena/the-dummys-guide-to-mfcc-aceab2450fd;
3. Lostanlen, Vincent; Cella, Carmine-Emanuele; Bittner, Rachel; Essid, Slim., Medley solos-DB: a cross-collection dataset for musical instrument recognition. Retrieved from https://zenodo.org/record/2582103#.XMTeI4opDmq;
4. Jafari, Amir., FasionMINST Project. Retrieved from https://github.com/amir-jafari/Deep-Learning/blob/master/Pytorch_/Mini_Project/FashionMNIST.py;
5. Shaikh, Faizan., Getting Started with Audio Data Analysis using Deep Learning (with case study). Retrieved from https://www.analyticsvidhya.com/blog/2017/08/audio-voice-processing-deep-learning/;
6. Hagan, Matin T., Neural Network Design, Chapter 25: Case Study 3: Pattern Recognition;
7. uohoruotsi., Music genre classification with LSTM Recurrent Neural Nets. Retrieved from https://github.com/ruohoruotsi/LSTM-Music-Genre-Classification;
8. Guthrie, Robert., Sequence Models and Long-Short Term Memory Networks. Retrieved from https://seba-1511.github.io/tutorials/beginner/nlp/sequence_models_tutorial.html
9. Zafar., Beginner's Guide to Audio Data. Retrieved from https://www.kaggle.com/fizzbuzz/beginner-s-guide-to-audio-data;
10. Petrou, Ted., Selection Subsets of Data in Pandas: Part 1. Retrieved from https://medium.com/dunder-data/selecting-subsets-of-data-in-pandas-6fcd0170be9c;
11. Chilamcurthy, Sasank., Data Loading and Processing Tutorial. Retrieved from https://pytorch.org/tutorials/beginner/data_loading_tutorial.html;
12. Nogueira W., Rode T., Büchner A. (2016) Optimization of a Spectral Contrast Enhancement Algorithm for Cochlear Implants Based on a Vowel Identification Model. In: van Dijk P., Başkent D., Gaudrain E., de Kleine E., Wagner A., Lanting C. (eds) Physiology, Psychoacoustics and Cognition in Normal and Impaired Hearing. Advances in Experimental Medicine and Biology, vol 894. Springer, Cham.