Deep Learning and Its Application
Final Project

2016-2017 Fall

Tianyao Chen, Runzhe Yang, Xingyuan Sun
Student ID: 5140309566, 5140309562, 5140309561

1 Introduction

In this report, we proposed some methods for processing data in tone classification problem, used several deep neural networks structure, did some experiments, and compared their performance and time consumptions among three popular deep learning toolkits: **Torch**[1] , **Theano**[2] and **Tensorflow**[3] .

The data set that our teaching assistances gave us has only 400 data for training. This is a number that is not so sufficient for very large or very deep network to train. Since then, we can only use some small networks. But the data is not so regular that small networks can not separate the original data. This situation forces us to find some way to preprocess our data, then to feed the preprocessed data into the networks.

2 Data Preprocessing

Our prior knowledge tells us that the tone recognition task only requires pitch contour information, which can be expressed by fundamental frequency. Fortunately, the feature `f0` as well as `engy` feature have already given to us. But it does not mean we can use the `f0` feature directly for three reasons: 1) there are some noise and redundant information in the raw `f0` feature, which would be great distraction; 2) there numerical values of `f0` feature varies in a large range so that it will increase difficulties when training; 3) the sequences of features are not of the same length so that it is not suitable for a wide range of models. Therefore, we can not circumvent data preprocessing part. In this section, we will introduce a pipeline of data preprocessing for normalization, removing redundant information, smoothing and denoising data and finally expanding to the same length. Experiments show that data preprocessing can influence performance of models significantly.

2.1 Mel Scale

We first convert the foundational frequency (`f0`) in the mel-scale. The reason we do that is the Mel frequency is much closer the non-linear sensing measurement of human listeners. We use a popular version[4] of mel-scale formula as

$$\texttt{Mel}(f) = 2595 \log_{10}(1 + f/700).$$

However, the frequencies in mel-scale are not very robust in the presence of additive noise, and so it is common to normalize them in speech recognition systems to lessen the influence of noise.

2.2 Standardization

When normalizing data, we only divide its standard derivation but do not subtract its mean, since there are some zeros in the data sequence. Besides, we conduct two types of standardization. One is called **local standardization**, in which divide the `std` of a certain feature it self, to avoid some data varies in a huge range. The other is to divide the `std` of the whole data set for easier training, called **global standardization**. We do the local standardization first, then do global standardization, for both `f0` and `engy`.

[1] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In BigLearn, NIPS Workshop, number EPFL-CONF-192376, 2011.

[2] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In Proc. SciPy, Austin, TX, 2010.

[3] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. and Ghemawat, S., 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467.

[4] `https://en.wikipedia.org/wiki/Mel-frequency_cepstrum`

2.3 Removing Redundancy

As we observe in the training data, there some redundancies in the `f0` features: Although the `engy` is very low (even is zero), the `f0` is still very high and behaves strangely. We think this part of `f0` is redundant because even human listeners cannot recognize a tone in very low volume. Further, this part of `f0` is more likely from environments instead of human speaker because of low corresponding `engy`. Hence, in data preprocessing, we discard the redundancies of `f0` such that corresponding $\texttt{standardized}(\texttt{engy}) < 1.0$.

We only keep the non-zero part of `f0` after processing as the output of this stage. This part of data still contains some noise. So in the next step, we are going to try to denoise the `f0` feature.

2.4 Smoothing

The extracted `f0` alway has some obvious double, half or random error point. We need to do further smoothing for offering better feature to the tone recognizer. Typically, we can use some approaches such as linear interpolation, moving average to make the frequencies varies smoothly. But those method cannot deal with the case that data containing continuous random error points which is quite common in our dataset. We use a search-based algorithm[5] to smooth the frequency data. In that paper, the author used this algorithm to decrease the recognition error rate by 40%.

We search from a smooth beginning of `f0` data, and adjust each data point in order. Let $f_1, f_2, \ldots, f_N$ be the frequencies of N continues frames. When we deal with the i-th data point f_i we first deal with the case of double of half frequency. If

$$|f_i/2 - f_{i-1}| < C_1, \text{ then we let } f_i' = f_i/2;$$

else if

$$|2 \cdot f_i - f_{i-1}| < C_1, \text{ then we let } f_i' = 2 \cdot f_i.$$

Then for random error point (noise), if

$$|f_i - f_{i-1}| > C_1 \text{ and } |f_{i+1} - f_{i-1}| > C_2, \text{, then we let } f_i' = 2 \cdot f_{i-1} - f_{i-2}.$$

else if

$$|f_i - f_{i-1}| > C_1 \text{ and } |f_{i+1} - f_{i-1}| \leq C_2, \text{, then we let } f_i' = 0.5 \cdot f_{i+1} + f_{i-1}.$$

otherwise, the data point is not an error point, we simply keep $f_i' = f_i$. When we deal with the frequency f_{i+1} of next frame, we substitute f_i with f_i'. By observing the smoothed `f0` in training set, we adopt $C_1 = 0.32$ and $C_2 = 0.67$ in our experimental settings.

The process above is good for the level tone, the rising tone and the falling tone, but for the falling-rising tone, it will change the shape of data curve. To solve this, we can do the same process as above but from the N-th frame to the first frame.

Finally, we use moving average with a window of size 5 frames to further smooth data. Moving averaging can not only remove the random error points but also keep the stepped transition between two smooth periods.

[5] Xiaoyan Zhu, Wang Yu, and Jun Liu. 2000. An approach of fundamental frequencies smoothing for chinese tone recognition. Journal of Chinese Information Processing, 15(2), May.

2.5 Data Expansion

By here, we have already get a denoised `f0`. But they are not of the same length and it is annoying if we want to use them to train a normal fully connected neural network or convolutional neural network. We come up with three ways to expand the data into the same length. Three ways corresponding three different datasets. They are

1. data.shift

   This dataset is made by shift the beginning of smoothed `f0` feature to its first non-zero element. Then we pad 0s at the end of the data to make its length be 128.

   The dataset is saved in file "shared data/data_shift.json".

2. data.linear

   In this dataset, we use expand the dataset by linear interpretation. Suppose the smoothed `f0` feature has N frames, we want to expand it to N' frame data, then the missing data f'_j between original smoothed `f0` data f_i and f_{i+1} should be

   $$f'_j = (f_{i+1} - f_i)\left(\frac{(j-1)(N-1)}{N'-1} - i + 1\right) + f_i$$

   and $f'_1 = f_1$, $f'_{N'} = f_N$ should hold as boundary condition.

   The dataset is saved in file "shared data/data_linear.json".

3. data.quad

   Another way is using a quadratic function to fit the curve of smoothed `f0` feature. If the fit curve obtained by least square method is $g(x) = a + bx + cx^2$, then the missing data f'_j between original smoothed `f0` data f_i and f_{i+1} should be scaled as

   $$f'_j = g[(j-1)(N-1)/(N'-1) + 1]$$

   The dataset is saved in file "shared data/data_quad.json".

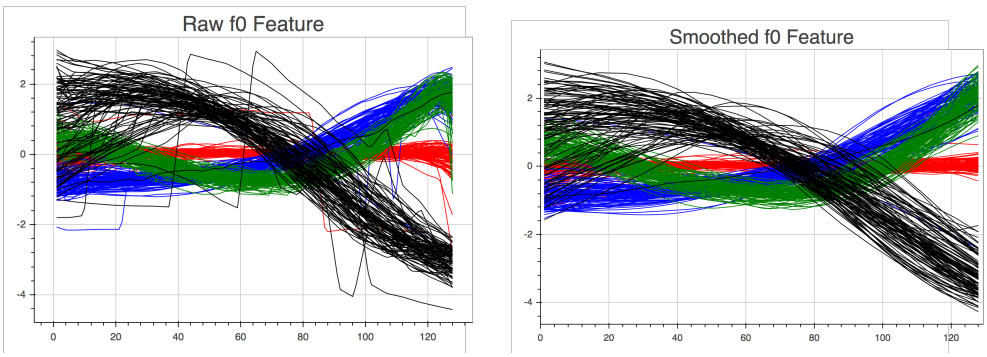

Figure 1: Left: the `f0` features in the training set without smoothing; Right: the `f0` features in the training set after smoothing; Both are expanded to the length of 128 frames by linear interpolation.

2.6 Centralization

For `data.linear` and `data.quad` we subtract their mean in the last step. The reason we do not apply centralization on `data.quad` is we pad zero at the end of the data and do not want that part contains any information. Up to now, we have finish the data preprocessing part.

2.7 Other Approaches

Admittedly, rule-based data processing always has some drawbacks and cannot denoising very well. Therefore we also tried some other approaches using deep learning technique. One of them is denoising autoencoder (DAE). This approach can provide us a robust feature extraction and generating more data so that we can do **data augmentation**. However in experiment we found that the noise of `f0` is not Gaussian and adding white noise cannot improve its robustness. We discard this way.

3 Models

3.1 Models in Comparison

We tested two models on the three different datasets we have made for comparing their performance on **Torch**, **Theano** and **Tensorflow**. These two models are

1. pure.fc

   Fully Connected Neural Networks: This model has just one fully connected layer. Suppose our input is just 1-dimensional raw vector x, with weight matrix W and bias vector b, the scores for the four tones are

   $$\text{score} = x \cdot W + b$$

   We use softmax loss and l2-norm regularization.

2. cnn.fc

   Temporal Convolutional Neural Networks: This model has one convolution layer with 64 kernels and 2-strided max-pooling layer followed by one fully connected layer. The input is one channel one-dimensional data and the output is its class index.

3.2 Other Models We Explored

In additional to those two simple models above, we also tried some complicated models. We implement these models in **Torch**. You can find them in folder "torch/extra".

3.2.1 Attention-based Recurrent Neural Networks

Since we meet sequence data in this problem, we want to check whether RNN is suitable for this problem. We first tried an RNN of hidden size 128 and it performs unstable. LSTM is also tried to achieve more stable performance. However, in experiment we find the generalization ability is very poor. The highest performance on `test_new` is about 50%.

We infer that the reason of poor generalization ability maybe lies on that the simple RNN or LSTM does not catch the history information well. Therefore we try to add the attention mechanism.

Since it is a "Sequence to One" problem, the design of attention could be very easy, just the weighted average of hidden layers at different time instance.

$$a = \sum_{t=1}^{T} w_t h_t, \text{ and } w_t = \frac{exp(f(h_t))}{\sum_{k=1}^{T} exp(f(h_t))}$$

where a is the attention output, and f is a vector-to-scaler function trained as a perceptron. Using this model, we can achieve accuracy 96.12% on `train`, 100.00% on `test` and 86.40% on `test_new`[6].

3.2.2 Temporal CNN + Attention-based LSTM

The Attention-based Recurrent Neural Network has two obvious shortcomings: 1) Too slow. The input sequence is of length 128 and it will cost very long time to train LSTM in one epoch. 2) Low ability of feature extraction. To solve this two problem, we tried to add an convolutional layer with pooling layer before the data flows into attention-based LSTM. Because of convolutional layer it can extract more useful information from input data, and because of pooling layer the input sequences to the LSTM are much shorter then speed up the training process. We use a convolutional layer with 4 4×1 kernels and a pooling layer of size 4×1 and 4-strided. This model can steadily perform an accuracy of 91.22% and even sometimes higher on `test_new`.

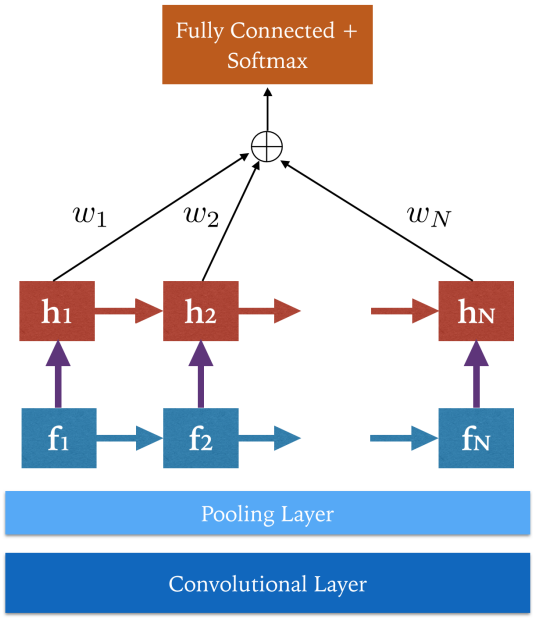

Figure 2: Right: The structure of Temporal CNN + Attention-based LSTM

4 Experiments

We tried each model on each dataset, implemented by each framework. Note that for accuracies we take the maximum over all trails and for time consumption, we ran each program for ten times, and then calculate its mean and standard deviation. The CPU is Intel Xeon E3-1230 V2 @ 3.3GHz.

[6]This result obtained by attention-based lstm of hidden size 128 with one layer fully connected layer of size 64.

4.1 Hyper-parameters

We use following hyper-parameters settings for experiments.

	value
regularization strength	0.01
batch size	10
optimizer	SGD
learning rate	3×10^{-5}
learning rate decay	N/A
number of epochs	2000

Table 1: Hyperparameters settings for "pure.fc" model.

	value
filter size	3
number of channels	64
regularization strength	0.002
batch size	10
optimizer	SGD
learning rate	1×10^{-4}
learning rate decay	N/A
number of epochs	400

Table 2: Hyperparameters settings for "cnn.fc" model.

4.2 Experiments result

Now we show the performance and time consumption. The detailed analysis can be found in section 5.

		Tensorflow	Torch	Theano
pure.fc	training set accuracy	85.25%	74.50%	**85.50**%
	test set accuracy	**80.00**%	74.00%	77.50%
	test_new set accuracy	55.26%	**57.46**%	**57.46**%
	time	34.45 ± 0.95s	$\mathbf{21.58 \pm 2.13}$s	22.60 ± 0.22s
cnn.fc	training set accuracy	84.75%	**93.00**%	87.50%
	test set accuracy	70.00%	**95.00**%	85.00%
	test_new set accuracy	54.82%	**63.15**%	56.58%
	time	$\mathbf{30.18 \pm 2.33}$s	34.14 ± 0.28s	36.03 ± 0.37s

Table 3: Performance and time consumption on data.shift.

		Tensorflow	Torch	Theano
pure.fc	training set accuracy	91.00%	**98.00**%	93.50%
	test set accuracy	**100.00**%	**100.00**%	**100.00**%
	test_new set accuracy	92.54%	**94.74** %	93.86%
	time	35.39 ± 1.02 s	$\mathbf{20.87 \pm 0.87}$ s	23.45 ± 0.16 s
cnn.fc	training set accuracy	93.25%	**98.00**%	95.25%
	test set accuracy	90.00%	**100.00**%	**100.00**%
	test_new set accuracy	90.35%	89.04%	**94.74**%
	time	$\mathbf{34.44 \pm 1.35}s$	34.77 ± 3.67s	35.83 ± 0.15 s

Table 4: Performance and time consumption on data.linear.

		Tensorflow	Torch	Theano
pure.fc	training set accuracy	92.75%	**96.75**%	93.75%
	test set accuracy	**100.00**%	95.00%	**100.00** %
	test_new set accuracy	92.98%	93.42%	**93.86** %
	time	37.71 ± 1.77s	$\mathbf{20.79 \pm 0.84}$ s	23.28 ± 0.32s
cnn.fc	training set accuracy	90.25%	**97.75**%	95.50 %
	test set accuracy	95.00%	**100.00** %	**100.00** %
	test_new set accuracy	**94.74**%	87.28 %	94.30 %
	time	37.21 ± 4.39s	$\mathbf{34.76 \pm 2.39}$ s	35.89 ± 0.87 s

Table 5: Performance and time consumption on data.quad.

5 Conclusion

5.1 Time consumption

From the experiment results, we can see that **Torch** is fastest when we apply "pure.fc" model. But for the "cnn.fc" model, there are not too much difference between different models. Although **Theano** beats **Tensorflow** on time criterion in the most cases, remember that we do not take the compiling time into account, **Theano** could be very slow when compiling big models.

5.2 Accuracy

The best accuracy preformance on test new dataset is 94.74%, and the best accuracy performance on test dataset is 100%. This result could be caused by the cleaness of test dataset. There might be some outliers inside test new dataset.

5.3 Analysis

From the experiment results we can see that whatever model we apply, we get the worst accuracy performance on "data.shift" dataset, no matter it is training, test or test new dataset.

And if we focus only on "data.shift" dataset, we can see that there are huge gaps between training accuracy or test accuracy and test new accuracy. This could mean a terrible overfitting problem. Since there are only 400 training samples, this problem is understandable.

But compare this result with other preprocessed dataset, we can see the effectiveness of preprocessing.

5.4 Difficulties in Implementation

Torch: It could be very straightforward if you want to construct a normal neural network. Torch provides you with perfect interface, but, in lua.

Theano: It would be somewhat messy if you use Theano without Keras. You need to implement every low level details for constructing NNs and training it well.

Tensorflow: Full-functional interface in Python, it is the most attractive feature of Tensorflow. And its document is quite thorough.

5.5 The Triumph of Simple Models

We only compared performance among different simple models (pure.fc and cnn.fc) and found that even the simplest (single layer network) model can have highest performance our preprocessed dataset. Actually, we have tried many other complex models on this task (see 3.2), but they cannot outperform simple model for three reasons: 1) The small dataset we use is not sufficient to train a complex model. We try to do some data augmentation to solve this problem while it will even make the performance worse (see). 2) Complex model has much larger capacity than our simple classification task requires. 3) Hyper-parametesrs of complex model is more difficult to tune compared with a simple model. We failed to find a good hyper-parametesrs setting to outperform simple models.