Homework 2

Yunian Pan

March 20, 2019

1 Backpropagation

1.1 case 1

Sol:

Given $x_i = g(y_j) = \frac{1}{1+e^{-\sum_j w_{ji}y_j}}$, and $\sum_i \frac{\partial E}{\partial x_i} = -\sum_i (\frac{t_i}{x_i} - \frac{1-t_i}{1-x_i})$ apply the chain rule, first take x_i and y_j as fixed to compute the gradient for w_{ji}, then take z_k and y_j as fixed to compute the gradient for w_{kj} we have

$$\begin{aligned}
\frac{\partial E}{\partial w_{ji}} &= \frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial w_{ji}} \\
&= (\frac{1-t_i}{1-x_i} - \frac{t_i}{x_i})\frac{\partial g(w_{ji}, y_j)}{\partial w_{ji}} \\
&= (\frac{1-t_i}{1-x_i} - \frac{t_i}{x_i})x_i^2 y_j e^{-\sum_j w_{ji}y_j} \\
&= (\frac{1-t_i}{1-x_i} - \frac{t_i}{x_i})x_i y_j (1-x_i) \\
&= (x_i - t_i)y_j
\end{aligned}$$

$$
\begin{aligned}
\frac{\partial E}{\partial w_{kj}} &= \sum_i \left(\frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial y_j}\right)\frac{\partial y_j}{\partial w_{kj}} \\
&= \sum_i \frac{\partial E}{\partial x_i}(-w_{ji})x_i(1-x_i)(-z_k)y_j(1-y_j) \\
&= \sum_i \frac{x_i - x_i t_i - t_i + x_i t_i}{(1-x_i)x_i}(-(-w_{ji})x_i(1-x_i))(-(-z_k)y_j(1-y_j)) \\
&= \sum_i (x_i - t_i)w_{ji}y_j(1-y_j)z_k
\end{aligned}
$$

Thus, we have $\quad \sum_i \frac{\partial E}{\partial w_{ji}} = \sum_i \delta_j^i y_j, \qquad \sum_j \frac{\partial E}{\partial w_{kj}} = \sum_j \delta_k^j z_k.$

1.2 case 2

Sol:

Given the cross-entropy $E = -\sum_i t_i \log(x_i)$ and softmax activation function $x_i = \frac{e^{\sum_j w_{ji}y_j}}{\sum_i e^{\sum_j w_{ji}y_j}} = f(w_{11}, \ldots, w_{j1}, \ldots, w_{ji}, \ldots, w_{jm}, \ldots, y_j, \ldots)$, we have

$$
\frac{\partial E}{\partial x_i} = -\frac{t_i}{x_i}
$$

$$
\begin{aligned}
\frac{\partial x_m}{\partial w_{ji}} &= \frac{y_j(\delta(i-m)e^{\sum_j w_{jm}y_j}(\sum_i e^{\sum_j w_{ji}y_j}) - e^{\sum_j w_{jm}y_j}e^{\sum_j w_{ji}y_j})}{(\sum_i e^{\sum_j w_{ji}y_j})^2} \\
&= y_j x_m(\delta(i-m) - x_i)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial w_{ji}} &= \sum_m \frac{\partial E}{\partial x_m}\frac{\partial x_m}{\partial w_{ji}} \\
&= \sum_m y_j\left(-\frac{t_m}{x_m}\right)x_m(\delta(i-m) - x_i) \\
&= y_j\left(\sum_m t_m \cdot x_i - t_i\right)
\end{aligned}
$$

$$\begin{aligned}\frac{\partial E}{\partial w_{kj}} &= \sum_i (\sum_m (\frac{\partial E}{\partial x_m}\frac{\partial x_m}{\partial y_j})\frac{\partial y_j}{\partial w_{kj}}) \\ &= \sum_i (w_{ji}(\sum_m t_m x_i - t_i)) y_j(1-y_j) z_k \\ &= \sum_i (\sum_m t_m x_i - t_i)(w_{ji} y_j (1-y_j)) z_k\end{aligned}$$

Here $\delta_j^i = \sum_m t_m \cdot x_i - t_i$, $\delta_k^j = \sum_i (\sum_m t_m x_i - t_i)(w_{ji} y_j (1-y_j))$.

2 Vowpal Wabbit

Data format: there are one example per line in the train/test file, each formatted as [label] | [features], where $label \in \{1, \ldots, 26\}$, $feature \in R^{617}$, features are expressed as $n : x_n$, where n is the dimension and x_n is a real number.

VW commands are executed through shell scripts:

```
#!/bin/bash

# creat a loss file to record the training and testing
  results
loss_file=oaa_loss.csv
if [ -e "$loss_file" ]; then
    rm $loss_file
    echo "train/test,learning_rate,passes,average_loss"
        >>$loss_file
fi

# loop for training
for pss in `seq 1 20` # passes up to 20
do
    for n in `seq -4 0` # learning rate
    do
        l_rate=`echo "scale=2; 2^$n" | bc -l`

        echo -e "*** train; learning rate: 2^$n; passes
            : $pss.  *** \n"

        # train model and save terminal output to
            train_log.txt
```

```
            vw --oaa 26 isolet_train.vw --cache_file
                cache_train --sgd -l $l_rate --passes $pss -
                f oaa.model >train_log.txt 2>&1

            # extract average train loss using regular
                expression
            avg_loss=`cat train_log.txt | grep "average␣
                loss␣=␣" | grep -o -E "([0-9]+.[0-9]+)
                |((\+|-)nan)"`
            echo "train,${l_rate},${pss},${avg_loss}" >>
                $loss_file

            echo -e "***␣test␣***␣\n"

            # test model and save terminal output to
                test_log.txt
            vw -t -c isolet_test.vw -i oaa.model >test_log.
                txt 2>&1

            # extract average test loss using regular
                expression
            avg_loss=`cat test_log.txt | grep "average␣loss
                ␣=␣" | grep -o -E "[0-9]+.[0-9]+"`
            echo "test,${l_rate},${pss},${avg_loss}" >>
                $loss_file

        done
    done

    rm test_log.txt train_log.txt
```

For ect model we can change the keyword 'oaa' into 'ect'. the script for one-against-all model iteratively(from 1 to 20 passes and different learning rates) executes vw training commands:

```
vw --oaa 26 isolet_train.vw -c --sgd -l $l_rate --passes
    $pss -f oaa.model >train_log.txt 2>&1
```

which use SGD as the solver, parameter *l_rate* as the learning rate, *pss* as the passes to train an one-against-all model extracted to the binary file *oaa.model*, with all the results output to a log file *train_log.txt*, then the script uses regular expression to parse the average losses, learning rates and passes and output them to loss file *oaa_loss.csv*. During each loop, after trainning the script does VW commands:

```
vw -t -c isolet_test.vw -i oaa.model >test_log.txt 2>&1
```

which test *oaa.model* on the test data *isolet_test.vw* and save the terminal output to file *test_log.txt*, then the scripts parses the results into loss file.

After the steps above, I could find the minimum in *oaa_loss.csv* and *ect_loss.csv*, in the first place I tried learning rates $\{0.001, 0.01, 0.1, 1\}$, and it turned out setting 0.1 has the best performance of all passes settings:

mini oaa loss

	train/test	learning_rate	passes	average_loss
101	test	0.1	13	0.048346

mini ect loss

	train/test	learning_rate	passes	average_loss
85	test	0.1	11	0.156489
93	test	0.1	12	0.156489
101	test	0.1	13	0.156489
109	test	0.1	14	0.156489
117	test	0.1	15	0.156489
125	test	0.1	16	0.156489
133	test	0.1	17	0.156489
141	test	0.1	18	0.156489
149	test	0.1	19	0.156489
157	test	0.1	20	0.156489

Therefore I explored several different learning rate settings around the neighbor of 0.1, using $learning\ rate = 2^n$ with n from -4 to 0: To summarize, the oaa model

mini oaa loss

	train/test	learning_rate	passes	average_loss
197	test	0.06	15	0.038168

mini ect loss

	train/test	learning_rate	passes	average_loss
155	test	0.06	12	0.151399
169	test	0.06	13	0.151399
183	test	0.06	14	0.151399
197	test	0.06	15	0.151399
211	test	0.06	16	0.151399
225	test	0.06	17	0.151399
229	test	0.25	17	0.151399
239	test	0.06	18	0.151399
253	test	0.06	19	0.151399
267	test	0.06	20	0.151399
271	test	0.25	20	0.151399

performs better than ect model on this dataset, with its minimum loss being around 0.04, the ect reaches its minimum loss around 0.15 after passes increased to 11 or 12, 0.06 is generally better than other learning rate settings, while the best passes setting is around 15.

3 Classification

The network is shown in 3.1, each hidden layer has 20 Relu nodes,

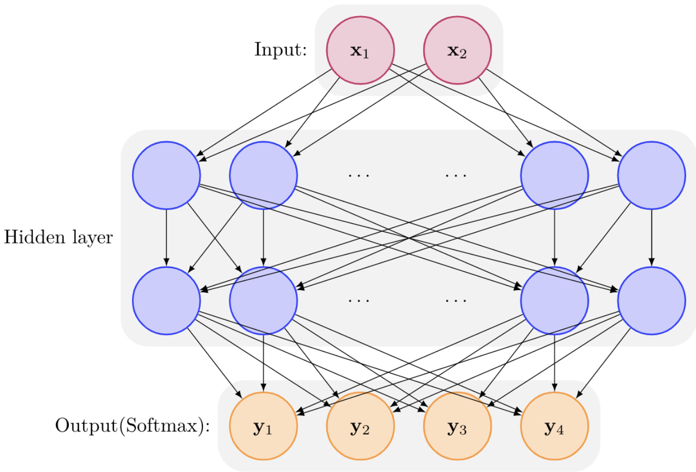
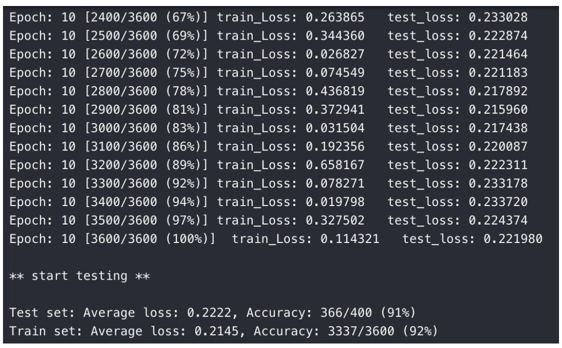


Figure 3.1: fc-net

Each of the output nodes assigns a probability value computed through softmax, we choose the dimension of maximum probability as prediction of the multi-class, the log-likelihood loss is setting learning rate as 0.01, using momentum SGD with batch size = 10 and momentum = 0.5, after 10 epoch we can get following results as shown in 3.2, with average test and train loss being around 0.2 and both accuracy being around 92%.



```
Epoch: 10 [2400/3600 (67%)] train_Loss: 0.263865   test_loss: 0.233028
Epoch: 10 [2500/3600 (69%)] train_Loss: 0.344360   test_loss: 0.222874
Epoch: 10 [2600/3600 (72%)] train_Loss: 0.026827   test_loss: 0.221464
Epoch: 10 [2700/3600 (75%)] train_Loss: 0.074549   test_loss: 0.221183
Epoch: 10 [2800/3600 (78%)] train_Loss: 0.436819   test_loss: 0.217892
Epoch: 10 [2900/3600 (81%)] train_Loss: 0.372941   test_loss: 0.215960
Epoch: 10 [3000/3600 (83%)] train_Loss: 0.031504   test_loss: 0.217438
Epoch: 10 [3100/3600 (86%)] train_Loss: 0.192356   test_loss: 0.220087
Epoch: 10 [3200/3600 (89%)] train_Loss: 0.658167   test_loss: 0.222311
Epoch: 10 [3300/3600 (92%)] train_Loss: 0.078271   test_loss: 0.233178
Epoch: 10 [3400/3600 (94%)] train_Loss: 0.019798   test_loss: 0.233720
Epoch: 10 [3500/3600 (97%)] train_Loss: 0.327502   test_loss: 0.224374
Epoch: 10 [3600/3600 (100%)]  train_Loss: 0.114321   test_loss: 0.221980

** start testing **

Test set: Average loss: 0.2222, Accuracy: 366/400 (91%)
Train set: Average loss: 0.2145, Accuracy: 3337/3600 (92%)
```

Figure 3.2: Classification result

The train loss and test loss after each input of the mini batch is as shown in 3.3

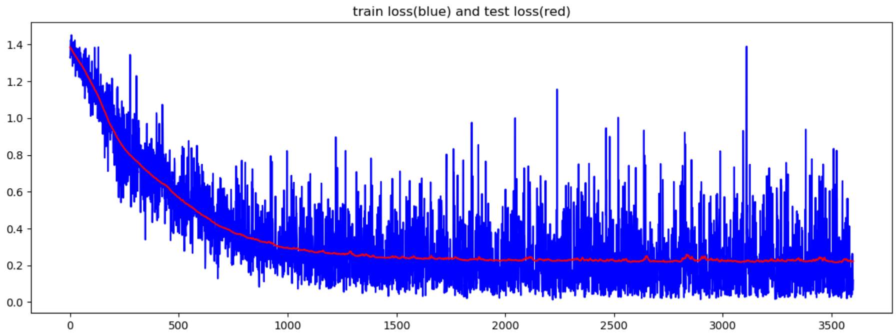


Figure 3.3

4 Regression

The network is shown in 4.1, each hidden layer has 20 tanh nodes,

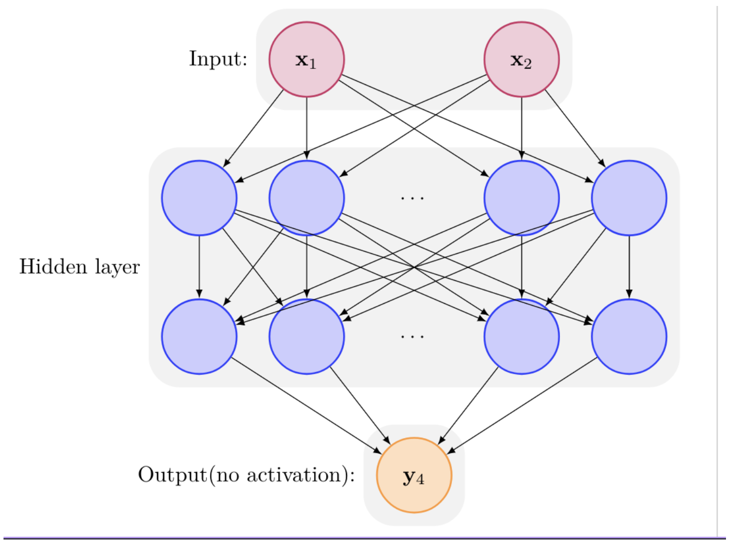


Figure 4.1: fc-net-2

Data samples are uniformly generated from $[-10, 10] \times [-10, 10]$ plane, since x and y is independent, we can genrate them respectively. Using Adam and gradient clipping (the loss is extremely large), after 150 training epoch we get the loss evolution curves in 4.2.

The final train and test loss are shown in 4.3.

To intuitively show the fit, first we show the data samples in 4.4

The model output fit is shown in 4.5

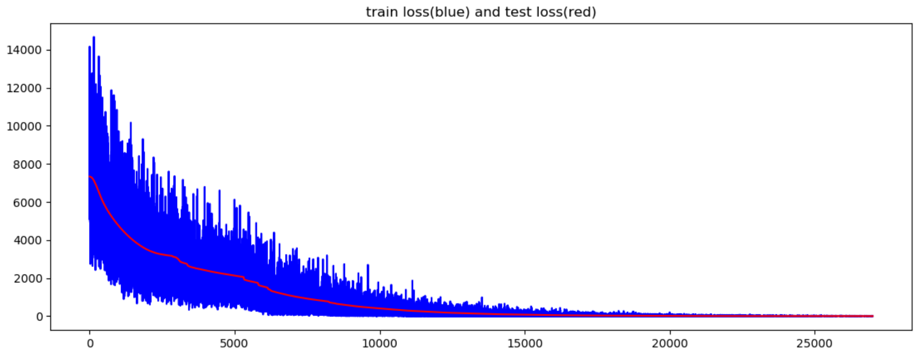


Figure 4.2

```
Epoch: 150 [4050/4500 (90%)]    train_Loss: 4.942859    test_loss: 1.760661
Epoch: 150 [4075/4500 (91%)]    train_Loss: 0.420680    test_loss: 1.748489
Epoch: 150 [4100/4500 (91%)]    train_Loss: 2.275429    test_loss: 1.673854
Epoch: 150 [4125/4500 (92%)]    train_Loss: 0.821406    test_loss: 1.569854
Epoch: 150 [4150/4500 (92%)]    train_Loss: 3.046631    test_loss: 1.599438
Epoch: 150 [4175/4500 (93%)]    train_Loss: 0.420706    test_loss: 1.665714
Epoch: 150 [4200/4500 (93%)]    train_Loss: 0.804503    test_loss: 1.682507
Epoch: 150 [4225/4500 (94%)]    train_Loss: 1.265893    test_loss: 1.645989
Epoch: 150 [4250/4500 (94%)]    train_Loss: 0.875513    test_loss: 1.544408
Epoch: 150 [4275/4500 (95%)]    train_Loss: 0.600644    test_loss: 1.425088
Epoch: 150 [4300/4500 (96%)]    train_Loss: 0.795179    test_loss: 1.383483
Epoch: 150 [4325/4500 (96%)]    train_Loss: 0.300245    test_loss: 1.403112
Epoch: 150 [4350/4500 (97%)]    train_Loss: 0.372322    test_loss: 1.439287
Epoch: 150 [4375/4500 (97%)]    train_Loss: 0.266637    test_loss: 1.421241
Epoch: 150 [4400/4500 (98%)]    train_Loss: 0.320801    test_loss: 1.399609
Epoch: 150 [4425/4500 (98%)]    train_Loss: 0.404778    test_loss: 1.438636
Epoch: 150 [4450/4500 (99%)]    train_Loss: 8.528111    test_loss: 1.560983
Epoch: 150 [4475/4500 (99%)]    train_Loss: 22.966002   test_loss: 1.641173
Epoch: 150 [4500/4500 (100%)]   train_Loss: 0.260145    test_loss: 1.695824

** start testing **

Test set: Average loss: 1.6958
Train set: Average loss: 1.6820
```

Figure 4.3: Regression result

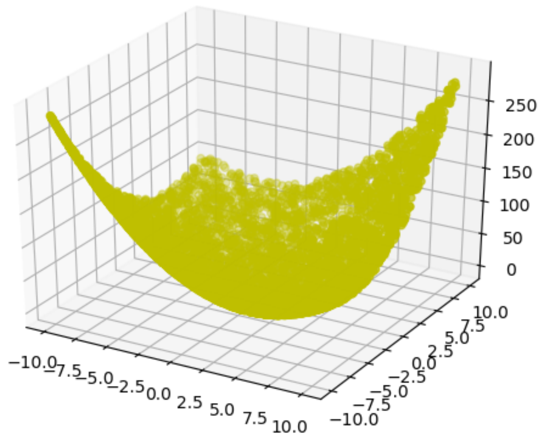


Figure 4.4: data samples

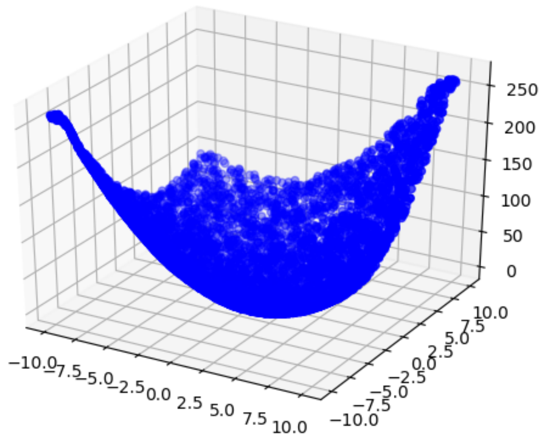


Figure 4.5: model fit

5 CNN

In MNIST there are 60000 training samples and 10000 test samples, take as required randomly 10% of the training samples as training set and randomly 10% of the test samples as test set, one can leverage CNN and F-C to train a network that recognizes if the 2 input images represent the same digit or not, but mutually combining elements of the whole dataset is time and memory costly, an easy way to do this is given an index when training, generating a random pair from either the same label set or the complementary label sets, and when it comes to testing, randomly generating positive or negative pair from MNIST.test_data, thus the dataset is modified to tackle the learning task.

The architecture is as follows:

- Input: $batch\_size \times 2\ plane\ counts \times Height \times Width$
- Convolutional layer: channels: $2 \rightarrow 20$, kernel: 5×5, stride: 1×1
- Relu
- Maxpooling: 2×2
- Convolutional layer: channels: $20 \rightarrow 40$, kernel: 5×5, stride: 1×1
- Relu
- Maxpooling: 2×2
- Flattening(3D to 1D): $40 \times 4 \times 4 \rightarrow 640$
- Full-connected layer: $640 \rightarrow 20$
- Relu
- Full-connected layer: $20 \rightarrow 1$
- Sigmoid
- Binary cross entropy loss

I used ADAM as its optimizer in order to make it converge faster.

In the first place, I tried 30 epochs, 5.1 demonstrates the train and test loss curves.

The final evaluation 5.2:

Continue training, the score will be higher, after 300 epochs,(fairly long time in my laptop) I get 5.3, while the loss evolution is as shown in 5.4

Note that the accuracy improvement is slow, and has been stuck in 98% after 230 epochs.

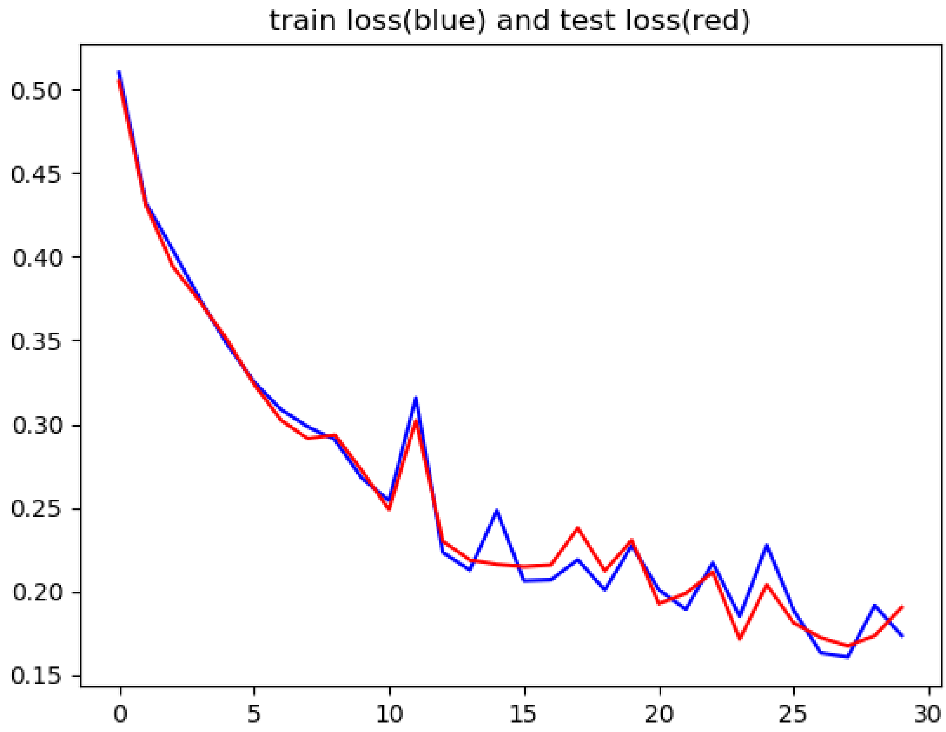


Figure 5.1: learning curve

```
** start training **

Train Epoch: 30 [0/6000 (0%)] Train_Loss: 0.208432
Train Epoch: 30 [640/6000 (11%)]  Train_Loss: 0.203390
Train Epoch: 30 [1280/6000 (22%)] Train_Loss: 0.177258
Train Epoch: 30 [1920/6000 (32%)] Train_Loss: 0.240630
Train Epoch: 30 [2560/6000 (43%)] Train_Loss: 0.152810
Train Epoch: 30 [3200/6000 (54%)] Train_Loss: 0.158519
Train Epoch: 30 [3840/6000 (65%)] Train_Loss: 0.174297
Train Epoch: 30 [4480/6000 (75%)] Train_Loss: 0.164138
Train Epoch: 30 [5120/6000 (86%)] Train_Loss: 0.128179
Train Epoch: 30 [5760/6000 (97%)] Train_Loss: 0.126980

** start testing **


Test set: Average loss: 0.1905,  Accuracy: 928/1000 (93%)   Train set: Average loss: 0.1736,   Accuracy: 5537/6000 (92%)
```

Figure 5.2

```
** start training **

Train Epoch: 300 [0/6000 (0%)]  Train_Loss: 0.004040
Train Epoch: 300 [640/6000 (11%)]       Train_Loss: 0.011570
Train Epoch: 300 [1280/6000 (22%)]      Train_Loss: 0.007282
Train Epoch: 300 [1920/6000 (32%)]      Train_Loss: 0.111136
Train Epoch: 300 [2560/6000 (43%)]      Train_Loss: 0.039400
Train Epoch: 300 [3200/6000 (54%)]      Train_Loss: 0.045840
Train Epoch: 300 [3840/6000 (65%)]      Train_Loss: 0.031541
Train Epoch: 300 [4480/6000 (75%)]      Train_Loss: 0.026480
Train Epoch: 300 [5120/6000 (86%)]      Train_Loss: 0.027079
Train Epoch: 300 [5760/6000 (97%)]      Train_Loss: 0.112205

** start testing **


Test set: Average loss: 0.0651,  Accuracy: 980/1000 (98%)       Train set: Average loss: 0.0465,        Accuracy: 5850/6000 (98%)
```

Figure 5.3

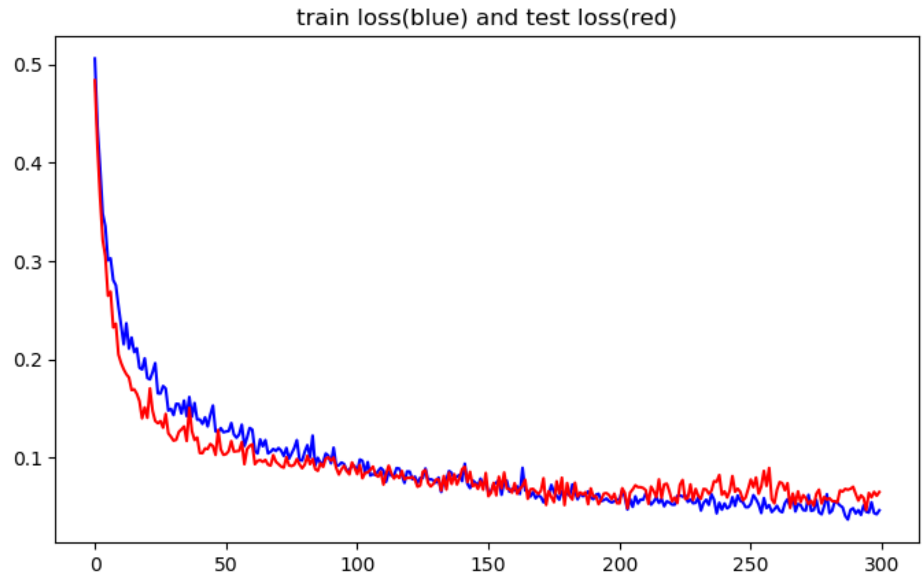


Figure 5.4: loss evolution