School of EECS

CS519: Deep Learning

CIFAR-10 Image Classification using Fully Connected Neural Network

Professor: Fuxin Li

Kaibo Liu * 932-976-427 *

Nov. 11th, 2016

Overview

File List & Introduction:

1. MLP_KaiboLiu.py #excute **python MLP_KaiboLiu.py** to run
2. normalization.py # normalize the data for better convergence.
3. report.pdf

This report is divided into two part according the assignment requirements: Decision Tree, and Random Forest.

- In the first part, I would explain how I build a network on the training dataset with loss function. Specifically how to determine the best parameter and the corresponding threshold for accuracy and smoothness. At last, I show the experiment result given different tuning plan and would give an explanation.
- In the second part, I would discuss some math tricks and precious experience from the time-consuming implementation.

Fully Connected Neural Network

1. Problem Description

The task is to implement a one hidden layer fully connected neural network using Python.

Given N training examples in 2 categories $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, I need to implement backpropagation using the cross-entropy loss on top of a softmax layer: $(p(c_1|x) = \frac{1}{1+\exp(-f(x))}, p(c_2|x) = \frac{1}{1+\exp(f(x))})$, where the output is $f(x) = \mathbf{w}_2^T g(\boldsymbol{W}_1^T \mathbf{x} + b) + c$, and $g(x) = \max(0, x)$ is the ReLU activation function. To be simple, I give some symbols to all positions on the network. The training network works on the 2-class dataset extracted from CIFAR-10. The data has 10,000 training examples in 3072 dimensions and 2,000 testing examples. For this assignment, we just treat each dimension as **uncorrelated** to each other.

$$
\begin{aligned}
\mathbf{z}_1 &= \boldsymbol{W}_1^T \mathbf{x} + b \\
\mathbf{z}_2 &= g(\mathbf{z}_1) = \max(0, \mathbf{z}_1) \\
z_3 &= \mathbf{w}_2^T \mathbf{z}_2 + c \\
p &= \sigma(z_3) = \frac{1}{1 + e^{-z_3}} \\
E &= -y \log(p) - (1 - y) \log(1 - p)
\end{aligned}
$$

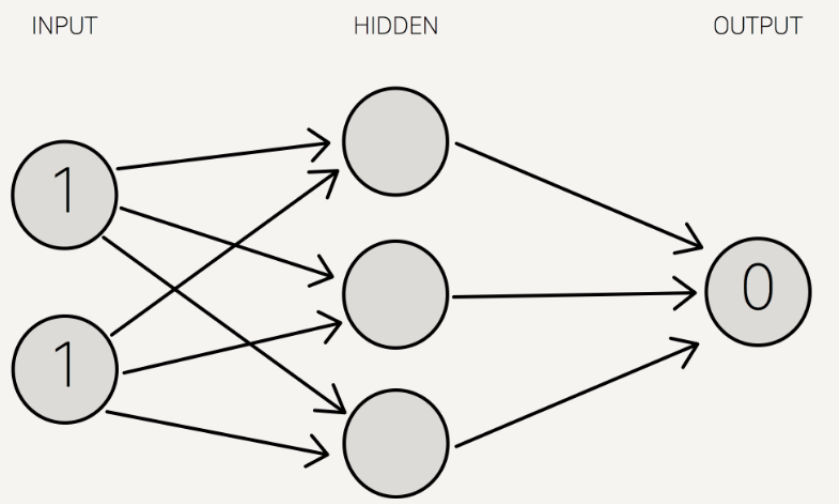

Figure 1: momentum

2. Network Structure

I construct a network structure with the data structure of class in Python from the given skeleton. In order to accelerate the running, I put examples within one mini-batch into a matrix, extending the input $\mathbf{x}$ to matrix $\boldsymbol{x}$, and dimension $[d, 1] \rightarrow [d, batch]$. So Fi.2 the list illustrate the dimension of all data and parameters:

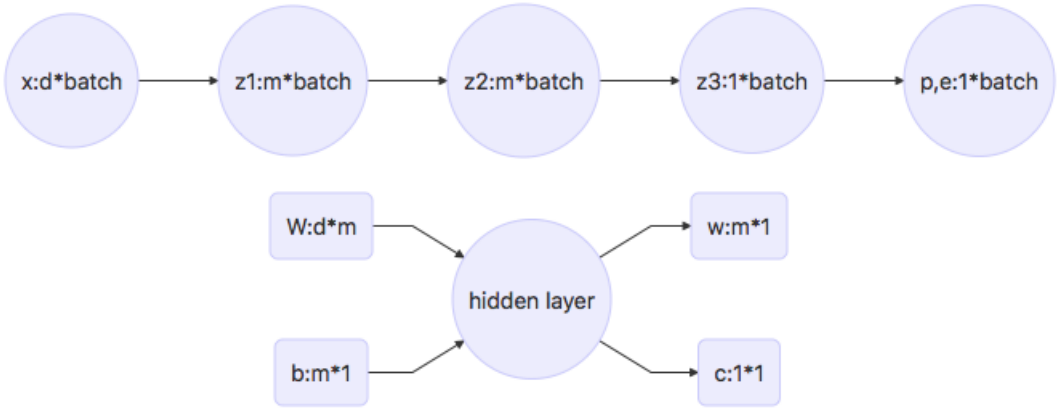

Figure 2: dimension in the network

(1) **class LinearTransform()**
Listing 1 shows the detailed structure of a Linear Transform layer, and its initialization.

Listing 1: class for Linear Transform layer

```
class LinearTransform(object):
    # vector elements in this class are stored in
       matrix[m,1] not ndarray [m,]
    def __init__(self, input_dims, hidden_units):
        self.d ← input_dims
        self.m ← hidden_units
        self.batch ← 1
        self.W ← np.random.uniform(-1,1,(self.d, self
           .m))/10    # W = [d,m], matrix even m==1
        self.b ← np.random.uniform(-1,1,(self.m,1))
           /10
        self.back_w, self.back_b, self.back_x ← 0,0,0
        self.dw, self.db ← 0,0
    def forward(self, x, size_batch):
        # [d,batch]->[m,batch]
    def backward(self,grad_output,
        learning_rate=0.0,
        momentum=0.0):
     ##### batch have nothing to do with W,b, after
        mean, W and b keep their dimensions, while x
        is extended 1 dimension
```

The forward part is easy to implement. Here are some functions for backprorogation.

– Output layer
In output layer, the forward direction is $\mathbf{z}_2 \rightarrow z_3$, and the backward direction is from SigmoidCrossEntropy layer$\rightarrow$ *back_out* of output layer.

$$\begin{cases} \dfrac{\partial \boldsymbol{E}^j}{\partial \mathbf{w}_2^i} = \dfrac{\partial \boldsymbol{E}^j}{\partial p^j}\dfrac{\partial p^j}{\partial z_3^j}\dfrac{\partial z_3^j}{\partial \mathbf{w}_2^i} = (p-y)^j \mathbf{z}_2^{ij} \\ \dfrac{\partial \boldsymbol{E}^j}{\partial c} = \dfrac{\partial \boldsymbol{E}^j}{\partial p^j}\dfrac{\partial p^j}{\partial z_3^j}\dfrac{\partial z_3^j}{\partial c} = (p-y)^j \\ \dfrac{\partial \boldsymbol{E}^j}{\partial \mathbf{z}_2^{ij}} = \dfrac{\partial \boldsymbol{E}^j}{\partial p^j}\dfrac{\partial p^j}{\partial z_3^j}\dfrac{\partial z_3^j}{\partial \mathbf{z}_2^{ij}} = (p-y)^j \mathbf{w}_2^i \end{cases}$$

– Hidden layer

$$\begin{cases} \dfrac{\partial \boldsymbol{E}}{\partial \mathbf{W}_1} = \dfrac{\partial \boldsymbol{E}}{\partial \mathbf{z}_2}\dfrac{\partial \mathbf{z}_2}{\partial \mathbf{z}_1}\dfrac{\partial \mathbf{z}_1}{\partial \mathbf{W}_1} \\ \dfrac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}} = \dfrac{\partial \boldsymbol{E}}{\partial \mathbf{z}_2}\dfrac{\partial \mathbf{z}_2}{\partial \mathbf{z}_1}\dfrac{\partial \mathbf{z}_1}{\partial \mathbf{b}} \end{cases}$$

To simplify, I use matrix to implement and treat the two linear transform layer as one group of expression. Here, b is denoted as input dimension, m is denoted as output dimension after linear combination.

$$\begin{cases} \dfrac{\partial \boldsymbol{E}}{\partial \boldsymbol{W}} = \sum_{j=1}^{batch} \mathbf{z}[:,j] \cdot \mathbf{grad\_output}[:,j]^T / batch & (1) \\ \dfrac{\partial \boldsymbol{E}}{\partial \boldsymbol{b}} = \sum_{j=1}^{batch} \mathbf{grad\_output}[:,j] / batch & (2) \\ \dfrac{\partial \boldsymbol{E}}{\partial \mathbf{z}} = \boldsymbol{W} \cdot \mathbf{grad\_output} & (3) \end{cases}$$

Here, **grad_output** means the next layer's backward output, can be either a vector or matrix, in output layer or hidden layer, respectively.
In equation (1), $\partial \boldsymbol{E}/\partial \boldsymbol{W}$ is $[d,m] = [d,1] \cdot [1,m]$, then get mean on each batch(all j).
In equation (2), $\partial \boldsymbol{E}/\boldsymbol{b}$ is $[m,1]$, derivatived from the mean of a $[m, batch]$ on each batch(all j).
In equation (3),$\partial \boldsymbol{E}/\boldsymbol{z}$ is $[d, batch]$, derivated from the produt of $[d,m] \cdot [m, batch]$.

In the part of backward, I used momentum like Listing3.

Listing 2: Parameter update with momentum

```
# Momentum update
# mu is momentum, lr is learning_rate
d_W ← mu*d_W - lr*partial_E_W        # delta_W
W ← W + d_W                          # update W
```

Introduction of the momentum rate allows the attenuation of oscillations in the gradient descent. The geometric idea behind this idea can probably best be understood in terms of an eigenspace analysis in the linear case. If the ratio

between lowest and largest eigenvalue is large then performing a gradient descent is slow even if the learning rate large due to the conditioning of the matrix. The momentum introduces some balancing in the update between the eigenvectors associated to lower and larger eigenvalues.[1]:

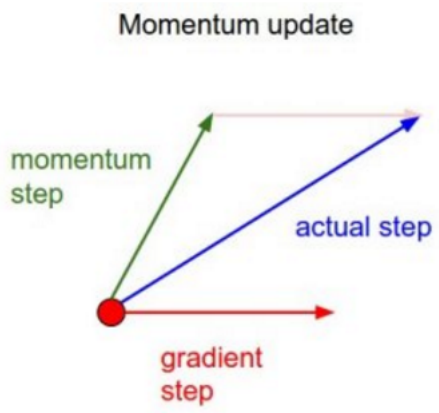

Figure 3: momentum

(2) **class ReLU()**

Listing 3 shows the detailed structure of a ReLU layer, including its forward and backward calculation.

Listing 3: class for ReLU layer

```
# This is a class for a ReLU layer max(x,0)
class ReLU(object):
    def __init__(self, x=0):
        self.layer_i = x
        self.layer_o, self.back = [],[]
    def forward(self, x):      # x is [m,batch]
        self.layer_i = x         # [m,batch]
        self.layer_o = deepcopy(x)
        self.layer_o[self.layer_o < 0] = 0
    def backward(self,grad_output): # [m,batch]
        y = deepcopy(self.layer_i)
        y[y > 0] = 1
        y[y == 0] = 0.5
        y[y < 0] = 0
        self.back = grad_output * y
        # [m,batch]=[m,batch]*[m,batch] element wise
```

(3) **class SigmoidCrossEntropy()**

Listing 4 shows the pseudocode for constructing a layer for probability and loss function calculation.
This layer

Listing 4: class for sigmoid and cross Entropy layer

```
class SigmoidCrossEntropy(object):
    def __init__(self, x=0):
        self.layer_i ← x
        self.layer_o ← 0   # p, sigmoid output
        self.label ← 0     # true label
        self.loss ← 0      # cross Entropy
        self.back ← 0      # back output
    def forward(self, x,y): # [1,batch]->[1,batch]
    def backward(self):     #  [1,batch]->[1,batch]
```

(5) **class network**

Listing 5 shows the detailed structure of the whole network, including two linear transform layer ***l1***,***l3***, one ReLU layer ***l2*** and one final layer ***l4*** from output to sigmoid and cross Entropy.

Listing 5: class for the whole network

```
class network(object):
    def __init__(self,input_dims, hidden_units):
        self.l1 ←LinearTransform(input_dims,
           hidden_units)
        self.l2 ←ReLU()
        self.l3 ←LinearTransform(hidden_units, 1)
        self.l4 ←SigmoidCrossEntropy()
    def forward(self,x,label,size_batch):
    def backward(self,learning_rate, momentum):
    def evaluate(self):
```

(6) **RunMain(x,train_y,test_x,test_y,num_epochs,num_batches,hidden_units,lr,mu)**

Full round of train is applied in this function, with determined parameters *num_epochs*, *num_batches*, *hidden_units*, and *learning rate*.

Listing 6 shows the pseudocode of logical sequence for running all epochs in training.

Listing 6: def RunMain

```
for epoch in xrange(num_epochs):
    for b in xrange(num_batches):
        batch_x ← b^th batch in x
        batch_y ← b^th batch in train_y
        n.forward(batch_x.T,batch_y,size_batch)
        n.backward(lr,mu)
        update total_loss
        update train_loss
        print()
```

(7) **WriteData**
I use this function for saving data and figure results into files. Due to different type of tuning conditions, the permutation of parameters should be corresponding to the condition. This is why I need this part.

Network training

1. Training Monitoring

I used print and savefig commends to monitor all the training loss, testing loss,training accuracy, and testing accuracy(from misclassifies) calculated. More results are shown in subsequent tuning sections.

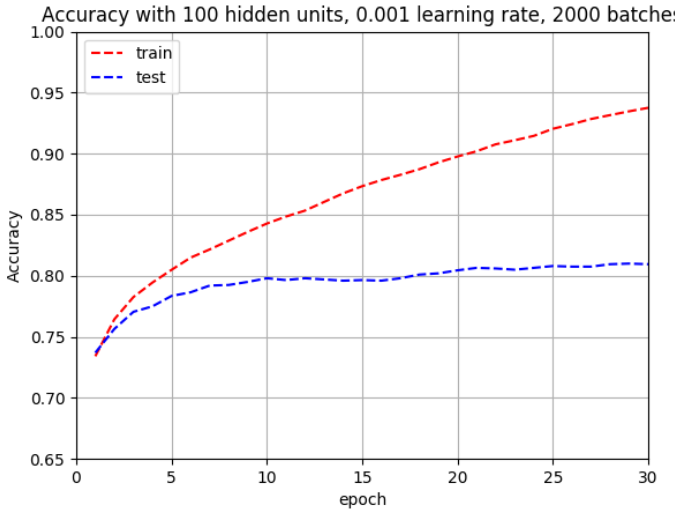

Figure 4: Accuracy for train and test

2. Tuning learning rate

In this part, I tuned the learning rate with proper values, and fix other parameters:

```
learning rate = [0.005,0.001,0.0005,0.0001]
hidden units=100
num_batch=1000
momentum=0.8
```

The results are shown in Fig.5-8, which are train loss,test loss,train accuracy and test accuracy against epoch with different learning rate, respectively.

We can see from the results that, with smaller learning rate, the better stability will the trend of loss and accuracy be. 0.005 seems a good learning rate with fast convergence and higher accuracy within 30 epochs. But it will impact the testing accuracy in Fig.6. That means this learning rate causes some sort of over-fitting.

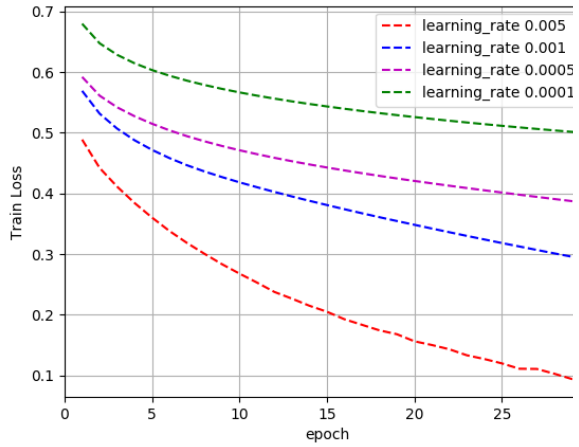

Figure 5: Train Loss with learning rate

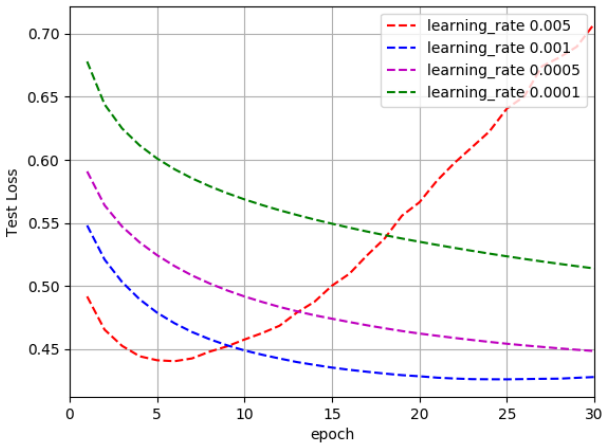

Figure 6: Test Loss with learning rate

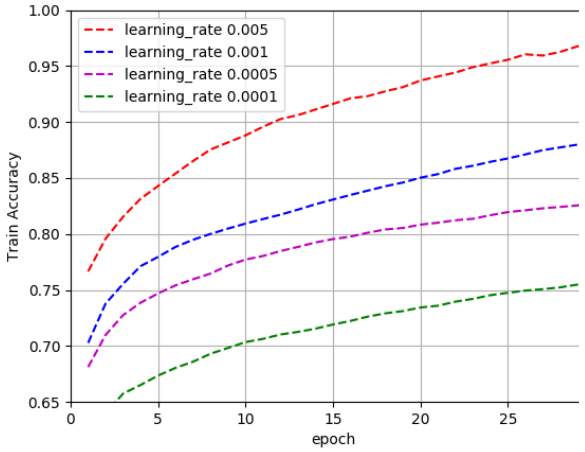

Figure 7: Train Accuracy with learning rate

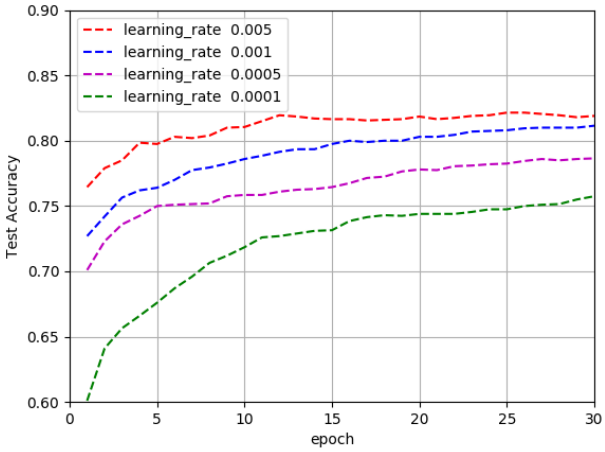

Figure 8: Test Accuracy with learning rate

3. Tuning number of hidden units

In this part, I tuned the number of hidden units with proper values, and fix other parameters with:

```
learning rate = 0.001
hidden units = [10,100,1000]
num_batch = 1000
momentum = 0.8
```

The results are shown in Fig.9-12, which are train loss,test loss,train accuracy and test accuracy against epoch with different number of hidden units, respectively.

We can see from the results that, with more hidden units, the better accuracy we can get. 1000 seems a good hidden units with fast convergence and higher accuracy within 30 epochs. But it will impact the testing accuracy in Fig.10. That means this number of hidden units causes some sort of over-fitting. So a reasonable hidden units cannot be too big, making 100 is a good choice in my future trail.

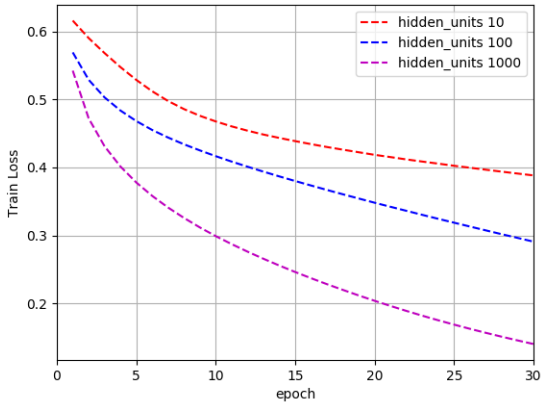

Figure 9: Train Loss with hidden units

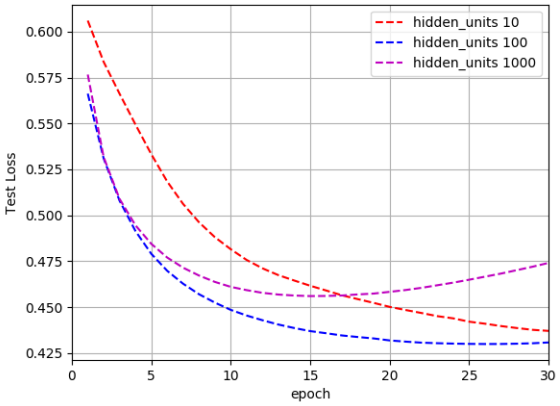

Figure 10: Test Loss with hidden units

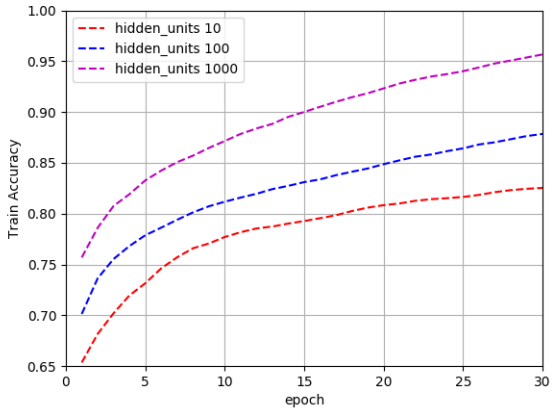

Figure 11: Train Accuracy with hidden units

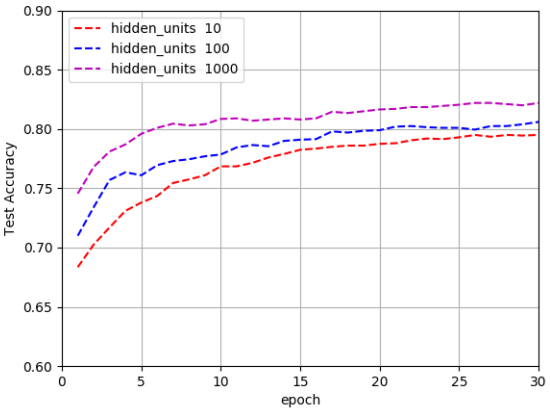

Figure 12: Test Accuracy with hidden units

4. Tuning the number of batches

In this part, I tuned the number of hidden units with proper values, and fix other parameters with:

```
learning rate = 0.001
hidden units = 100
num_batch = [100,500,1000,2000]
momentum = 0.8
```

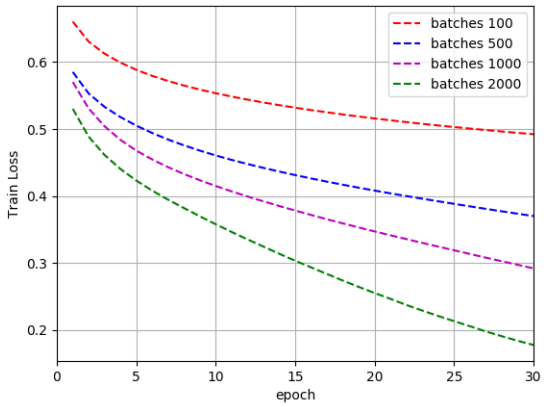

Figure 13: Train Loss with batches

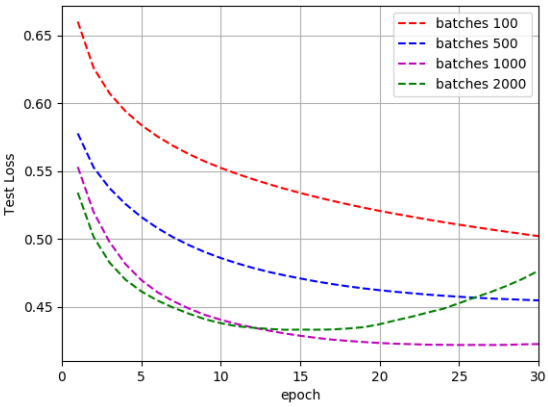

Figure 14: Test Loss with batches

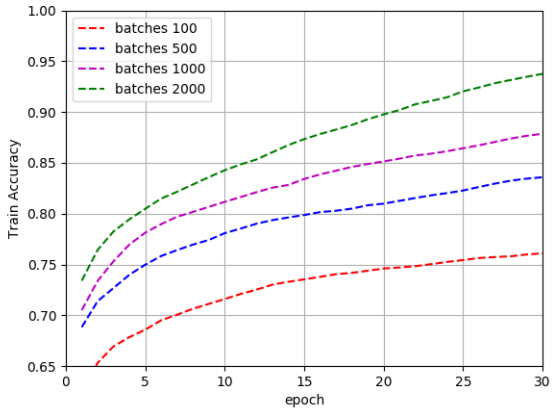

Figure 15: Train Accuracy with batches

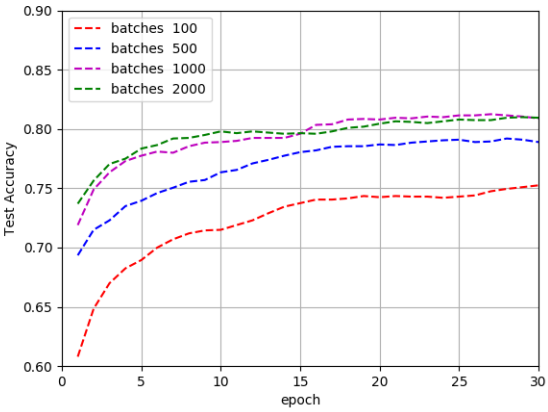

Figure 16: Test Accuracy with batches

The results are shown in Fig.13-16, which are train loss,test loss,train accuracy and test accuracy against epoch with different number of batches, respectively.

We can see from the results that, with more number of batches, the better stability will the trend of loss and accuracy be. 2000 seems a good number of batches with fast convergence and higher accuracy within 30 epochs. But it will impact the testing accuracy in Fig.14. The reason is that, if number of batches are big enough to the number of examples, the network will update every backward procedure of example, so

the last status of a epoch is mainly related to the last example, which can not reflect the whole dataset. So a reasonable number of batches cannot be too big, making 1000 is a good choice.

5. Runtime for each tuning

The statistic for the runtime of each tunning is shown in following tables, we can see the runtime on server is acceptable. The biggest factor impacting the runtime is the number of hidden units, which will enlarge the size of matrices in hidden layer. From the middle table, when I have 1000 hidden units, I need more than 1 hour to train the network with 30 epochs.

hidden units=100 num_batch=1000	
learning rate	running time for 1 epoch(s)
0.005	11.1305
0.001	10.6034
0.0005	10.8551
0.0001	10.3944

learning rate=0.0010 num_batch=1000	
hidden units	running time for 1 epoch(s)
10	1.5880
100	10.2849
1000	152.8437

learning rate=0.0010 hidden units=100	
batch numa	running time for 1 epoch(s)
10	7.2926
500	8.6152
1000	10.1885
2000	13.0935

6. performance new neural network

From section 3 to 5, I can get the best performance of 82.2% testing accuracy in 30 epochs with parameters:

```
learning rate = 0.001
hidden units = 1000
num_batch = 1000
momentum = 0.8
```

Discussion

1.Initialization

In my strategy,elements in weight $\boldsymbol{W}$ is $[-1, 1]$, then after fist epoch, average loss is 2.64, accuracy is 54%. However if I set the initial value in $\boldsymbol{W} \in [-0.1, 0.1]$, after first epoch, the average loss is 0.55, accuracy is 74%. So making the initialization of weights can help fast convergence.

2. Overflow in calculation

Even though I set initial values for weights and bias very small, they can grow after several updates if learning rate is not small enough, and they eventually accumulate to z, the output of hidden layer. It is possible that $z < -50$, making the output of softmax layer, $p = \sigma(z)$, naturally to 0. However, considering the intermediate calculation e^{-z}, I would encounter some NAN results due to overflow. To solve this problem, I need to put every calculation in safe place, and use a better equation for both Sigmoid and Croos-Entropy[2].

Sigmoid:

$$p = \sigma(z) = \frac{1}{1+e^{-z}} \begin{cases} \dfrac{1}{1+e^{-|z|}} & , z > 0 \\ \dfrac{e^{-|z|}}{1+e^{-|z|}} & , z < 0 \end{cases}$$

Cross-Entropy:

$$\begin{aligned} E &= -y\log(p) - (1-y)\log(1-p) \\ &= \log(1+e^{-z}) - zy + z && \%z > 0, \text{otherwise would overflow} \\ &= \log(1+e^{z}) - zy && \%z < 0, \text{otherwise would overflow} \\ &= \log(1+e^{-|z|}) - zy + \max(z, 0) && \% \text{ overflow free} \end{aligned}$$

3. Strategy of loss and acc calculation

In one epoch, I run by the order of batches. With the value of $num\_batches$, we know there are $size\_batch = \frac{num\_examples}{num\_batches}$. For the i^{th} batch, I can get loss from the current parameters after calling $\boldsymbol{forward}$ and get the result of $size\_batch$ examples. So $loss_i$ is based on the i^{th} parameter within this epoch, while $loss_j$ is based on the j^{th} parameter within this epoch. If I accumulate $loss_i$ and $loss_j$ for total loss of this epoch, they may reflect different status of parameters. For final monitoring, I think loss (or average loss) and accuracy in each batch should based on same status of parameters $\boldsymbol{W}$ and $\mathbf{b}$, which will also make final curves more smoother. So my strategy is, after all updates, I come to the last batch of this epoch, then I run $\boldsymbol{forward}$ with whole epoch data to get objectives and accuracies from the same standard and parameters.

Reference

[1] Stanford CS231. http://cs231n.github.io/neural-networks-3/#sgd.

[2] Tensorflow git. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/g3doc/api_docs/python/functions_and_classes/shard5/tf.nn.sigmoid_cross_entropy_with_logits.md.