1.1 By Bayes 'rule, we know: $p(t=1|x) = \frac{p(x|t=1)\,p(t=1)}{p(x)}$

By Product rule, we know: $= \frac{\alpha\, p(x|t=1)}{\alpha\, p(x|t=1) + (1-\alpha) p(x|t=0)}$

$$= \frac{1}{1 + \frac{1-a}{a} \frac{p(x|t=0)}{p(x|t=1)}}$$

Since we are given that $t = \{0, 1\}$ and all x_i are independent, and $x_i|t \sim \mathcal{N}(\mu_{it}, \sigma_i^2)$

we can write: $p(x|t=0) = \prod_{i=1}^{D} p(x_i|t=0)$

$$= \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}\right)$$

$$= \left(\frac{1}{\sqrt{2\pi\sigma_i^2}}\right)^D \exp\left(\sum_{i=1}^{D} -\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}\right)$$

similarly, we know $P(x|t=1) = \left(\frac{1}{\sqrt{2\pi\sigma_i^2}}\right)^D \exp\left(\sum_{i=1}^{D} -\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}\right)$

we can now compute $\frac{1-a}{a} \frac{p(x|t=0)}{p(x|t=1)}$

$$= \frac{1-a}{a} \frac{\exp\left(-\sum_{i=1}^{D} \frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}\right)}{\exp\left(-\sum_{i=1}^{D} \frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}\right)}$$

$$= \frac{1-a}{a} \exp\left(-\sum_{i=1}^{D} \frac{(x_i-\mu_{i0})^2}{2\sigma_i^2} - \frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}\right)$$

$$= \frac{1-a}{a} \exp\left(-\sum_{i=1}^{D} \frac{2\mu_{i1}-2\mu_{i0}}{2\sigma_i^2} x_i + \frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}\right)$$

$$= \frac{1-a}{a} \exp\left(-\sum_{i=1}^{D} \frac{2\mu_{i1}-2\mu_{i0}}{2\sigma_i^2} x_i - \sum_{i=1}^{D} \frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}\right)$$

$$= \exp\left(-\sum_{i=1}^{D} \frac{2\mu_{i1}-2\mu_{i0}}{2\sigma_i^2} x_i - \left(\left(\sum_{i=1}^{D} \frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}\right) - \log\frac{1-\alpha}{\alpha}\right)\right)$$

$\Rightarrow p(t=1|x) = \frac{1}{1+\exp\left(-\sum_{i=1}^{D} w_i x_i - b\right)}$ where $w_i = \frac{2\mu_{i1}-2\mu_{i0}}{2\sigma_i^2} = \frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}$

$$b = \left(\sum_{i=1}^{D} \frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}\right) - \log\frac{1-\alpha}{\alpha}$$

1.2 $$L(w,b) = -\log \prod_{i=1}^{N} p(t^{(i)}=1|x^{(i)},w,b)^{t^{(i)}} \left(1-p(t^{(i)}=1|x^{(i)},w,b)\right)^{1-t^{(i)}}$$

$$= -\sum_{i=1}^{N} t^{(i)} \log p(t^{(i)}=1/x^{(i)},w,b) + (1-t^{(i)})\log\left(1-p(t^{(i)}=1|x^{(i)},w,b)\right)$$

$$= -\sum_{i=1}^{N} t^{(i)} \log \frac{1}{1+\exp(-w^Tx^{(i)}-b)} + (1-t^{(i)})\log\left(1-\frac{1}{1+\exp(-w^Tx^{(i)}-b)}\right)$$

$$= -\sum_{i=1}^{N} \log\left(1-\frac{1}{1+\exp(-w^Tx^{(i)}-b)}\right) + t^{(i)}\log\left(\frac{1}{1+\exp(-w^Tx^{(i)}-b)}\right)\left(1-\frac{1}{1+\exp(-w^Tx^{(i)}-b)}\right)^{-1}$$

$$= -\sum_{i=1}^{N} \log\left(1-\frac{1}{1+\exp(-w^Tx^{(i)}-b)}\right) + t^{(i)}(w^Tx^{(i)}+b)$$

$$\frac{\partial}{\partial w}L(w,b) = -\sum_{i=1}^{N} \frac{\partial}{\partial w}\left(\log\left(1-\frac{1}{1+\exp(-w^Tx^{(i)}-b)}\right)\right) + \frac{\partial}{\partial w}\left(t^{(i)}(w^Tx^{(i)}+b)\right)$$

$$= -\sum_{i=1}^{N} -\frac{1}{1+e^{w^Tx+b}} e^{w^Tx^{(i)}+b} x^{(i)} + t^{(i)}x^{(i)}$$

$$\frac{\partial}{\partial b}L(w,b) = -\sum_{i=1}^{N} \frac{\partial}{\partial b}\left(\log\left(1-\frac{1}{1+\exp(-w^Tx^{(i)}-b)}\right)\right) + \frac{\partial}{\partial b}\left(t^{(i)}(w^Tx^{(i)}+b)\right)$$

$$= -\sum_{i=1}^{N} -\frac{1}{1+e^{w^Tx+b}} e^{w^Tx^{(i)}+b} + t^{(i)}$$

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

Since we know $p(w,b|t^{(1)},\dots,t^{(N)}) \propto p(w)\,p(b)\,p(t^{(1)},\dots,t^{(N)}|w,b)$

we can write $L_{post}(w,b) = -\log\prod_{i=1}^{D} p(w_i)\,p(b_i)\,p(t^{(i)}|w,b)$

we know $p(b)=1$ and $p(w) = \mathcal{N}(w_i|0, 1/\lambda) = \sqrt{\frac{\lambda}{2\pi}}\, e^{-\frac{\lambda w_i^2}{2}}$

1.2 answer

$$L_{post}(w,b) = -\sum_{i=1}^{D} \log\sqrt{\frac{\lambda}{2\pi}} + \log e^{-\frac{\lambda w_i^2}{2}} - \sum_{i=1}^{D} \log p(t^{(i)}|w,b)$$

$$= L(w,b) + \sum_{i=1}^{D} \frac{\lambda w_i^2}{2} - \log\sqrt{\frac{\lambda}{2\pi}}$$

$$= L(w,b) + \frac{\lambda}{2}\sum_{i=1}^{D} w_i^2 - \log\sqrt{\frac{\lambda}{2\pi}} \quad \text{where } C = -\log\sqrt{\frac{\lambda}{2\pi}}$$

$$\frac{\partial}{\partial w} L_{post}(w,b) = \frac{\partial}{\partial w} L(w,b) + \frac{\partial}{\partial w}\left[\frac{\lambda}{2}\sum_{i=1}^{D} w_i^2 - \log\sqrt{\frac{\lambda}{2\pi}}\right]$$

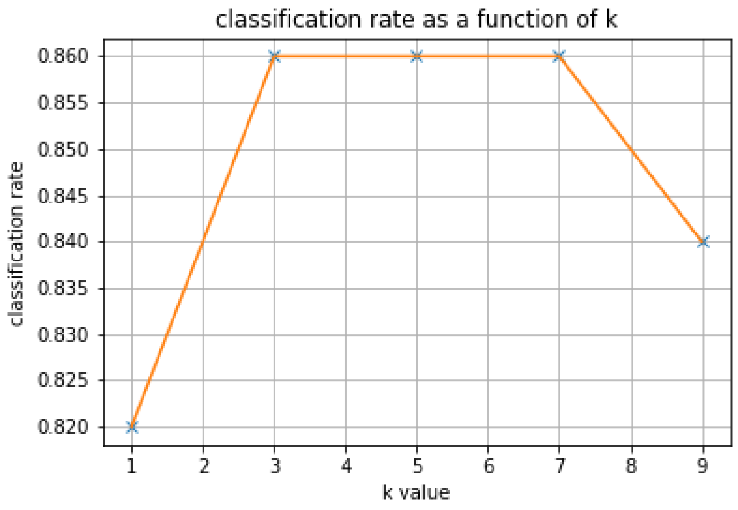
$$= \frac{\partial}{\partial w} L(w,b) + \frac{\lambda}{2}\sum_{i=1}^{D} 2w_i$$

$$= \frac{\partial}{\partial w} L(w,b) + \lambda\sum_{i=1}^{D} w_i \quad \text{where } \frac{\partial}{\partial w}L(w,b) \text{ is answer from 1.2}$$

$$\frac{\partial}{\partial b} L_{post}(w,b) = \frac{\partial}{\partial b} L(w,b) + \frac{\partial}{\partial b}\left[\frac{\lambda}{2}\sum_{i=1}^{D} w_i^2 - \log\sqrt{\frac{\lambda}{2\pi}}\right]$$

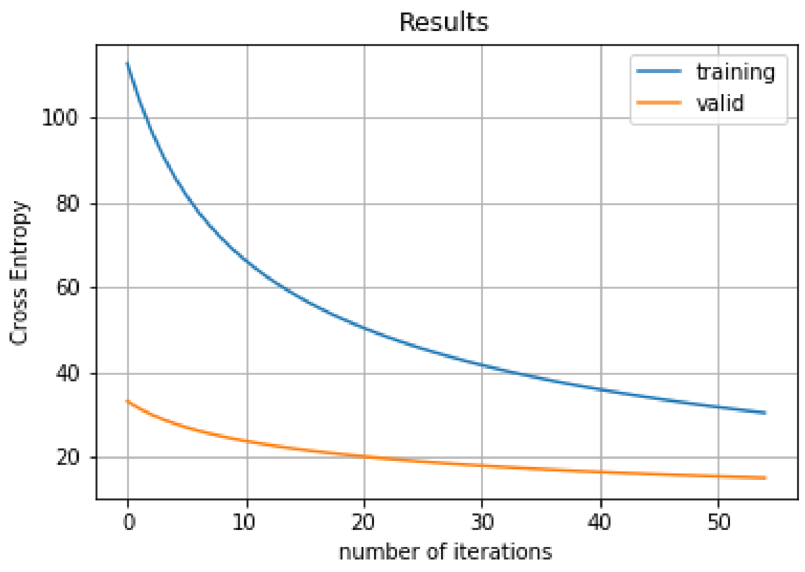
$$= \frac{\partial}{\partial b} L(w,b) \quad \text{where } \frac{\partial}{\partial b}L(w,b) \text{ is answer from 1.2.}$$

2.1 The performance of my classifier is pretty good as it gives us a classification rate of over 80%. I would pick k = 5, since the classification rate is one of the highest. Also, for k-2 and k+2, which is k=3 and k=7 they both have the same classification rate as k=5. Therefore, k=5 would be the perfect choice since it allows some room and prevent underfitting and overfitting. Note that the test performance for these k corresponds to validation performance since we can see a peak at k = 5 , k-2 = 3, and k+2 = 7.



```
import numpy as np
import matplotlib.pyplot as plt
from utils import load_train, load_valid
from run_knn import run_knn

trainData = load_train()
validData = load_valid()

kRange = [1,3,5,7,9]
results = []
for k in kRange:
    temp = run_knn(k, trainData[0],trainData[1],validData[0])
    results.append(temp)

def classificationRate(validSet, trainResult):
    return np.sum(validSet == trainResult)/len(validSet)

classificationRateResults = [classificationRate(validData[1], i) for i in
results]

fig, graph = plt.subplots()
graph.plot(kRange, classificationRateResults, 'x')
graph.plot(kRange, classificationRateResults)
graph.set(xlabel='k value', ylabel='classification rate',
       title='classification rate as a function of k')
graph.grid()
fig.savefig("q2_1.png")
plt.show()
```

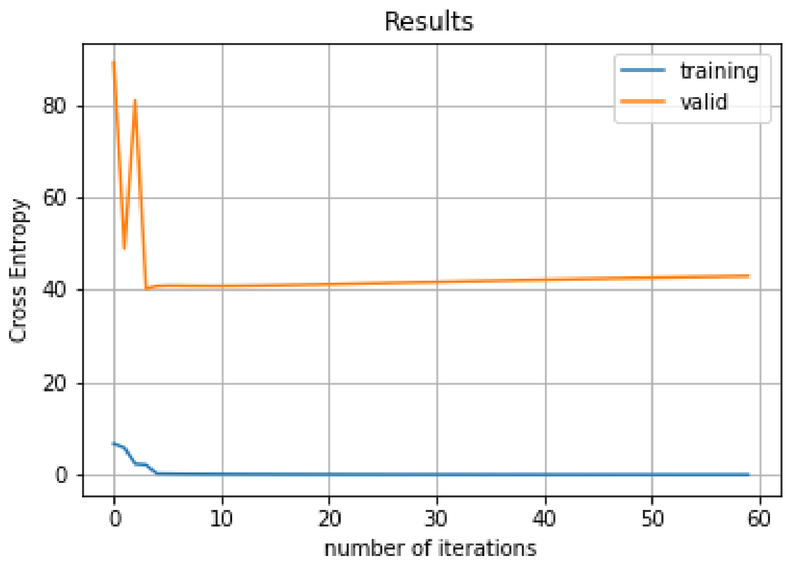
2.2 For mnist_train, I found that the best hyperparameters were: 0.1 learning rate, 55 iterations and a weight that is initialized as 0.1 * np.random.randn(len(train_inputs[0])+1, 1) to create a N x M weights matrix. The results on the last iteration is as follow:



ITERATION: 40 TRAIN NLOGL:0.06 TRAIN CE:9.923683 TRAIN FRAC:100.00 VALID CE:11.175484 VALID FRAC:90.00 TEST CE:13.535786 TEST FRAC:92.00

From the plot, we can see that the cross entropy decreases as number of iterations increases, and it has a logarithmic trend in which it seems to unable to go lower than 15. After re-running the code several times, the general trend does not change much.

For mnist_train_small, I found that the best hyperparameters were: 0.7 learning rate, 60 iterations and a weight that is initialized as 0.01 * np.random.randn(len(train_inputs[0])+1, 1) to create a N x M weights matrix. The results on the last iteration is as follow:



ITERATION: 60 TRAIN NLOGL:0.00 TRAIN CE:0.049714 TRAIN FRAC:100.00 VALID CE:43.004112 VALID FRAC:70.00 CE:49.152572 TEST FRAC:78.00

From the plot, we can see that the cross validation reaches a minimum and then slowly increase. On the other hand, our accuracy for the test data is reaching 78% which is very good. However, the couple spikes in the beginning of the graph maybe due to high learning rate, but this choice of parameter created a relatively high accuracy. Therefore, for other general set, it is better to tune down the learning rate.

Therefore, in order to choose for the best hyperparameters, I would start with a lower learning rate and then slowly increase it and number of iterations. Also keep in mind to reduce the cross entropy.

Snippet code for logistic.py

```
def logistic_predict(weights, data):
    z = np.dot(data, weights[:len(data[0])])
    y = sigmoid(z)
    return y

def evaluate(targets, y):
    ce = np.sum(-np.multiply(targets, np.log(y)) - np.multiply((1-targets),
np.log(1-y)))
    normalizedY = [[1] if i > 0.5 else [0] for i in y]
    frac_correct = np.sum(normalizedY == targets)/len(targets)
    return ce, frac_correct

def logistic(weights, data, targets, hyperparameters):
    y = logistic_predict(weights, data)
    f , percent = evaluate(targets, y)
    dfdw = np.dot(data.T, y - targets)
    dfdb = [np.zeros(weights[-1].shape)]
    df = np.append(dfdw, dfdb, axis=0)
    return f, df, y
```

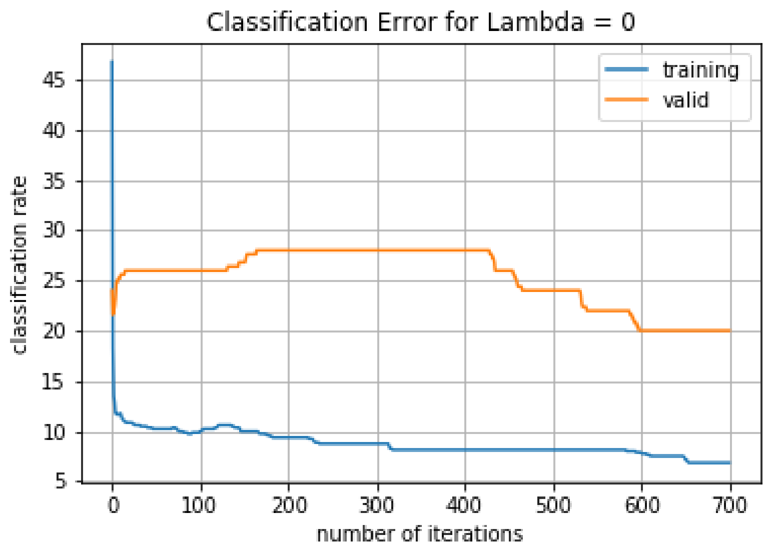
Snippet code for logistic regression template.py

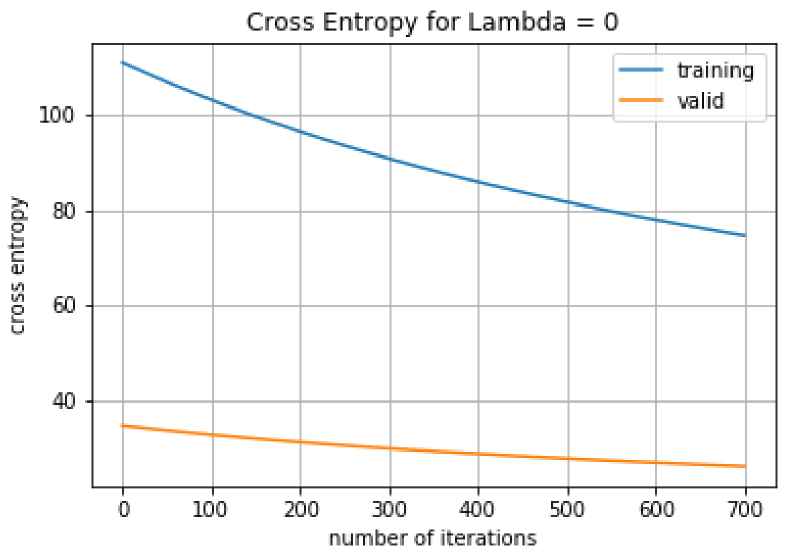
```
weights = 0.01 * np.random.randn(len(train_inputs[0])+1, 1)
    tList = np.arange(0,hyperparameters['num_iterations'],1)
    train_CEList = []
    valid_CEList = []
    for t in range(hyperparameters['num_iterations']):
        # TODO: you may need to modify this loop to create plots, etc.

        # Find the negative log likelihood and its derivatives w.r.t. the
weights.
        f, df, predictions = logistic(weights, train_inputs, train_targets,
hyperparameters)

        # Evaluate the prediction.
        cross_entropy_train, frac_correct_train = evaluate(train_targets,
predictions)

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

        # update parameters
        weights = weights - hyperparameters['learning_rate'] * df / N
        # Make a prediction on the valid_inputs.
        predictions_valid = logistic_predict(weights, valid_inputs)
        # Make a prediction on test inputs
        predictions_test = logistic_predict(weights, test_inputs)

        # Evaluate the prediction.
        cross_entropy_valid, frac_correct_valid = evaluate(valid_targets,
predictions_valid)
        # Evaluate the prediction for test.
        cross_entropy_test, frac_correct_test = evaluate(test_targets,
predictions_test)

        train_CEList.append(cross_entropy_train)
        valid_CEList.append(cross_entropy_valid)
        # print some stats
        print(("ITERATION:{:4d}  TRAIN NLOGL:{:4.2f}  TRAIN CE:{:.6f} "
               "TRAIN FRAC:{:2.2f}  VALID CE:{:.6f}  VALID FRAC:{:2.2f} "
               "TEST CE:{:.6f}  TEST FRAC:{:2.2f}").format(
                   t+1, f / N, cross_entropy_train, frac_correct_train*100,
                   cross_entropy_valid, frac_correct_valid*100,
                   cross_entropy_test, frac_correct_test*100))
    fig, graph = plt.subplots()
    graph.plot(tList, train_CEList, label="training")
    graph.plot(tList, valid_CEList, label="valid")
    graph.set(xlabel='number of iterations', ylabel='Cross Entropy',
           title="Results")
    graph.grid()
    graph.legend()
    fig.savefig("q2_2.png")
    plt.show()
```

2.3 For mnist_train, I found that the best hyperparameters were: 0.001 learning rate, 700 iterations and a weight that is initialized as 0.0001 * np.random.randn(len(train_inputs[0])+1, 1) to create a N x M weights matrix. The results on the last iteration of each run is as follow:

λ = 0:

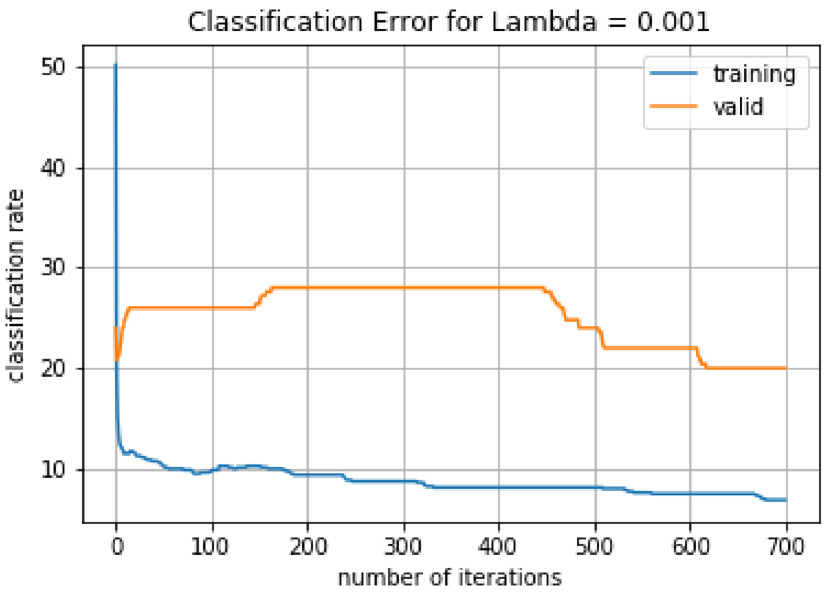
```
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.566827 TRAIN
FRAC:93.12  VALID CE:26.149396  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.568467 TRAIN
FRAC:93.12  VALID CE:26.148824  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.569547 TRAIN
FRAC:93.12  VALID CE:26.149230  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.570664 TRAIN
FRAC:93.12  VALID CE:26.150035  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.570393 TRAIN
FRAC:93.12  VALID CE:26.146994  VALID FRAC:80.00
```

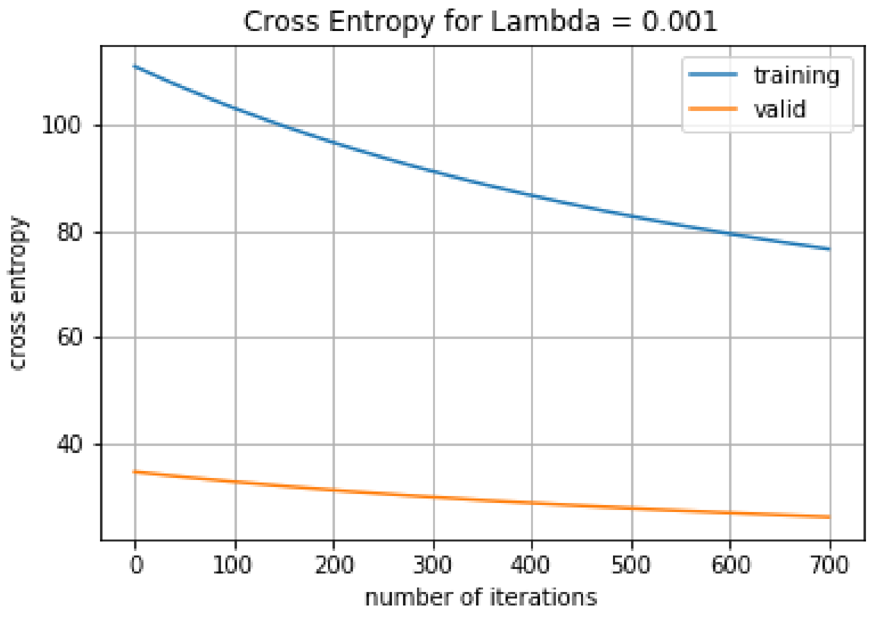




λ = 0.001:

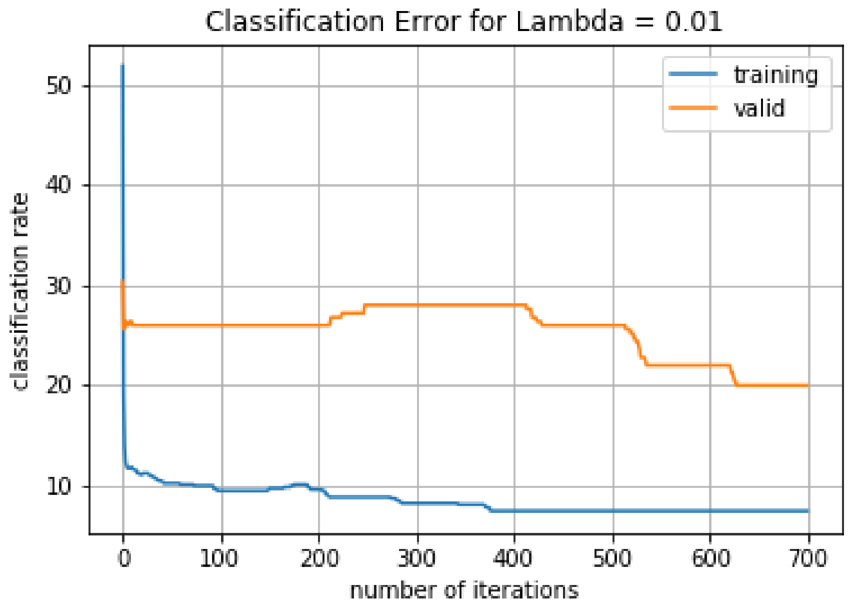
```
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.572389 TRAIN
FRAC:93.12  VALID CE:26.149155  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.573468 TRAIN
FRAC:93.12  VALID CE:26.150391  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.565162 TRAIN
FRAC:93.12  VALID CE:26.148334  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.574172 TRAIN
FRAC:93.12  VALID CE:26.151249  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.569868 TRAIN
FRAC:93.12  VALID CE:26.148677  VALID FRAC:80.00
```

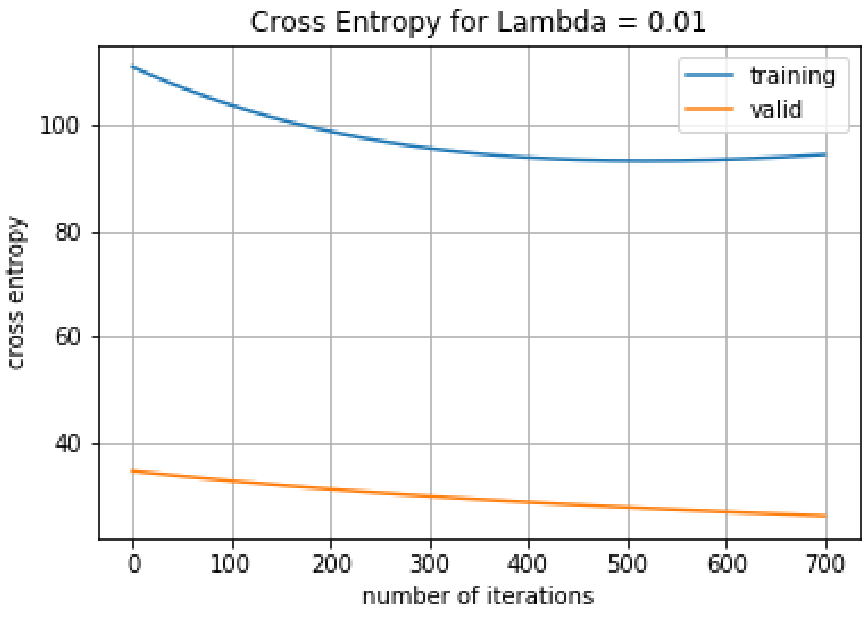




λ = 0.01:

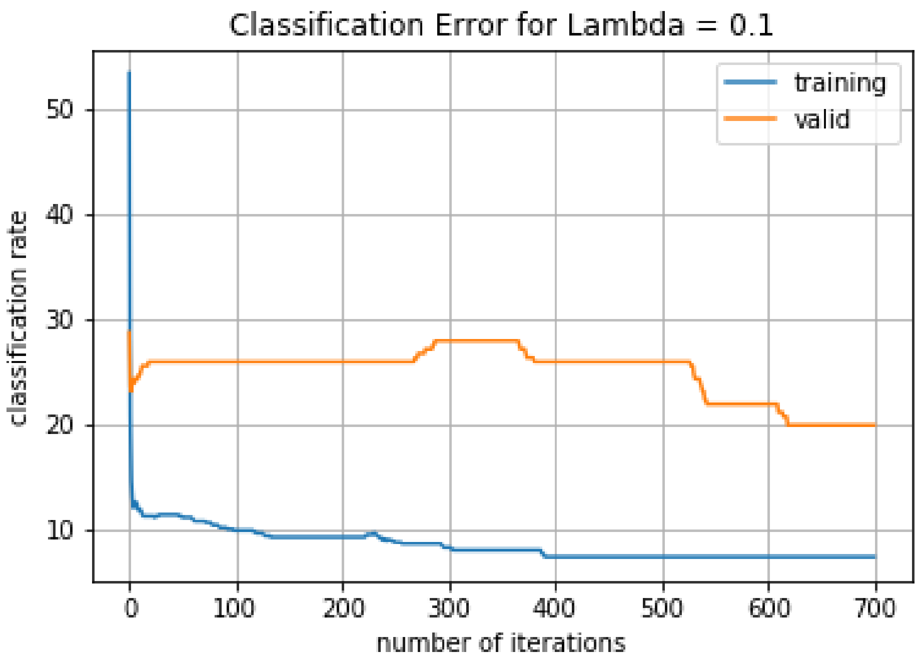
```
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.597194 TRAIN
FRAC:92.50  VALID CE:26.154684  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.586457 TRAIN
FRAC:92.50  VALID CE:26.148769  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.589075 TRAIN
FRAC:92.50  VALID CE:26.153018  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.585326 TRAIN
FRAC:92.50  VALID CE:26.149716  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.589362 TRAIN
FRAC:92.50  VALID CE:26.150401  VALID FRAC:80.00
```

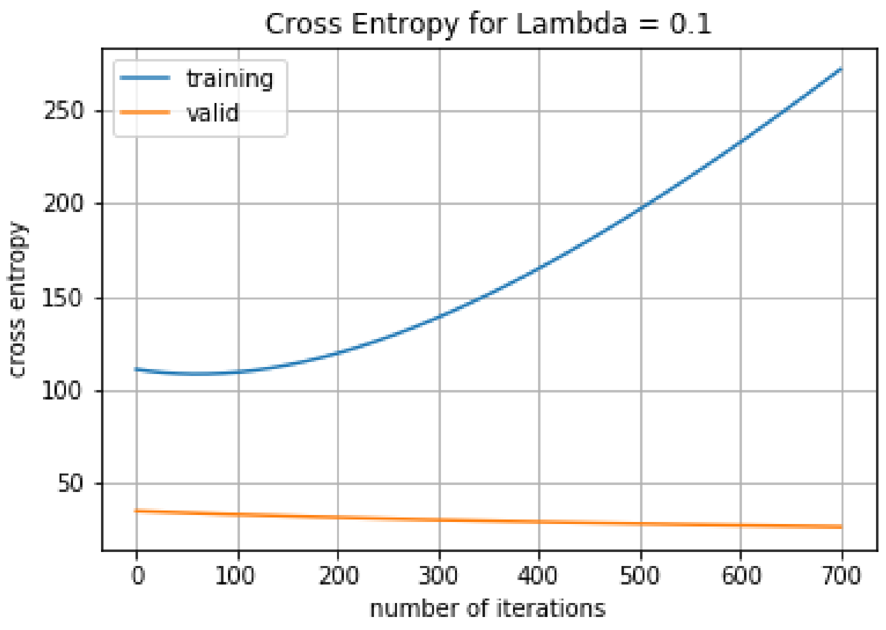




λ = 0.1:

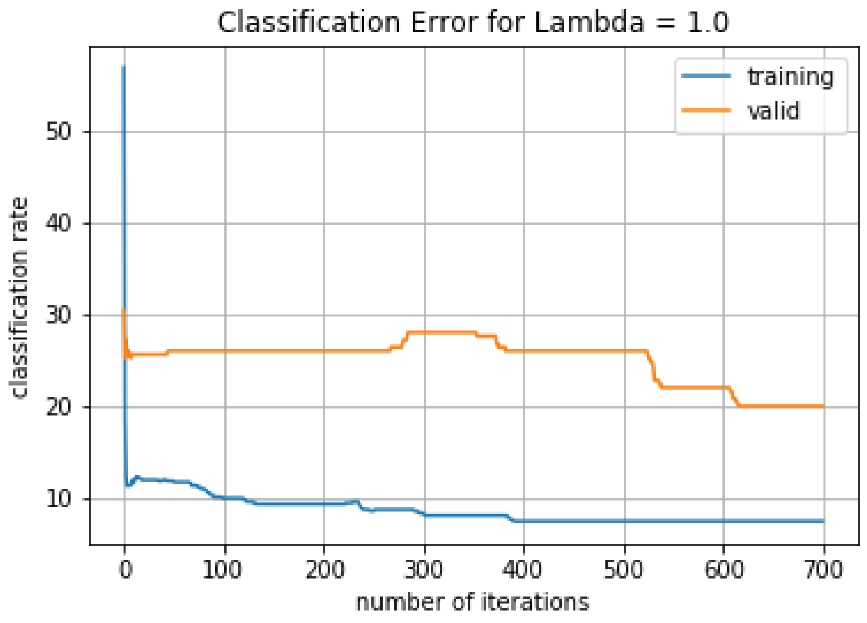
```
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.588937 TRAIN
FRAC:92.50  VALID CE:26.151304  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.593339 TRAIN
FRAC:92.50  VALID CE:26.151275  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.599905 TRAIN
FRAC:92.50  VALID CE:26.153442  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.587060 TRAIN
FRAC:92.50  VALID CE:26.153024  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.47  TRAIN CE:74.586471 TRAIN
FRAC:92.50  VALID CE:26.152119  VALID FRAC:80.00
```

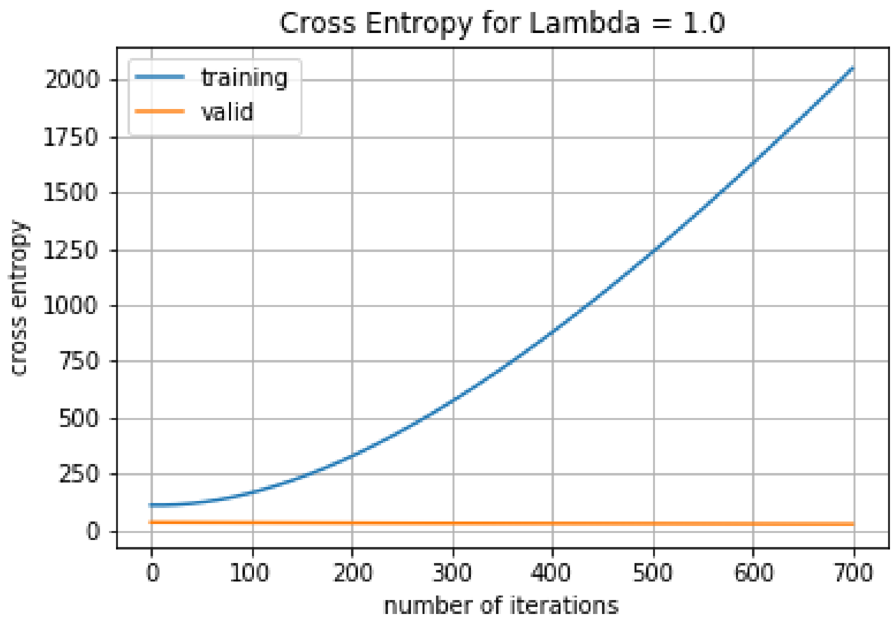




λ = 1.0:

```
ITERATION: 700  TRAIN NLOGL:0.54  TRAIN CE:74.596676 TRAIN
FRAC:92.50  VALID CE:26.150739  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.54  TRAIN CE:74.583762 TRAIN
FRAC:92.50  VALID CE:26.151616  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.54  TRAIN CE:74.592070 TRAIN
FRAC:92.50  VALID CE:26.151440  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.54  TRAIN CE:74.589866 TRAIN
FRAC:92.50  VALID CE:26.152365  VALID FRAC:80.00
ITERATION: 700  TRAIN NLOGL:0.54  TRAIN CE:74.587671 TRAIN
FRAC:92.50  VALID CE:26.151401  VALID FRAC:80.00
```

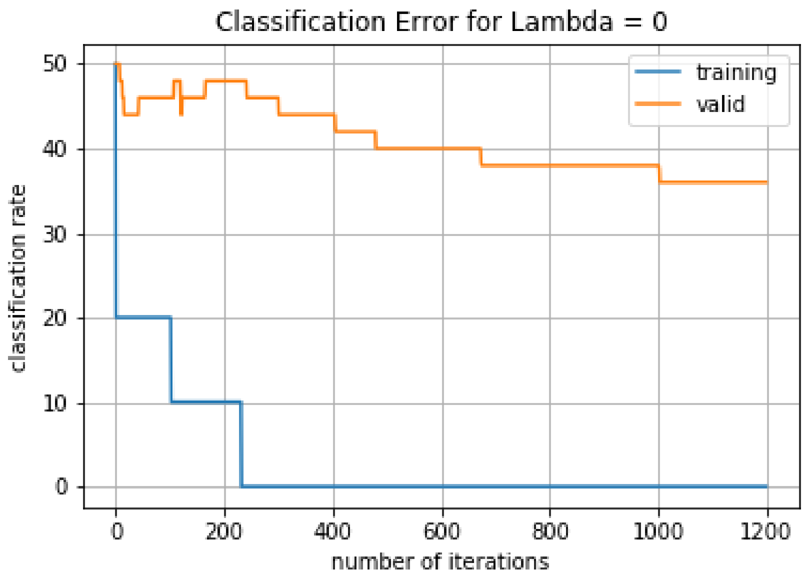


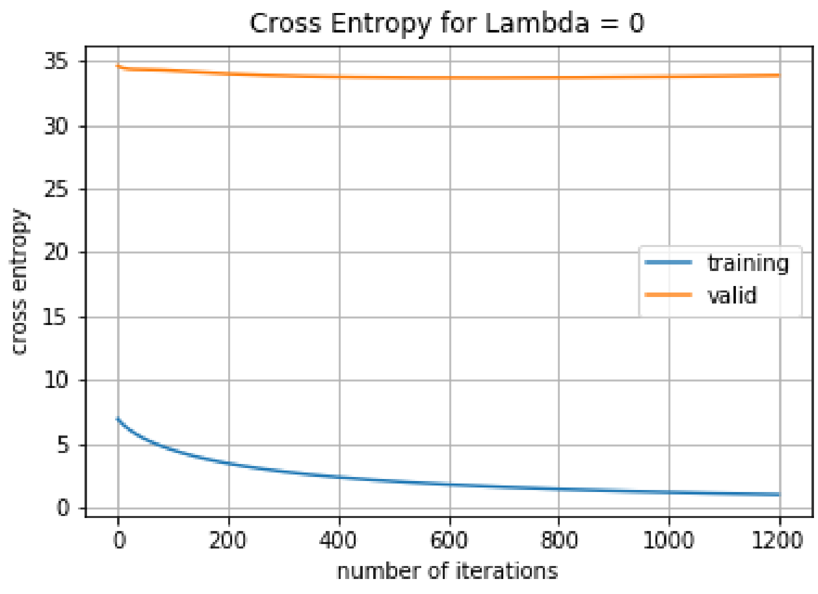


For mnist_train_small, I found that the best hyperparameters were: 0.002 learning rate, 1200 iterations and a weight that is initialized as 0. 00001 * np.random.randn(len(train_inputs[0])+1, 1) to create a N x M weights matrix. The results on the last iteration of each run is as follow:

λ = 0:

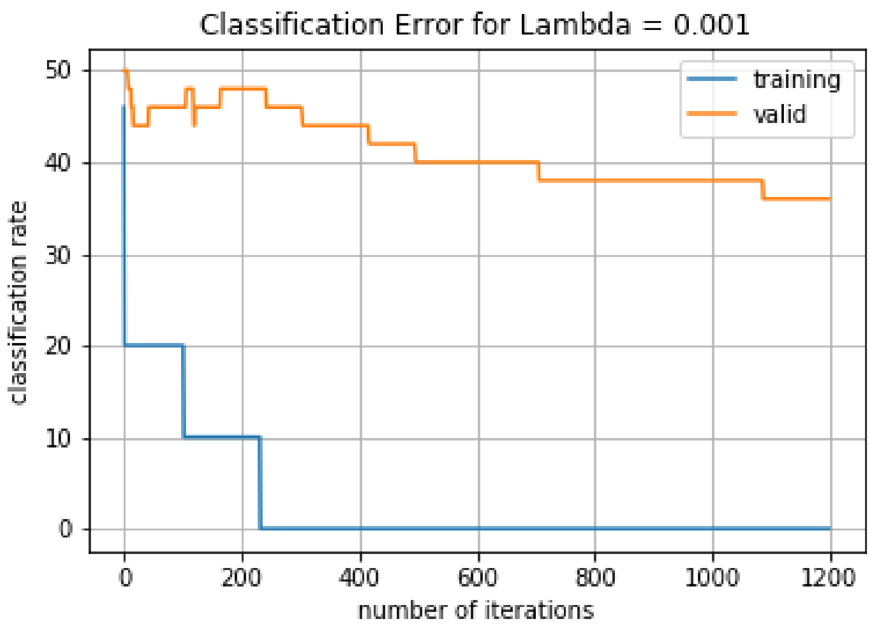
```
ITERATION:1200  TRAIN NLOGL:0.10  TRAIN CE:1.000040 TRAIN
FRAC:100.00  VALID CE:33.885092  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.10  TRAIN CE:1.000046 TRAIN
FRAC:100.00  VALID CE:33.886169  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.10  TRAIN CE:1.000050 TRAIN
FRAC:100.00  VALID CE:33.885465  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.10  TRAIN CE:1.000057 TRAIN
FRAC:100.00  VALID CE:33.885655  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.10  TRAIN CE:1.000044 TRAIN
FRAC:100.00  VALID CE:33.885859  VALID FRAC:64.00
```

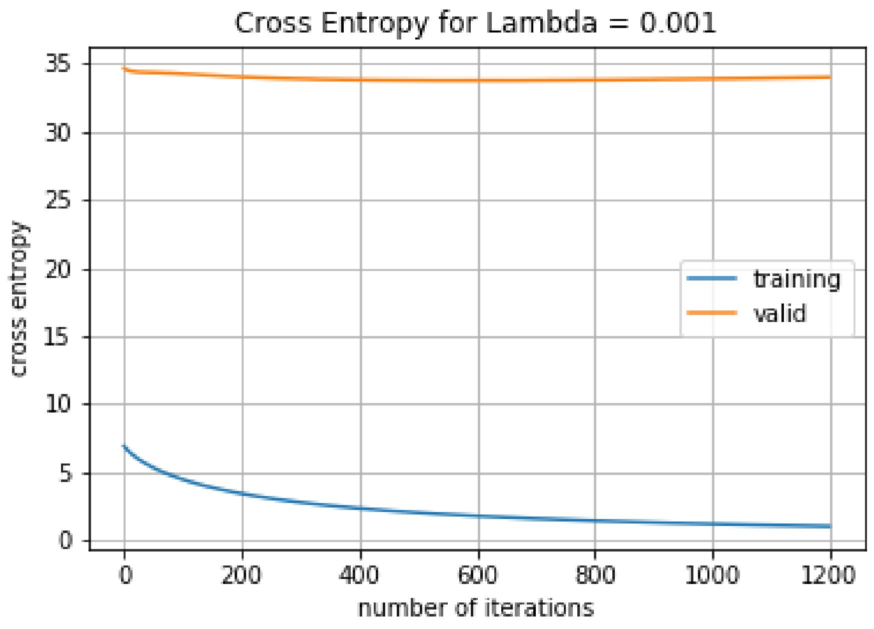




λ = 0.001:

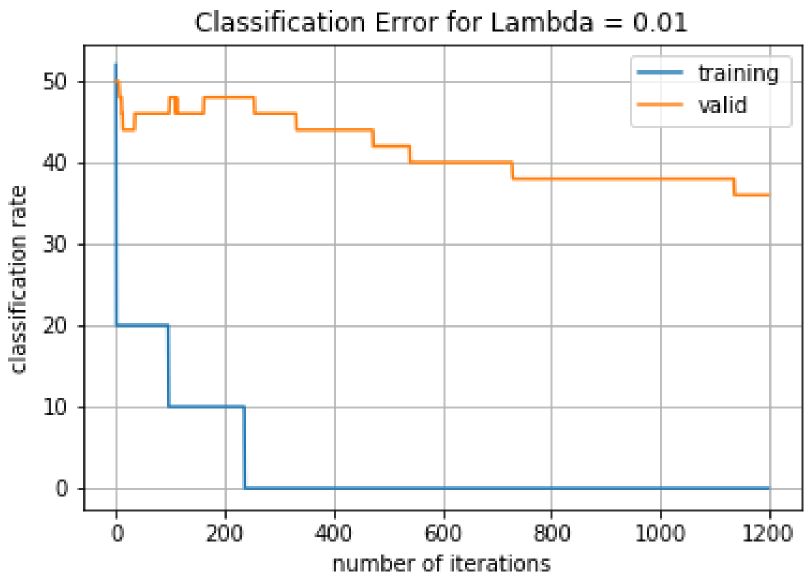
```
ITERATION:1200  TRAIN NLOGL:0.11  TRAIN CE:1.002456 TRAIN
FRAC:100.00  VALID CE:33.984513  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.11  TRAIN CE:1.002449 TRAIN
FRAC:100.00  VALID CE:33.984423  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.11  TRAIN CE:1.002453 TRAIN
FRAC:100.00  VALID CE:33.984810  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.11  TRAIN CE:1.002448 TRAIN
FRAC:100.00  VALID CE:33.984135  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.11  TRAIN CE:1.002452 TRAIN
FRAC:100.00  VALID CE:33.984158  VALID FRAC:64.00
```

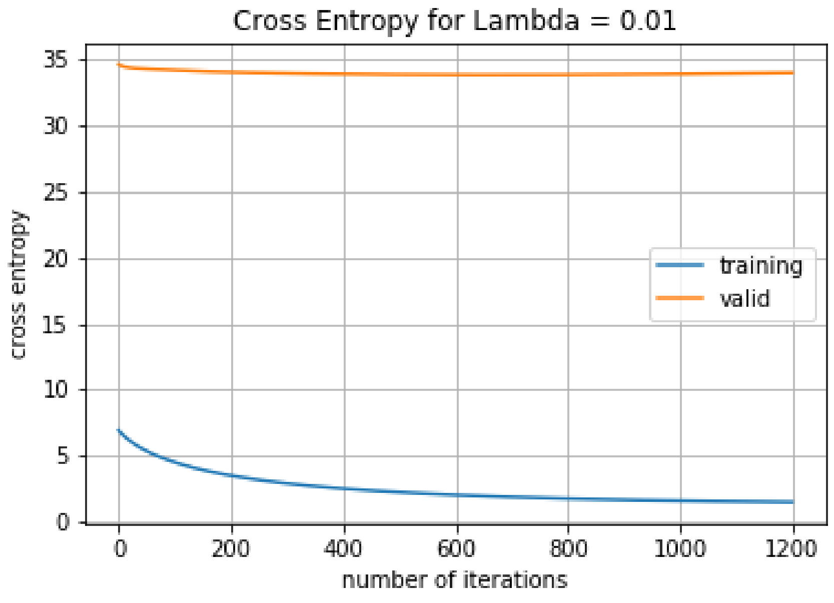




λ = 0.01:

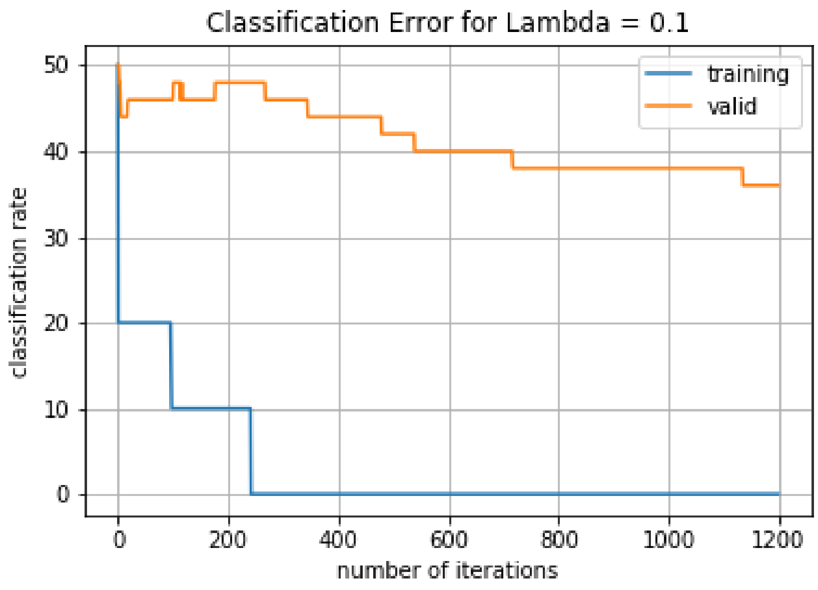
```
ITERATION:1200  TRAIN NLOGL:0.15  TRAIN CE:1.003877 TRAIN
FRAC:100.00  VALID CE:33.994472  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.15  TRAIN CE:1.003879 TRAIN
FRAC:100.00  VALID CE:33.994371  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.15  TRAIN CE:1.003893 TRAIN
FRAC:100.00  VALID CE:33.994736  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.15  TRAIN CE:1.003879 TRAIN
FRAC:100.00  VALID CE:33.995020  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.15  TRAIN CE:1.003891 TRAIN
FRAC:100.00  VALID CE:33.994663  VALID FRAC:64.00
```

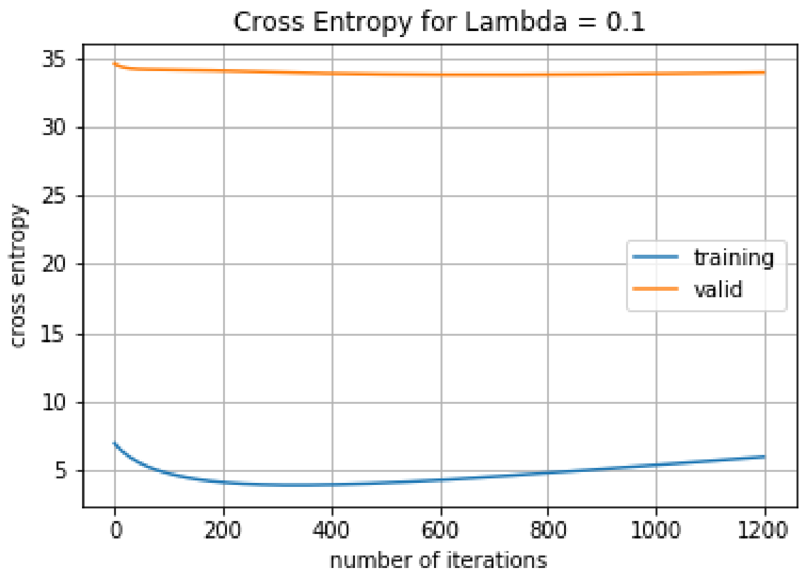




λ = 0.1:

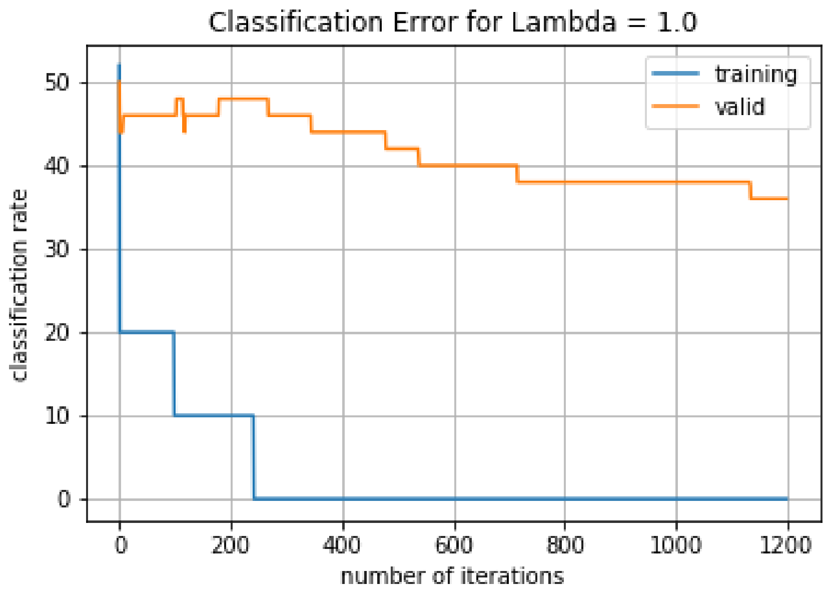
```
ITERATION:1200  TRAIN NLOGL:0.59  TRAIN CE:1.003723 TRAIN
FRAC:100.00  VALID CE:33.986043  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.59  TRAIN CE:1.003728 TRAIN
FRAC:100.00  VALID CE:33.986008  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.59  TRAIN CE:1.003721 TRAIN
FRAC:100.00  VALID CE:33.986050  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.59  TRAIN CE:1.003728 TRAIN
FRAC:100.00  VALID CE:33.986014  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:0.59  TRAIN CE:1.003733 TRAIN
FRAC:100.00  VALID CE:33.986098  VALID FRAC:64.00
```

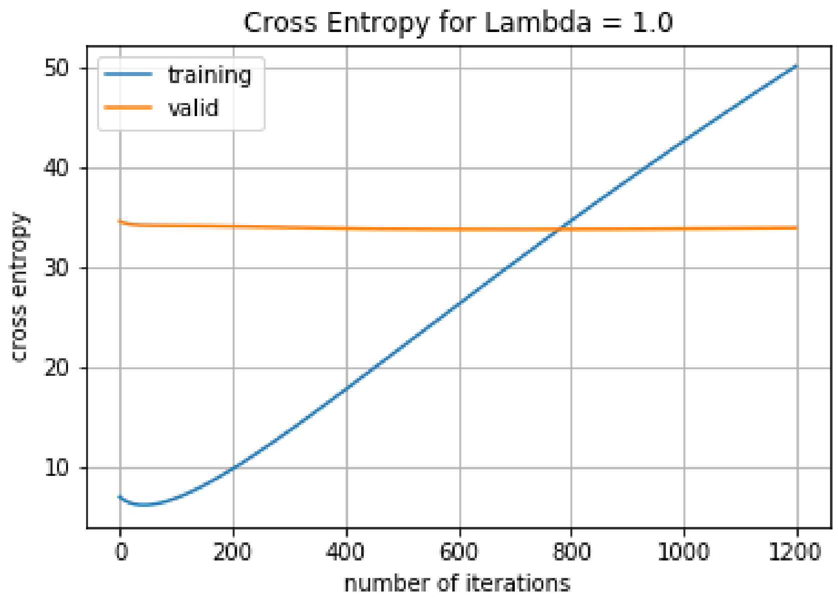




λ = 0.1:

```
ITERATION:1200  TRAIN NLOGL:5.02  TRAIN CE:1.003719 TRAIN
FRAC:100.00  VALID CE:33.985183  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:5.02  TRAIN CE:1.003712 TRAIN
FRAC:100.00  VALID CE:33.985317  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:5.02  TRAIN CE:1.003704 TRAIN
FRAC:100.00  VALID CE:33.985903  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:5.02  TRAIN CE:1.003710 TRAIN
FRAC:100.00  VALID CE:33.985325  VALID FRAC:64.00
ITERATION:1200  TRAIN NLOGL:5.02  TRAIN CE:1.003724 TRAIN
FRAC:100.00  VALID CE:33.986069  VALID FRAC:64.00
```





From the plots, we can observe that the higher the lambda, the faster the cross entropy reaches its minimum and then begin to increase due to over fitting.
For example, with lambda = 0, we can see that the cross entropy starts off as the minimum which can be sort of reflected in the classification rate graph where the first part of the graph is a long flat line, indicating a possible convergence has already happened.

On the other hand, the higher the lambda, the more fit the classification error is, for example, in the lambda = 0, we can see a flat top from 200 to 400 iterations. However, in the lambda = 1, the same flat top becomes a bump from 250 to 375 iterations, which it tries to fit more.

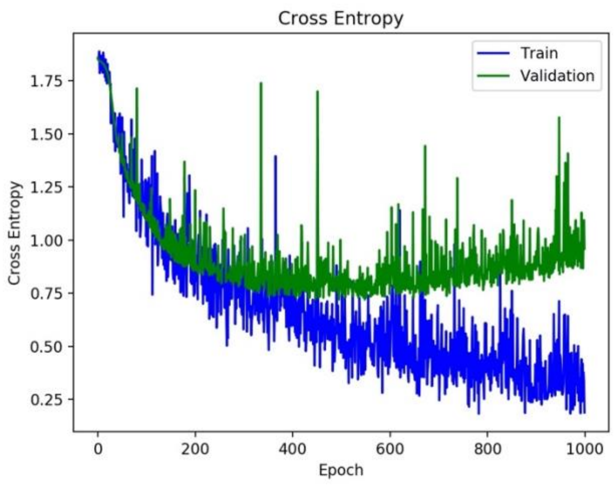
I would say the best lambda value would be 0.001, because it does not make the cross entropy go to convergence too fast yet it allows penalties to the weights which is what we want to achieve.

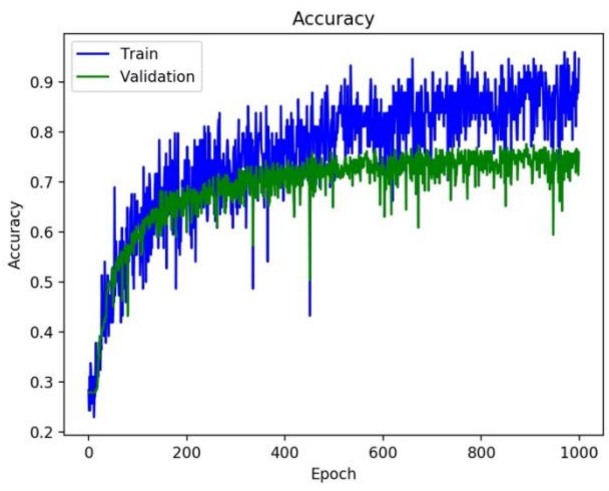
Compare to the results without penalty, mnist_train performs better with penalty while mnist_train_small performs worse with penalty. I think this is due to the size of the data set, adding penalty to a small data set is not ideal because it would just skew the data points. However if we add penalty to a large data set, it will be even out and achieve a better fitting.

```
def logistic_pen(weights, data, targets, hyperparameters):
    lambd =
hyperparameters['lambd']*hyperparameters['weight_regularization']
    regulator = lambd/2 * np.sum(weights[:-1]**2)
    regulatordw = lambd/2 * np.sum(2*weights[:-1])
    y = logistic_predict(weights, data)
    f , percent = evaluate(targets, y)
    f += (regulator*hyperparameters['weight_regularization'])
    dfdw = np.dot(data.T, y - targets) + regulatordw
    dfdb = [np.zeros(weights[-1].shape)]
    df = np.append(dfdw, dfdb, axis=0)
return f, df, y
```

```
lambdaRange = [0, 0.001, 0.01, 0.1, 1.0]
tList = np.arange(0,hyperparameters['num_iterations'],1)
numRerun = 5
for l in lambdaRange:
    hyperparameters['lambd'] = l
    train_CEList_avg = []
    valid_CEList_avg = []
    train_CRList_avg = []
    valid_CRList_avg = []
    for i in range(numRerun):
        weights = 0.00001 * np.random.randn(len(train_inputs[0])+1, 1)
        valid_CEList = []
        train_CEList = []
        valid_CRList = []
        train_CRList = []
        for t in range(hyperparameters['num_iterations']):
            f, df, predictions = logistic_pen(weights, train_inputs,
train_targets, hyperparameters)

            cross_entropy_train, frac_correct_train =
evaluate(train_targets, predictions)

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

            weights = weights - hyperparameters['learning_rate'] * df / N
            predictions_valid = logistic_predict(weights, valid_inputs)
            cross_entropy_valid, frac_correct_valid =
evaluate(valid_targets, predictions_valid)

            train_CRList.append((1-frac_correct_train)*100)
            valid_CRList.append((1-frac_correct_valid)*100)
            train_CEList.append(f)
            valid_CEList.append(cross_entropy_valid)
        print(("ITERATION:{:4d}  TRAIN NLOGL:{:4.2f}  TRAIN CE:{:.6f} "
                   "TRAIN FRAC:{:2.2f}  VALID CE:{:.6f}  VALID
FRAC:{:2.2f}").format(t+1, f / N, cross_entropy_train,
frac_correct_train*100, cross_entropy_valid, frac_correct_valid*100))
        valid_CEList_avg.append(valid_CEList)
        train_CEList_avg.append(train_CEList)
        valid_CRList_avg.append(valid_CRList)
        train_CRList_avg.append(train_CRList)
    train_CEList_avg = np.mean(train_CEList_avg, axis=0)
    valid_CEList_avg = np.mean(valid_CEList_avg, axis=0)
    train_CRList_avg = np.mean(train_CRList_avg, axis=0)
    valid_CRList_avg = np.mean(valid_CRList_avg, axis=0)
    fig, graph = plt.subplots()
    graph.plot(tList, train_CEList_avg, label="training")
    graph.plot(tList, valid_CEList_avg, label="valid")
    graph.set(xlabel='number of iterations', ylabel='cross entropy',
           title="Cross Entropy for Lambda = {}".format(l))
    graph.grid()
    graph.legend()
    fig.savefig("q2_3_CE_l={}.png".format(l))
    plt.show()
    fig, graph = plt.subplots()
    graph.plot(tList, train_CRList_avg, label="training")
    graph.plot(tList, valid_CRList_avg, label="valid")
    graph.set(xlabel='number of iterations', ylabel='classification rate',
           title="Classification Error for Lambda = {}".format(l))
    graph.grid()
    graph.legend()
    fig.savefig("q2_3_CR_l={}.png".format(l))
    plt.show()
```

3.1





From the plots, we can see that for both data sets, their cross entropy and accuracy both have a logarithmic trend. With cross entropy, the training set decreases as epoch increases, this is because during each epoch, when we calculate the back propagation, it will decrease the cross entropy. However it reaches a lower bound of around 0.2 and stay relatively constant. On the other hand, the validation cross entropy reached a minimum and then slightly increases when epoch increases, this may be due to possible over fitting. Since we can see from the accuracy graph, around 500 epochs, the accuracy for validation stopped increasing. However, we can also see that the accuracy continue to increase as epoch increases, because we are adjusting the weight during back propagation which will have an upper bound of 1.0.
Code Snippet:

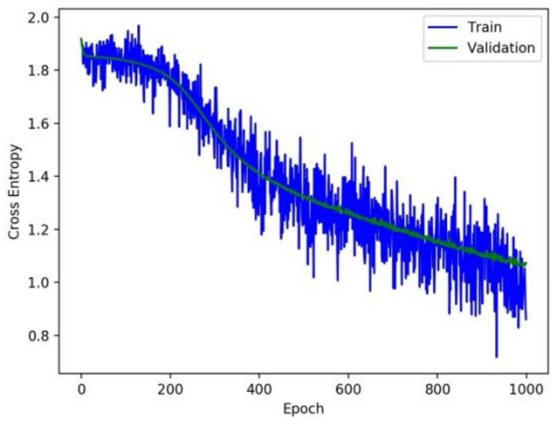
```
def AffineBackward(grad_y, h, w):
    grad_h = grad_y @ w.T
    grad_w = h.T @ grad_y
    grad_b = np.sum(grad_y, axis=0)
    return grad_h, grad_w, grad_b
```

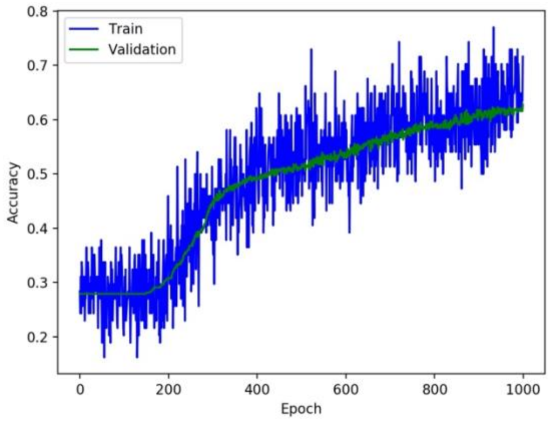
```
def ReLUBackward(grad_h, z):
    grad_z = grad_h * np.vectorize(lambda ReLU: 1 if ReLU > 0 else 0)(z)
    return grad_z
```

```
def NNUpdate(model, eps, momentum):
    model['V_W1'] = momentum * model['V_W1'] + (1 - momentum) *
model['dE_dW1']
    model['V_W2'] = momentum * model['V_W2'] + (1 - momentum) *
model['dE_dW2']
    model['V_W3'] = momentum * model['V_W3'] + (1 - momentum) *
model['dE_dW3']
    model['V_b1'] = momentum * model['V_b1'] + (1 - momentum) *
model['dE_db1']
    model['V_b2'] = momentum * model['V_b2'] + (1 - momentum) *
model['dE_db2']
    model['V_b3'] = momentum * model['V_b3'] + (1 - momentum) *
model['dE_db3']
    model['W1'] = model['W1'] - eps * model['V_W1']
    model['W2'] = model['W2'] - eps * model['V_W2']
    model['W3'] = model['W3'] - eps * model['V_W3']
    model['b1'] = model['b1'] - eps * model['V_b1']
    model['b2'] = model['b2'] - eps * model['V_b2']
    model['b3'] = model['b3'] - eps * model['V_b3']
    pass
```

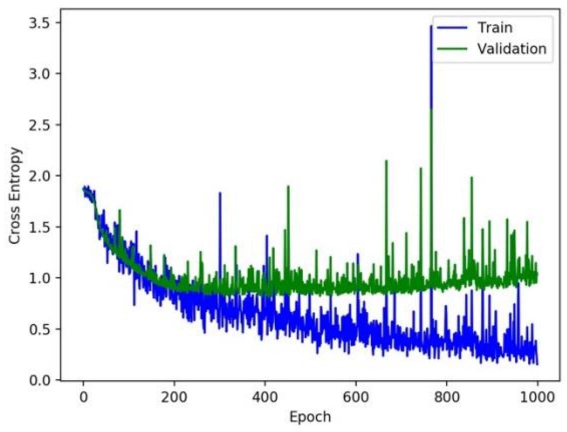
3.2

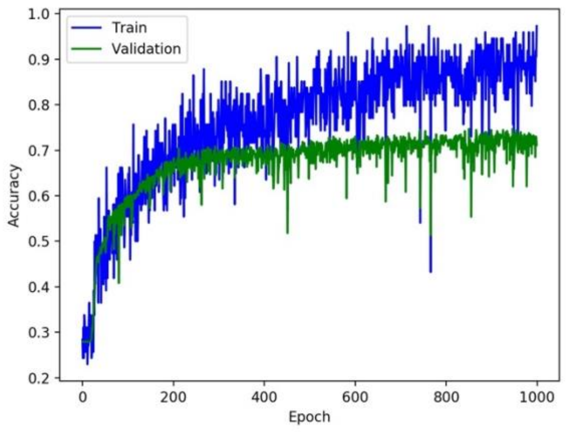
For ε = 0.001



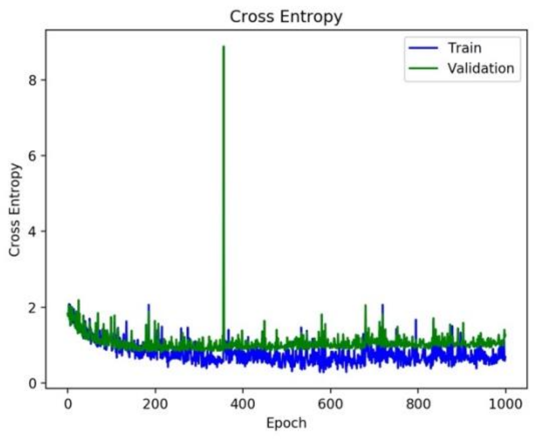


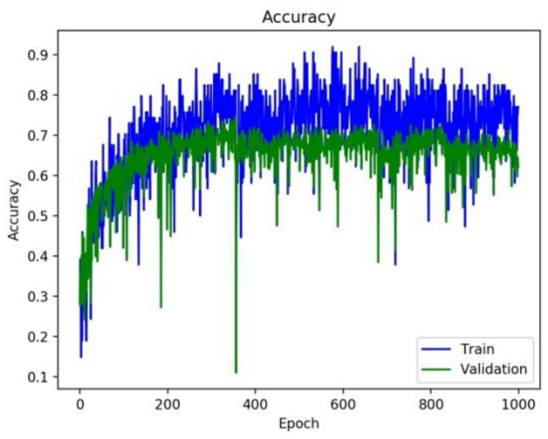
For ε = 0.01



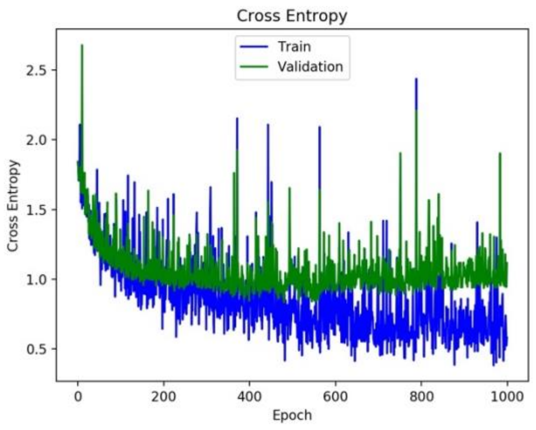


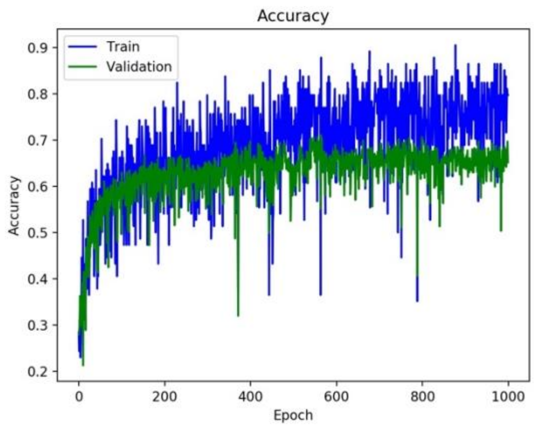
For ε = 0.1



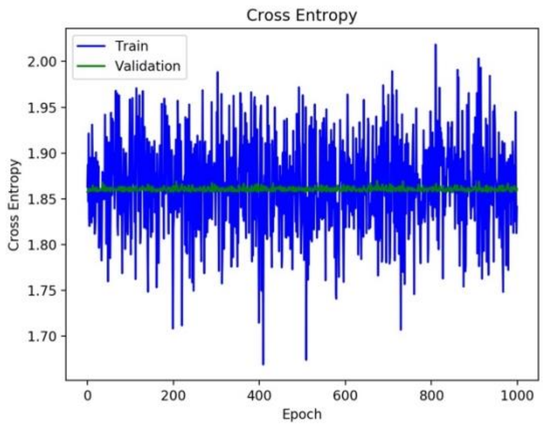


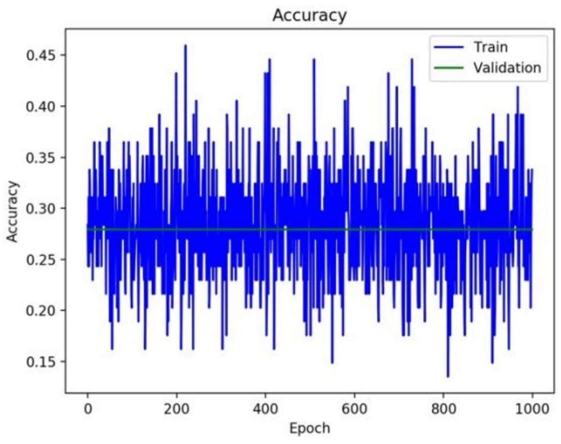
For ε = 0.5





For ε = 1.0



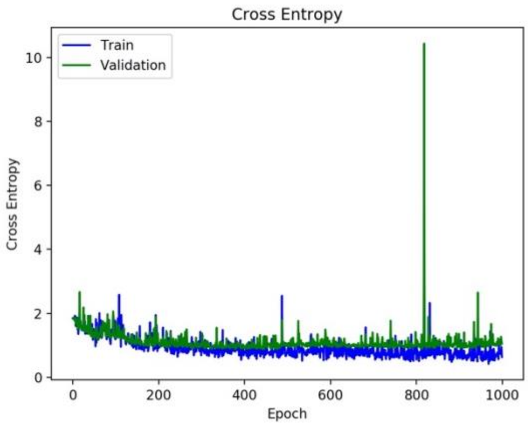


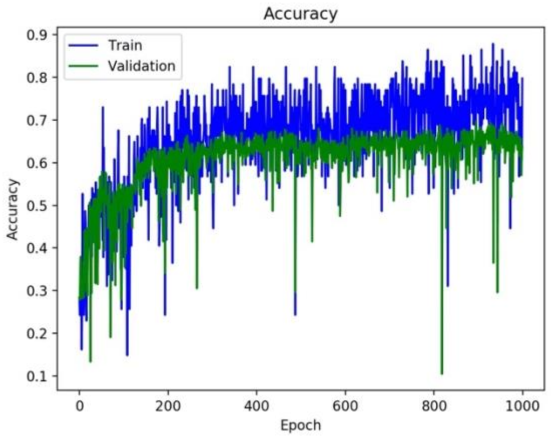
For validation set

ε =	0.001	0.01	0.1	0.5	1.0
Accuracy	0.61337	0.68492	0.62712	0.67239	0.27942
Cross Entropy	1.12031	1.05727	1.2393	1.09235	1.85782

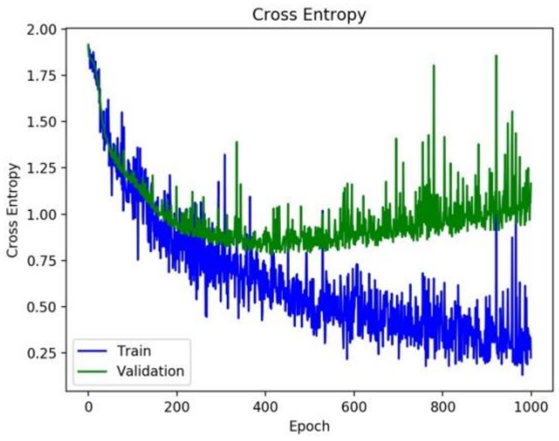
For this experiment batch size is fixed at 100, momentum is 0.0.
From the graphs, we can see that the higher the learning rate is, the faster the cross entropy and accuracy reaches their lower and upper bound. For example, when we set learning rate to 1, it immediately reaches the upper and lower bound, therefore the validation on both graph is a horizontal line since its already at the minimum in cross entropy and maximum in accuracy.
Also, it is worth noting that at very small and large learning rate, the validation on both graphs seem to have less oscillations, which indicates a lower variance.

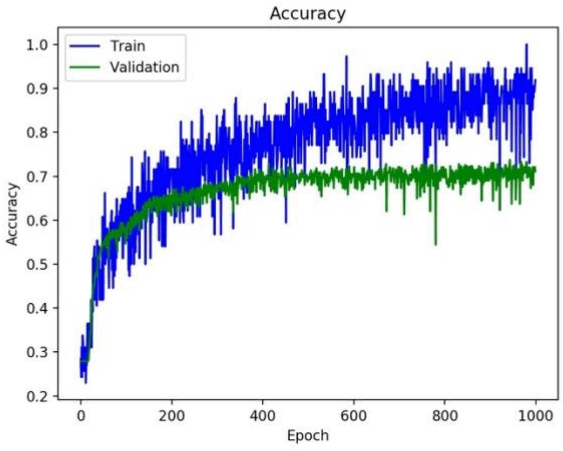
For momentum = 0.0



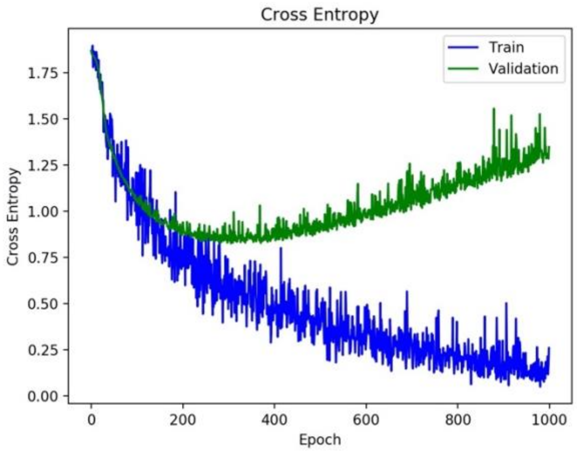


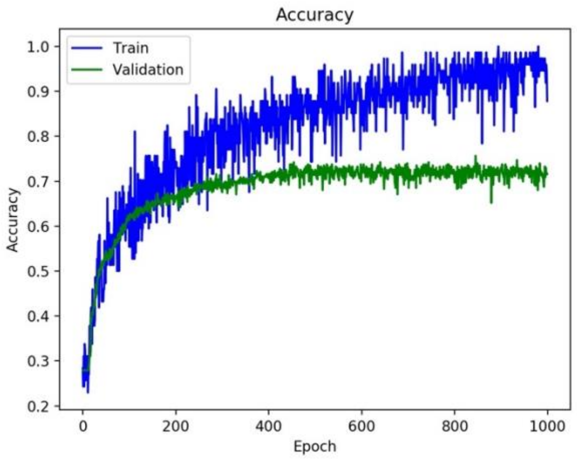
For momentum = 0.5





For momentum = 0.9





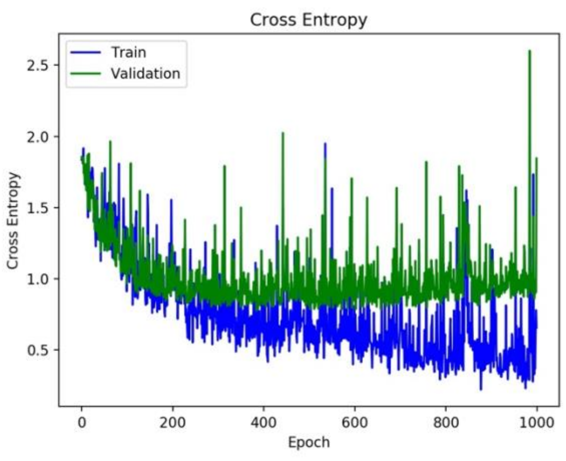
For validation set:

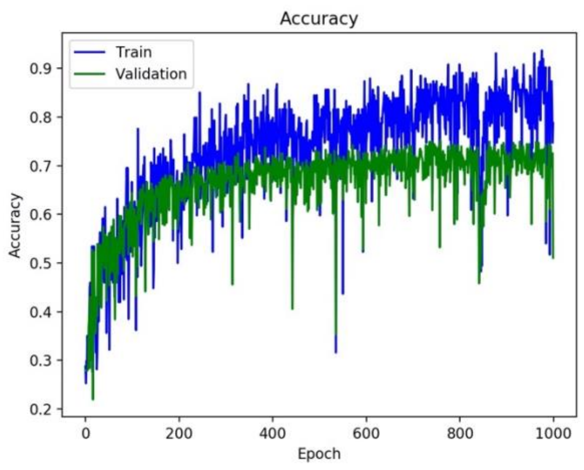
Momentum =	0.0	0.5	0.9
Accuracy		0.71122	0.69812
Cross Entropy		1.16566	1.24982

For this experiment batch size is fixed at 100, learning rate is 0.01.

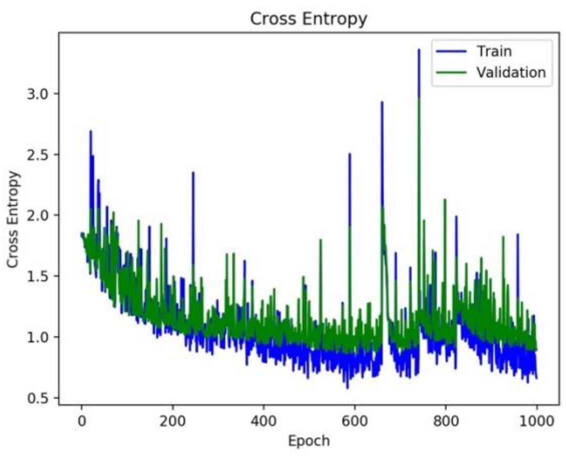
From the plots, we can observe that the higher the momentum, the higher the convergence rate. For example with momentum=0.9, the cross entropy for validation reaches the minimum at around 250 epochs and then continue to increase, which is reflecting that accuracy reaches the upper bound at around 250.
We can also observe that the higher the momentum, the lower the variance in vertical direction. For example, we can see that in higher momentum, the vertical oscillation of the curves are lower. This is due to higher momentum average out positive and negative numbers, so the average will be close to zero.

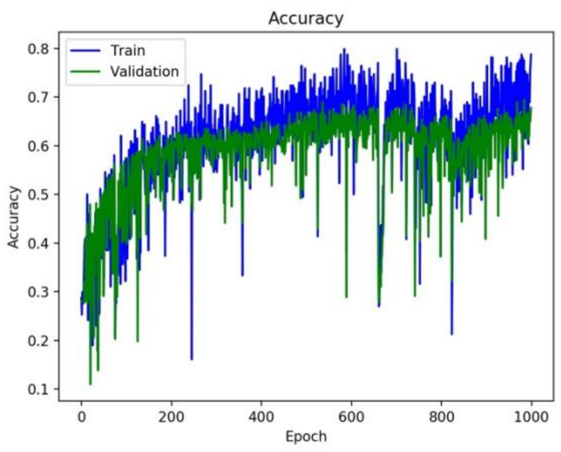
For batch size = 200:



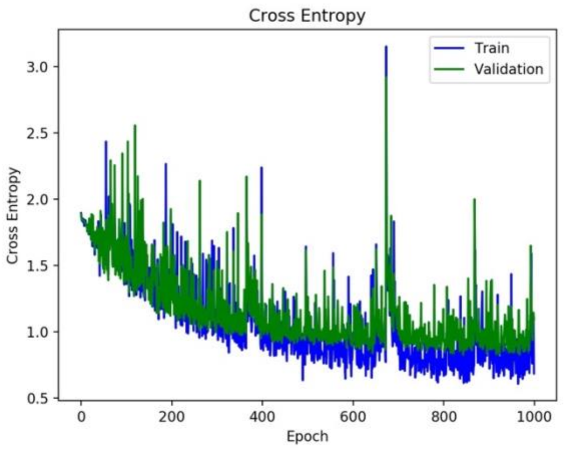


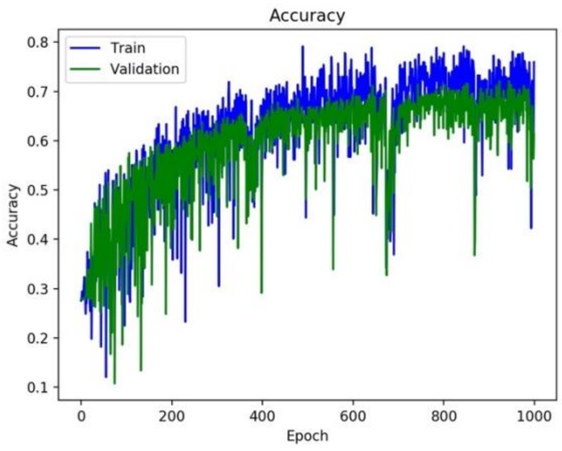
For batch size = 400:



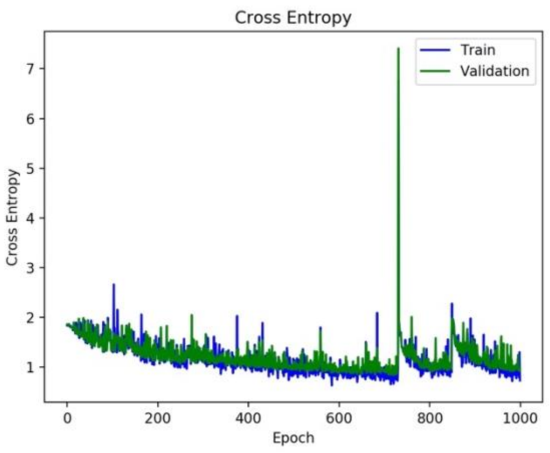


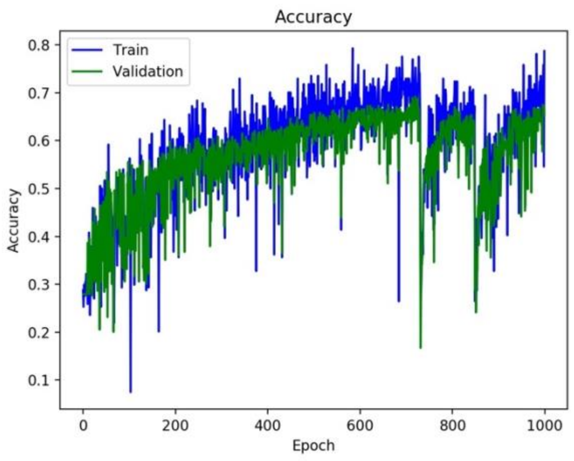
For batch size = 600:



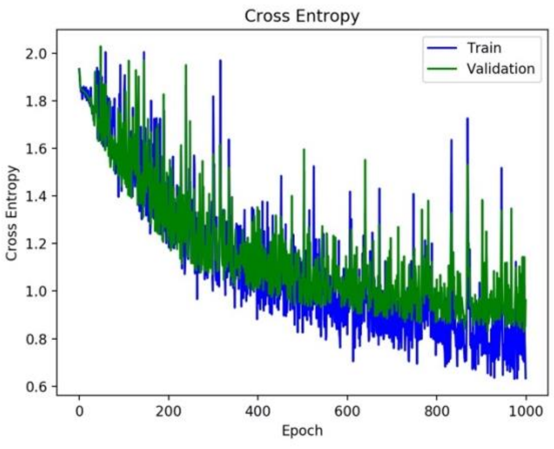


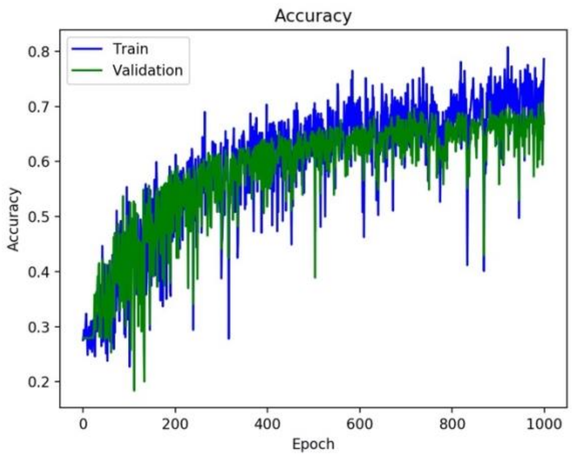
For batch size = 800:





For batch size = 1000:





For validation set

Batch size =	200	400	600	800	1000
Accuracy	0.69081	0.64823	0.62391	0.65283	0.66813
Cross Entropy	0.98235	1.13329	0.99214	1.12417	0.97134

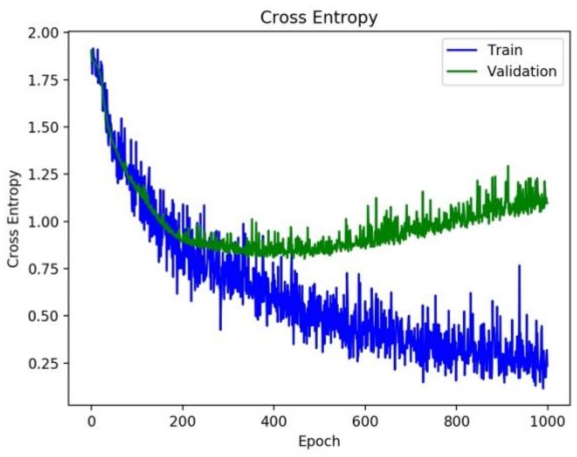
For this experiment the momentum is fixed at 0.0, learning rate is 0.01.

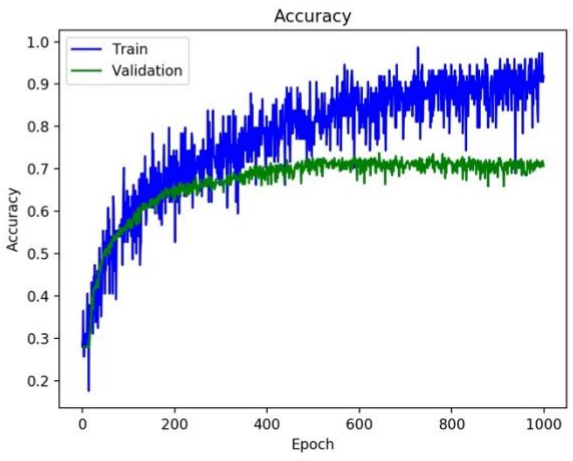
Following from the lectures, we know that if batch size is too large, more computations and memory are needed. On the other hand, if the batch size is too small, then there should be higher variance. Looking at the plots, we can kind of observe said phenomenon. For example, with bath size = 1000, the validation accuracy almost have identical shape as the training set. While for batch size = 200, we can see that they do not share the same shape and have differences in the accuracy and cross entropy.

So in conclusion, depending on the size of input, we can pick about 5% as the batch size and start with a low momentum and learning rate. If the cross entropy and accuracy does not converge, then slowly increase it and keep an eye on the final accuracy and cross entropy and try to minimize these two.

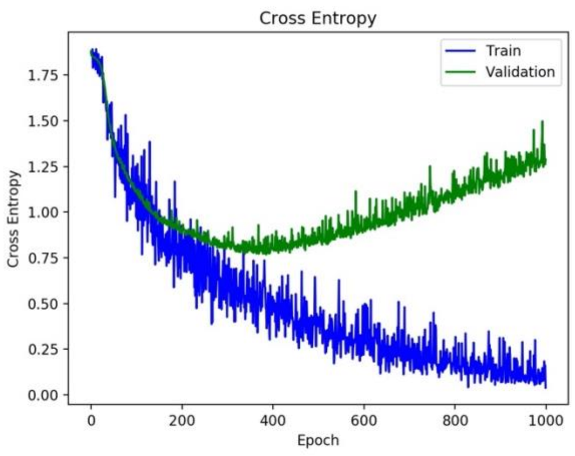
3.3

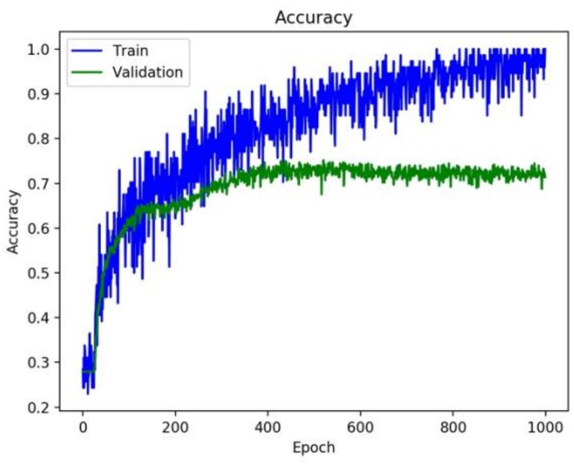
For hidden layers = [8,48]



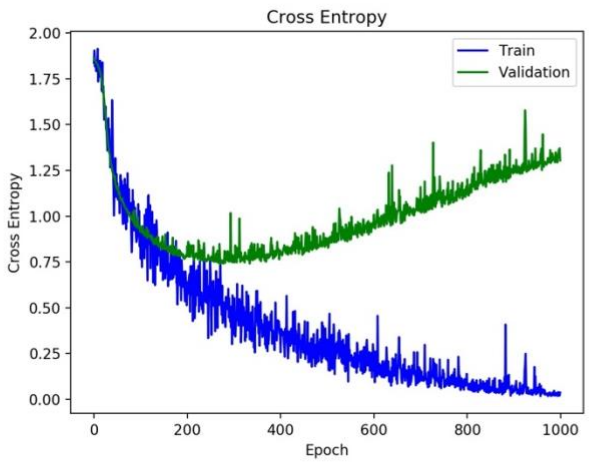


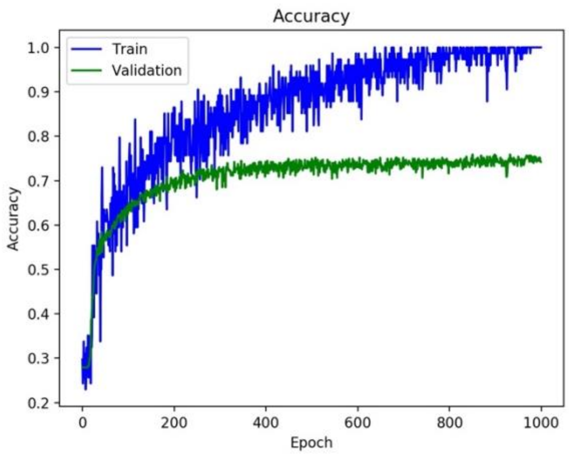
For hidden layers = [16,48]



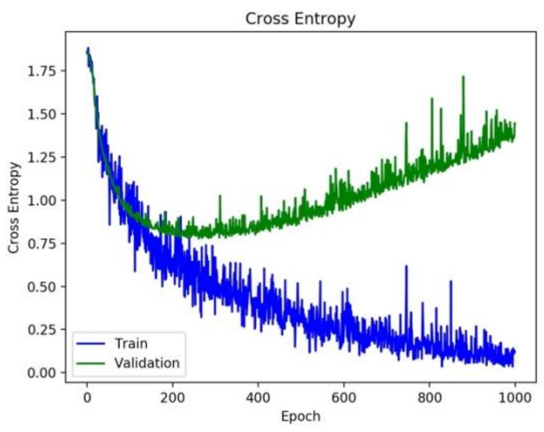


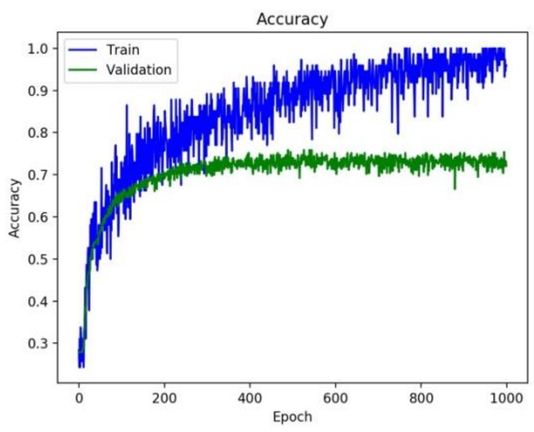
For hidden layers = [32, 48]



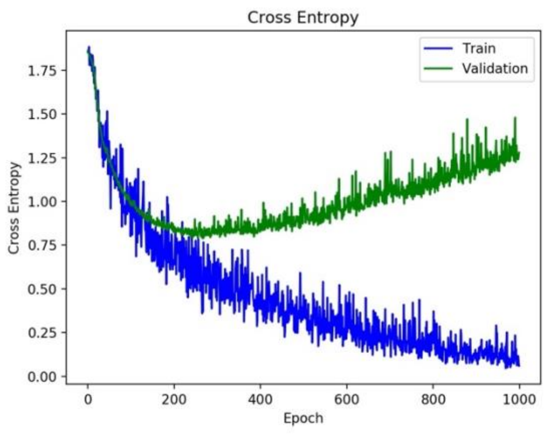


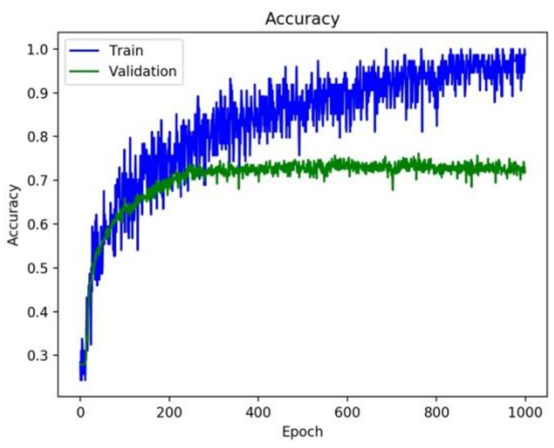
For hidden layers = [16,100]



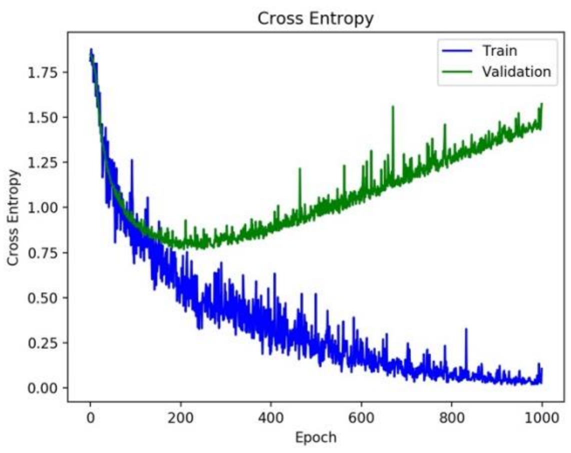


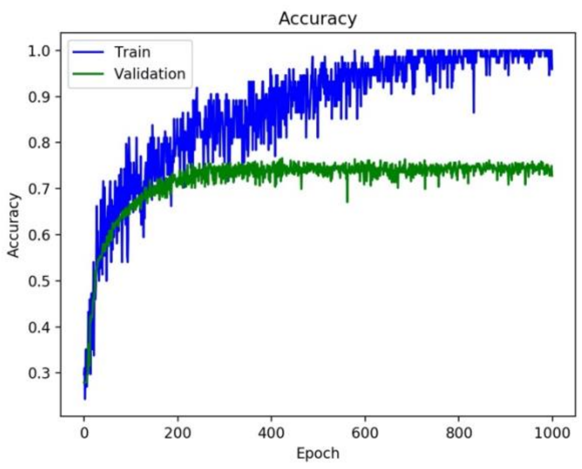
For hidden layers = [16, 72]





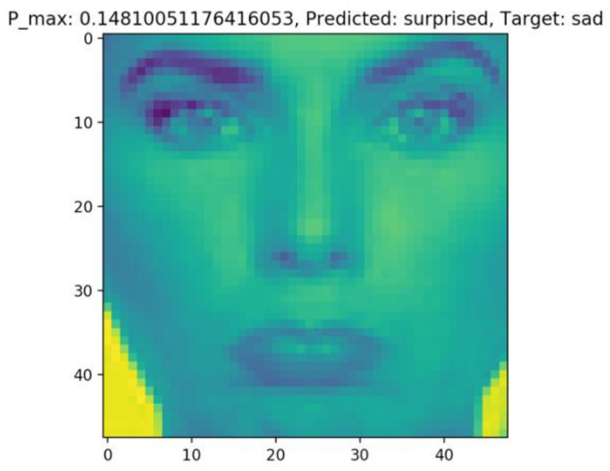
For hidden layers = [48,72]

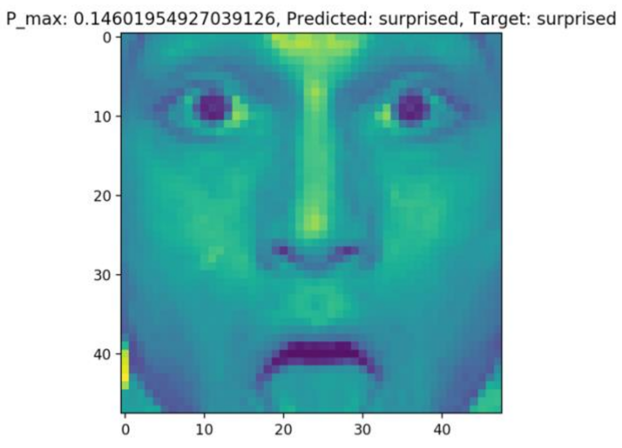


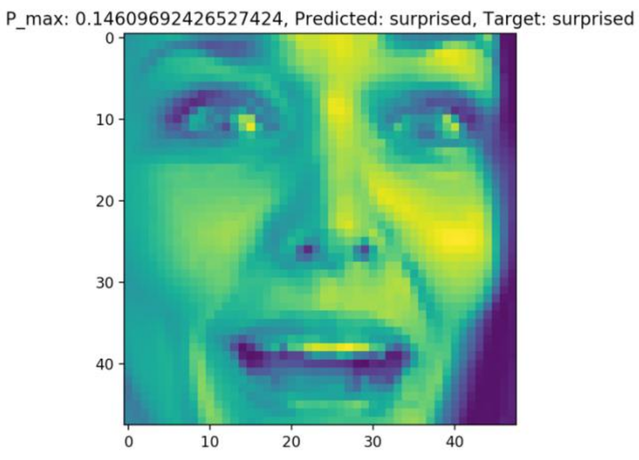


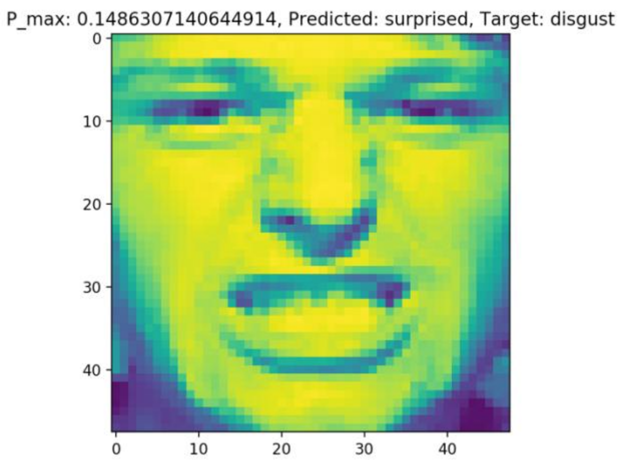
In theory, we know that the wider the hidden layers are, the better they are at memorization, however they would be worse in generalization. We can see this effect with [48,72], the accuracy of training set hit 100% at around 700 epochs yet the validation accuracy is still constant since it cannot generalize to the validation data set.
Therefore, we should pick size that's not too big but not too small because it will over generalize and result in low accuracy. A good starting point is the size of output, we can try take the multiple of the output size, for example maybe [2*7, 4*7].

3.4


P_max: 0.14810051176416053, Predicted: surprised, Target: sad


P_max: 0.14601954927039126, Predicted: surprised, Target: surprised


P_max: 0.14609692426527424, Predicted: surprised, Target: surprised


P_max: 0.1486307140644914, Predicted: surprised, Target: disgust

So from the above pictures, I picked out some samples where the neural network is not confident with the P_max printed on top which all these top score is lower than 0.5, which is the threshold I picked.
Noticed that two samples here still correctly predicted the target yet it is not so confident with a probability of only 14%, which was a surprise for me since the lower left one seems to be a more like a disgusted face to me as a human.
However on the other hand, it is also predicting surprised for the other 2 samples. Note that these faces are quite ambiguous for me, i.e. these faces do not look like sad and disgust to me, therefore it is normal for the neural network to output such a low probability.
Also, I noticed that most of these errors come from the neural network predicting surprised, which I assume it is because the face of surprised is a mix of other faces, such as a mix of happy and anger.
Therefore, the classifiers will not be very correct if it outputs these top scoring class. Since the neural network still has quite an amount of uncertainty that can be improved.

Code taken from piazza which I used (Bret)

```
def plot_uncertain_images(x, t, prediction, threshold=0.5):
    """
    """
    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
```