Neural Network Hw2
Individual Written Part

Hao-en Sung (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

1 Problem 1

Since it is assumed that the targets follow the Gaussian distribution with noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$, the probability relationship between target t^n and data x^n can be expressed as

$$p(t^n|x^n) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t^n - h(x^n))^2}{2\sigma^2}}.$$

In view that we model data with linear approximation, we have $y = \sum_{i=0}^{K} w_i x_i$, and thus one can write down the probability for whole data as

$$L = \prod_{n=1}^{N} p(t^n|x^n) = \prod_{n=1}^{N} \left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(t^n - \sum_{i=0}^{K} w_i x_i\right)^2}{2\sigma^2}} \right).$$

At the end, one can tell from the error, which is defined as the negative log-version of overall probability, as follows.

$$\begin{aligned} E &= -\ln \left[\prod_{n=1}^{N} \left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(t^n - \sum_{i=0}^{K} w_i x_i\right)^2}{2\sigma^2}} \right) \right] \\ &= \frac{1}{2\sigma^2} \sum_{n=1}^{N} \left(t^n - \sum_{i=0}^{K} w_i x_i \right)^2 + N \cdot \ln(\sqrt{2\pi}\sigma) \end{aligned}$$

It is clear that the minimum value w_i in E can be found by solving an equal minimization problem

$$w^* = \arg\min_{w} \left(t^n - \sum_{i=0}^{K} w_i x_i \right)^2$$

2 Problem 2

(a)

Following the definition of $\delta_i = -\frac{\partial E}{\partial a_i}$ and usage of least squared error function, i.e. $E = \sum_k \frac{1}{2}(t_k - y_k)^2$, I can derive the derivation of δ for both output layer (δ_k) and hidden layers (δ_j) as follows.

$$\begin{aligned}
\delta_k = -\frac{\partial E}{\partial a_k} &= -\frac{\partial E}{\partial g(a_k)} \cdot \frac{\partial g(a_k)}{\partial a_k} \\
&= \frac{t_k - y_k}{g'(a_k)} \cdot g'(a_k) \\
&= t_k - y_k \\
\delta_j = -\frac{\partial E}{\partial a_j} &= -\sum_k \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial a_j} \\
&= -\sum_k \frac{\partial E}{\partial a_k} \cdot \frac{\partial a_k}{\partial z_k} \cdot \frac{\partial z_j}{\partial a_j} \\
&= -h'(a_j) \cdot \sum_k -\delta_k \cdot w_{jk} \\
&= h'(a_j) \cdot \sum_k \delta_k \cdot w_{jk}
\end{aligned}$$

(b)

Similar to what I did in *Problem 2 (a)*, I can derive the gradient for both output layer (w_{jk}) and hidden layers (w_{ij}) with multiple input error function, i.e. $E = \sum_n \sum_k \frac{1}{2}(t_k^n - y_k^n)^2$, as follows.

$$\begin{aligned}
\frac{\partial E}{\partial w_{jk}} &= \sum_n \frac{\partial E^n}{\partial a_k^n} \cdot \frac{\partial a_k^n}{\partial w_{jk}} \\
\frac{\partial E}{\partial w_{ij}} &= \sum_n \frac{\partial E^n}{\partial a_j^n} \cdot \frac{\partial a_j^n}{\partial w_{ij}}
\end{aligned}$$

In view that our purpose is to reduce the error, I can update the weight for output layer w_{jk} and the weight for hidden layer w_{ij} with gradient descent as follows.

$$\begin{aligned}
w_{jk} &= w_{jk} - \eta \frac{\partial E}{\partial w_{jk}} \\
&= w_{jk} - \eta \sum_n \frac{\partial E^n}{\partial a_k^n} \cdot \frac{\partial a_k^n}{\partial w_{jk}} \\
&= w_{jk} + \eta \sum_n \delta_k^n \cdot z_j^n \\
w_{ij} &= w_{ij} - \eta \frac{\partial E}{\partial w_{ij}} \\
&= w_{ij} - \eta \sum_n \frac{\partial E^n}{\partial a_j^n} \cdot \frac{\partial a_j^n}{\partial w_{ij}} \\
&= w_{ij} + \eta \sum_n h'(a_j^n) \cdot \left(\sum_k \delta_k^n \cdot w_{jk} \right) \cdot x_i^n
\end{aligned}$$

(c)

For output layer update, it is clear that we can stack the transpose of column vector $\boldsymbol{w}_k^{\text{out}}$, constant value y_k^n and t_k^n row-wisely into huge matrices, as shown in the following. Note that element of matrix A at r-th row c-th column is represented as A_r^c. It is also noticeable that symbol $\odot$ indicates the element-wise product between two matrices with same sizes; while symbol $\times$ means the standard matrix multiplication.

$$\begin{aligned}
\begin{bmatrix} \vdots \\ (\boldsymbol{w}_k^{\text{out}})^\intercal \\ \vdots \end{bmatrix} &= \begin{bmatrix} \vdots \\ (\boldsymbol{w}_k^{\text{out}})^\intercal \\ \vdots \end{bmatrix} + \eta \cdot \sum_n \left(\begin{bmatrix} \vdots \\ t_k^n \\ \vdots \end{bmatrix} - \begin{bmatrix} \vdots \\ y_k^n \\ \vdots \end{bmatrix} \right) \odot (\boldsymbol{z}^n)^\intercal \\
&= \begin{bmatrix} \vdots \\ (\boldsymbol{w}_k^{\text{out}})^\intercal \\ \vdots \end{bmatrix} + \eta \cdot \left(\begin{bmatrix} & \vdots & \\ \cdots & t_k^n & \cdots \\ & \vdots & \end{bmatrix} - \begin{bmatrix} & \vdots & \\ \cdots & y_k^n & \cdots \\ & \vdots & \end{bmatrix} \right) \odot \begin{bmatrix} \cdots & (\boldsymbol{z^n})^\intercal & \cdots \end{bmatrix}
\end{aligned}$$

Similarly, the hidden layer update can be written as

$$\begin{aligned}
\begin{bmatrix} \vdots \\ (\boldsymbol{w}_j^{\text{hide}})^\intercal \\ \vdots \end{bmatrix} &= \begin{bmatrix} \vdots \\ (\boldsymbol{w}_j^{\text{hide}})^\intercal \\ \vdots \end{bmatrix} + \eta \cdot \sum_n \begin{bmatrix} \vdots \\ h'(a_j) \\ \vdots \end{bmatrix} \odot \left(w^{\text{out}} \times \left(\begin{bmatrix} \vdots \\ t_k^n \\ \vdots \end{bmatrix} - \begin{bmatrix} \vdots \\ y_k^n \\ \vdots \end{bmatrix} \right) \right) \odot (\boldsymbol{x}^n)^\intercal \\
&= \begin{bmatrix} \vdots \\ (\boldsymbol{w}_j^{\text{hide}})^\intercal \\ \vdots \end{bmatrix} + \eta \cdot \begin{bmatrix} & \vdots & \\ \cdots & h'(a_j) & \cdots \\ & \vdots & \end{bmatrix} \\
&\quad \odot \left(w^{\text{out}} \times \left(\begin{bmatrix} & \vdots & \\ \cdots & t_k^n & \cdots \\ & \vdots & \end{bmatrix} - \begin{bmatrix} & \vdots & \\ \cdots & t_k^n & \cdots \\ & \vdots & \end{bmatrix} \right) \right) \odot \begin{bmatrix} \cdots & (\boldsymbol{x^n})^\intercal & \cdots \end{bmatrix}
\end{aligned}$$

Neural Network Hw2: Team Programming Assignment

Hao-en Sung (wrangle1005@gmail.com), Haifeng Huang (hah086@ucsd.edu)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

In this homework, we first implement the basic forward and backpropogation procedure of neural network with Python. Later, we apply many modern and effective tricks, including a better way to initialize weighting matrix, usage of mini-batch, improvement of hidden layer activation function, and momentum techniques.

3 Classification

(d) To check the correctness of our gradient descent, we perform this sanity check for both output layer and hidden layer with different activation functions. The checking procedure is implemented as function *checkGradient()* in Code 1.

The results, shown as Table 1, are recorded in terms of maximum difference between real gradients and numerical approximation for all weighting neurons. It is clear that our gradient descents are very close to the numerical approximation, and thus, it successively passes the test.

	Output Layer	Hidden Layer
Sigmoid	0.0000015771	0.0000006979
Funny tanh	0.0000014305	0.0000043544

Table 1: Maximum Difference between Gradient and Numerical Approximation

(e) We first follow the instructions to load the data as (a), normalize the data as (b), append a column vector filled with 1s, and then shuffle-split the training set into sub-train set and validation set wtih 50000 and 10000 instances, respectively.

At the start of the model training procedure, we initialize two layer weighting matrices with uniform distribution between -1 and 1. For the learning rate η, we initialize it as 0.0001, and it is going to decay with the number of iterations T as $\frac{\eta}{1+0.01T}$.

The structure of model follows exactly as the statement (c): the hidden layer is a simple *Logistic Regression* layer; while the output layer uses the *Softmax* activation function.

The stop condition of our model is designed to stop after three consecutive increases of validation error. With 40 neurons, our model stops at the maximum 1000 iterations. The accuracy and error through each iterations are shown as Fig. 1 and Fig. 2; while the final convergence performance is recorded in Table 2.

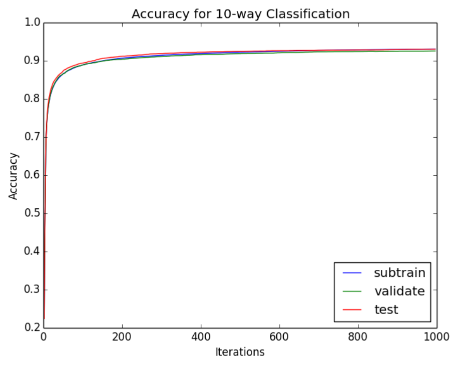

Figure 1: Model Accuracy through Iterations

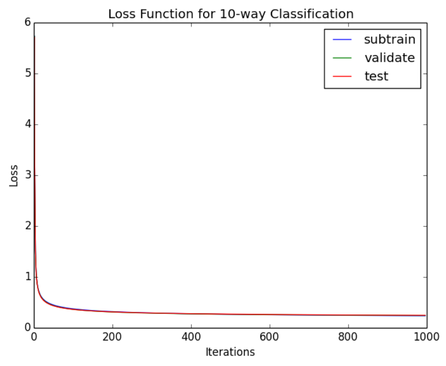

Figure 2: Model Error through Iterations

Subtrain	Validate	Test
0.9309 (0.2372)	0.9257 (0.2424)	0.9305 (0.2469)

Table 2: Model Performance: Accuracy (Error)

4 Adding the "Tricks of the Trade"

To examine the effect of different tricks apart from Problem 3, we run multiple experiments separately.

(a) In this setting, we use $\eta = 0.002$ and change to use *mini-batches* framework, where the batch size is 128. With this special stochastic gradient strategy, we can efficiently accelerate our model convergence speed but with poorer performance. We believe the reason that *mini-batches* strategy does not work so well is that MNIST data is not so large and contain few redundant information. However, on the other hand, the model with mini-batches run faster than the model with *full-batches* framework, which is benefited from the reduce of learning instances. Time comparison between these two models is further recorded in Table 3.

	Real	User	System
Problem 3	19m 39.300s	27m 17.545s	2m 52.916s
Problem 4 (a)	16m 50.099s	20m 46.239s	2m 15.051s

Table 3: Time Comparison between Problem 3 (full-batches) and Problem 4 (mini-batches)

The accuracy and error through each iterations are shown as Fig. 3 and Fig. 4; while the final convergence performance is recorded in Table 4.

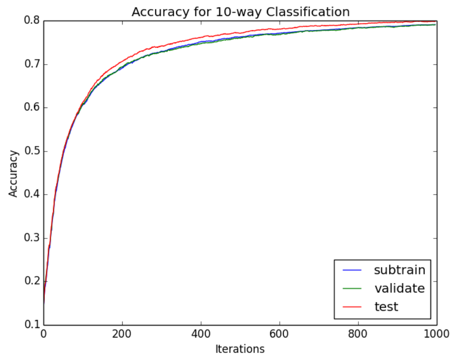

Figure 3: Model Accuracy through Iterations (mini-batches)

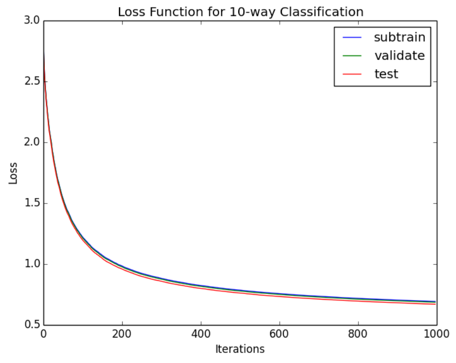

Figure 4: Model Error through Iterations (mini-batches)

(c) In this setting, we use $\eta = 0.00005$ and change the hidden layer activation function from simple logistic regression to *funny tanh*, which calculates

$$y(x) = 1.7159 \cdot \tanh(\frac{2}{3}x). \quad (1)$$

The accuracy and error through each iterations are shown as Fig. 5 and Fig. 6; while the final convergence performance is recorded in Table 4. From the table we can find out that the model does not improve with this trick and even get worse. We believe it is caused by the bad weighting initialization, which causes *funny tanh* has worse performance than simple logistic regression.

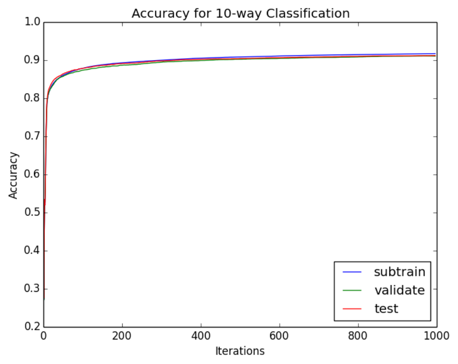

Figure 5: Model Accuracy through Iterations (funny tanh)

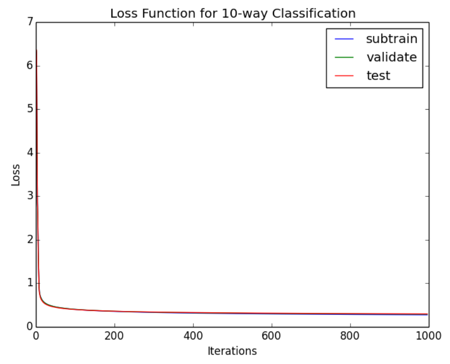

Figure 6: Model Error through Iterations (funny tanh)

(d) In this setting, we use $\eta = 0.0001$ and initialize all weighting matrices with zero mean and $\frac{1}{\sqrt{\text{fan-in}}}$ variance instead of uniform distribution used in Problem 3. The accuracy and error through each iterations are shown as Fig. 7 and Fig. 8; while the final convergence performance is recorded in Table 4. It can be found that this trick works well and effectively enhance the model performance.

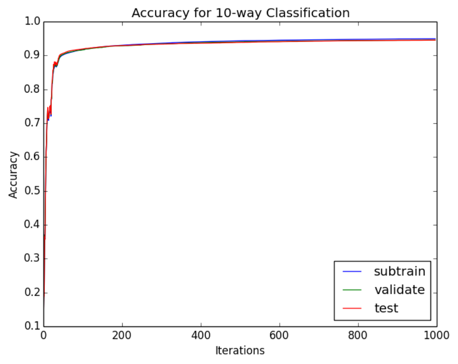

Figure 7: Model Accuracy through Iterations (better initialization)

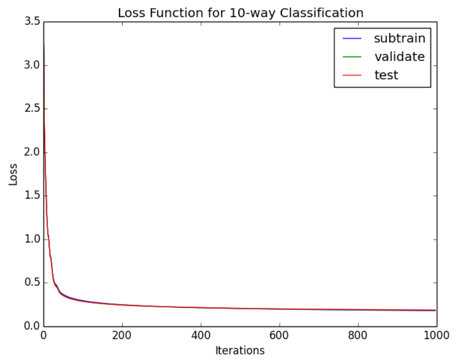

Figure 8: Model Error through Iterations (better initialization)

(e) In this setting, we use $\eta = 0.0001$ and apply momentum strategy while updating the gradient direction: add the weight matrix of previous iteration to the updated term multiplied by a fix value $\mu = 0.9$. This strategy can efficiently improve the gradient direction and hence accelerate the model convergence and performance. The accuracy and error through each iterations are shown as Fig. 9 and Fig. 10; while the final convergence performance is recorded in Table 4. It can be found that this trick works well and effectively enhance the model performance.

On top of that, we also run the experiment on model with every trick, i.e. (a) + (c) + (d) + (e); however, the result is still worse than Problem 3, just as what we get in extended questions in Problem 5. The accuracy and error through each iterations are shown as Fig. 11 and Fig. 12; while the final

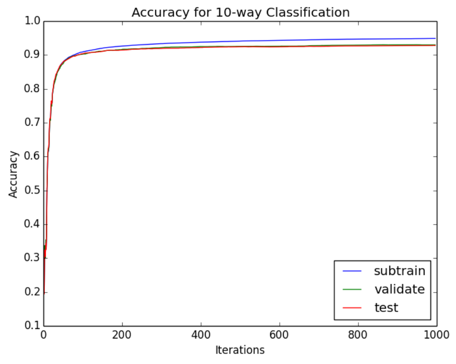

Figure 9: Model Accuracy through Iterations (momentum)

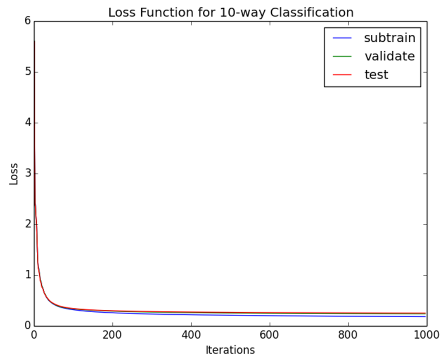

Figure 10: Model Error through Iterations (momentum)

convergence performance is recorded in Table 4. We believe the reason is that MNIST data is not so large and contain few redundant information. Thus, *mini-batches* is not a effective trick here, which can even degrade our model performance. To prove this, we put an additional experiment with every trick except mini-batches, i.e. (c) + (d) + (e), and we get the best result among all other settings, as recorded in Table 4 with red color.

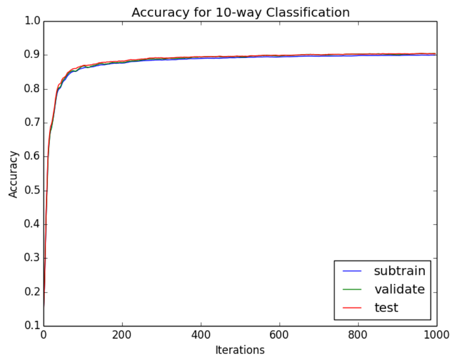

Figure 11: Model Accuracy through Iterations (full tricks)

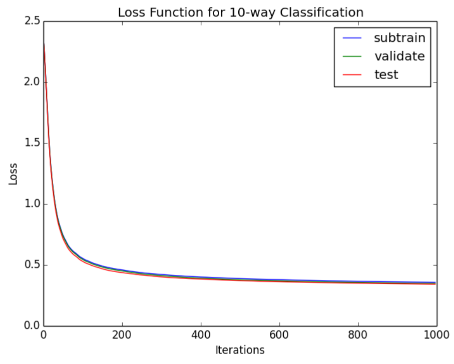

Figure 12: Model Error through Iterations (full tricks)

	Subtrain	Validate	Test
no trick	0.9309 (0.2372)	0.9257 (0.2424)	0.9305 (0.2469)
(a) mini-batches	0.7907 (0.6930)	0.7904 (0.6864)	0.7981 (0.6702)
(c) funny tanh	0.9173 (0.2772)	0.9107 (0.2917)	0.9122 (0.2989)
(d) initialization	0.9495 (0.1805)	0.9455 (0.1849)	0.9452 (0.1879)
(e) momentum	0.9488 (0.1821)	0.9298 (0.2389)	0.9279 (0.2523)
(a), (c), (d), (e)	0.8997 (0.3575)	0.9041 (0.3478)	0.9036 (0.3422)
(c), (d), (e)	0.9902 ((0.0407)	0.9689 (0.1033)	0.9696 (0.1009)

Table 4: Model Performance Comparison: Accuracy (Error)

5 Experiment with Network Topology

(a) On top of the fourth problem, we change the number of the hidden neurons to 80 and 20, and use initial learning rate 0.0001 because we didn't divide 128 when calculating the update term for weight matrices. We record the accuracy and loss for 80 hidden neurons in Fig. 13 and Fig. 14; while accuracy and loss for 20 hidden neurons is shown as Fig. 15 and Fig. 16.

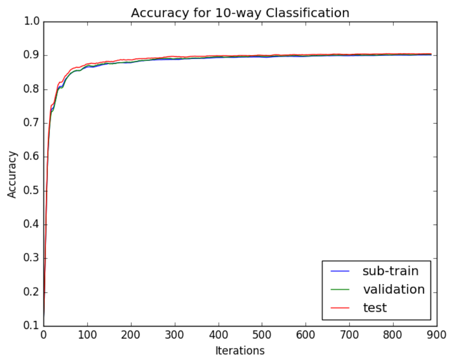

Figure 13: Accuracy for 80 Hidden Neurons (mini-batch)

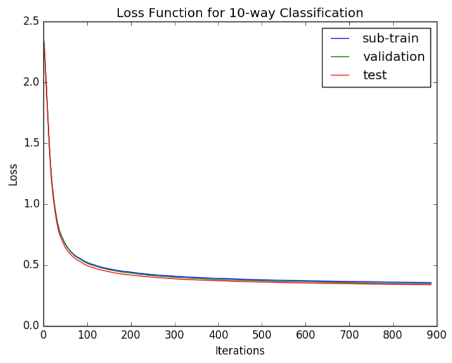

Figure 14: Error for 80 Hidden Neurons (mini-batch)

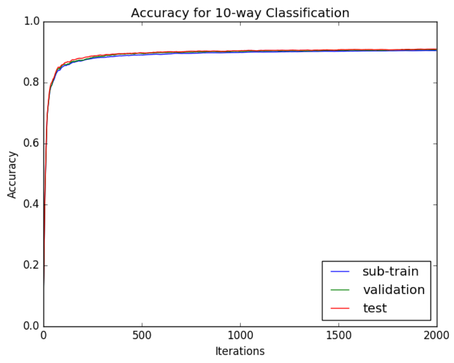

Figure 15: Accuracy for 20 Hidden Neurons (mini-batch)

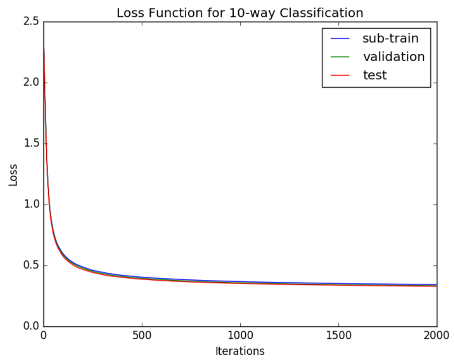

Figure 16: Error for 20 Hidden Neurons (mini-batch)

The accuracy and error comparison for mini-batches for single layer is shown as Table 5.

	Subtrain	Validate	Test
80 (mini-batch)	0.90094 (0.35616)	0.9028 (0.34733)	0.9047 (0.33886)
20 (mini-batch)	0.90478 (0.34282)	0.9077 (0.33330)	0.9098 (0.32899)
(a), (c), (d), (e)	0.8997 (0.3575)	0.9041 (0.3478)	0.9036 (0.3422)

Table 5: Accuracy (Error) for Mini-Batches for Single Hidden Layer

Since we found that mini-batch has worse performance than full batch, we experimented on full batch for the above architecture of neural networks. For all the following experiments, we set $\eta = 0.00002$. The accuracy and loss results for 80 hidden neurons are shown as Fig. 17 and Fig. 18; while the accuracy and loss results for 20 hidden neurons are shown as Fig. 19 and Fig. 20.

The additional accuracy and error comparison for single layer with full-batches is shown as Table 6.

Discussion:

1. Full batch learning has better performance than mini-batches, because the redundancy of the dataset is not too great and the instances of dataset is big enough.
2. For 20 and 80 hidden neurons, the accuracy is almost the same as 40 hidden neurons in the fourth problem and takes more time to train. If there are too many hidden neurons, the

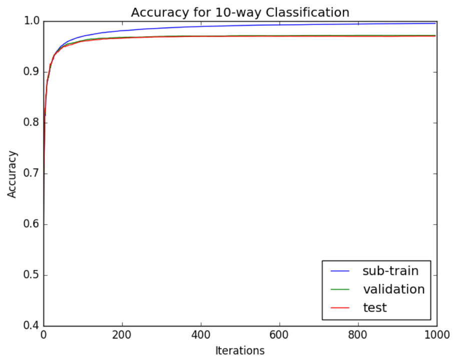

Figure 17: Accuracy for 80 Hidden Neurons (full batch)

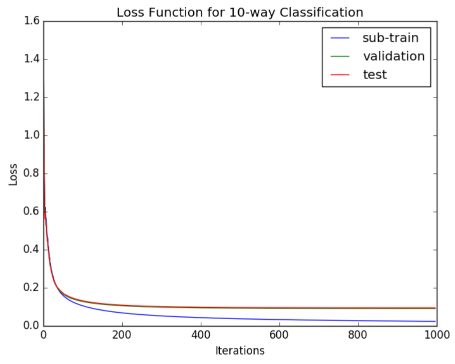

Figure 18: Error for 80 Hidden Neurons (full batch)

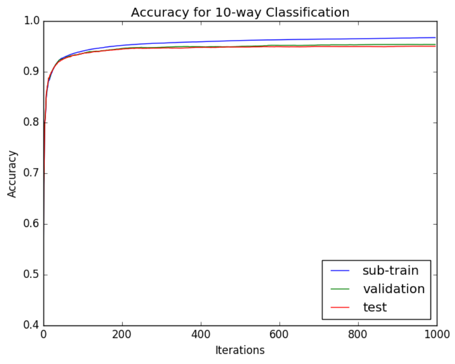

Figure 19: Accuracy for 20 Hidden Neurons (full batch)

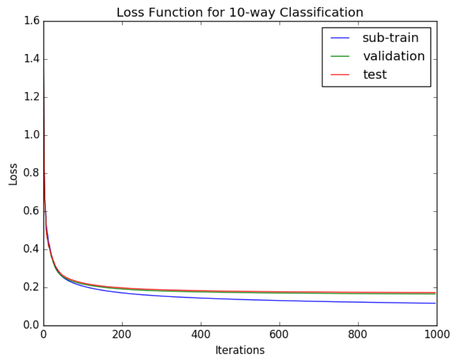

Figure 20: Error for 20 Hidden Neurons (full batch)

	Subtrain	Validate	Test
80 (full batch)	0.99552 (0.02385)	0.9718 (0.09123)	0.9704 (0.09488)
20 (full batch)	0.9673 (0.11658)	0.9538 (0.16529)	0.9503 (0.17247)
(c), (d), (e)	0.9902 ((0.0407)	0.9689 (0.1033)	0.9696 (0.1009)

Table 6: Accuracy (Error) for Full-Batches for Single Hidden Layer

overfit may occur and more time is needed to converge; if there are not enough hidden neurons, the representation of the input patterns is not fully learned by the neural network. Thus we need to choose the moderate hidden neurons to make the neural network work better.

(b) After that, we use two layers of hidden neurons, with 40 hidden neurons for both hidden layers, and learning rate is 0.00001. The accuracy of two hidden layers is shown as Fig. 21. The loss function of two hidden layers is shown as Fig. 22.

The accuracy and error comparison for two hidden layers with mini-batches is shown as Table 7.

For full batch, the accuracy of two hidden layers with 40 neurons is shown as Fig. 23. The loss function of two hidden layers with 40 neurons is shown as Fig. 24. Here we use 0.00001 as initial learning rate.

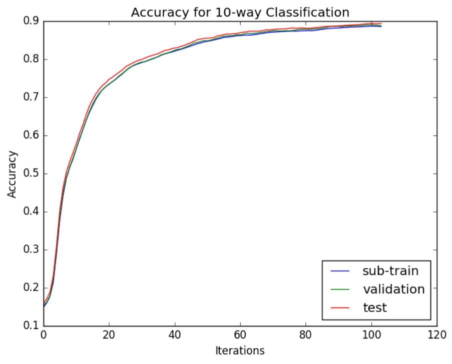

Figure 21: Accuracy for Two Hidden Neurons (mini-batch)

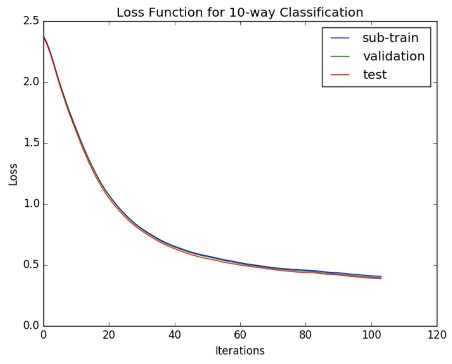

Figure 22: Error for Two Hidden Neurons (mini-batch)

	Subtrain	Validate	Test
two hidden layers (mini-batch)	0.88564 (0.40682)	0.8879 (0.39607)	0.8937 (0.38792)
(a), (c), (d), (e)	0.8997 (0.3575)	0.9041 (0.3478)	0.9036 (0.3422)

Table 7: Accuracy (Error) for Mini-Batches for Two Hidden Layers

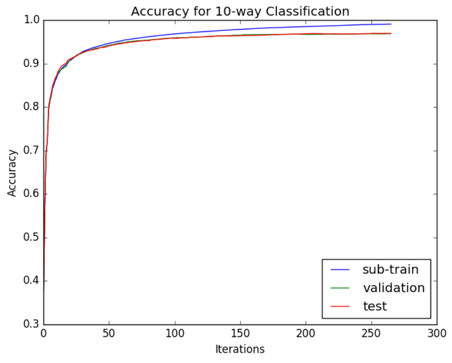

Figure 23: Accuracy for Two Hidden Neurons (full batch)

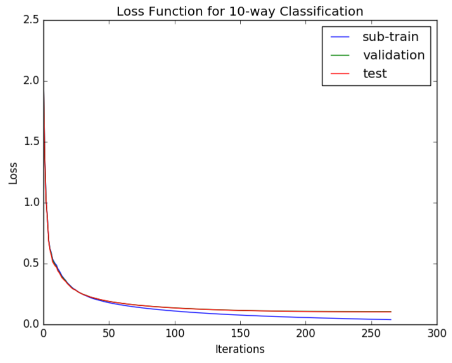

Figure 24: Error for Two Hidden Neurons (full batch)

The accuracy and error comparison for full-batches for two hidden layers is shown as Table 8.

	Subtrain	Validate	Test
two hidden layers (full batch)	0.9907 (0.03931)	0.9687 (0.10544)	0.9693 (0.10268)
(c), (d), (e)	0.9902 ((0.0407)	0.9689 (0.1033)	0.9696 (0.1009)

Table 8: Accuracy (Error) for Full-Batches for Two Hidden Layers

Discussion:

1. For two hidden layers, we didn't get enough performance improvement. We believe it is caused by error propagation in deeper model during learning. So it will be easier to overfit than single hidden layer, so we can try using shared weights to avoid overfit.

6 Work Distribution

Hao-en Sung is responsible for problem 3 and $4a)$, Haifeng Huang is responsible for $4c)$, $4d)$, $4e)$, $4f)$, $5a)$ and $5b)$.

Appendix

Implementation Codes

For code clarity and easiness of maintenance, we write all solutions in one file but within different functions. One can easily alter the definition *problem* in line 27 to certain keyword for specific task. For example: 'prob_3' for Problem 3, 'prob_4' for Problem 4, 'prob_5' for Problem 5, and 'gradient' for gradient correctness check.

Code Listing 1: Code for Homework 2

```
import os, struct
from array import array as pyarray
import numpy as np
import matplotlib.pyplot as plt
import math

from sklearn import preprocessing
from sklearn import model_selection
from sklearn.model_selection import ShuffleSplit

np.random.seed(514)
np.set_printoptions(suppress=True, precision=10)

INF = 10000000
MAX_ITER = 1000
eps = 1e-10

tn_n = 60000
tt_n = 10000
ft_n = 784

nh = 40
nh_1 = 40
nh_2 = 40
batch = 128

problem = 'prob_4' # gradient, prob_3, prob_4, prob_5
trick = 'a' # a, c, d, e, all
img_folder = '../res/'

if problem == 'prob_3':
    eta = 0.00001
elif problem == 'prob_4':
    # trick (a)
    if trick == 'a':
        eta = 0.002
    elif trick == 'c':
        eta = 0.00005
    elif trick == 'd':
        eta = 0.0001
    elif trick == 'e':
        eta = 0.0001
    else:
        eta = 0.0001
    # trick (e)
    if trick == 'e' or trick == 'all':
        mu = 0.9
    else:
        mu = 0.0
else:
    eta = 0.0001
    mu = 0.9

def load_mnist(dataset, digits=np.arange(10), path='../dat/'):
    if dataset == 'training':
        fname_img = os.path.join(path, 'train-images-idx3-ubyte')
        fname_lbl = os.path.join(path, 'train-labels-idx1-ubyte')
    elif dataset == 'testing':
        fname_img = os.path.join(path, 't10k-images-idx3-ubyte')
        fname_lbl = os.path.join(path, 't10k-labels-idx1-ubyte')
    else:
        raise ValueError("dataset must be 'testing' or 'training'")

    flbl = open(fname_lbl, 'rb')
    magic_nr, size = struct.unpack('>II', flbl.read(8))
    lbl = pyarray('b', flbl.read())
    flbl.close()

    fimg = open(fname_img, 'rb')
    magic_nr, size, rows, cols = struct.unpack('>IIII', fimg.read(16))
    img = pyarray('B', fimg.read())
    fimg.close()

    ind = [ k for k in range(size) if lbl[k] in digits ]
    N = len(ind)

    images = np.zeros((N, rows, cols), dtype=np.uint8)
    labels = np.zeros((N, 1), dtype=np.int8)
    for i in range(len(ind)):
        images[i] = np.array(img[ ind[i]*rows*cols : \
                (ind[i]+1)*rows*cols ]).reshape((rows, cols))
        labels[i] = lbl[ind[i]]

    return images, labels

def calERR(x, y, p):
    assert(x.shape[0] == y.shape[0])
    val = 0.0
    E_mat = y * np.log(p + eps)
    val = - E_mat.sum()/float(x.shape[0])
    return val

def calACC(x, y, p):
    assert(x.shape[0] == y.shape[0] and y.shape[0] == p.shape[0])
    num=0
    for row in range(p.shape[0]):
        largest=0
        index=0
        for column in range(y.shape[1]):
            if p[row,column]>largest:
                index=column
                largest=p[row,column]
        if y[row,index]==1:
            num+=1
    return 1.0 / x.shape[0] * num

# single hidden layer
def predict_s(W_ij, W_jk, x):
    A_j = np.dot(x, W_ij)
    # trick (c)
    if problem == 'prob_4' and (trick == 'c' or trick == 'all'):
        Y_j = 1.7159 * np.tanh(2.0 / 3 * A_j)
    else:
        Y_j = 1.0 / (1 + np.exp(-A_j))
    X_k = np.hstack((Y_j, np.ones((Y_j.shape[0],1))))
    A_k = np.dot(X_k, W_jk)
    E_k = np.matrix(np.exp(A_k))
    Y_k = np.array(E_k/(E_k.sum(1)+eps))
    return Y_k, X_k, Y_j, A_j

# two hidden layer
def predict_t(W_ij, W_jk, W_kl, x):
    A_j = np.dot(x, W_ij)
    Y_j = 1.7159 * np.tanh(2.0 / 3 * A_j)  # tanh
    X_k = np.hstack((Y_j, np.ones((Y_j.shape[0],1))))
    A_k = np.dot(X_k, W_jk)
    Y_k = 1.7159 * np.tanh(2.0 / 3 * A_k)  # tanh
    X_l = np.hstack((Y_k, np.ones((Y_k.shape[0],1))))
    A_l = np.dot(X_l, W_kl)
    E_l = np.matrix(np.exp(A_l))
    Y_l = np.array(E_l/(E_l.sum(1)+eps))
    return Y_l, X_l, A_k, X_k, A_j

def checkGradient(x, y):
    x_d, y_d = x.shape[1], y.shape[1]
    W_ij = np.random.normal(0, 1.0/(x_d**0.5), (x_d, nh))
    W_jk = np.random.normal(0, 1.0/((nh+1)**0.5), (nh+1, y_d))
    Yk, Xk, Yj, Aj = predict_s(W_ij, W_jk, x)

    nW_jk = np.dot(Xk.transpose(), y - Yk)
    nW_ij = np.dot(x.transpose(), \
            np.dot(y - Yk, W_jk.transpose()[:,:nh]) \
            * Yj * (1 - Yj))

    epsilon = 0.01

    # output layer
    max_diff_output = 0.0
    for j in range(nh+1):
        for k in range(y.shape[1]):
            W_p = np.copy(W_jk)
            W_p[j,k] += epsilon
            Yp, _, _, _ = predict_s(W_ij, W_p, x)
            ep = calERR(x, y, Yp)

            W_m = np.copy(W_jk)
            W_m[j,k] -= epsilon
            Ym, _, _, _ = predict_s(W_ij, W_m, x)
            em = calERR(x, y, Ym)

            diff = abs(nW_jk[j,k] + (ep-em) / (2 * epsilon))
            max_diff_output = max(max_diff_output, diff)

    # hidden layer
    max_diff_hidden= 0.0
    for i in range(x.shape[1]):
        for j in range(nh):
            W_p = np.copy(W_ij)
            W_p[i,j] += epsilon
            Yp, _, _, _ = predict_s(W_p, W_jk, x)
            ep = calERR(x, y, Yp)

            W_m = np.copy(W_ij)
            W_m[i,j] -= epsilon
            Ym, _, _, _ = predict_s(W_m, W_jk, x)
            em = calERR(x, y, Ym)

            diff = abs(nW_ij[i,j] + (ep-em) / (2 * epsilon))
```

```
                max_diff_hidden = max(max_diff_hidden, diff)

    # results
    print('Maximum Difference for Output Layer: ' \
            + str(max_diff_output))
    print('Maximum Difference for Hidden Layer: ' \
            + str(max_diff_hidden))

def backProp_prob_3(sx, sy, vx, vy, tx, ty):
    assert(sx.shape[0] == sy.shape[0])
    assert(vx.shape[0] == vy.shape[0])
    assert(tx.shape[0] == ty.shape[0])

    # initialization
    sx_d, sy_d = sx.shape[1], sy.shape[1]
    W_ij = np.random.uniform(-1.0, 1.0, (sx_d, nh))
    W_jk = np.random.uniform(-1.0, 1.0, (nh+1, sy_d))

    Ys_k, Xs_k, Ys_j, As_j = predict_s(W_ij, W_jk, sx)
    vld_v = INF
    cnt = 0

    # results
    stn_err = []
    vld_err = []
    tt_err = []
    stn_acc = []
    vld_acc = []
    tt_acc = []

    for t in xrange(MAX_ITER):
        n_eta = eta / (1 + 0.01 * t)

        # gradient
        nW_jk = np.dot(Xs_k.transpose(), sy - Ys_k)
        nW_ij = np.dot(sx.transpose(), \
                np.dot(sy - Ys_k, W_jk.transpose()[:,:nh]) \
                * Ys_j * (1 - Ys_j))

        # update
        W_jk = W_jk + n_eta * nW_jk
        W_ij = W_ij + n_eta * nW_ij

        # predict
        Ys_k, Xs_k, Ys_j, As_j = predict_s(W_ij, W_jk, sx)
        Yv_k, Xv_k, Yv_j, Av_j = predict_s(W_ij, W_jk, vx)
        Yt_k, Xt_k, Yt_j, At_j = predict_s(W_ij, W_jk, tx)

        # calculate err
        n_stn_v = calERR(sx, sy, Ys_k)
        n_vld_v = calERR(vx, vy, Yv_k)
        n_tt_v = calERR(tx, ty, Yt_k)
        stn_err.append(n_stn_v)
        vld_err.append(n_vld_v)
        tt_err.append(n_tt_v)

        # calculate acc
        n_stn_a = calACC(sx, sy, Ys_k)
        n_vld_a = calACC(vx, vy, Yv_k)
        n_tt_a = calACC(tx, ty, Yt_k)
        stn_acc.append(n_stn_a)
        vld_acc.append(n_vld_a)
        tt_acc.append(n_tt_a)

        # stop condition
        if n_vld_v >= vld_v:
            cnt += 1
        else:
            bW_ij = np.copy(W_ij)
            bW_jk = np.copy(W_jk)
            vld_v = n_vld_v
            cnt = 0

        if cnt == 3:
            print 'Stop at #' + str(t)
            return bW_ij, bW_jk, stn_err[:-3], vld_err[:-3], \
                tt_err[:-3], stn_acc[:-3], vld_acc[:-3], tt_acc[:-3]

    print 'Maximum iterations are reached'
    return bW_ij, bW_jk, stn_err[:-3], vld_err[:-3], \
        tt_err[:-3], stn_acc[:-3], vld_acc[:-3], tt_acc[:-3]

def backProp_prob_4(sx, sy, vx, vy, tx, ty):
    assert(sx.shape[0] == sy.shape[0])
    assert(vx.shape[0] == vy.shape[0])
    assert(tx.shape[0] == ty.shape[0])

    # initialization
    sx_d, sy_d = sx.shape[1], sy.shape[1]

    # trick (d)
    if trick == 'd' or trick == 'all':
        W_ij = np.random.normal(0, 1.0/(sx_d)**0.5, (sx_d, nh))
        W_jk = np.random.normal(0, 1.0/(nh+1)**0.5, (nh+1, sy_d))
    else:
        W_ij = np.random.uniform(-1.0, 1.0, (sx_d, nh))
        W_jk = np.random.uniform(-1.0, 1.0, (nh+1, sy_d))

    nW_ij = np.zeros((sx_d, nh))
    nW_jk = np.zeros((nh+1, sy_d))

    vld_v = INF
    cnt = 0

    # results
    stn_err = []
    vld_err = []
    tt_err = []
    stn_acc = []
    vld_acc = []
    tt_acc = []

    num_batch = int(math.ceil(sx.shape[0]/batch))
    for t in range(MAX_ITER):
        i = t % num_batch

        # trick (a)
        if trick == 'a' or trick == 'all':
            sx_batch = sx[i*batch: min((i+1)*batch, sx.shape[0]), :]
            sy_batch = sy[i*batch: min((i+1)*batch, sx.shape[0]), :]
        else:
            sx_batch = sx;
            sy_batch = sy;

        n_eta = eta / (1 + 0.01 * t)

        # gradient
        Ys_k, Xs_k, Ys_j, As_j = predict_s(W_ij, W_jk, sx_batch)
        nW_jk = np.dot(Xs_k.transpose(), sy_batch - Ys_k) + mu * nW_jk
        # trick (c)
        if trick == 'c' or trick == 'all':
            nW_ij = np.dot(sx_batch.transpose(), \
                    np.dot(sy_batch - Ys_k, W_jk.transpose()[:,:nh]) \
                    * 1.7159 * 8 * np.exp(4.0 / 3 * As_j) / 3 / \
                    (np.exp(4.0 / 3 * As_j) + 1) ** 2) + mu * nW_ij
        else:
            nW_ij = np.dot(sx_batch.transpose(), \
                    np.dot(sy_batch - Ys_k, W_jk.transpose()[:,:nh]) \
                    * Ys_j * (1 - Ys_j)) + mu * nW_ij

        # update
        W_jk = W_jk + n_eta * nW_jk
        W_ij = W_ij + n_eta * nW_ij

        # predict
        Ys_k, Xs_k, Ys_j, As_j = predict_s(W_ij, W_jk, sx)
        Yv_k, Xv_k, Yv_j, Av_j = predict_s(W_ij, W_jk, vx)
        Yt_k, Xt_k, Yt_j, At_j = predict_s(W_ij, W_jk, tx)

        # calculate err
        n_stn_v = calERR(sx, sy, Ys_k)
        n_vld_v = calERR(vx, vy, Yv_k)
        n_tt_v = calERR(tx, ty, Yt_k)
        stn_err.append(n_stn_v)
        vld_err.append(n_vld_v)
        tt_err.append(n_tt_v)

        # calculate acc
        n_stn_a = calACC(sx, sy, Ys_k)
        n_vld_a = calACC(vx, vy, Yv_k)
        n_tt_a = calACC(tx, ty, Yt_k)
        stn_acc.append(n_stn_a)
        vld_acc.append(n_vld_a)
        tt_acc.append(n_tt_a)

        # stop condition
        if n_vld_v >= vld_v:
            cnt += 1
        else:
            bW_ij = np.copy(W_ij)
            bW_jk = np.copy(W_jk)
            vld_v = n_vld_v
            cnt = 0

        if cnt == 3:
            print 'Stop at #' + str(t)
            return bW_ij, bW_jk, stn_err[:-3], vld_err[:-3], \
                tt_err[:-3], stn_acc[:-3], vld_acc[:-3], tt_acc[:-3]

    print 'Maximum iterations are reached'
    return bW_ij, bW_jk, stn_err[:-3], vld_err[:-3], \
        tt_err[:-3], stn_acc[:-3], vld_acc[:-3], tt_acc[:-3]

def backProp_prob_5(sx, sy, vx, vy, tx, ty):
    assert(sx.shape[0] == sy.shape[0])
    assert(vx.shape[0] == vy.shape[0])
    assert(tx.shape[0] == ty.shape[0])

    # initialization
    sx_d, sy_d = sx.shape[1], sy.shape[1]
    W_ij = np.random.normal(0, 1.0/(sx_d)**0.5, (sx_d, nh_1))
    W_jk = np.random.normal(0, 1.0/(nh_1+1)**0.5, (nh_1+1, nh_2))
    W_kl = np.random.normal(0, 1.0/(nh_2+1)**0.5, (nh_2+1, sy_d))
    nW_ij = np.zeros((sx_d, nh_1))
```

```
nW_jk = np.zeros((nh_1+1, nh_2))
nW_kl = np.zeros((nh_2+1, sy_d))

vld_v = INF
cnt = 0

# results
stn_err = []
vld_err = []
tt_err = []
stn_acc = []
vld_acc = []
tt_acc = []

num_batch = int(math.ceil(sx.shape[0]/batch))
for t in range(MAX_ITER):
    i = t % num_batch
    sx_batch = sx[i*batch: min((i+1)*batch, sx.shape[0]), :]
    sy_batch = sy[i*batch: min((i+1)*batch, sx.shape[0]), :]

    n_eta = eta / (1 + 0.01 * t)

    # gradient
    Ys_l, Xs_l, As_k, Xs_k, As_j = \
            predict_t(W_ij, W_jk, W_kl, sx_batch)

    delta_l = sy_batch - Ys_l
    nW_kl = np.dot(Xs_l.transpose(), delta_l) + mu * nW_kl

    delta_k = np.dot(delta_l, W_kl.transpose()[:,:nh_2]) \
            * 1.7159 * 8 * np.exp(4.0 / 3 * As_k) / 3 / \
            (np.exp(4.0 / 3 * As_k) + 1) ** 2
    nW_jk = np.dot(Xs_k.transpose(), delta_k) + mu * nW_jk

    delta_j = np.dot(delta_k, W_jk.transpose()[:,:nh_1]) \
            * 1.7159 * 8 * np.exp(4.0 / 3 * As_j) / 3 / \
            (np.exp(4.0 / 3 * As_j) + 1) ** 2
    nW_ij = np.dot(sx_batch.transpose(), delta_j) + mu * nW_ij

    # update
    W_kl = W_kl + n_eta * nW_kl
    W_jk = W_jk + n_eta * nW_jk
    W_ij = W_ij + n_eta * nW_ij

    # predict
    Ys_l, Xs_l, As_k, Xs_k, As_j = predict_t(W_ij, W_jk, W_kl, sx)
    Yv_l, Xv_l, Av_k, Xv_k, Av_j = predict_t(W_ij, W_jk, W_kl, vx)
    Yt_l, Xt_l, At_k, Xt_k, At_j = predict_t(W_ij, W_jk, W_kl, tx)

    # calculate err
    n_stn_v = calERR(sx, sy, Ys_l)
    n_vld_v = calERR(vx, vy, Yv_l)
    n_tt_v = calERR(tx, ty, Yt_l)
    stn_err.append(n_stn_v)
    vld_err.append(n_vld_v)
    tt_err.append(n_tt_v)

    # calculate acc
    n_stn_a = calACC(sx, sy, Ys_l)
    n_vld_a = calACC(vx, vy, Yv_l)
    n_tt_a = calACC(tx, ty, Yt_l)
    stn_acc.append(n_stn_a)
    vld_acc.append(n_vld_a)
    tt_acc.append(n_tt_a)
```

```
        # stop condition
        if n_vld_v >= vld_v:
            cnt += 1
        else:
            bW_ij = np.copy(W_ij)
            bW_jk = np.copy(W_jk)
            bW_kl = np.copy(W_kl)
            vld_v = n_vld_v
            cnt = 0

        if cnt == 3:
            print 'Stop at #' + str(t)
            return bW_ij, bW_jk, bW_kl, \
                    stn_err[:-3], vld_err[:-3], tt_err[:-3], \
                    stn_acc[:-3], vld_acc[:-3], tt_acc[:-3]

    print 'Maximum iterations are reached'
    return bW_ij, bW_jk, bW_kl, \
            stn_err[:-3], vld_err[:-3], tt_err[:-3], \
            stn_acc[:-3], vld_acc[:-3], tt_acc[:-3]

## read dataset
tn_x, tn_y = load_mnist('training')
tn_x, tn_y = tn_x[:tn_n, :, :], tn_y[:tn_n, :]
tn_x = tn_x.reshape(tn_n, ft_n)
tn_cy = np.zeros((tn_n,10))
for i in range(tn_n):
    tn_cy[i,tn_y[i,0]]=1.0

tt_x, tt_y = load_mnist('testing')
tt_x, tt_y = tt_x[:tt_n, :, :], tt_y[:tt_n, :]
tt_x = tt_x.reshape(tt_n, ft_n)
tt_cy = np.zeros((tt_n,10))
for i in range(tt_n):
    tt_cy[i,tt_y[i,0]]=1.0

## preprocessing
tn_x = 1.0 * (tn_x - np.mean(tn_x, axis=1, keepdims=True)) / 255
tt_x = 1.0 * (tt_x - np.mean(tt_x, axis=1, keepdims=True)) / 255

## append 1s
tn_x = np.hstack([tn_x, np.ones((tn_n, 1))])
tt_x = np.hstack([tt_x, np.ones((tt_n, 1))])

# hold out 10000 train as validation
ss = ShuffleSplit(n_splits=1, test_size=1.0/6, random_state=0)
stn_idx, vld_idx = \
        list(ss.split(np.arange(tn_x.shape[0])))[0]
stn_x = tn_x[stn_idx, :]
stn_y = tn_cy[stn_idx, :]
vld_x = tn_x[vld_idx, :]
vld_y = tn_cy[vld_idx, :]

if problem == 'gradient':
    checkGradient(tn_x[[0], :], tn_cy[[0], :])
    exit(0)
elif problem == 'prob_3':
    W_ij, W_jk, stn_err, vld_err, tt_err, \
            stn_acc, vld_acc, tt_acc = backProp_prob_3( \
            stn_x, stn_y, vld_x, vld_y, tt_x, tt_cy)
elif problem == 'prob_4':
    W_ij, W_jk, stn_err, vld_err, tt_err, \
            stn_acc, vld_acc, tt_acc = backProp_prob_4( \
            stn_x, stn_y, vld_x, vld_y, tt_x, tt_cy)
else:
    W_ij, W_jk, W_kl, stn_err, vld_err, tt_err, \
            stn_acc, vld_acc, tt_acc = backProp_prob_5( \
            stn_x, stn_y, vld_x, vld_y, tt_x, tt_cy)

if problem == 'prob_4':
    problem = problem + '_' + trick

# plot err
plt.figure()
plt.plot(stn_err, label='subtrain')
plt.plot(vld_err, label='validate')
plt.plot(tt_err, label='test')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss Function for 10-way Classification')
plt.legend(loc=1)
plt.savefig(img_folder + problem + '_err.png')

# plot acc
plt.figure()
plt.plot(stn_acc, label='subtrain')
plt.plot(vld_acc, label='validate')
plt.plot(tt_acc, label='test')
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.title('Accuracy for 10-way Classification')
plt.legend(loc=4)
plt.savefig(img_folder + problem + '_acc.png')

# output performance
print(problem + ', eta(' + str(eta) + '), nh(' + str(nh) + ')')
print('subtrain: ' + str(stn_acc[-1]) + ' (' + str(stn_err[-1]) + ')')
print('validate: ' + str(vld_acc[-1]) + ' (' + str(vld_err[-1]) + ')')
print('test: ' + str(tt_acc[-1]) + ' (' + str(tt_err[-1]) + ')')
```