HOMEWORK2 - CSC311 Fall 2019

Nicole Xin Yue Wang

UtorID: wangnic8, #1004235339

Question 1

1.1 By Bayes' rule, we have:

$$p(t=1|x) = \frac{p(x|t=1)\,p(t=1)}{p(x)}$$

$\because x_i|t \sim \mathcal{N}(\mu_{it}, \sigma_i^2)$

$$\therefore p(x_i|t=1) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left\{-\frac{1}{2\sigma_i^2}(x_i-\mu_{i1})^2\right\}$$

$$p(x_i|t=0) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left\{-\frac{1}{2\sigma_i^2}(x_i-\mu_{i0})^2\right\}$$

since $t \in \{0, 1\}$, then

$$p(x) = p(x|t=1)\,p(t=1) + p(x|t=0)\,p(t=0)$$

$$\Rightarrow p(t=1|x) = \frac{(\alpha)\prod_{i=1}^{D}\frac{1}{\sqrt{2\pi\sigma_i^2}}\exp\left\{-\frac{1}{2\sigma_i^2}(x_i-\mu_{i1})^2\right\}}{(\alpha)\prod_{i=1}^{D}\frac{1}{\sqrt{2\pi\sigma_i^2}}\exp\left\{-\frac{1}{2\sigma_i^2}(x_i-\mu_{i1})^2\right\} + (1-\alpha)\prod_{i=1}^{D}\frac{1}{\sqrt{2\pi\sigma_i^2}}\exp\left\{-\frac{1}{2\sigma_i^2}(x_i-\mu_{i0})^2\right\}}$$

$$= \left\{1 + \frac{1-\alpha}{\alpha} \cdot \frac{\prod_{i=1}^{D}\frac{1}{\sqrt{2\pi\sigma_i^2}}\exp\left\{-\frac{1}{2\sigma_i^2}(x_i-\mu_{i0})^2\right\}}{\prod_{i=1}^{D}\frac{1}{\sqrt{2\pi\sigma_i^2}}\exp\left\{-\frac{1}{2\sigma_i^2}(x_i-\mu_{i1})^2\right\}}\right\}^{-1}$$

$$= \left\{1 + \frac{1-\alpha}{\alpha} \cdot \prod_{i=1}^{D}\exp\left\{-\frac{1}{2\sigma_i^2}\left[(x_i-\mu_{i0})^2 - (x_i-\mu_{i1})^2\right]\right\}\right\}^{-1}$$

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

$$= \left\{1 + \exp\left\{\ln\left[\frac{1-\alpha}{\alpha}\prod_{i=1}^{D}\exp\left\{-\frac{1}{2\sigma_i^2}\left(\mu_{i0}^2 - \mu_{i1}^2 + (-2\mu_{i0}+2\mu_{i1})x_i\right)\right\}\right]\right\}\right\}^{-1}$$

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

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

Therefore, $p(t=1|x)$ takes the form of a logistic function:

$$\frac{1}{1+\exp\left(-\sum_{i=1}^{D} w_i x_i - b\right)}$$

where $w_i = \frac{-\mu_{i0}+\mu_{i1}}{\sigma_i^2}$ and $b = \sum_{i=1}^{D}\left(\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}\right) - \ln\frac{1-\alpha}{\alpha}$

1.2. $p(t^{(1)}, t^{(2)}, \dots, t^{(n)} | x^{(1)}, \dots, x^{(n)}, w, b)$

$$= \prod_{j=1}^{n} p(t^{(j)} | x^{(j)}, w, b)$$

$$= \prod_{j=1}^{n} p(t^{(j)}=1 | x^{(j)}, w, b)^{t^{(j)}} \; p(t^{(j)}=0 | x^{(j)}, w, b)^{1-t^{(j)}}$$

$$= \prod_{j=1}^{n} p(t^{(j)}=1 | x^{(j)}, w, b)^{t^{(j)}} \left[1 - p(t^{(j)}=1 | x^{(j)}, w, b)\right]^{1-t^{(j)}}$$

Bernoulli r.v.

we know that $p(t^{(j)}=1 | x^{(j)}, w, b) = \dfrac{1}{1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)}$

Let $z^{(j)} = \sum_{i=1}^{D} w_i x_i^{(j)} + b$, so $p(t^{(j)}=1 | x^{(j)}, w, b) = \dfrac{1}{1+\exp(-z)}$

$$\Rightarrow p(t^{(1)}, \dots, t^{(n)} | x^{(1)}, \dots, x^{(n)}, w, b) = \prod_{j=1}^{n} \left(\frac{1}{1+\exp(-z)}\right)^{t^{(j)}} \left(\frac{1+\exp(-z)-1}{1+\exp(-z)}\right)^{1-t^{(j)}}$$

$$= \prod_{j=1}^{n} [1+\exp(-z)]^{-t^{(j)}} \left[\frac{\exp(-z)}{1+\exp(-z)}\right]^{1-t^{(j)}}$$

$$\Rightarrow L(w,b) = -\log p(t^{(1)}, \dots, t^{(n)} | x^{(1)}, \dots, x^{(n)}, w, b)$$

$$= \sum_{j=1}^{n} \left\{ + t^{(j)} \log[1+\exp(-z^{(j)})] + (1-t^{(j)}) \left\{ + z^{(j)} + \log[1+\exp(-z^{(j)})] \right\} \right\}$$

By chain rule: $\dfrac{\partial L}{\partial w} = \dfrac{\partial L}{\partial z^{(j)}} \cdot \dfrac{\partial z^{(j)}}{\partial w}$ $\qquad \dfrac{\partial L}{\partial b} = \dfrac{\partial L}{\partial z^{(j)}} \cdot \dfrac{\partial z^{(j)}}{\partial b}$

$$\because z^{(j)} = \sum_{i=1}^{D} w_i x_i^{(j)} + b$$

$$\therefore \frac{\partial z^{(j)}}{\partial w} = \sum_{i=1}^{D} x_i^{(j)} \qquad \frac{\partial z^{(j)}}{\partial b} = 1$$

$$\frac{\partial L}{\partial z^{(j)}} = \sum_{j=1}^{n} t^{(j)} \frac{-\exp(-z^{(j)})}{1+\exp(-z^{(j)})} + (1-t^{(j)}) \left[1 + \frac{-\exp(-z^{(j)})}{1+\exp(-z^{(j)})}\right]$$

$$= \sum_{j=1}^{n} -t^{(j)} \frac{\exp(-z^{(j)})}{1+\exp(-z^{(j)})} + 1 - \frac{\exp(-z^{(j)})}{1+\exp(-z^{(j)})} - t^{(j)} + t^{(j)} \frac{\exp(-z^{(j)})}{1+\exp(-z^{(j)})}$$

$$= \sum_{j=1}^{n} 1 - \frac{\exp(-z^{(j)})}{1+\exp(-z^{(j)})} - t^{(j)}$$

$$\therefore \frac{\partial L}{\partial w} = \sum_{j=1}^{n} \left[1 - \frac{\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)}{1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)} - t^{(j)}\right] \cdot \sum_{i=1}^{D} x_i^{(j)}$$

$$\frac{\partial L}{\partial b} = \sum_{j=1}^{n} \left[1 - \frac{\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)}{1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)} - t^{(j)}\right]$$

$$L(w,b) = \sum_{j=1}^{n} \left\{ + t^{(j)} \log\left[1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)\right] + (1-t^{(j)}) \right.$$

$$\left. * \left\{ \sum_{i=1}^{D} w_i x_i^{(j)} - b + \log\left[1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)\right] \right\} \right\}$$

1.3. $\because p(w_i) \sim \mathcal{N}(w_i|0, 1/\lambda)$

$\therefore p(w) = \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi/\lambda}} \exp\left[-\frac{\lambda}{2}(w_i)^2\right]$

$\Rightarrow p(w,b|t^{(1)},\dots,t^{(n)}) \propto p(w)\, p(b)\, p(t^{(1)},\dots,t^{(n)}|w,b)$

$= \left\{\prod_{i=1}^{D} \frac{1}{\sqrt{2\pi/\lambda}} \exp\left[-\frac{\lambda}{2}(w_i)^2\right]\right\} \cdot (1) \cdot p(t^{(1)},\dots,t^{(n)}|w,b)$

$\Rightarrow L_{post}(w,b) = -\log\left[p(w,b|t^{(1)},\dots,t^{(n)})\right]$

$= -\log \prod_{i=1}^{D} \frac{1}{\sqrt{2\pi/\lambda}} \exp\left(-\frac{\lambda}{2} w_i^2\right) - \log p(t^{(1)},\dots,t^{(n)}|w,b)$

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

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

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

$\therefore \frac{\partial L_{post}}{\partial w} = \frac{\partial L}{\partial w} + \frac{\partial}{\partial w}\left(\frac{\lambda}{2}\sum_{i=1}^{D} w_i^2\right)$

$= \sum_{j=1}^{n}\left\{\left[1 - \frac{\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)}{1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)} - t^{(j)}\right] \cdot \sum_{i=1}^{D} x_i^{(j)}\right\} + \lambda \sum_{i=1}^{D} w_i$

$\frac{\partial L_{post}}{\partial b} = \frac{\partial L}{\partial b} = \sum_{j=1}^{n}\left[1 - \frac{\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)}{1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)} - t^{(j)}\right]$

It is shown that $L_{post}(w,b) = L(w,b) + \frac{\lambda}{2}\sum_{i=1}^{D} w_i^2 + C$ where $C = -\frac{D}{2}\log\left(\frac{\lambda}{2\pi}\right)$.

Question 2

2.1. Script that runs kNN for values of *k* and plots the classification rate on the validation set.

```
if __name__ == '__main__':
        """ Script that runs kNN for different values of k and plots the clasification rate on the
validation set. """
            train_input, train_target = load_train()
            valid_input, valid_target = load_valid()

            for k in [1,3,5,7,9]:
               pred_val = run_knn(k, train_input, train_target, valid_input)
               val_rate = np.mean(pred_val == valid_target)
               print "k =", k, ", classification rate on validation set =", val_rate, "
```

Output:

```
k = 1 , classification rate on validation set = 0.82
k = 3 , classification rate on validation set = 0.86
k = 5 , classification rate on validation set = 0.86
k = 7 , classification rate on validation set = 0.86
k = 9 , classification rate on validation set = 0.84
```

Predictions resulted in an averagely high classification rates while *k=3,5,7* shares the same highest classification rate which is 0.86. Since mnist_train contains only 80 examples in total, it could be very misleading if we choose some big values of *k*. Therefore, I would choose 5 as *k**, while both *k*+2* and *k*-2* would both have rate of 0.86.

For test performance, output would be:

```
k = 1 , classification rate on set set = 0.88
k = 3 , classification rate on set set = 0.92
k = 5 , classification rate on set set = 0.94
k = 7 , classification rate on set set = 0.94
k = 9 , classification rate on set set = 0.88
```

Although it follows a similar pattern, test performance with k=3 results in a smaller rate than with k=5,7, showing that test performance does not correspond to the validation performance.

2.2 logistic.py

```
def logistic_predict(weights, data):
    # TODO: Finish this function
    y = sigmoid(np.dot(data, weights[:-1]) + weights[-1])
    return y.reshape((len(y), 1))   # reshape to keep vector's shape consistent

def evaluate(targets, y):
    # TODO: Finish this function
    ce = np.dot(targets.T, -1 * np.log(y))
    frac_correct = np.mean((y > 0.5) == targets)
    return ce, frac_correct

def logistic(weights, data, targets, hyperparameters):
    N = len(targets)
    # TODO: Finish this function
    y = logistic_predict(weights, data)
    z = np.dot(data, weights[:-1]) + weights[-1]
    f = np.dot(targets.T, -1 * np.logaddexp(0, -1*z)) + np.dot((1-targets).T, np.logaddexp(0,z))
    # add one more column of ones for bias
    data_ca = np.concatenate((data, np.ones((N, 1))), axis=1)
    df = np.dot(data_ca.T, y-targets)
    return f, df, y
```

logistic_regression_template.py

```
def run_logistic_regression(l, w, i, p, penalty=False):
    #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': l,
                'weight_regularization': w,
                'num_iterations': i,
                'penalty': p
              }
    # Logistic regression weights
    # TODO:Initialize to random weights here.
    weights = np.random.normal(0, hyperparameters['weight_regularization'], M+1).reshape(M+1, 1)
```

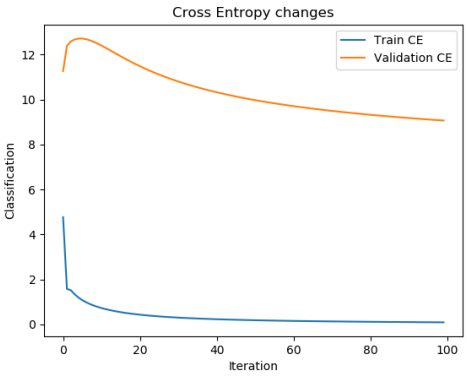
```
# Verify that your logistic function produces the right gradient.
# diff should be very close to 0.
#run_check_grad(hyperparameters)
ce_train_list = []
ce_val_list = []
# Begin learning with gradient descent
for t in xrange(hyperparameters['num_iterations']):

   # 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.
   if penalty:
      f, df, predictions = logistic_pen(weights, train_inputs, train_targets, hyperparameters)
   else:
      f, df, predictions = logistic(weights, train_inputs, train_targets, hyperparameters)

   # Evaluate the prediction.
   cross_entropy_train, frac_correct_train = evaluate(train_targets, predictions)
   ce_train_list.append(cross_entropy_train[0][0])

   if np.isnan(f) or np.isinf(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)

   # Evaluate the prediction.
   cross_entropy_valid, frac_correct_valid = evaluate(valid_targets, predictions_valid)

   #pred_test = logistic_predict(weights, test_inputs)
   #cross_entropy_test, frac_correct_test = evaluate(test_targets, pred_test)

   ce_val_list.append(cross_entropy_valid[0][0])

   #print some stats
   if t + 1 == hyperparameters['num_iterations']:
      print ("ITERATION:{:4d}  TRAIN NLOGL:{:4.2f}  TRAIN CE:{:.6f} "
        "TRAIN FRAC:{:2.2f}  VALID CE:{:.6f}  VALID FRAC:{:2.2f}").format(
          t+1, float(f / N), float(cross_entropy_train), float(frac_correct_train*100),
          float(cross_entropy_valid), float(frac_correct_valid*100))
      #print "TEST CE=", cross_entropy_test, "acc=", frac_correct_test
return ce_train_list, ce_val_list
```

When training with mnist_train_small, the best hyperparameter setting found is:

Learning rate: 0.1

Initialize weight: normal distribution with mean=0, standard deviation=0.1

Number of iterations: 100



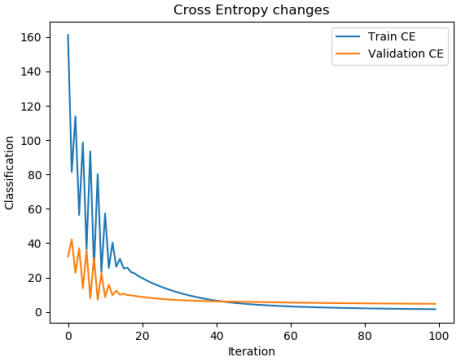
Results	Cross Entropy	Classification error
Train	0.096896	0.0
Validation	9.063846	28.0
Test	4.41671567	22.0

When training with mnist_train, the best hyperparameter setting found:

Learning rate: 1.0

Initialize weight: normal distribution with mean=0, standard deviation=1.0

Number of iterations: 100



Results	Cross Entropy	Classification error
Train	1.558408	0.62
Validation	4.706549	8.0
Test	1.22859344	10.0

When we run the code several times, the results of best set of hyperparameters changes. This is due to our initial assignments to the weights. The weights are normally random distributed. However, we could still find the best set of hyperparameters by limiting the standard deviation of the distribution to a smaller number so that difference between weights would we smaller, so that results of best set of hyper parameters would be similar every time we run the code. Then , we could choose the best parameter settings.

2.3 logistic_pen

```
def logistic_pen(weights, data, targets, hyperparameters):
    # TODO: Finish this function
    N = len(targets)
    reg = hyperparameters['penalty']
    lf, ldf, y = logistic(weights, data, targets, hyperparameters)

    f = lf + ((reg/2) * np.dot(weights.T, weights)) - (np.log(reg/(2*np.pi))*N/2)
    df = ldf + (reg * weights)
    return f, ldf, y
```

When training with **mnist_train_small**, the best hyperparameter setting found is:

Learning rate: 0.1

Initialize weight: normal distribution with mean=0, standard deviation=0.1

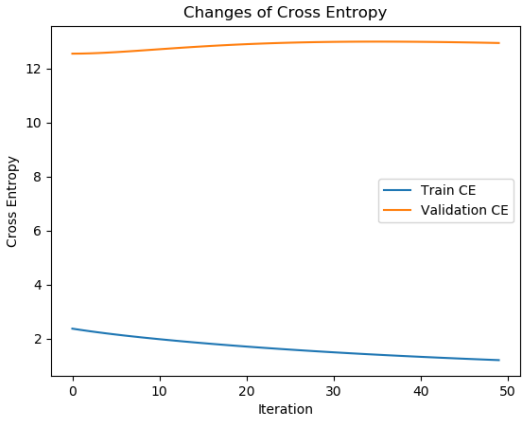
Number of iterations: 100

Penalty = 0 :

Wouldn't work since when calculating the sum of the loss over all data points, log(penalty) is needed which has no result.

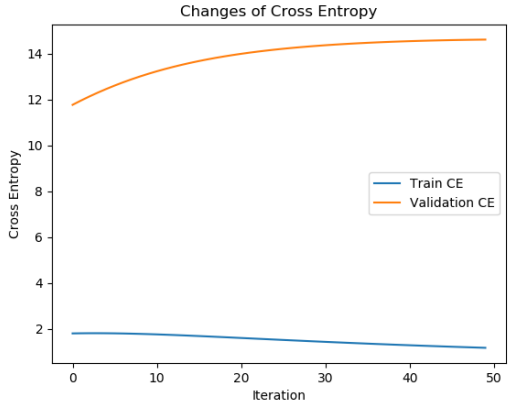
Penalty = 0.1

Results	Cross Entropy	Classification error
Train	1.213881	10.0
Validation	12.940203	40.0



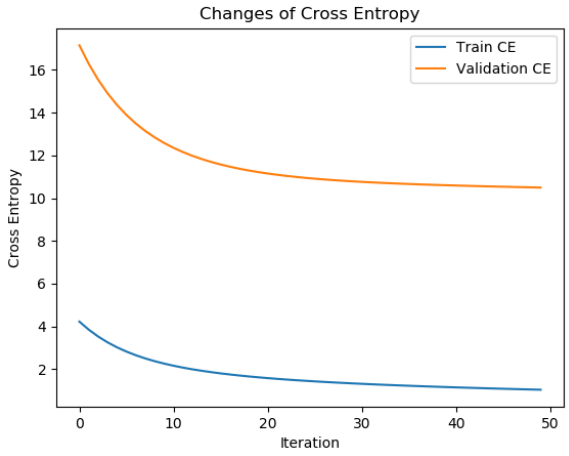
Penalty = 0.01

Results	Cross Entropy	Classification error
Train	1.178782	10.0
Validation	14.614325	48.0



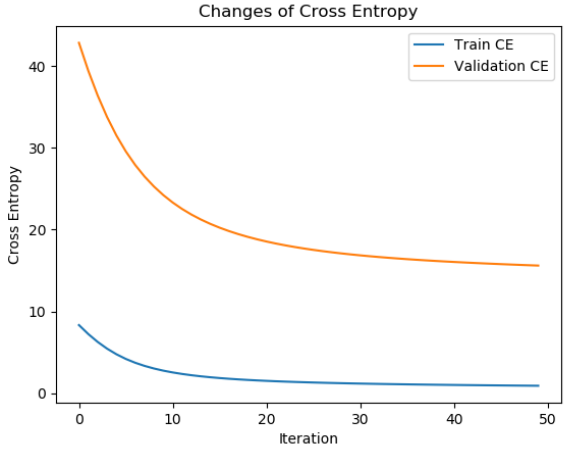
Penalty = 0.001

Results	Cross Entropy	Classification error
Train	1.039547	10.0
Validation	10.494749	46.0



Penalty = 1.0

Results	Cross Entropy	Classification error
Train	0.926470	0.0
Validation	15.608910	52.0



When training with **mnist_train**, the best hyperparameter setting found is:

Learning rate: 1.0

Initialize weight: normal distribution with mean=0, standard deviation=1.0

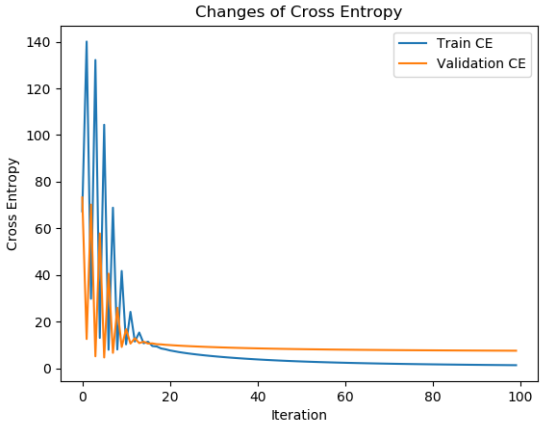
Number of iterations: 100

Penalty = 0 :

Wouldn't work since when calculating the sum of the loss over all data points, log(penalty) is needed which has no result.

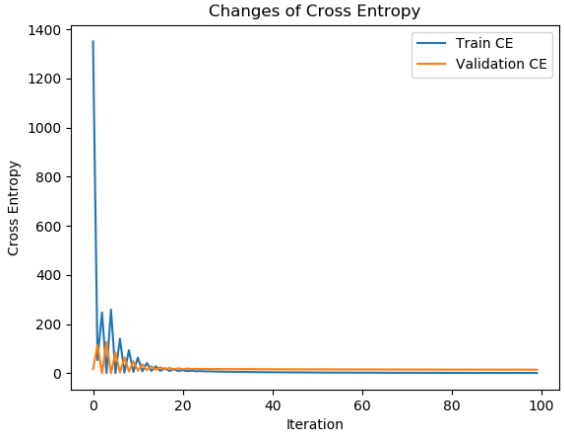
Penalty = 1.0

Results	Cross Entropy	Classification error
Train	1.356983	0.0
Validation	7.558003	12.0



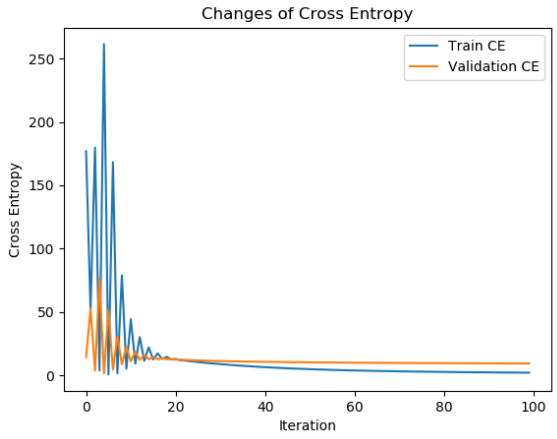
Penalty = 0.1

Results	Cross Entropy	Classification error
Train	1.511433	0.0
Validation	14.936302	10.0



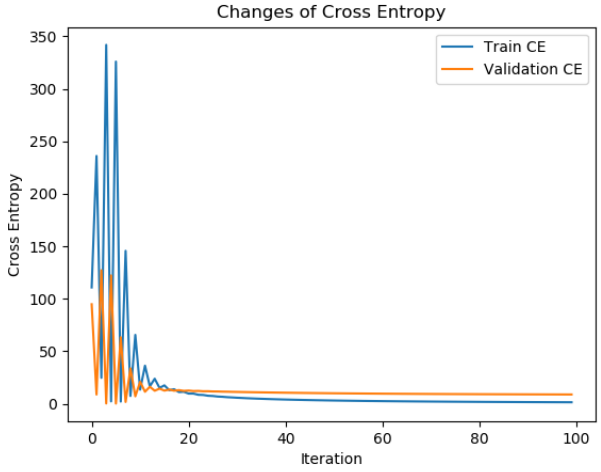
Penalty = 0.01

Results	Cross Entropy	Classification error
Train	2.049677	0.0
Validation	9.313509	8.0



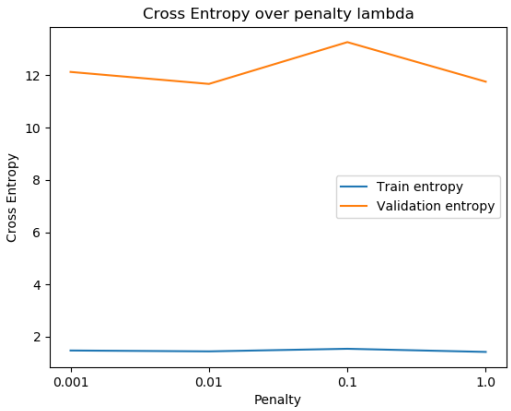
Penalty = 0.001

Results	Cross Entropy	Classification error
Train	1.496006	0.0
Validation	8.901392	14.0

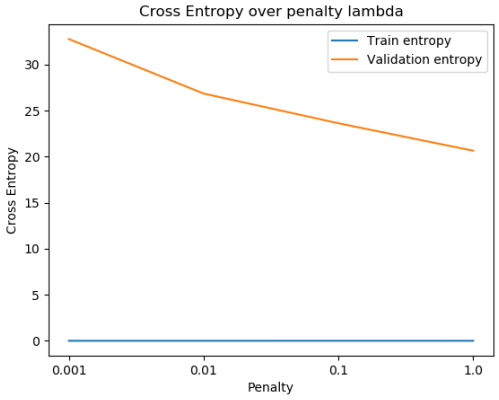


Cross Entropy changes as penalty increases

mnist_train:

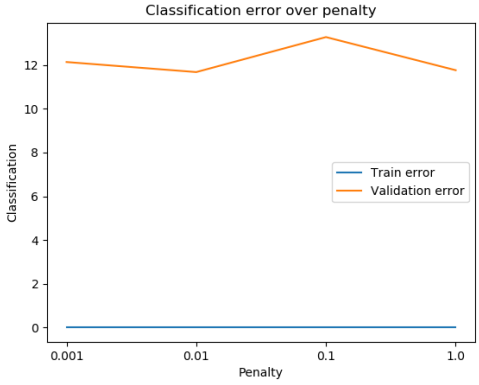


mnist_train_small:

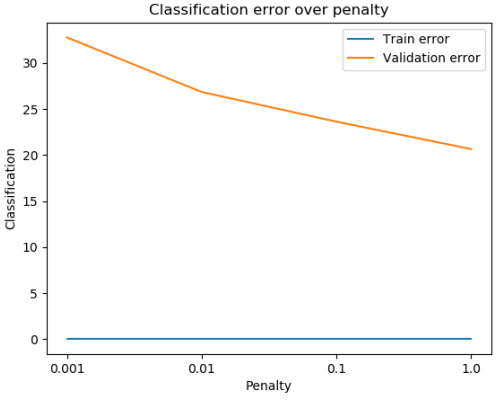


Classification error changes as penalty increases

mnist_train:



mnist_train_small



As the penalty increases, both the classification error and cross starts from above and gradually goes down. I believe it behaves this way because as the penalty increases, the weight is more regularized, which means its affect on the prediction decreases. Since we are only looking at penalties 0.001, 0.01, 0.1, 1.0, 1.0 is the largest among the all but still a very small number comparing to other numbers that may be too large causing too much penalty. Therefore, I would choose 1.0 to be the best λ.

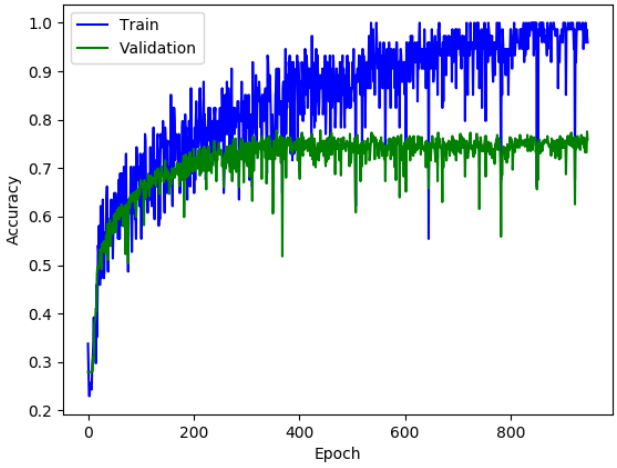
Question 3

Added codes:

```
def AffineBackward(grad_y, h, w):
    grad_h = np.dot(grad_y, w.T)
    grad_w = np.dot(h.T, grad_y)
    grad_b = np.dot(grad_y.T, np.ones(grad_y.shape[0]))
    return grad_h, grad_w, grad_b
```

```
def ReLUBackward(grad_h, z):
    grad_z = grad_h * (z > 0)
    return grad_z

def InitNN(num_inputs, num_hiddens, num_outputs):
    W1 = 0.1 * np.random.randn(num_inputs, num_hiddens[0])
    W2 = 0.1 * np.random.randn(num_hiddens[0], num_hiddens[1])
    W3 = 0.01 * np.random.randn(num_hiddens[1], num_outputs)
    V_W1 = np.zeros(W1.shape)
    V_W2 = np.zeros(W2.shape)
    V_W3 = np.zeros(W3.shape)
    b1 = np.zeros((num_hiddens[0]))
    b2 = np.zeros((num_hiddens[1]))
    b3 = np.zeros((num_outputs))
    V_b1 = np.zeros(b1.shape)
    V_b2 = np.zeros(b2.shape)
    V_b3 = np.zeros(b3.shape)
    model = {
        'W1': W1,
        'W2': W2,
        'W3': W3,
        'b1': b1,
        'b2': b2,
        'b3': b3,
        'V_W1': V_W1,
        'V_W2': V_W2,
        'V_W3': V_W3,
        'V_b1': V_b1,
        'V_b2': V_b2,
        'V_b3': V_b3
    }
    return model

    def NNUpdate(model, eps, momentum):
        # Update the velocities
        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']
        # Update the weights.
        model['W1'] = model['W1'] - eps * model['V_W1']
        model['W2'] = model['W2'] - eps * model['V_W2']
        model['W3'] = model['W3'] - eps * model['V_W3']
        # Update bias
        model['b1'] = model['b1'] - eps * model['V_b1']
        model['b2'] = model['b2'] - eps * model['V_b2']
        model['b3'] = model['b3'] - eps * model['V_b3']
```

3.1.



The network performs a higher accuracy learning in training set than the validation set, and the accuracy of the training set keeps on increasing while the validation set learns and gradually stops increasing its accuracy. This makes sense since the network learns form the training set directly, the more experience it has with the training set, the higher accuracy it has with predicting about the training set. While it learns from the validation set by predicting the values of the set and adjusting hyperparameters, which is probably going to be slower in learning. Moreover, as it learns more in more iterations, it may learn too much that it would cause overfitting. Since validation set is a 'brand new' set of data to training set, there would be a threshold were the network starts to preform worse on predicting the validation set.

3.2 Different settings of learning rate:

Learning rate	Other hyperparameters	Output
0.001	num_hiddens = [16,32] momentum = 0 num_epochs = 500 batch_size = 100	CE: Train 1.3458 Validation 1.33365 Test 1.34682 Acc: Train 0.5166 Validation 0.52029 Test 0.54286
0.01		CE: Train 0.56271 Validation 0.87353 Test 0.83644 Acc: Train 0.80083 Validation 0.71838 Test 0.7039
0.1		CE: Train 0.56147 Validation 0.88013 Test 0.93519 Acc: Train 0.79431 Validation 0.70883 Test 0.69351
0.5		CE: Train 1.86101 Validation 1.85833 Test 1.83837 Acc: Train 0.28542 Validation 0.27924 Test 0.31688
1.0		CE: Train 1.86222 Validation 1.85883 Test 1.83870 Acc: Train 0.28542 Validation 0.27924 Test 0.31688

The cross entropy decreases and accuracy increases when learning rate increases from 0.001 to 0.01, but then it starts increasing cross entropy and decreasing accuracy as learning rate keeps on increasing. When increasing learning rate from 0.5 to 1.0, convergence properties seems to have no big difference. Therefore, 0.01 is the threshold where larger learning rate would be learning too fast, and smaller learning rate would be learning too slow. In this case, I would choose 0.01 to be the learning rate.

Different momentum

Momemtum	Other hyperparameters	Output
0.0	num_hiddens = [16,32] learning_rate = 0.01 num_epochs = 500 batch_size = 100	CE: Train 0.56147 Validation 0.88013 Test 0.93519 Acc: Train 0.79431 Validation 0.70883 Test 0.69351
0.4		CE: Train 0.96963 Validation 1.19847 Test 1.14838 Acc: Train 0.64286 Validation 0.57757 Test 0.56623
0.9		CE: Train 0.17057 Validation 1.581 Test 1.4999 Acc: Train 0.92916 Validation 0.72076 Test 0.71169

Momentum being 0.9 has the best performance.

Different mini-batch sizes

Batch_size	Other hyperparameters	Output
100	num_hiddens = [16,32] learning_rate = 0.01 momentum = 0 num_epochs = 500	CE: Train 0.17057 Validation 1.581 Test 1.4999 Acc: Train 0.92916 Validation 0.72076 Test 0.71169
300		CE: Train 0.41774 Validation 0.9318 Test 0.98559 Acc: Train 0.83373 Validation 0.7136 Test 0.69091
500		CE: Train 0.47648 Validation 1.05824 Test 0.97692 Acc: Train 0.81654 Validation 0.7136 Test 0.71429
700		CE: Train 0.58027 Validation 0.99361 Test 0.971 Acc: Train 0.79253 Validation 0.70167 Test 0.67792
1000		CE: Train 0.43939 Validation 0.74372 Test 0.72308 Acc: Train 0.84914 Validation 0.74702 Test 0.72468

The mini-batch size doesn't seem to have a big affect of the convergence of the network. The cross entropies are all low and high accuracies are quite high for all training, validation, and test sets. In this case, I would choose a medium batch size, maybe 300-500, in order to keep the workflow not overwhelming.

3.3.

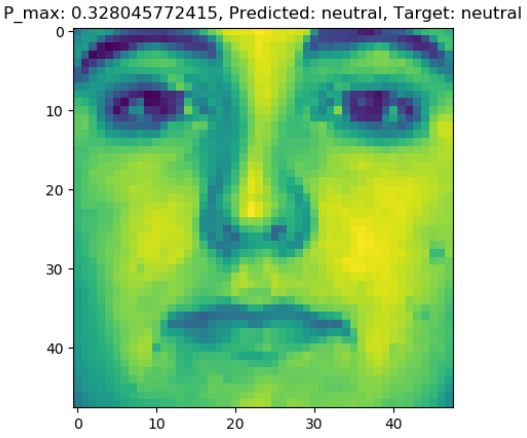
num_hiddens	Other hyperparameters	Output
[8,48]	Momentum = 0.9 learning_rate = 0.01 num_epochs = 1000 batch_size = 100	CE: Train 0.39975 Validation 0.93326 Test 0.83170 Acc: Train 0.85536 Validation 0.72792 Test 0.70909
[48,48]		CE: Train 0.07769 Validation 1.16882 Test 0.82415 Acc: Train 0.98400 Validation 0.74224 Test 0.76364
[90,48]		CE: Train 0.03894 Validation 1.08037 Test 0.73476 Acc: Train 0.99526 Validation 0.76134 Test 0.78961
[48,8]		CE: Train 0.26961 Validation 1.16429 Test 0.92757 Acc: Train 0.89301 Validation 0.71599 Test 0.71688
[48,60]		CE: Train 0.07298 Validation 0.93157 Test 0.82014 Acc: Train 0.98666 Validation 0.76372 Test 0.76623
[48,99]		CE: Train 0.08062 Validation 1.01334 Test 0.79309 Acc: Train 0.98429 Validation 0.76850 Test 0.77922

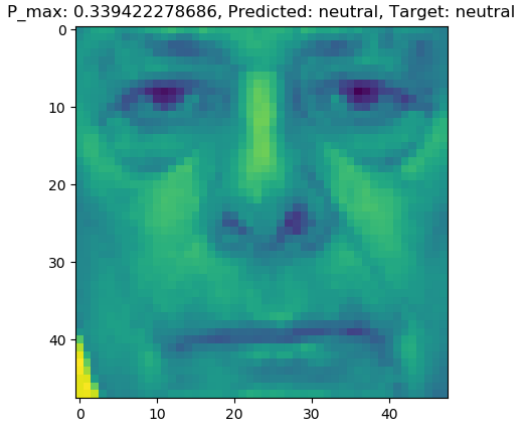
By increasing the number of hidden units for each layer of the network, the accuracy slightly increases for all sets. Cross entropy for training and test sets decreases, while in validation set, it

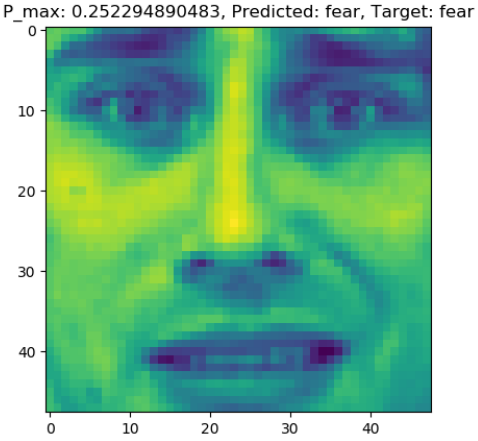
increases with more hidden units then decreases as number of hidden units keeps increasing. Exchanging the number of hidden units between the two layers doesn't make much different. This makes sense since every hidden layer could be viewed as a single feature learning process, so the order of learning different features doesn't matter. As long as every feature is learned, the result will be the same. The reason why incrementing number of hidden units of each layer makes better performance could be that when there are more units in one layer, the weight of every unit becomes smaller, so every single unit has smaller affect to the units in the next layer, and so the last hidden layer has smaller affect to the outputs. Therefore, the more the number of hidden units increases, the more the network is regularized.

3.4. We acknowledge that the network is not confident with the classification by using the softmax result. Softmax returns us a list of probability that this image belongs to each of the 7 classes, and the network will predict the classification by choosing the class with highest probability. Classification is correct only if the maximum probability is above some threshold. I would set the threshold to be greater than 0.5 since if some max probability is 0.49, there still could be another class which shares the same probability, then it's unclear which class to choose.

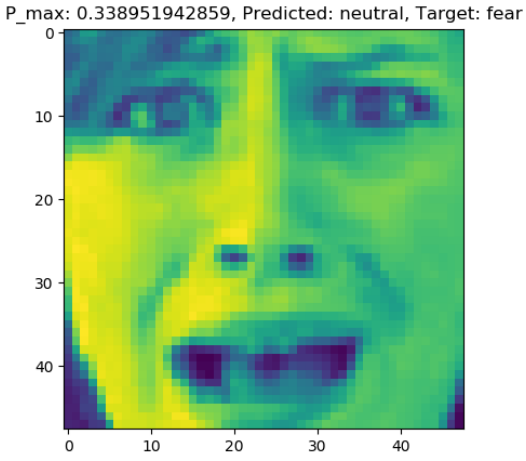
Here are some examples of when the neural network is not confident of the classification output.

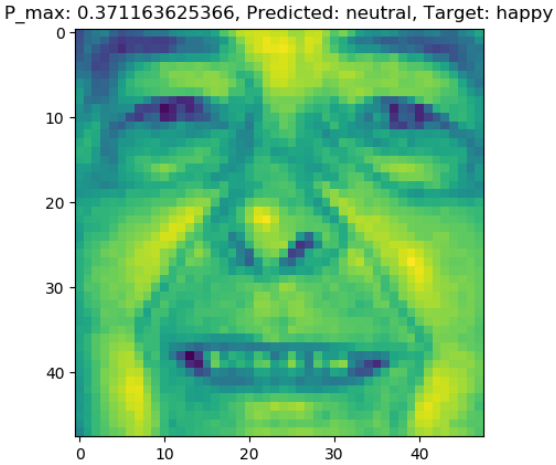




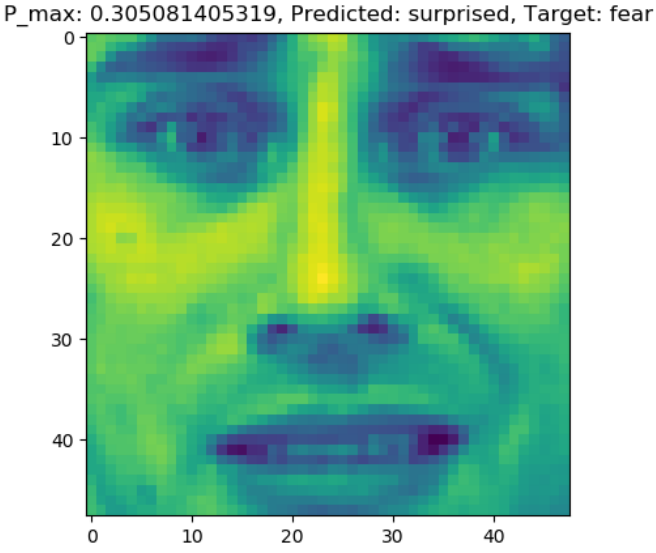


In these three examples, although the network is not confident of the classification output, it does predicts the correct output.





In these examples, the top score is below the threshold and the network predicts the wrong answer.



This example, which we've seen before on previous page, predicted correctly with a top score of about 0.252. However, in later epochs, the top score changes and was classified in a different class. In this case, the network predicted wrong.

Hence we can see that when the neural network is not confident to its prediction, classification is not reliable. The class with maximum score may not be the correct answer, and there may be more than one classes with the same maximum score. However, by efficient amount of learning, the network may be trained to have better calculations on the probabilities of the image belonging to each class in order to perform better in generating classifications.