DEEP LEARNING COURSE FINAL PROJECT

Tensorflow[1], MXNet[2], Theano[3]

Yiwei Bai, 5140309316
Lequn Chen, 5140309565
Huichen Li, 5140309194

January 15, 2017

Contents

1 Task formalization

The task is to design a tone classifier to output the tone label.

1.1 Data

We are given the f0/engy files as the input. The f0 and energy features are extracted from corresponding wav files. Each wav file is the pronunciation of a single Chinese character.

1.2 Challenges

Small Data Amount The data amount in this task is very limited. So models that are too complex cannot be used here.

Various Lengths The lengths of different wav files are different. Hence, the f0/engy files corresponding to different characters also have different length. For the same character, the f0 file has exactly the same length as the engy file.

Unconformity in Train and Test The training data is clean, pronounced by professional speakers, while the test data has noise in it, with characters spelled not so accurately. Thus, the training samples and the test samples are not extracted from the same distribution. This is different from normal tasks where we have the assumption that training data has the same distribution as test data.

2 Sparks for bonus

- Various Feature Selection techniques(see Section Feature Selection).
- A Multi-GPU training (see Section Multi-GPU training) was implemented.
- A well designed code architecture (see Section Design of the framework).

3 Model Evolution

We decided to construct a Deep Neural Network to do the classification in our project.

Sliding Window To solve the challenge of the lengths of features being diverse, we adopt the method of sliding windows to make features be the same length.

3.1 Basic DNN on Original Feature

Having made the features equal length, we built roughly two types of DNNs, one with two layers and one with three layers. In both cases, relu is used as the activation function. Every layer is fully connected layer. Tuning the hyper parameters gives us the following model:

3.1.1 Two-layer DNN: $12 - 40 - 4$

The input has dimension 12 as we use sliding windows to partition each feature file into 12 blocks. The hidden layer is of dimension 40. And in order to output one of the four tone labels, the output dimension is set to be 4.

layer	fully connected 1	fully connected 2
dimension	12×40	40×4

3.1.2 Three-layer DNN: $12 - 80 - 15 - 4$

The input and output dimension is the same with the two-layer DNN model. We changed one hidden layer of 40 units to two hidden layers, one with 80 units, the other 15.

layer	fully connected 1	fully connected 2	fully connected 3
dimension	12×80	80×15	15×4

3.1.3 Results and Comparison

The table below shows the performance on accuracy of the two networks when we divide the input into different sizes. The accuracy values are averages over 5 trainings. Based on this comparison, we chose 12 as our input size.The number of hidden units are tuned during training.

input size	$? - 40 - 4$ DNN	$? - 80 - 15 - 4$ DNN
8	85.27%	86.60%
12	88.84%	88.83%
15	81.25%	85.26%
18	81.25%	86.16%

Unsatisfactory Results The performance on test data cannot go above 90 percent accuracy in our basic DNN models, which is not very good.

3.1.4 Trying to Elevate

Adjusting direction We also take more complex models into account, for example, CNN.

3.2 Convolutional Neural Network

The fragments are relatively short, so we consider using CNN instead of RNN or LSTM. We are using complex models while avoiding overfitting or other drawbacks.
After careful design and fine tuning, the CNN structure we picked is the following:

layer	kernel number	kernel/weight size
convolution 1	96	(1×10)
maxpool 1	None	(1×2)
convolution 2	108	(1×10)
maxpool 2	None	(1×2)
convolution 3	108	(1×10)
maxpool 3	None	(1×2)
convolution 4	16	(1×10)
flatten layer	None	None

Followed by

layer	dropout probability	fully connected dimension
fully connected 1	None	$(? \times 128)$
dropout	0.5	None
fully connected 2	None	(128×40)
fully connected 3	None	(40×4)

3.2.1 L2 Regularization

The average accuracy of CNN and DNN with/without regularization:

Accuracy	With L2 reg	Without L2 reg
DNN	88.96%	88.83%
CNN	92.10%	89.90%

3.2.2 Adjusting Output Layer

This experiment is an extension of a bug in the model code. Normally the number of output layer is the same with the number of final classes. But adjusting the number of output layers gives us some interesting results, both on the variance and the accuracy. Adding output units has different impacts on CNN and DNN. The performance may get higher, or fall lower. But on the whole, the variance is reduced. See the table below:

	layers	4	10	20
Var	DNN	3.09458	2.9599	0.71506
	CNN	3.49015	0.5126	0.12702
Accuracy	DNN	88.83%	87.75%	89.45%
	CNN	92.10%	89.638%	89.0172%

3.2.3 Trying to Elevate

Adjusting direction Given data that is so sparse, the task kind of degenerate to the time when machine learning was limited by small amount of data, and people aided computers by selecting the 'important features' for the algorithm by their domain specific knowledge. We realize maybe we should generate more useful features from the original data file for the model to learn. This is where feature selection comes in.

3.3 Feature Selection

In training neural networks, the f0 data is what we really use. But there are some problems with the original f0 data.

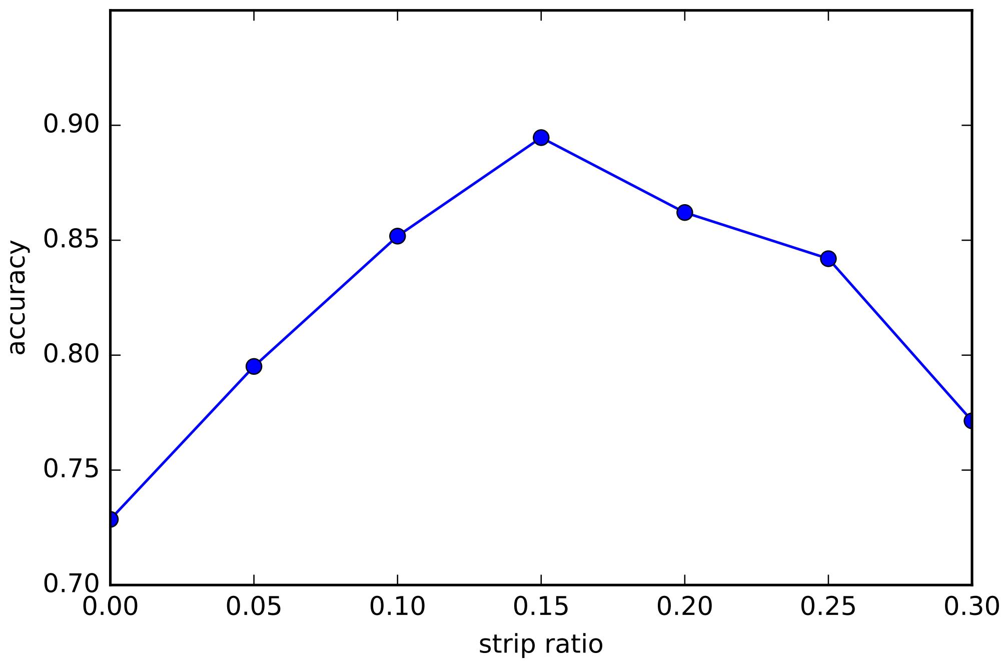
- They contain noise for our network model. There are features that consist of noise rather than useful patterns.
- They are too sparse, the sample rate is low so there are not enough data points.

In selecting features, what we are actually doing is throwing away useless information so that the remaining features capture the real underlying patterns of the audio that we need in recognition. After that, we use some methods to create new data points based on the remaining useful information after throwing away noise.

3.3.1 Cubic Spline Interpolation

The interpolation is used here because the original data points are too sparse. In the project we did the followings:

- Let the maximum value in an .engy file be M. Discard the data points that are below a threshold of $0.15M$ for we treat them as noise. The corresponding points in f0 file are discarded simultaneously. The comparison of performance when choosing different thresholds is plotted below



- Use cubic spline interpolation to fit engy and f0 points. The data before and after the interpolation is plotted in Figure3.1 and Figure 3.2 below.
- Pick points in the fitted f0 function equidistantly as the newly generated features.

3.3.2 MFCC

In the tutorial[4], a method called MFCC is introduced. MFCC is inspired by human auditory system and used for extracting features in sound processing. Here are several key ideas about MFCC:

- For simplicity, although an audio signal is constantly changing, we make the assumption that it doesn't change much during short time intervals statistically.
- Identifying which frequencies like what the human cochlea does so we calculate the power spectrum of each frame.
- The human ears couldn't discern the difference between two closely spaced frequencies so we take clumps of periodogram bins and sum them up to get an idea of how much energy exists in various frequency regions. As the frequencies get higher our filters get wider as we become less concerned about variations.
- As human being don't hear loudness on a linear scale, we take the logarithm of the filterbank energies.

The number of MFCCs and f0s we use

- for DNN model:

	MFCC	f0
number	6	12

- for CNN model:

	MFCC	f0
number	6	100

After a linear combination of MFCC and f0(simple concatenation) is used as the feature, the results are

Numcep	13(default)	6	None(Without mfcc)
DNN	89.8%	87.9%	88.83%
CNN	89.69%	92.10%	90.11%

Plots before and after cubic spline interpolation

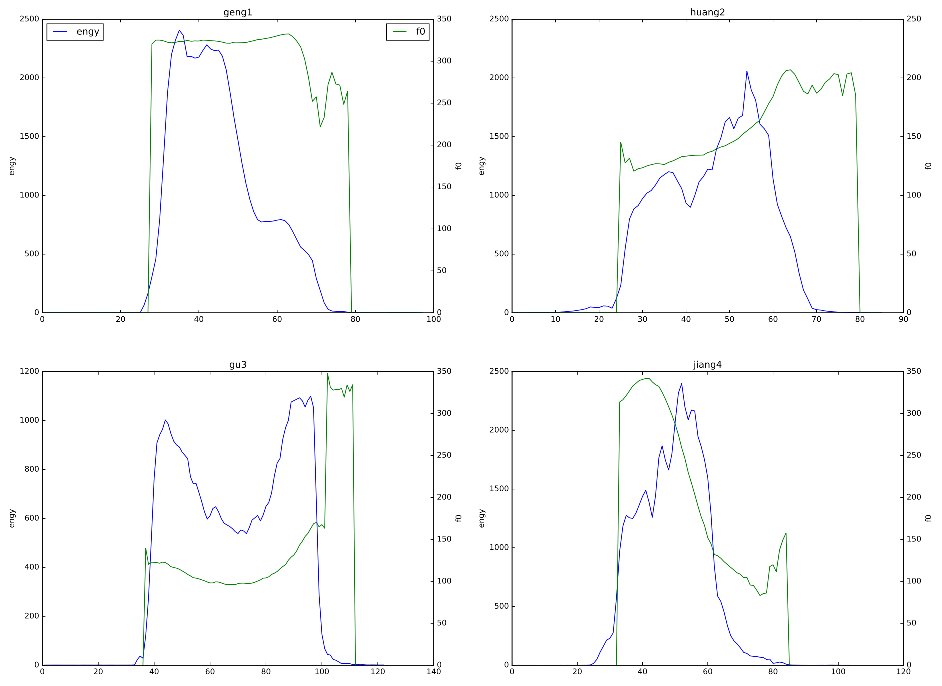


Figure 3.1: Raw engy and f0 data

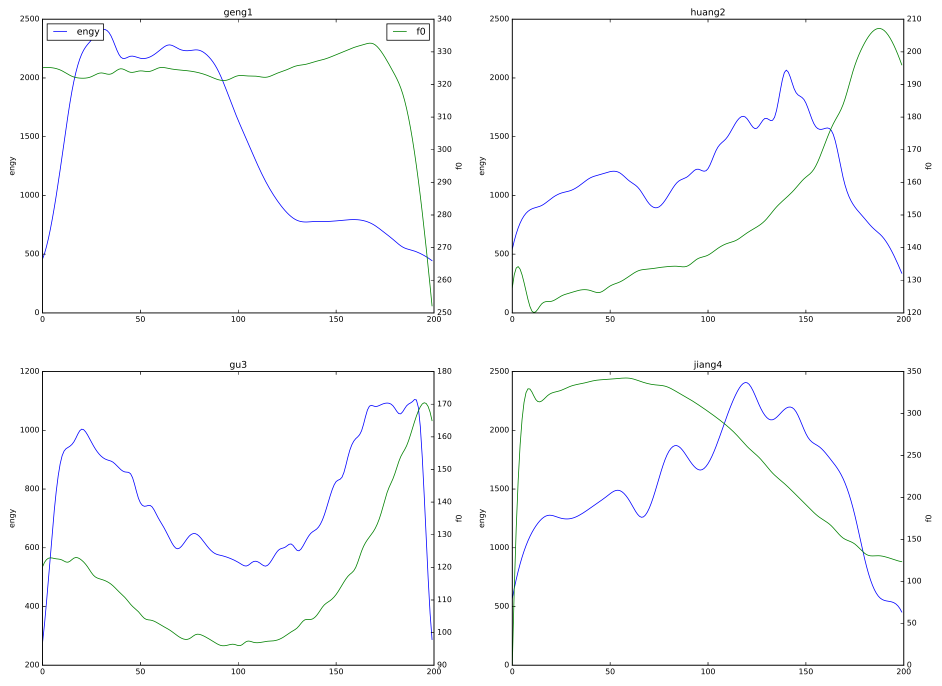


Figure 3.2: Processed engy and f0 data

3.3.3 Dynamic Programming

We adopt this method based on the assumption that naturally, the audio of Chinese characters should be relatively smooth.
As the original data contains noise, we want to manually evaluate its smoothness and throw away noise based on the evaluation. The evaluation function we chose is

$$(i - pre_{pos})^m + (f0[i] - pre_{f0})^n$$

pre_{pos} is the position of the last data point we picked. m and n are hyper parameters we tuned. We get $m = 4$ and $n = 2$ in our experiment. The impact on performance is shown below:

Accuracy	With DP	Without DP
DNN	87.96%	88.83%
CNN	90.63%	92.10%

3.3.4 Finite Impulse Response

In signal processing, a finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time.[6]

Accuracy	With FIR	Without FIR
DNN	89.52%	88.83%
CNN	91.51%	92.10%

3.3.5 Trying to Elevate

Up to now, the best accuracy of our CNN and DNN models are:

	Best Accuracy
DNN	92.40%
CNN	95.04%

The structure of the best model(click to see):

- DNN structure

layer	fully connected 1	fully connected 2	fully connected 3
dimension	18×80	80×15	15×4

This structure differs a little from the previous DNN structure because we add MFCC feature here. Actually $18 = 12 +$ number of MFCC points $= 12 + 6$.

- CNN structure

The feature for the best model(click to see):

- DNN feature
- CNN feature

The parameters that we use are:

model	learning rate	lr decay	optimizer
DNN	0.0004	1.0	SGD
CNN	0.001	1.0	Adam

The code of the project is posted in Github[7].

ADJUSTING DIRECTION Not only do we want better results on accuracy, faster speed is also what we are pursuing all the time. We want to make use of the multi GPUs to speed up the training.

4 CODE ARCHITECTURE

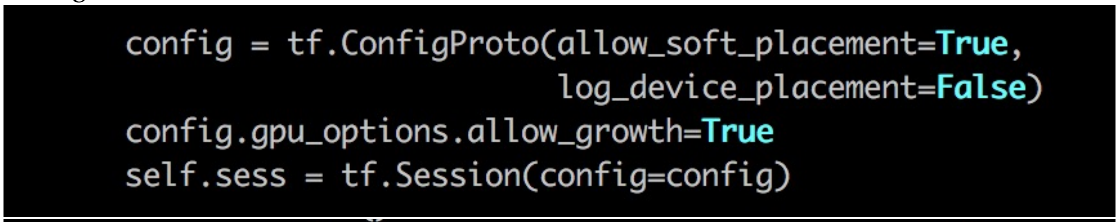
4.1 MULTI-GPU TRAINING

As we are in an era of what people call 'Big Data', it's important to know how to do multi-GPU training. The support and ability of scheduling job on multi-GPUs of DL platforms are critical here.

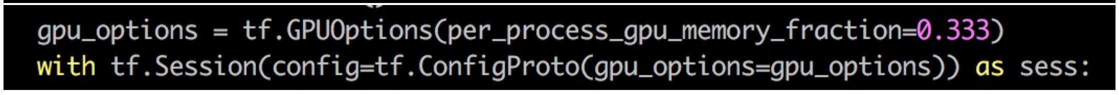
4.1.1 TENSORFLOW

We implemented a simple data parallel multi-GPU model on Tensorflow. Every mini batch is dispatched to different GPUs where the loss is calculated and sent back. The parameters are updated based on the loss.

POINT TO NOTICE When using Tensorflow on GPU, we need to be very careful. TF was designed to use up all the memory in GPU if we don't declare usage explicitly. So we need to set the 'gpu_options' manually. We allow GPU memory to grow by the fraction we set. See code fragments below for more information:



```
config = tf.ConfigProto(allow_soft_placement=True,
                        log_device_placement=False)
config.gpu_options.allow_growth=True
self.sess = tf.Session(config=config)
```



```
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.333)
with tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)) as sess:
```

4.1.2 MXNet

Thanks to the intermediate representation mechanism adopted by MXNet, it is relatively easy to implement a multi-GPU version on MXNet. The optimazation can be done with high efficiency.

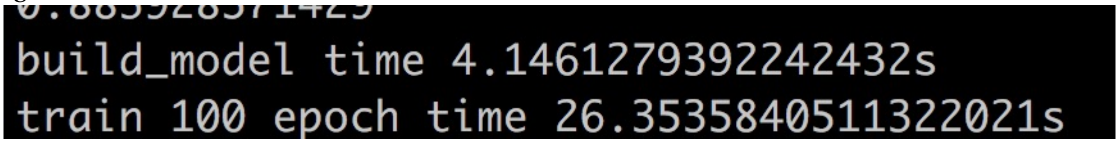
```python
import mxnet as mx
model = mx.model.FeedForward(ctx=[mx.gpu(0), mx.gpu(2)], ...)
```

4.1.3 Theano

Theano doesn't have much support for multi-GPU, so we have to write many low level code explicitly, like how to transfer data between CPU and GPU, how to store parameters on GPUs, etc.

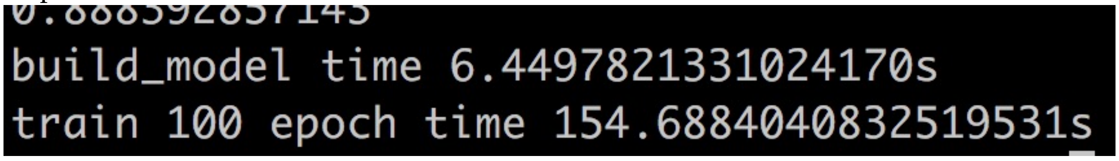
4.1.4 Results

Empirically, using eight GPUs can achieve a speed up of about 4 to 5 times on a complex model when the cost of transferring data is constrained by careful design. However, in small projects like this one, the cost of overhead in data transmission surpasses other improvements, so the overall performance falls instead. The time cost is shown below:
Single GPU:



```
build_model time 4.1461279392242432s
train 100 epoch time 26.3535840511322021s
```

Multiple GPU:



```
build_model time 6.4497821331024170s
train 100 epoch time 154.6884040832519531s
```

4.2 Design of the framework

- The separation of model and training. We separate model and training so that the reuse of model is convenient.
- The explicit processing of data. We choose to process first each time before training.

5 Platform Comparison

5.1 Encapsulation

Theano

- Theano was not invented for deep learning specific usage. So some basic operations in DL were not included. For example,
  - In the function 'conv2d'[5], there were no arguments like "padding='SAME'", which makes it a little inconvenient to use.
- Theano is implemented in low level compared to the others. This makes it hard to use for newcomers, but it also allows for inspiration.

TENSORFLOW

- Tensorflow has the highest capacity. It has a great variety of functions encapsulated inside with plenty of adjustable arguments.

MXNET

- As a young project, short of funding and support as the founders claim officially, MXNet is in bad need for more detailed documents and tutorials. There is only API Reference for newcomers.
- MXNet has only a limited number of layers encapsulated, and there are only a few parameters that can be tuned compared to Tensorflow.

5.2 SPEED

model	platform	time to build model	time to train(200 epochs)
DNN	Tensorflow	0.3806691169738770s	20.3759698867797852s
	Theano	0.5839521884918213s	3.0449399948120117s
	MXNet	0.0004229545593262s	19.3520941734313965s
CNN	Tensorflow	0.9123530387878418s	271.7300508022308350s
	Theano	3.4809811115264893s	876.6990518569946289s
	MXNet	0.0007259845733643s	1459.4064869880676270s

5.3 COMPUTATIONAL GRAPH

The expression ability of the three are comparable. Their workflow are similar: after building the computational graph, they would take care of the derivatives for us. However, there are some slight difference between them.
We can even write a library on our own and build model on this basis. By choosing the backend, we can transfer to the platform we want easily.

5.3.1 DIMENSION

For example, if there are a couple of convolution layers,

- In Theano, both the input dimension and the number of filters for the current layer and the previous layer are needed.

- In Tensorflow, the input dimension is not needed explicitly, but we need to write out the number of filters for the current layer and the previous layer.
- In MXNet, neither the input dimension nor the number of filters for the previous layer is needed. All we have to write out explicitly is the number of filters for the current layer.

platform	Theano	Tensorflow	MXNet
input dimension	required	not required	not required
number of filters for current layer	required	required	required
number of filters for previous layer	required	not required	not required

TENSORFLOW We can declare dimension for 'Placeholder's in Tensorflow, while in the other platforms this can't be done.

MXNET Usually we don't need to calculate dimensions on our own. MXNet will take care of the dimension transformation for us.

THEANO All dimensions must be declared clearly.

5.4 NUMERICAL STABILITY

In the experiment, we realize that the numerical stability of the three platforms are different.

- The validation loss of the same model using the same parameters on the three platforms over epochs is plotted below in Figure 5.1 and 5.2.
- By examining these oscillation curves, it's clear that among the three platforms, MXNet is the most unstable numerically. Theano comes next in instability. Tensorflow is the best among the three.

Validation Loss:
same model, same parameters except for the optimizers,
different platforms, over epochs

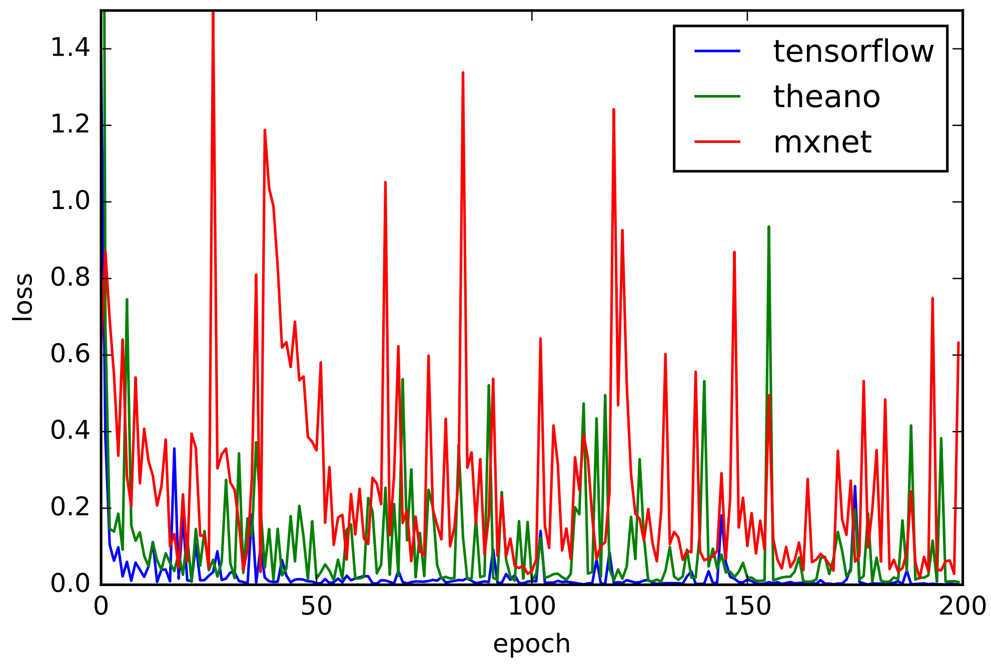


Figure 5.1: Training CNN using Adam

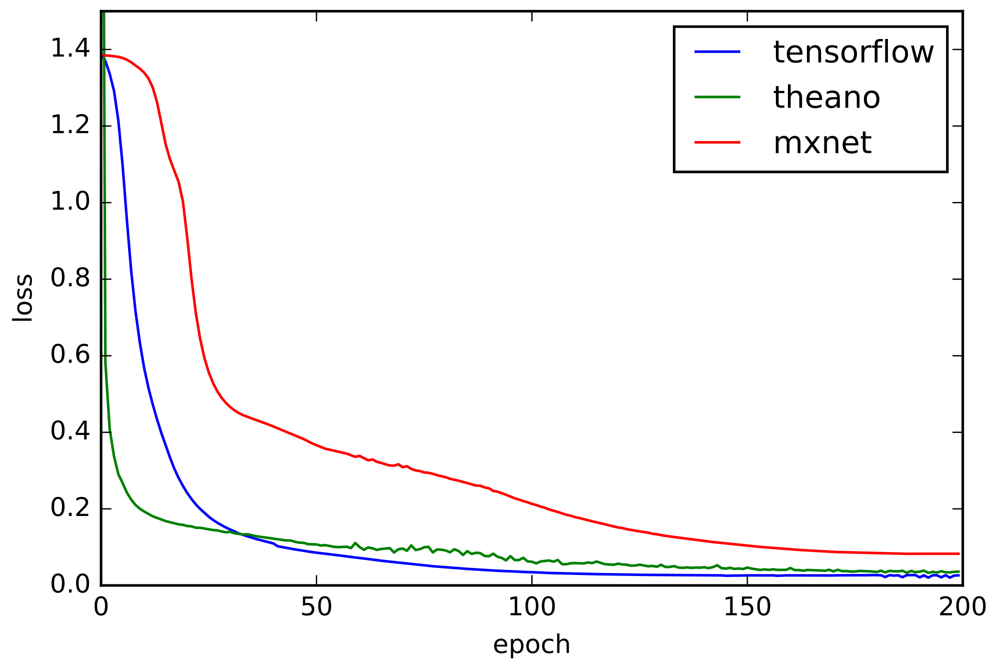


Figure 5.2: Training DNN using SGD

6 Reference

References

[1] Tensorflow, https://www.tensorflow.org/api_docs/

[2] Flexible and Efficient Library for Deep Learning, http://mxnet.io/

[3] Theano Docs, http://deeplearning.net/software/theano/#

[4] Mel Frequency Cepstral Coefficient (MFCC) tutorial, http://www.practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/

[5] Theano Convolution, http://deeplearning.net/software/theano/library/tensor/signal/conv.html

[6] Finite Impulse Response, https://en.wikipedia.org/wiki/Finite_impulse_response?oldformat=true

[7] Project code, https://github.com/bywbilly/DeepLearning-Final-Project