CSC311 HOMEWORK 2

Zhuoyue Lyu

Due Thu, Oct. 17, 23:59 PM

1. Logistic Regression

1.1. Bayes Rule

Use Bayes rule to show that $p(t = 1|\mathbf{x})$ takes the form of a logistic function:

$$p(t = 1|\mathbf{x}) = \sigma(\mathbf{w^T x} + b) = \frac{1}{1 + exp(-\sum_{i=1}^{D} w_i x_i - b)}$$

Derive expressions for the weights $\mathbf{w} = (w_1, ..., w_D)^T$ and the bias b in terms of the parameters of the class likelihoods and priors (i.e., $\mu_{i0}, \mu_{i1}, \sigma_i$ and α)

Solution :

$$\begin{aligned}
p(t = 1|\mathbf{x}) &= \frac{p(\mathbf{x}|t = 1)p(t = 1)}{p(\mathbf{x})} && \text{(By Bayes' Rule)} \\
&= \frac{p(\mathbf{x}|t = 1)p(t = 1)}{p(\mathbf{x}|t = 1)p(t = 1) + p(\mathbf{x}|t = 0)p(t = 0)} \\
&= \frac{1}{1 + \frac{p(\mathbf{x}|t=0)p(t=0)}{p(\mathbf{x}|t=1)p(t=1)}} = \frac{1}{1 + exp\big(ln\frac{p(\mathbf{x}|t=0)p(t=0)}{p(\mathbf{x}|t=1)p(t=1)}\big)} \\
&= \frac{1}{1 + exp\big(ln\frac{p(\mathbf{x}|t=0)}{p(\mathbf{x}|t=1)} + ln\frac{p(t=0)}{p(t=1)}\big)} \\
&= \frac{1}{1 + exp\big(ln(\prod_{i=1}^{D}\frac{p(x_i|t=0)}{p(x_i|t=1)}) + ln\frac{1-\alpha}{\alpha}\big)} \\
& && \text{(Since } \mathbf{x} \text{ are conditionally independent given } t) \\
&= \frac{1}{1 + exp\big(\sum_{i=1}^{D}(ln\frac{p(x_i|t=0)}{p(x_i|t=1)}) + ln\frac{1-\alpha}{\alpha}\big)} \\
&= \frac{1}{1 + exp\big(\sum_{i=1}^{D}(ln\frac{\frac{1}{\sqrt{2\pi}\sigma}exp(-\frac{(x_i-\mu_{i0})^2}{2\sigma^2})}{\frac{1}{\sqrt{2\pi}\sigma}exp(-\frac{(x_i-\mu_{i1})^2}{2\sigma^2})}) + ln\frac{1-\alpha}{\alpha}\big)}
\end{aligned}$$

(continued on the next page)

*Q*1.1 *continued* :

$$
\begin{aligned}
p(t=1|\mathbf{x}) &= \frac{1}{1+exp\Big(\sum_{i=1}^{D}\big(ln\big(exp(\frac{(x_i-\mu_{i1})^2}{2\sigma^2}-\frac{(x_i-\mu_{i0})^2}{2\sigma^2}\big)\big)+ln\frac{1-\alpha}{\alpha}\Big)} \\
&= \frac{1}{1+exp\Big(\sum_{i=1}^{D}\big(\frac{(x_i-\mu_{i1})^2}{2\sigma^2}-\frac{(x_i-\mu_{i0})^2}{2\sigma^2}\big)+ln\frac{1-\alpha}{\alpha}\Big)} \\
&= \frac{1}{1+exp\Big(\sum_{i=1}^{D}\frac{\mu_{i0}-\mu_{i1}}{\sigma^2}x_i+\sum_{i=1}^{D}\frac{\mu_{i1}^2-\mu_{i0}^2}{2\sigma^2}+ln\frac{1-\alpha}{\alpha}\Big)} \\
&= \frac{1}{1+exp(-\sum_{i=1}^{D}w_ix_i-b)}
\end{aligned}
$$

which takes the form of logistic function with $w_i=\frac{\mu_{i1}-\mu_{i0}}{\sigma^2}$, $b=-\sum_{i=1}^{D}\frac{\mu_{i1}^2-\mu_{i0}^2}{2\sigma^2}-ln\frac{1-\alpha}{\alpha}$.■

1.2. Maximum Likelihood Estimation

Derive an expression for $L(\mathbf{w},b)$, the negative log-likelihood of $t^{(1)},\ldots,t^{(n)}$ given $\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(n)}$ and the model parameters, then derive expressions for the derivatives of L with respect to each of the model parameters.

Solution :

$$
\begin{aligned}
L(\mathbf{w},b) &= -ln\Big(\prod_{i=1}^{N}p(t^i=0|x^i)^{1-t^i}p(t^i=1|x^i)^{t^i}\Big) && \text{(Since } t^i \text{ is either 1 or 0)} \\
&= -\sum_{i=1}^{N}ln\Big(p(t^i=0|x^i)^{1-t^i}p(t^i=1|x^i)^{t^i}\Big) \\
&= -\sum_{i=1}^{N}ln\Big(p(t^i=0|x^i)^{1-t^i}\Big)-\sum_{i=1}^{N}ln\Big(p(t^i=1|x^i)^{t^i}\Big) \\
&= -\sum_{i=1}^{N}(1-t^i)ln\Big(p(t^i=0|x^i)\Big)-\sum_{i=1}^{N}(t^i)ln\Big(p(t^i=1|x^i)\Big) \\
&= -\sum_{i=1}^{N}(1-t^i)ln\Big(1-\frac{1}{1+exp(-z^i)}\big)\Big)-\sum_{i=1}^{N}(t^i)ln\Big(\frac{1}{1+exp(-z^i)}\big)\Big) \\
&\quad \text{(Since } p(t=1|\mathbf{x}) \text{takes the form of logistic function with } z^i=\textstyle\sum_{j=1}^{D}w_jx_j^{(i)}+b) \\
&= \sum_{i=1}^{N}(1-t^i)\Big(ln(1+exp(-z^i))-ln(exp(-z^i))\Big)+\sum_{i=1}^{N}(t^i)ln\Big(1+exp(-z)\big)\Big) \\
&= \sum_{i=1}^{N}\Big(ln(1+exp(-z^i))+(t^i-1)(-z^i)\Big)
\end{aligned}
$$

As for the derivatives

*Q*1.2 *continued* :

$$\begin{aligned}
\frac{\partial L(\mathbf{w},b)}{\partial w_j} &= \frac{\partial L(\mathbf{w},b)}{\partial z^i}\cdot\frac{\partial z^i}{\partial w_j} && \text{(By the Chain Rule)}\\
&= \sum_{i=1}^{N}\frac{\partial\big(ln(1+exp(-z^i))+(t^i-1)(-z^i)\big)}{\partial z_i}\cdot x_j^i\\
&= \sum_{i=1}^{N}\big(-\frac{exp(-z^i)}{1+exp(-z^i)}+(1-t^i)\big)\cdot x_j^i\\
&= \sum_{i=1}^{N}\big(\frac{1}{1+exp(-z^i)}-t^i\big)\cdot x_j^i
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L(\mathbf{w},b)}{\partial b} &= \frac{\partial L(\mathbf{w},b)}{\partial z^i}\cdot\frac{\partial z^i}{\partial b} && \text{(By the Chain Rule)}\\
&= \sum_{i=1}^{N}\frac{\partial\big(ln(1+exp(-z^i))+(t^i-1)(-z^i)\big)}{\partial z_i}\cdot 1\\
&= \sum_{i=1}^{N}\big(-\frac{exp(-z^i)}{1+exp(-z^i)}+(1-t^i)\big)\\
&= \sum_{i=1}^{N}\big(\frac{1}{1+exp(-z^i)}-t^i\big)
\end{aligned}$$

1.3. L2 Regularization

Derive an expression that is proportional to $p(\mathbf{w},b|D)$. Show that $L_{post}(\mathbf{w},b)$ takes the form:

$$L_{post}(\mathbf{w},b) = L(\mathbf{w},b)+\frac{\lambda}{2}\sum_{i=1}^{D}w_i^2+C$$

What are the derivatives of L_{post} with respect to each of the model parameters?

Solution :

In Bayesian statistics

$$\text{Posterior} = \frac{\text{Likelihood}\times\text{Prior}}{\text{Evidence}}$$

So

$$p(\mathbf{w},b|D) = \frac{p(D|\mathbf{w},b)\times p(\mathbf{w},b)}{p(D)}$$

Since $p(D)$ is a constant, we have

$$\begin{aligned}
p(\mathbf{w},b|D) &\propto p(D|\mathbf{w},b)\times p(\mathbf{w},b)\\
&= p(\{(\mathbf{x}^{(1)},t^{(1)}),\ldots,(\mathbf{x}^{(N)},t^{(N)})\}|\mathbf{w},b)\times p(\mathbf{w})\times p(b)\\
&= p(\mathbf{t}|\mathbf{w},b)\times p(\mathbf{w})\times p(b)
\end{aligned}$$

Then we can derive the negative logarithm of this posterior

$$
\begin{aligned}
L_{post}(\mathbf{w}, b) &= -ln\big(p(D|\mathbf{w}, b) \times p(\mathbf{w}, b)\big) \\
&= -ln\big(p(\mathbf{t}|\mathbf{w}, b) \times p(\mathbf{w}) \times p(b)\big) \\
&= -ln\big(p(\mathbf{t}|\mathbf{w}, b) \times \prod_{i=1}^{D} p(w_i) \times 1\big) && \text{(Since p(b) = 1)} \\
&= -ln\big(p(\mathbf{t}|\mathbf{w}, b)\big) - \sum_{i=1}^{D} ln\big(p(w_i)\big) \\
&= L(\mathbf{w}, b) - \sum_{i=1}^{D} ln\left(\frac{1}{\sqrt{2\pi\frac{1}{\lambda}}} exp\Big(\frac{-(w_i - 0)^2}{2(\frac{1}{\lambda})}\Big)\right) && \text{(Since } p(w_i) = \mathcal{N}(w_i|0, 1/\lambda)\text{)} \\
&= L(\mathbf{w}, b) - \sum_{i=1}^{D} \left(\left(ln(1) - ln\sqrt{2\pi\frac{1}{\lambda}}\right) + \frac{-\lambda w_i^2}{2}\right) \\
&= L(\mathbf{w}, b) + \frac{\lambda}{2}\sum_{i=1}^{D} w_i^2 + \sum_{i=1}^{D} ln\sqrt{2\pi\frac{1}{\lambda}} \\
&= L(\mathbf{w}, b) + \frac{\lambda}{2}\sum_{i=1}^{D} w_i^2 + C
\end{aligned}
$$

The derivatives with respect to each parameters

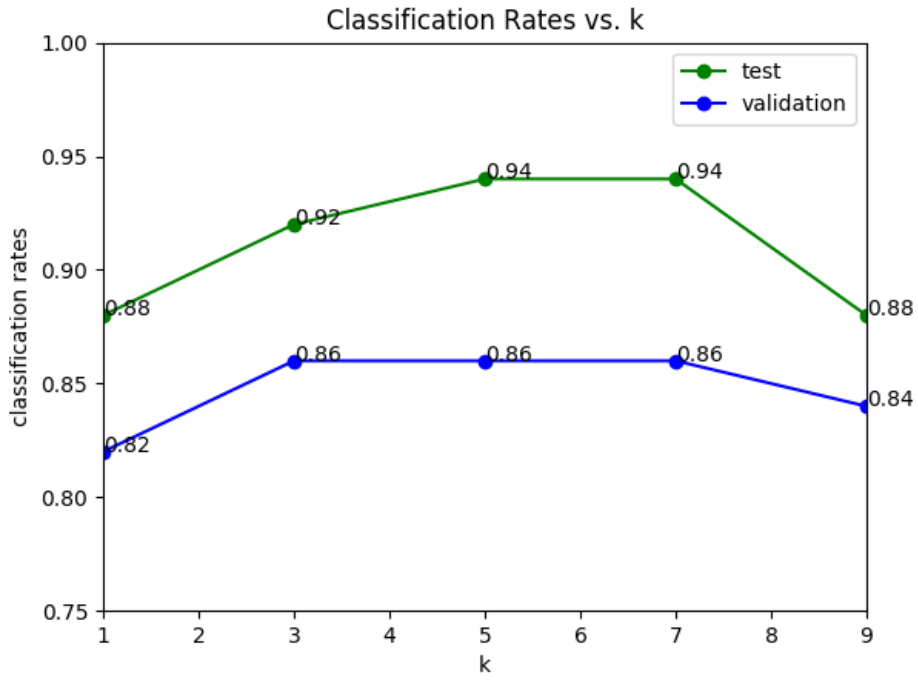
$$
\begin{aligned}
\frac{\partial L_{post}(\mathbf{w}, b)}{\partial b} &= \frac{\partial L(\mathbf{w}, b)}{\partial b} + 0 \\
&= \sum_{i=1}^{N} \Big(\frac{1}{1 + exp(-z^i)} - t^i\Big) && \text{(We've derived } \tfrac{\partial L(\mathbf{w},b)}{\partial b} \text{ in Q1.2)}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L_{post}(\mathbf{w}, b)}{\partial w_i} &= \frac{\partial L(\mathbf{w}, b)}{\partial w_i} + \lambda \sum_{i=0}^{i=D} w_i \\
&= \sum_{i=1}^{N} \Big(\frac{1}{1 + exp(-z^i)} - t^i\Big) \cdot x_j^i + \lambda \sum_{i=0}^{i=D} w_i && \text{(We've derived } \tfrac{\partial L(\mathbf{w},b)}{\partial w_i} \text{ in Q1.2)}
\end{aligned}
$$

2. Logistic Regression vs. KNN

2.1. k-Nearest Neighbors

Write a script that runs KNN for different values of $k \in \{1, 3, 5, 7, 9\}$and plots the classification rate on the validation set as a function of k. Comment on the performance of the classifier and argue which value of k you would choose. What is the classification rate for k^*, your chosen value of k? Also compute the rate for $k^* + 2$ and $k^* - 2$. Does the test performance for these values of k correspond to the validation performance? Why or why not?



Comments :
The classification rates on the validation set is shown as the blue line. When k is small (k = 1), the model is quite sensitive to random idiosyncrasies in the training data thus performs badly on the test/validation set (overfit). While when k is large (k = 9) the model fail to capture important regularities thus performs badly as well (underfit).

Based on this graph, the best k is probably $k^* = 5$(or 7), which achieves the highest accuracy on both test and validation set (94% and 86% respectively).

The accuracy for $k^* - 2$ is 92% on test set, 86% on validation set; And for $k^* + 2$, is 94% on test set, 86% on validation set. The test performance correspond to the validation performance: low accuracy for small k or large k, while achieving highest accuracy at around $k = 5$, because KNN is a non-parametric method, the value of k is the only factor that influence the performance.

```
from l2_distance import l2_distance
from plot_digits import *
from utils import *
import numpy as np
import matplotlib.pyplot as plt


def run_knn(k, train_data, train_labels, valid_data):
    """..."""

    dist = l2_distance(valid_data.T, train_data.T)
    nearest = np.argsort(dist, axis=1)[:, :k]

    train_labels = train_labels.reshape(-1)
    valid_labels = train_labels[nearest]

    # note this only works for binary labels
    valid_labels = (np.mean(valid_labels, axis=1) >= 0.5).astype(np.int)
    valid_labels = valid_labels.reshape(-1, 1)

    return valid_labels


if __name__ == '__main__':

    # X is the input, t is target
    train_X, train_t = load_train()
    valid_X, valid_t = load_valid()
    test_X, test_t = load_test()

    k_list = [1, 3, 5, 7, 9]
    valid_err = []
    test_err = []

    # Classification rate on validation set and test set for each k
    for k in k_list:
        pred_valid = run_knn(k, train_X, train_t, valid_X)
        pred_test = run_knn(k, train_X, train_t, test_X)
        # np.mean will compare all the entries in the matrix and
        # return the rates of matches (correct prediction)
        valid_err.append(np.mean((valid_t == pred_valid)))
        test_err.append(np.mean((test_t == pred_test)))

    # Print and plot the classification rate
    print("Validation accuracy: \n", valid_err)
    print("Test accuracy: \n", test_err)
```

```
    plt.axis([1, max(k_list), 0.75, 1])
    plt.plot(k_list, test_err, 'go-')
    plt.plot(k_list, valid_err, 'bo-')

    # Show the values of each point
    for a, b in zip(k_list, test_err):
        plt.text(a, b, str(b))
    for a, b in zip(k_list, valid_err):
        plt.text(a, b, str(b))

    # Modify the style of the plot
    plt.title("Classification Rates vs. k")
    plt.xlabel("k")
    plt.ylabel("classification rates")
    plt.legend(['test', 'validation'], loc='upper right')
    plt.savefig("./hw2_2_1.png")
    plt.clf()
```

2.2. Logistic regression

a. Complete the implementation of logistic regression by providing the missing part of `logistic`.

```
""" Methods for doing logistic regression."""

import numpy as np
from utils import sigmoid


def logistic_predict(weights, data):
    """
    Compute the probabilities predicted by the logistic classifier.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:   (M+1) x 1 vector of weights, where the last element
                   corresponds to the bias (intercepts).
        data:      N x M data matrix where each row corresponds
                   to one data point.
    Outputs:
        y:         :N x 1 vector of probabilities. This is the output of the
            classifier.
    """
    # data^T is M*N, np.ones(N) is 1*N, so x will become (M+1)*N
```

```
    # with a new row of "1" at the bottom
    N, M = data.shape
    x = np.append(np.transpose(data), [np.ones(N)], axis=0)
    # z = wx + b (bias is the last element of w)x:N*(M+1), w:(M+1)*1 so z is N*1
    z = np.dot(np.transpose(x), weights)
    y = sigmoid(z)
    return y


def evaluate(targets, y):
    """
    Compute evaluation metrics.
    Inputs:
        targets : N x 1 vector of targets.
        y       : N x 1 vector of probabilities.
    Outputs:
        ce          : (scalar) Cross entropy. CE(p, q) = E_p[-log q].
                      Here we want to compute CE(targets, y)
        frac_correct : (scalar) Fraction of inputs classified correctly.
    """
    # This formula for Lce is given on Page 23 of the lecture slides 4
    # And len(y) is the number of targets
    ce = np.sum(np.squeeze(- targets * np.log(y)
                           - (1 - targets) * np.log(1 - y))) / len(y)
    frac_correct = (targets == (y > 0.5)).mean()
    return ce, frac_correct


def logistic(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to weights.
    Also return the predictions.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:   (M+1) x 1 vector of weights, where the last element
                   corresponds to bias (intercepts).
        data:      N x M data matrix where each row corresponds
                   to one data point.
        targets:   N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.

    Outputs:
        f:      The sum of the loss over all data points. This is the objective
            that we want to minimize.
        df:     (M+1) x 1 vector of derivative of f w.r.t. weights.
```

```
        y:      N x 1 vector of probabilities.
    """
    # f = - sum(np.log(y) + (targets - 1) * z)
    # df = np.dot(x, y - targets)
    # weight_regularization is a bool

    N, M = data.shape
    x = np.append(np.transpose(data), [np.ones(N)], axis=0)
    z = np.dot(np.transpose(x), weights)
    y = sigmoid(z)
    f = evaluate(targets, y)[0] * len(y)
    df = np.dot(x, y - targets) # This formula is given on Page 33 of lecture 4
    return f, df, y


def logistic_pen(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to weights.
    Also return the predictions.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:   (M+1) x 1 vector of weights, where the last element
                   corresponds to bias (intercepts).
        data:      N x M data matrix where each row corresponds
                   to one data point.
        targets:   N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.

    Outputs:
        f:             The sum of the loss over all data points. This is the
            objective that we want to minimize.
        df:            (M+1) x 1 vector of derivative of f w.r.t. weights.
    """
    N, M = data.shape
    c = np.append(np.ones((M, 1)), [[0]], axis=0)
    f, df, y = logistic(weights, data, targets, hyperparameters)
    f = f + sum(
        ((weights * c) ** 2) * hyperparameters['weight_regularization'] / 2)
    df = df + weights * c * hyperparameters['weight_regularization']

    return f, df, y
```

b. Report which hyperparameter settings you found worked the best and the final cross entropy and classification error on the training, validation and test sets.

The process of selecting the best hyperparameters are shown below (based on *mnist_train*, the cross entropy and fraction correctness data can differ depends weights)

Learning_rate	num_iter	Weights (w_i)	cross_entr	frac_corr
0.001	100	Gaussian(0, 1)	400.807391	50.00
0.001	1000	Gaussian(0, 1)	54.946939	42.00
0.01	100	Gaussian(0, 1)	57.025413	64.00
0.01	70	Gaussian(0, 0.1)	11.567	76.00
0.01	200	Gaussian(0, 0.1)	10.0654	74
0.01	35	Gaussian(0,1)	12.998	62.00
0.1	100	Gaussian(0, 1)	12.858099	82.00
0.5	100	Gaussian(0, 1)	4.138122	88.00
1	100	Gaussian(0, 1)	15.162320	90.00
1	100	0.001	6.065	88.00
1	100	0.001	6.065	88.00
10	100	Gaussian(0, 1)	65.475161	88.00

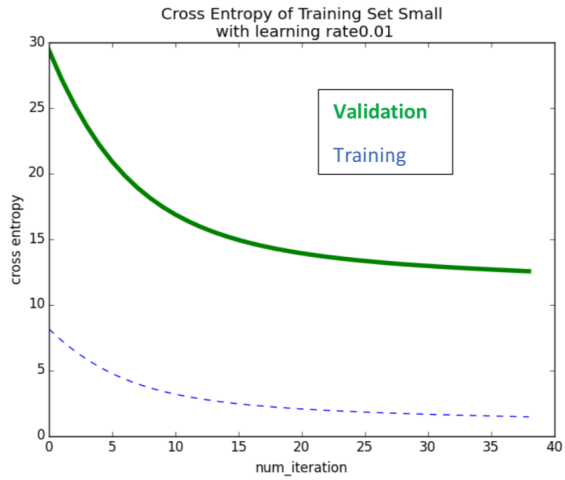
Answer : With learning rate is too small, the model with weights initialized as Gaussian(0, 1) will take longer to converge. Therefore, the cross entropy at the 100th run is still large. For learning rate equals to 0.001, number of iterations for the model to converge will be very large. At the 100th iteration, the cross entropy is large, which indicates that the classification has a very high uncertainty. Thus, we need to higher the number of iterations. However, if we set num_iter to 1000, the maximum frac_corr occurs at around the 700th iteration, with a relatively large cross entropy. Cross entropy keeps decreasing before the 1000th iteration, with fraction correctness bouncing below. It will be noticed that the curves are jumping by observing the curves of cross entropy and fraction correctness against the number of iterations.

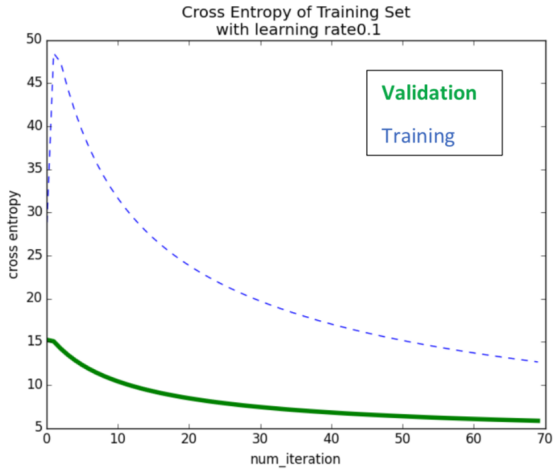
Reset the variables and repeat the steps to find a model with low cross entropy, high fraction correctness and smooth curves. After many attemps, a reasonable learning rate is found to be around 0.1 on mnist_train, and 0.01 on mnist_train_small.

c. Next look at how the cross entropy changes as training progresses. Submit 2 plots, one for each of `mnist_train` and `mnist_train_small` .

```
For minist_train_small:                  For mnist_train:

Learning rate: 0.1                       Learning rate: 0.1
Weight regularization: 1                 Weight regularization: 1
Number of iterations: 39                 Number of iterations: 40
Initial weights: Gaussian(0, 0.01)       Initial weights: Gaussian(0, 0.01)

Cross Entropy and Accuracy:              Cross Entropy and Accuracy:
Mnist_train_small Set: [1.2, 90]         Mnist_train_small Set: [13.04, 97.5]
Validation Set: [11.3, 54]               Validation Set: [8.06, 84]
Test Set: [9.6, 68]                      Test Set: [5.86, 90]
```

For minist_train_small:
Notice that both curves are converging, which indicates that the model is converging





at the minimum of loss function. This is a good model because the cross entropy values for both validation and training sets are low, which means the uncertainty of the classification is low. Also, the fraction correctness is pretty high for all data sets.

For minist_train :
Notice that both curves are converging, which indicates that the model is converging at the minimum of loss function. This is a good model because the cross entropy values for both validation and training sets are low, which means the uncertainty of the classification is low. Also, the fraction correctness is pretty high for all data sets.

In summary :
Comparing results from both training sets, the larger training set trains the model much better than the smaller one because the cross entropy for both sets are around the same level, but fraction correctness for large training set is much higher than the small.

The reason is that with more historical data, the model will get a higher accuracy on its classification. The results will change slightly if we run the program several times. This is because we are assigning the initial weights randomly at each call. If the variance of the Gaussian model is large, the results will differ more. Therefore, we can picked a small variance in order to restrict on the weights, thus changes in results can be reduced.

Alternatively, we can run the program several times and take the mean of the data in each run, to get an average on model performance.

2.3. Penalized logistic regression

a. Choose a hyperparameter setting which seems to work well (for learning rate, number of iterations, and weight initialization). With these hyperparameters, do the following for each value of the penalty parameter $\lambda \in \{0, 0.001, 0.01, 0.1, 1.0\}$:

```
import numpy as np
from check_grad import check_grad
from plot_digits import *
from utils import *
from logistic import *
import matplotlib.pyplot as mp

def run_logistic_pen_regression():
    train_inputs, train_targets = load_train()
    #train_inputs, train_targets = load_train_small()

    valid_inputs, valid_targets = load_valid()
    test_inputs, test_targets = load_test()

    N, M = train_inputs.shape

    # TODO: Set hyperparameters
    hyperparameters = {
                   'learning_rate': 0.01,
                   'weight_regularization': [0, 0.001,0.01,0.1,1.0],
                   'num_iterations': 80
                }

    ce_train = []
    ce_valid = []
    ce_test = []
    fe_train = []
    fe_valid = []
    fe_test = []

    for item in hyperparameters['weight_regularization']:

        hyperparameter = {
                         'learning_rate': hyperparameters['learning_rate'],
                         'weight_regularization': item,
                         'num_iterations': hyperparameters['num_iterations']
                       }

        ce_train_sub = []
        ce_valid_sub = []
        ce_test_sub = []
        fe_train_sub = []
```

```
fe_valid_sub = []
fe_test_sub = []


for i in range(10):

    # Logistic regression weights
    # TODO:Initialize to random weights here.
    #weights = np.transpose([np.random.normal(0,
        hyperparameter['weight_regularization'], M+1)])
    weights = np.transpose([np.random.normal(0, 0.1, M+1)])


    # Verify that your logistic function produces the right gradient.
    # diff should be very close to 0.
    run_check_grad(hyperparameter)


    # Begin learning with gradient descent
    for t in xrange(hyperparameter['num_iterations']):


        # TODO: you may need to modify this loop to create plots, etc.

        # Find the negative log likelihood and its derivatives w.r.t. the
            weights.
        f, df, predictions = logistic_pen(weights, train_inputs,
            train_targets, hyperparameter)
        ft, dft, predictionst = logistic_pen(weights, test_inputs,
            test_targets, hyperparameter)

        # Evaluate the prediction.
        cross_entropy_train, frac_correct_train = evaluate(train_targets,
            predictions)
        cross_entropy_test, frac_correct_test = evaluate(test_targets,
            predictionst)


        if np.isnan(f) or np.isinf(f):
            raise ValueError("nan/inf error")

        # update parameters
        weights = weights - hyperparameter['learning_rate'] * df / N

        # Make a prediction on the valid_inputs.
        predictions_valid = logistic_predict(weights, valid_inputs)

        # Evaluate the prediction.
```

```
            cross_entropy_valid, frac_correct_valid = evaluate(valid_targets,
                predictions_valid)

            # print some stats
            stat_msg = "ITERATION:{:4d} TRAIN NLOGL:{:4.2f} TRAIN CE:{:.6f} "
            stat_msg += "TRAIN FRAC:{:2.2f} VALID CE:{:.6f} VALID FRAC:{:2.2f}"
            print stat_msg.format(t+1,
                                  float(f / N),
                                  float(cross_entropy_train),
                                  float(frac_correct_train*100),
                                  float(cross_entropy_valid),
                                  float(frac_correct_valid*100))


            # Append the values for the last iteration:
            if t == (hyperparameter['num_iterations'] - 1):

                # Calculate the classification error:
                frac_error_train = 1 - frac_correct_train
                frac_error_valid = 1 - frac_correct_valid
                frac_error_test = 1 - frac_correct_test

                ce_train_sub = np.append(ce_train_sub, cross_entropy_train)
                ce_valid_sub = np.append(ce_valid_sub, cross_entropy_valid)
                ce_test_sub = np.append(ce_test_sub, cross_entropy_test)
                fe_train_sub = np.append(fe_train_sub, frac_error_train)
                fe_valid_sub = np.append(fe_valid_sub, frac_error_valid)
                fe_test_sub = np.append(fe_test_sub, frac_error_test)

    ce_train = np.append(ce_train, ce_train_sub.mean())
    ce_valid = np.append(ce_valid, ce_valid_sub.mean())
    ce_test = np.append(ce_test, ce_test_sub.mean())
    fe_train = np.append(fe_train, fe_train_sub.mean())
    fe_valid = np.append(fe_valid, fe_valid_sub.mean())
    fe_test = np.append(fe_test, fe_test_sub.mean())

    # Report test errrors:
    print "\n [0,0.001,0.01,0.1,1]"
    print "Training set:\n frac_error: ", fe_train, "\n ce: ", ce_train
    print "Validation set: \n frac_error: ", fe_valid, "\n ce: ", ce_valid
    print "Testing set: \n frac_error: ", fe_test, "\n ce: ", ce_test

# Plot cross entropy of training and testing sets
w = np.log(hyperparameters['weight_regularization'])
mp.plot(w, ce_train, label="training", linewidth=4)
mp.plot(w, ce_valid,label="validation", linestyle="--")
mp.xticks(w, hyperparameters['weight_regularization'])
mp.title("Plot of Average Cross Entropy")
```

```
    mp.xlabel("penalty parameters")
    mp.ylabel("avg. cross entropy")
    mp.savefig("2.3_ce_pen.png")
    mp.clf()


    # Plot fraction error of training and testing sets
    mp.plot(w, fe_train, label="training", linewidth=4)
    mp.plot(w, fe_valid,label="validation", linestyle="--")
    mp.xticks(w, hyperparameters['weight_regularization'])
    mp.title("Plot of Classification Error")
    mp.xlabel("penalty parameters")
    mp.ylabel("classification error")
    mp.savefig("2.3_fe_pen.png")
    mp.clf()

def run_check_grad(hyperparameters):
    """Performs gradient check on logistic function.
    """

    # This creates small random data with 20 examples and
    # 10 dimensions and checks the gradient on that data.
    num_examples = 20
    num_dimensions = 10

    weights = np.random.randn(num_dimensions+1, 1)
    data = np.random.randn(num_examples, num_dimensions)
    targets = np.round(np.random.rand(num_examples, 1), 0)

    diff = check_grad(logistic,   # function to check
                      weights,
                      0.001,      # perturbation
                      data,
                      targets,
                      hyperparameters)

    print "diff =", diff

if __name__ == '__main__':
    run_logistic_pen_regression()
```

b. For each of mnist_train and mnist_train_small you will have 2 plots, one plot for cross entropy and another plot for classification error. Each plot will have two curves one for training and one for validation. How do the cross entropy and classification error change when you increase λ? Do they go up, down, first up and then down, or down and then up? Explain why you think they behave this way. Which is the best value of λ, based on your experiments? Report the test error for the best value of λ.

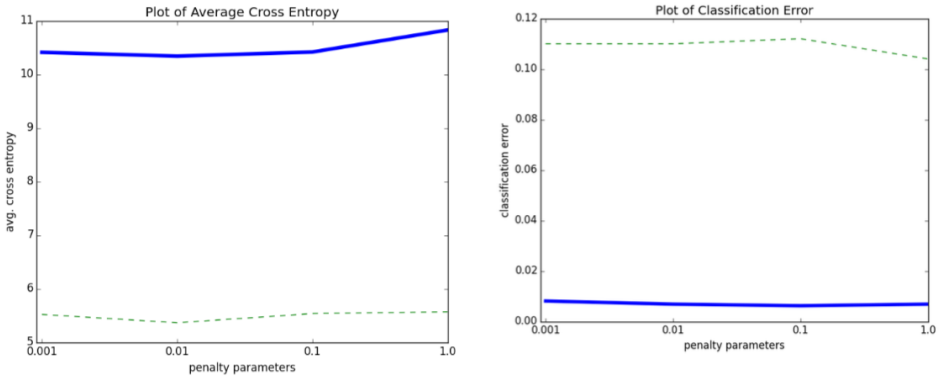
For minist_train:
Learning rate: 0.1
Weight regularizations: {0, 0.001, 0.01, 0.1, 1.0}
Number of iterations: 70
Initial weights: follows Gaussian(0, 0.1)

Training set:
frac_error: [0.02 0.02125 0.019375 0.015625]
ce: [12.91345334 12.94917253 12.95216837 12.73485931 13.13836072]

Validation set:
frac_error: [0.116 0.116 0.106 0.104 0.104]
ce: [6.01937512, 6.08488291 6.05699394 6.26602178 6.04909971]

Testing set:
frac_error: [0.087 0.076 0.096 0.082 0.078]
ce: [3.90846823 3.80139734 4.15496006 4.02944961 3.97020806]

(The green line is the Validation set, the blue line is the Training set)



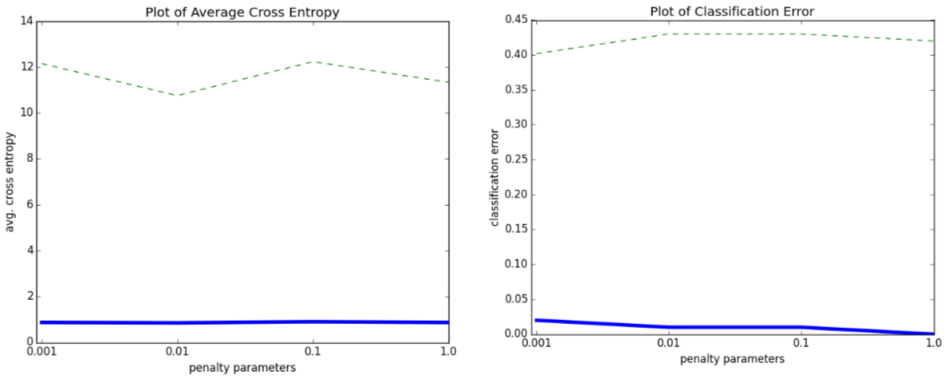
For minist_train_small:
Learning rate: 0.1
Weight regularizations: {0, 0.001, 0.01, 0.1, 1.0}
Number of iterations: 50
Initial weights: follows Gaussian(0, 0.1)

Training set:
frac_error: [0.02 0.02 0.01 0.01 0.]
ce: [0.86274859 0.87178067 0.8527974 0.90033255 0.86834025]

Validation set:

```
frac_error: [0.43 0.402 0.43 0.43 0.42 ]
ce: [12.08738561 12.14076983 10.74983207 12.22768712 11.33352384]

Testing set:
frac_error: [0.274 0.278 0.294 0.316 0.304]
ce: [8.68729451 8.91892895 8.329423 9.64514705 8.75605653]
```

(The green line is the Validation set, the blue line is the Training set)



Comments :
1. Models trained by using mnist_train data set perform better than the ones trained using mnist_train_small. This is shown by a lower entropy and fraction error. Classification error curves for validation sets is higher than the training sets in general.

2. In mnist_train plots, both average cross entropy and classification error curves are quiet stable. The best penalty parameter vary if we call the function multiple times. In minist_train_small plots, the curves for validation set fluctuates, with a minimum average cross entropy at 0.01.

3. The data on penalty parameters 0, 0.001, 0.01, 0.1, 1.0 reflects that as the penalty parameter increases, the classification error and the average cross entropy on the validation set both slightly decreases before λ 0.01, and increase afterwards. Therefore, the best λ is 0.01 based on the results with test error = 0.096 on Mnist_train and 0.294 on Mnist_train_small.

c. Compare the results with and without penalty. Which one performed better for which data set? Why do you think this is the case?

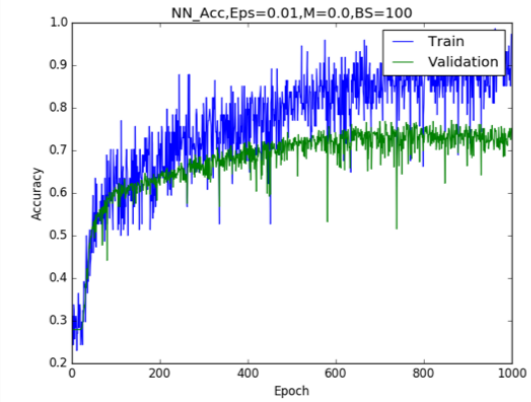
	Training set	Testing set	Cross entropy	Fraction error
Without regularization	Mnist_train	Validation	8.0579	0.16
		test	5.861	0.10
	Mnist_train_small	Validation	11.255	0.46
		test	9.599	0.32
With regularization $\lambda = 0.01$	Mnist_train	Validation	6.0570	0.106
		test	4.1549	0.096
	Mnist_train_small	Validation	10.7498	0.43
		Test	8.3294	0.294

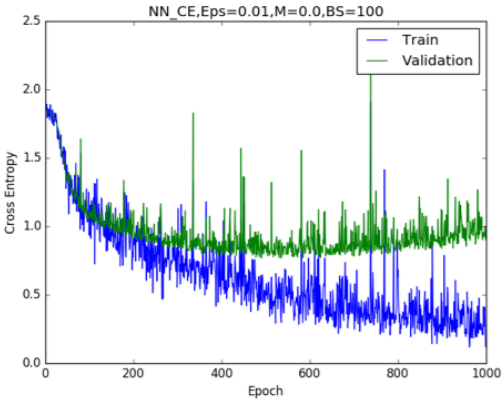
Answer :
The data suggests that the model with penalty performs better than without regularization. Penalty parameters are used in logistic regression to prevent the models from overfitting due to the existence of some extremely weighted parameters.A penalty term will shrink every weight parameter during the updating process (i.e. gradient descent).

3. Neural Networks

3.1. Basic generalization

Train a regular NN with the default set of hyperparameters. Examine the statistics and plots of training error and validation error (generalization). How does the networks performance differ on the training set versus the validation set during learning? Show a plot of error curves (training and validation).





Comment :
Trained with the default set of hyperparameters, the NN performs better on the training set than validation set. In the first 150ish epochs, the Cross Entropy drop very fast while the Accuracy increasing dramatically, indicating the model is updating fast and gaining performs better and better. Then the increase of Acc (decrease of CE) becomes much slower. The Cross Entropy for validation set appears to be increasing before 1000 epochs, which means that the model is likely overfitting the training data. So we'd probably stop before 1000 epochs, as the model seems to converge at around 500 epochs.

Code : (I only paste the methods that I modified, leave the provided methods as `"""Not change"""`)

```
from __future__ import division
from __future__ import print_function

from util import LoadData, Load, Save, DisplayPlot
import sys
import numpy as np


def InitNN(num_inputs, num_hiddens, num_outputs):
    """
    Initializes NN parameters.

    Args:
```

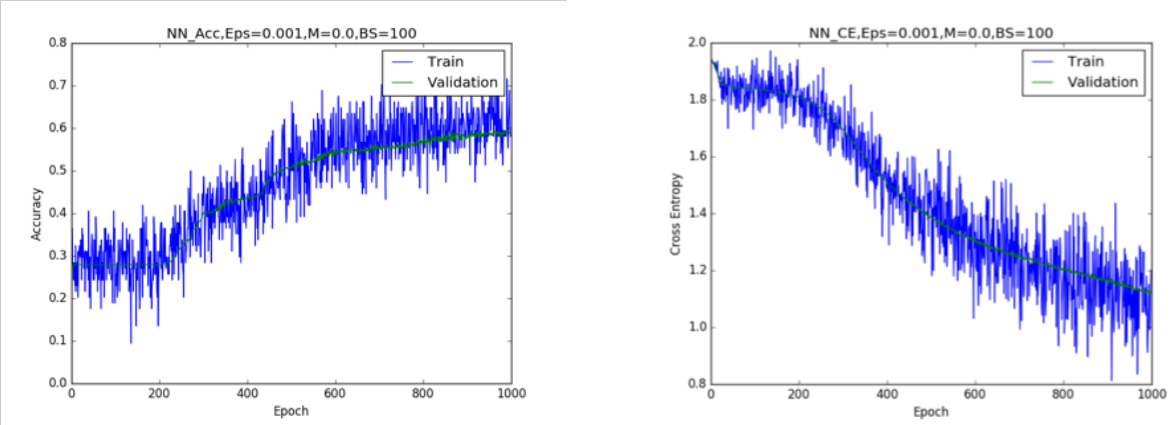
```
        num_inputs:   Number of input units.
        num_hiddens:  List of two elements, hidden size for each layer.
        num_outputs:  Number of output units.

        num_inputs = 2304
        num_hiddens = [16, 32] (since we have two layers)
        num_outputs = 7

    Returns:
        model:        Randomly initialized network weights.
    """
    # randn creates an array of specified shape (ex, the first one is 2304x16)
    # and fills it with random values as per standard
    # normal distribution (mean 0 and variance 1); dwi, dbi are differential

    # W1b1 is the model that connect input to the first layer
    W1 = 0.1 * np.random.randn(num_inputs, num_hiddens[0])
    b1 = np.zeros((num_hiddens[0]))
    dW1 = np.zeros(W1.shape)
    db1 = np.zeros(b1.shape)

    # W2b2 is the model that connect first layer to the second layer
    W2 = 0.1 * np.random.randn(num_hiddens[0], num_hiddens[1])
    b2 = np.zeros((num_hiddens[1]))
    dW2 = np.zeros(W2.shape)
    db2 = np.zeros(b2.shape)

    # W3b3 is the model that connect the second layer to the output
    W3 = 0.01 * np.random.randn(num_hiddens[1], num_outputs)
    b3 = np.zeros((num_outputs))
    dW3 = np.zeros(W3.shape)
    db3 = np.zeros(b3.shape)

    # Dictionary of all the weights, biases and differentials
    model = {
        'W1': W1, 'W2': W2, 'W3': W3,
        'b1': b1, 'b2': b2, 'b3': b3,
        'dW1': dW1, 'dW2': dW2, 'dW3': dW3,
        'db1': db1, 'db2': db2, 'db3': db3

    }
    return model


def AffineBackward(grad_y, h, w):
    """
    Computes gradients of affine transformation.
    Here we don't have activation function
```

```
    Args:
    y is nx1, h is nxm, w is mx1, b is nx1
        grad_y: gradient from last layer
        h: inputs from the hidden layer
        w: weights

    Returns:
        grad_h: Gradients wrt. the inputs/hidden layer.
        grad_w: Gradients wrt. the weights.
        grad_b: Gradients wrt. the biases.
    """
    ############################
    # so nx1 dot 1xm = nxm
    grad_h = grad_y.dot(w.T)
    grad_w = h.T.dot(grad_y)
    # axis = 0 along the column, 1 along the row.
    grad_b = np.sum(grad_y, axis=0)
    return grad_h, grad_w, grad_b
    ############################

def ReLUBackward(grad_y, z):
    """
    Computes gradients of the ReLU activation function
    wrt. the unactivated inputs.

    Returns:
        grad_z: Gradients wrt. the hidden state prior to activation.
    """
    ############################
    # This is the definition of the ReLU activation function
    # numpy.maximum: element-wise maximum of array elements.
    y = np.maximum(0, z)
    # Since z bar = y bar * y', and if z > 0, y = z and dy/dz = dz/dz = 1
    y[y > 0] = 1
    # "*" will return an nx1 array(apply each term individually),
    # while dot() will return a number
    grad_z = y * grad_y
    return grad_z
    ############################


def NNUpdate(model, eps, momentum):
    """
    Update each of the weights in the network so that they cause the actual
    output to be closer the target output, thereby minimizing the error for
    each output neuron and the network as a whole.
    Args:
```

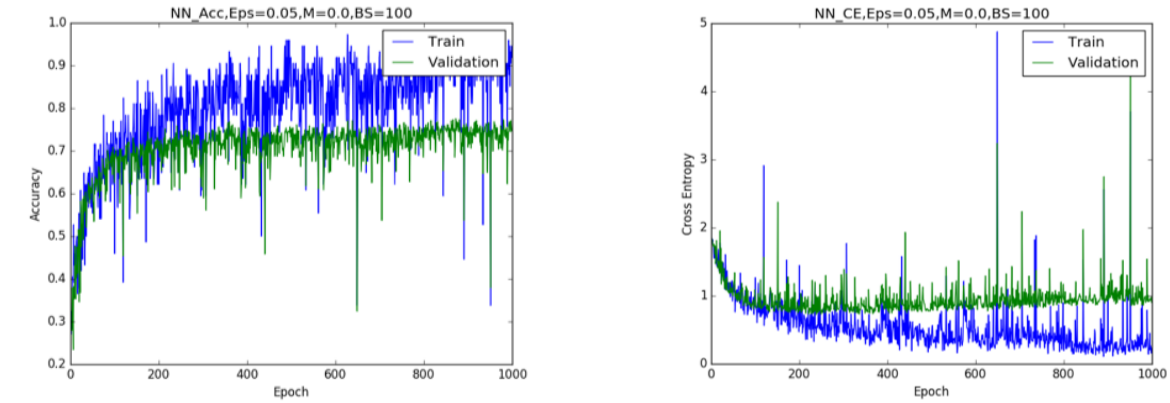
```
        model:   Dictionary of all the weights.
        eps:     Learning rate.
        momentum: Momentum.
    """
    ############################
    # Update all the W and b
    for i in ["1", "2", "3"]:
        for j in ["W", "b"]:
            model['d' + j + i] = momentum * model['d' + j + i] - eps * model[
                'dE_d' + j + i]
            model[j + i] += model['d' + j + i]
    ############################

def Affine(x, w, b):
"""Not change"""

def ReLU(z):
"""Not change"""

def Softmax(x):
"""Not change"""

def NNForward(model, x):
"""Not change"""

def NNBackward(model, err, var):
"""Not change"""

def Train(model, forward, backward, update, eps, momentum, num_epochs,
          batch_size):
"""Not change"""

def Evaluate(inputs, target, model, forward, batch_size=-1):
"""Not change"""

def CheckGrad(model, forward, backward, name, x):
"""Not change"""

def main():
"""Not change"""
```

3.2. Optimization

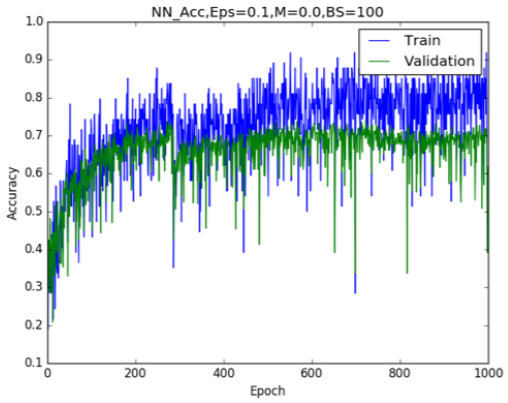
a. Try 5 different settings of ϵ from 0.001 to 1.0. What happens to the convergence properties of the algorithm (looking at both cross-entropy and percent-correct)?

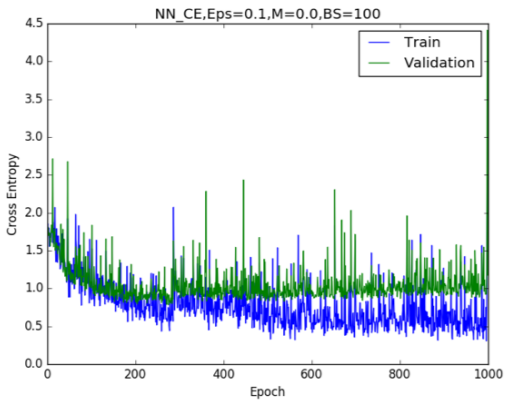


Epsilon = 0.001

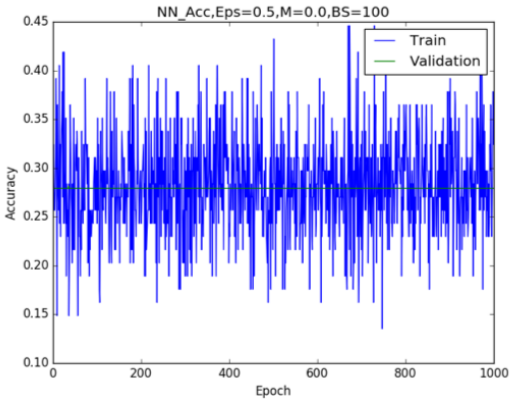


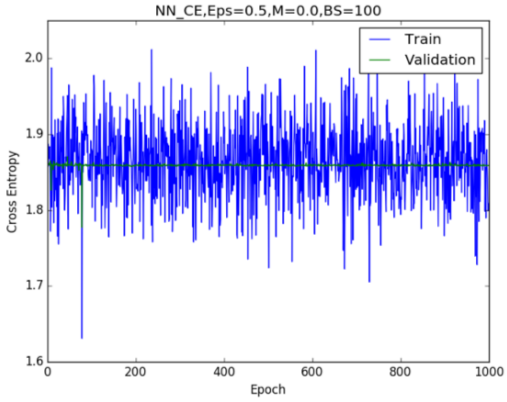
Epsilon = 0.05



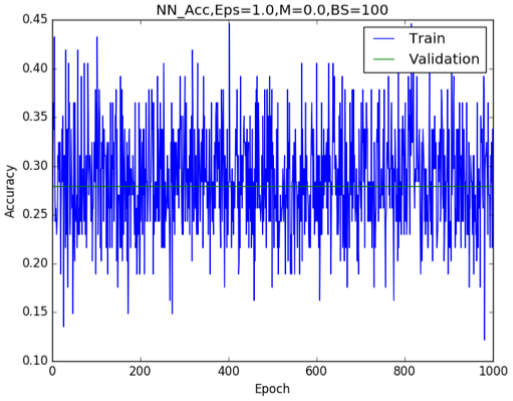


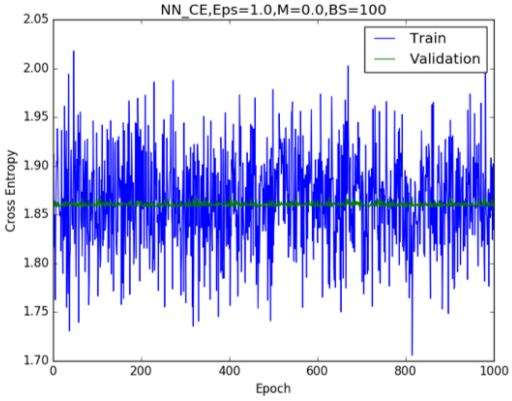
Epsilon = 0.1





Epsilon = 0.5



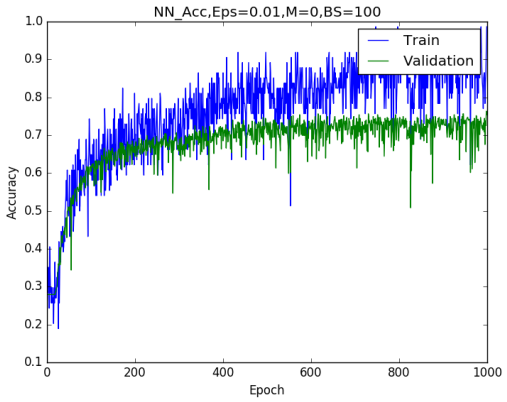


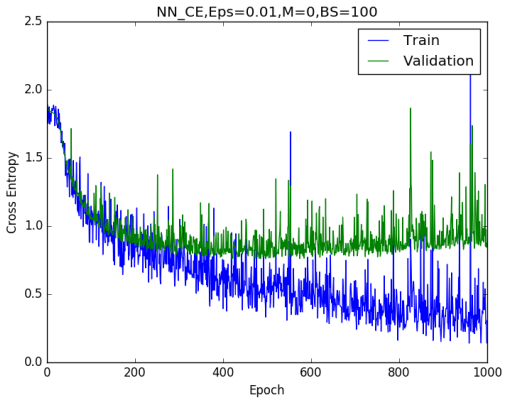
Epsilon = 1

*Answer to Q*3.2(*a*)
Compare the figures for epsilon = 0.001, 0.05 and 0.1, we can see that larger epsilon (within reasonable range) let the model converges faster, since for epsilon=0.001 it takes about 900 epochs, but for epsilon = 0.1, it only takes 150 epochs. But when the epsilon s too large (0.5 or 1), the performance do not improved at all. it's very noisy and does not seems to be converged on both cross entropy and accuracy.

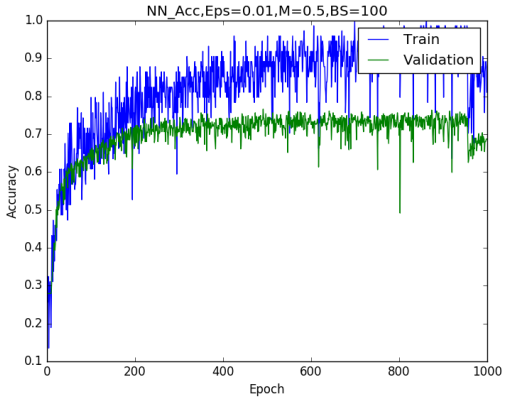
Compare to the default epsilon in Q3.1, we can see that although epsilon= 0.05 or 0.1 learns faster, they perform quite similar to the default epsilon. So maybe the main take-away here

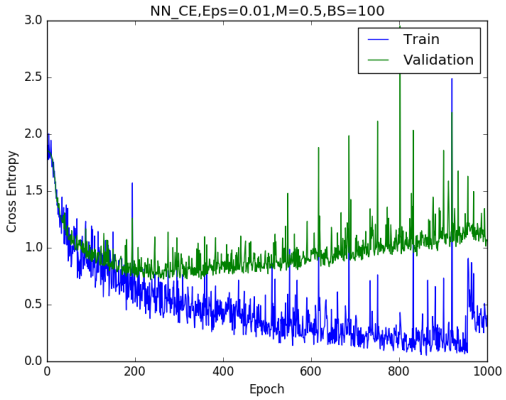
b. Try 3 values of momentum from 0.0 to 0.9. How does momentum affect convergence rate?



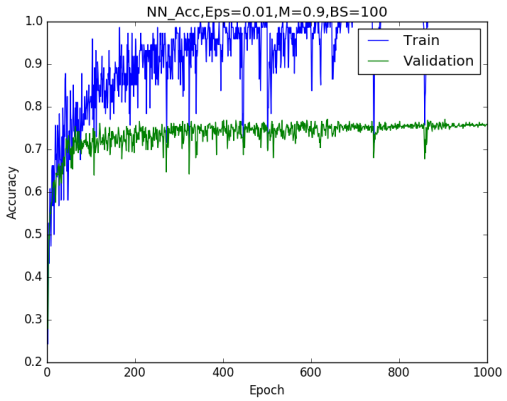


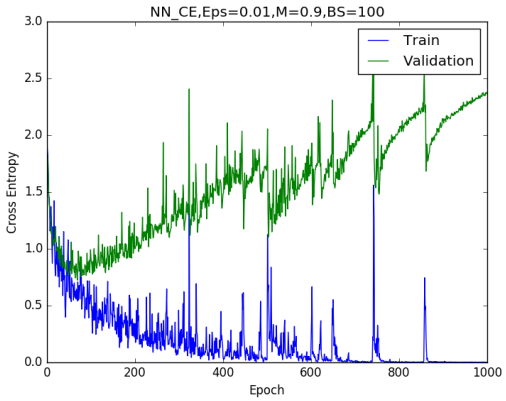
Momentum = 0





Momentum = 0.5



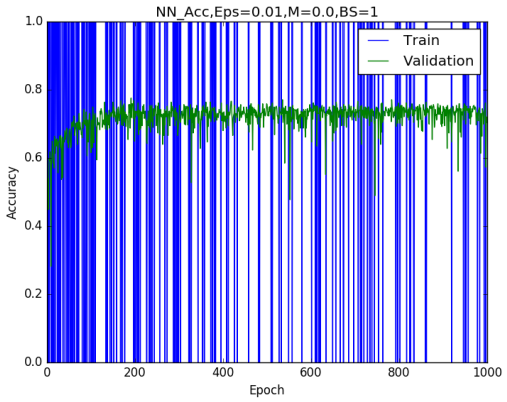


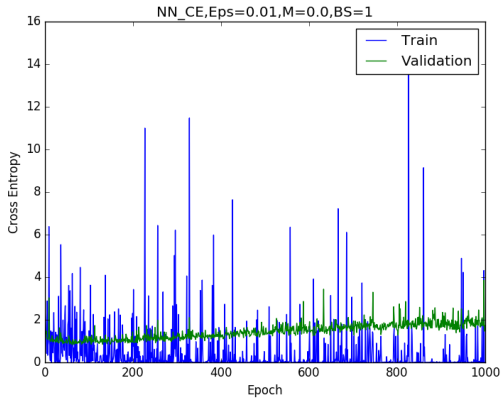
Momentum = 0.9

Answer to Q3.2(b)
As the momentum increases, the model reaches the peak of accuracy much earlier, i.e., the rate of convergence is higher. Cross entropy reaches the lowest point earlier as well, but becomes more diverge, and growing quickly (on the CE plot for Momentum = 0.5 and 0.9, we can see that the CE reaches the lowest point before 200 epoch but goes up again as epoch increases) which is a sign of overfitting (if we look closely to the plot, there is a little drop of accuracy in the accuracy plot for momentum = 0.5 at around 950 epoch).

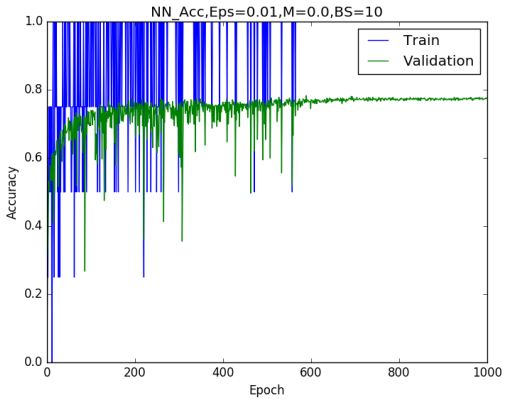
It suggest that higher momentum will let the model converge earlier but we should stop earlier as well, or we the model might be overfit to the training data (high convergence rate with the cost of cross entropy).

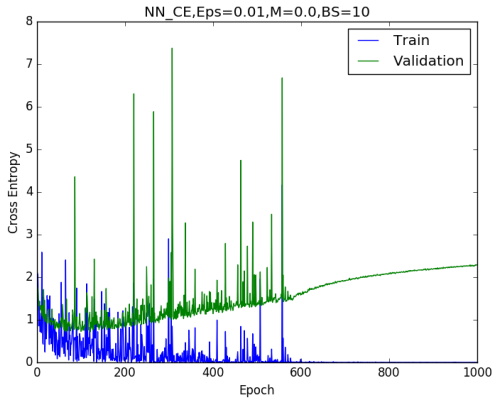
c. Try 5 different mini-batch sizes, from 1 to 1000. How does mini-batch size affect convergence?



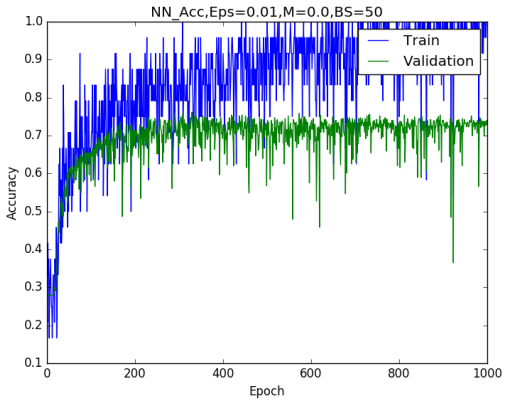


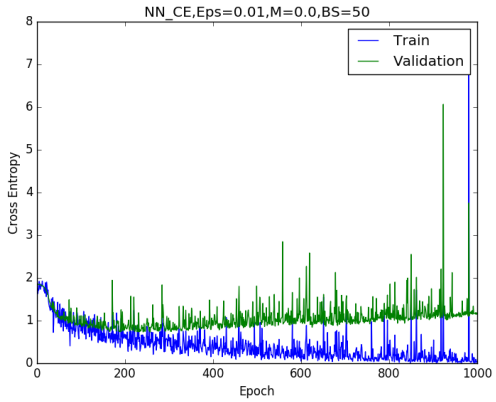
Batch Size = 1



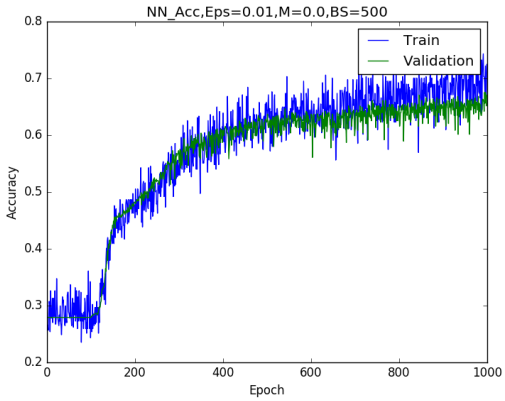


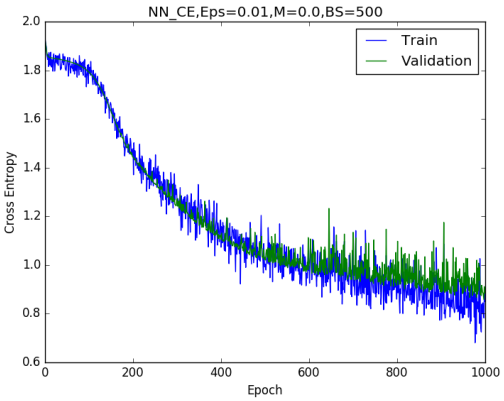
Batch Size = 10





Batch Size = 50

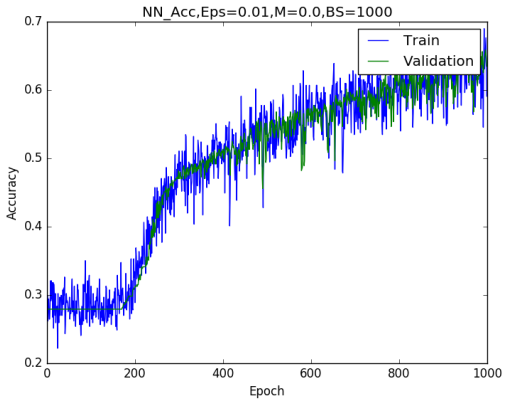


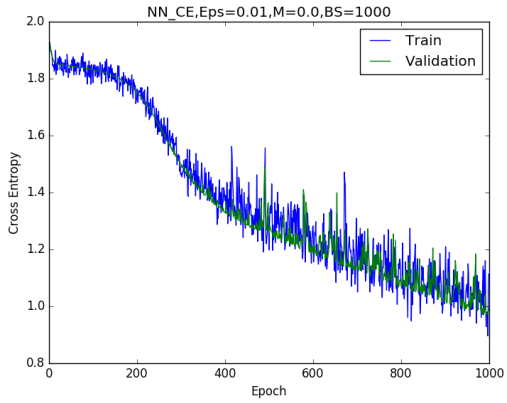


Batch Size = 500

*Answer to Q*3.2(*c*)
As the batch size increases, more samples are processed before the model is updated (i.e., before the gradient is evaluated), which leads to a smoother plot for training set (Just compare the graph of batch size = 1 to the 1000, the BS = 1's plot is very noisy on the training set) with a lower convergence rate on the validation set (for BS = 1, the model converge before 200 epoch, while on BS = 1000, it doesn't even converge at the end of 1000 epoch).

The accuracy for all the different batch size is quite similar, the only difference is the convergence happens earlier with lower batch size. It suggest that smaller batch size will increase the convergence rate but with the cost of cross entropy.





Batch Size = 1000

d. How would you choose the best value of these parameters?

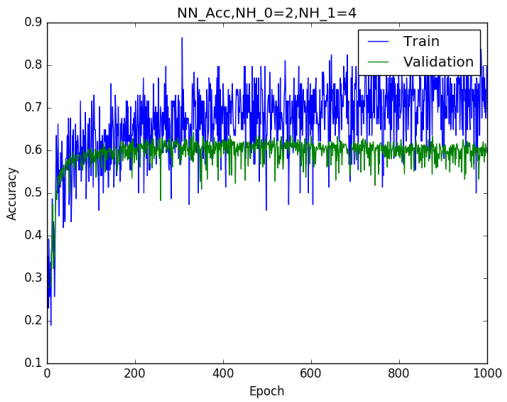
*Answer to Q*3.2(*d*)
As we've discussed above, higher epsilon, higher momentum and lower batch size will increase the convergence rate. But we can't go as much as we want, when epsilon or momentum is too large (eps = 1 or momentum > 0.9), or the batch size is too small (BS = 1), the performance do not improve at all, and become noisy.

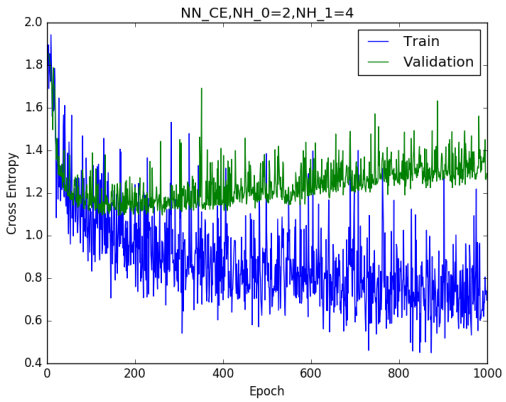
Based on my plots and the scenario, I would suggest eps = 0.05 to 0.1, momentum = 0.5 to 0.9, batch size = 10 to 500. But for other models and datasets, we might need to run several combinations of parameters and observe the trend.

```
################ Code for Q3.2 ################
def Optimization():
    """
    Try 5 different values of the learning rate from 0.001 to 1.0.
    Try 3 values of momentum from 0.0 to 0.9.
    Try 5 different mini-batch sizes, from 1 to 1000.
    Find the best value of these parameters
    """
    model_fname = 'nn_model.npz'
    stats_fname = 'nn_stats.npz'

    # Default hyper-parameters.
    num_hiddens = [16, 32]
    num_epochs = 1000
    eps = 0.01
    momentum = 0.0
    batch_size = 100

    # Input-output dimensions.
    num_inputs = 2304
    num_outputs = 7

    # Try different eps
    for eps_i in [0.001, 0.05, 0.1, 0.5, 1.0]:
        model = InitNN(num_inputs, num_hiddens, num_outputs)
        stats = Train(model, NNForward, NNBackward, NNUpdate, eps_i,
                      momentum, num_epochs, batch_size)
        Save('nn_model_eps'+str(eps_i)+'.npz', model)
        Save('nn_stats_eps'+str(eps_i)+'.npz', stats)

    # Try different momentum
    for momentum_i in [0, 0.5, 0.9]:
        model = InitNN(num_inputs, num_hiddens, num_outputs)
        stats = Train(model, NNForward, NNBackward, NNUpdate, eps,
                      momentum_i, num_epochs, batch_size)
        Save('nn_model_momentum'+str(momentum_i)+'.npz', model)
        Save('nn_stats_momentum'+str(momentum_i)+'.npz', stats)

    # Try different batch size
    for batch_size_i in [1, 10, 50, 500, 1000]:
        model = InitNN(num_inputs, num_hiddens, num_outputs)
        stats = Train(model, NNForward, NNBackward, NNUpdate, eps,
                      momentum, num_epochs, batch_size_i)
        Save('nn_model_batch_size'+str(batch_size_i)+'.npz', model)
        Save('nn_stats_batch_size'+str(batch_size_i)+'.npz', stats)
```

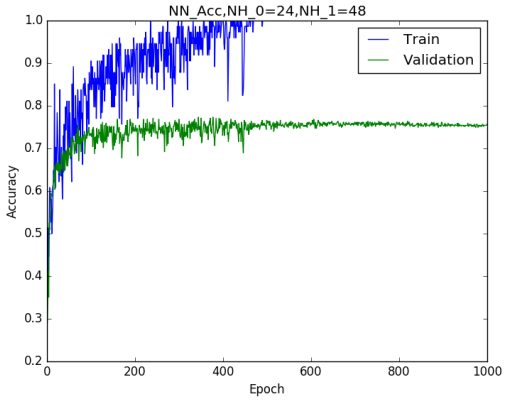
3.3. Model architecture

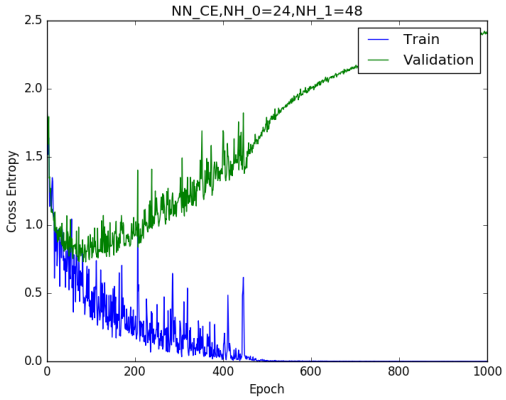
Fix momentum to be 0.9. Try 3 different values of the number of hidden units for each layer of the fully connected network (range from 2 to 100). You might need to adjust the learning rate and the number of epochs. Comment on the effect of this modification on the convergence properties, and the generalization of the network.



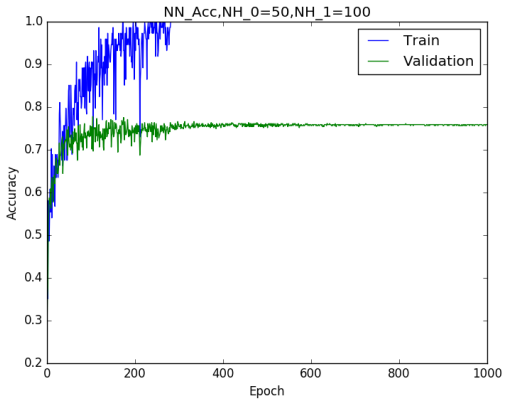


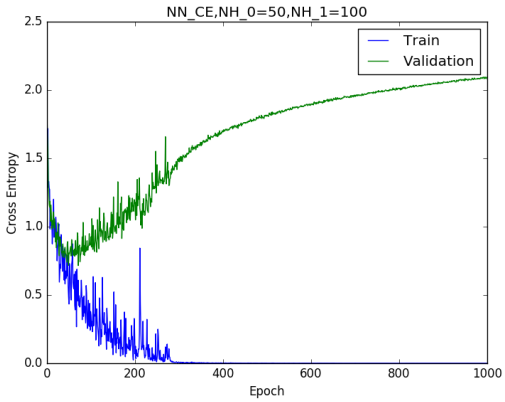
Hidde Units = (2, 4)





Hidde Units = (24, 48)





Hidde Units = (50, 100)

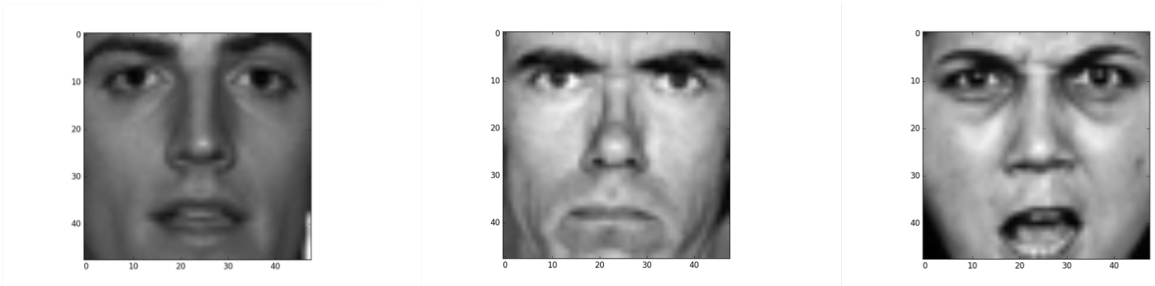
*Answer to Q*3.3
We can see that as the number of hidden units for each layer of NN increases, the model converges more quickly, while overfitting increases. For instance, with 50 and 100 hidden units in the two layers, the network converges after around 300 epochs, but also begins to overfit after less than 100. The overall accuracy is similar to the default neural network in this case, but the generalization is much worse (as the network obtains 100% accurracy on the training data very quickly and actually begins to increase in cross entropy error in the validation data concurrently). For 24 and 48 hidden units in the two layers the network converges after around 500 epochs, but also beings overfitting within 100 epochs. Finally, for 2 and 4 hidden units, the network is much less accurate, but does not overfit. It also seems to converge relatively quickly, but is far noisier than for the networks with more hidden units.

The trade-off is in performance however, as it is both less accurate and has more error than the default network as it is a far simpler model.

```
################ Code for Q3.3 ################
def ModelArchitecture():
    """
    Try 3 different values of the number of hidden units
    """
    eps = 0.01
    momentum = 0.9
    num_epochs = 1000
    batch_size = 100
    num_inputs = 2304
    num_outputs = 7

    # Try different values of # of hidden units
    for hidden_units_i in [[2, 4], [20, 40] , [50, 100]]:
        model = InitNN(num_inputs, hidden_units_i, num_outputs)
        stats = Train(model, NNForward, NNBackward, NNUpdate, eps,
                      momentum, num_epochs, batch_size)
```

3.4. Network Uncertainty

Plot some examples where the neural network is not confident of the classification output (the top score is below some threshold), and comment on them. Will the classifier be correct if it outputs the top scoring class anyways?



(a) #173 in train set (b) #170 in validation set (c) #198 in test set

Comments :
(a) #173 Softmax:
[0.23994943, 0.15444303, 0.05461541, 0.16983587, 0.15332245, 0.02243239 , 0.20540143]

The model thought the expression was angry with a softmax score of 0.23994943. Here the correct target is neutral, which it predicted with a softmax score of 0.205. So here the network was not confident and actually made the wrong prediction between its two highest options.

(b) #170 Softmax:
[0.27735642, 0.00162479, 0.16682891, 0.00140601, 0.19614639, 0.07920172, 0.27743576]

The model predicted with softmax score 0.2774 that this face was neutral. The correct target is in fact angry (this was the second highest softmax score which it predicted with probability 0.2773). In this case, the network is definitely not confident and unfortunately got the classification incorrect by outputing the top scoring result.

(c) #198 Softmax:
[0.25038703, 0.19903878, 0.20576177, 0.00046154, 0.05101528, 0.28049365,s 0.01284195]

The network predicted this face was surprise with a softmax score of 0.28049365. The correct label for this face is angry, which got a softmax score of 0.25038703. Here again, the network was uncertain as the top scores were all similar, but by going with the highest value, it was incorrect.

```
################ Code for Q3.4 ################
def plot_uncertain_images(x, t, prediction, threshold):
    """
    Provided on Piazza, plot the uncertain images
    """
    low_index = np.max(prediction, axis=1) < threshold
    class_names = ['anger', 'disgust', 'fear', 'happy', 'sad', 'surprised',
                   'neutral']
    if np.sum(low_index) > 0:
        for i in np.where(low_index > 0)[0]:

            plt.figure()
            img_w, img_h = int(np.sqrt(2304)), int(
                np.sqrt(2304)) # 2304 is input size
            plt.imshow(x[i].reshape(img_h, img_w))
            plt.title('P_max: {}, Predicted: {}, Target: {}'.format(
                np.max(prediction[i]),
                class_names[np.argmax(prediction[i])],
                class_names[np.argmax(t[i])]))
            plt.show()
            input("press enter to continue")
    return
```