CSC311 A2 FALL 2019

Yujie Wu, 1003968904

1.1 Since the dimension of x are conditionally independent given t, thus

$$p(x|t) = \prod_{i=1}^{D} p(x_i|t)$$

By Bayes' rule: $p(t=1|x) = \dfrac{p(t=1)\,p(x|t=1)}{p(x)}$

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

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

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

$$= \frac{1}{1 + \dfrac{1-\alpha}{\alpha} \dfrac{\prod_{i=1}^{D} p(x_i|t=0)}{\prod_{i=1}^{D} p(x_i|t=1)}}$$

$$= \frac{1}{1 + \exp\left(\ln\left(\dfrac{1-\alpha}{\alpha} \cdot \dfrac{\prod_{i=1}^{D} p(x_i|t=0)}{\prod_{i=1}^{D} p(x_i|t=1)}\right)\right)}$$

$$= \frac{1}{1 + \exp\left(\ln\left(\dfrac{1-\alpha}{\alpha}\right) + \ln\left(\dfrac{\prod_{i=1}^{D} p(x_i|t=0)}{\prod_{i=1}^{D} p(x_i|t=1)}\right)\right)}$$

$$= \frac{1}{1+\exp\left(\ln\frac{1-\alpha}{\alpha} + \sum_{i=1}^{D}\ln\frac{P(x_i|t=0)}{P(x_i|t=1)}\right)}$$

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

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

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

where $w = \sum_{i=1}^{D}\frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}$

$$b = -\ln\frac{1-\alpha}{\alpha} - \sum_{i=1}^{D}\frac{\mu_{i1}^2-\mu_{i2}^2}{2\sigma_i^2}$$

1.2 $L(w,b) = -\sum_{j=1}^{n} \log\left(\frac{1}{1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)}\right)^{t_j} \cdot \left(1 - \frac{1}{1+\exp(-\sum_{i=1}^{D} w_i x_i^{(j)} - b)}\right)^{1-t_j}$

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

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

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

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

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

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

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

$= -\sum_{j=1}^{n} \left(t_j \left(\sum_{i=1}^{D} w_i x_i^{(j)} + b\right) - \left(\sum_{i=1}^{D} w_i x_i^{(j)} + b\right) - \right.$

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

$= -\sum_{j=1}^{n} \left((t_j - 1)(w^T x_j + b) - \log(1 + \exp(-w^T x_j - b))\right)$

Derivative: $\frac{\partial L}{\partial w} = -\sum_{j=1}^{n} \left((t_j - 1)x_j - \frac{\exp(-w^T x_j - b)}{1+\exp(-w^T x_j - b)}\right)$

$\frac{\partial L}{\partial b} = -\sum_{j=1}^{n} \left(t_j - 1 - \frac{\exp(-b - w^T x_j)}{1+\exp(-w^T x_j - b)}\right)$

1.3 $L_{post}(w,b) = -\log P(w)P(b)P(t^{(1)},\dots,t^{(n)}|w,b)$

$= -\log P(w) - \log P(b) - \log P(t^{(1)},\dots,t^{(n)}|w,b)$

$= -\sum_{i=1}^{D} \log \frac{1}{\sqrt{2\pi\frac{1}{\lambda}}} \exp\left(-\frac{\lambda w_i^2}{2}\right) - 0 + L(w,b)$

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

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

Derivative:

$$\frac{\partial L_{post}}{\partial w} = \lambda \sum_{i=1}^{D} w_i - \sum_{j=1}^{n} \left((t_j - 1)x_j - \frac{\exp(w^T x_j - b)}{1+\exp(-w^T x_j - b)}\right)$$

$$\frac{\partial L_{post}}{\partial b} = \sum_{j=1}^{n} \left(t_j - 1 - \frac{\exp(-b - w^T x_j)}{1+\exp(-w^T x_j - b)}\right)$$

2.1 The performance of classifer has an bell shape as K increases, with maximum being 0.86 an $k=\{3,5,7\}$. I would choose $k=5$ as k^*, since we already computed k^*-2 and k^*+2 and they all have the value of 0.86. The test performance does correspond to the validation performance, because they are also the top 3 among all K values.

```
from q2_materials.run_knn import run_knn
from q2_materials.utils import *
import matplotlib.pyplot as plt


def get_classification_rate(k, train_data, train_label, valid_data, valid_label):
    val_label = run_knn(k, train_data, train_label, valid_data)
    correct = 0
    for i in range(len(val_label)):
        if val_label[i] == valid_label[i]:
            correct += 1
    return correct / len(val_label)


def get_test_data_rate(k, train_data, train_label, test_data, test_label):
    tst_label = run_knn(k, train_data, train_label, test_data)
    correct = 0
    for i in range(len(tst_label)):
        if tst_label[i] == test_label[i]:
            correct += 1
    return correct / len(tst_label)


if __name__ == "__main__":
    k_set = [1, 3, 5, 7, 9]
    train_d, train_l = load_train()
    valid_d, valid_l = load_valid()
    test_d, test_l = load_test()
    rate = []
    test_rate = []
    for k_val in k_set:
        rate.append(get_classification_rate(k_val, train_d, train_l, valid_d, valid_l))
        test_rate.append(get_test_data_rate(k_val, train_d, train_l, test_d, test_l))
    print(rate)
    plt.plot(rate)
    plt.show()
    print(test_rate)
```

2.2 Hyperparameters for mnist_train:

'learning_rate': 0.15

'weight_regularization': 0

'num_iterations': 768

Final Cross entropy:

Train: 0.015530 Valid: 0.183114 Test: 0.219312

Classification error:

Train: 0% Valid: 12% Test: 8%

Hyperparameters for mnist_train_small:

'learning_rate': 0.01

'weight_regularization': 0

'num_iterations': 205

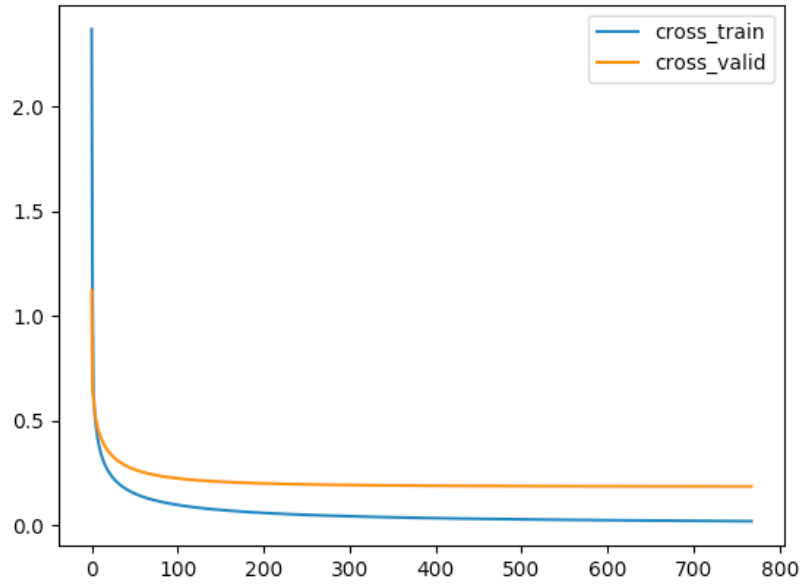
Final Cross entropy:

Train: 0.116636 Valid: 0.735973 Test: 0.560604

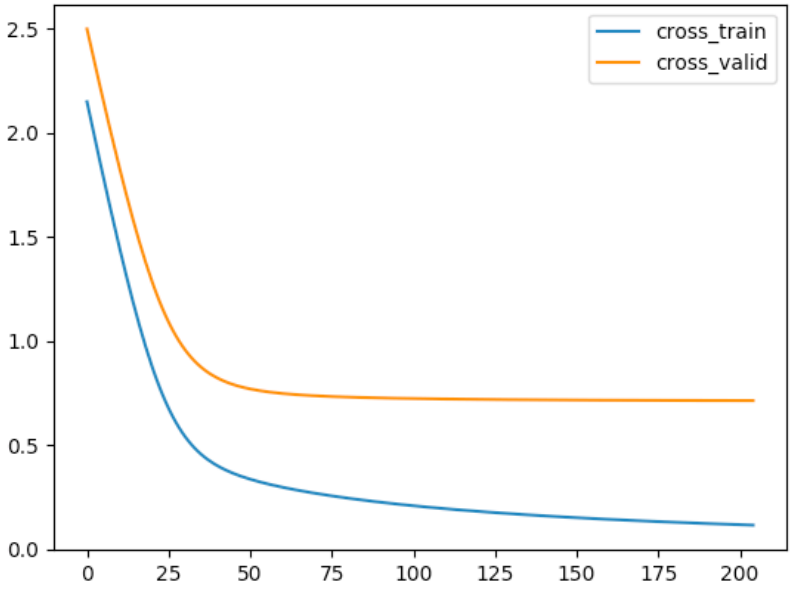
Classification error:

Train: 0% Valid: 38% Test: 26%

The graph will change since W is initialized randomly, the only thing we can do is choose the one minimizes CE



mnist_train



mnist_train_small

logistic_regression_template.py

```
import numpy as np
from scipy._lib.six import xrange
import matplotlib.pyplot as plt

from q2_materials.check_grad import check_grad
from q2_materials.utils import *
from q2_materials.logistic import *

def run_logistic_regression():
    train_inputs, train_targets = load_train()
    #train_inputs, train_targets = load_train_small()
    valid_inputs, valid_targets = load_valid()
    #valid_inputs, valid_targets = load_test()

    N, M = train_inputs.shape

    # TODO: Set hyperparameters
    hyperparameters = {
                'learning_rate': 0.1,
                'weight_regularization': 0,
                'num_iterations': 2000
            }

    # Logistic regression weights
    # TODO:Initialize to random weights here.
    weights = np.random.rand(M + 1, 1) / 10
    # Verify that your logistic function produces the right gradient.
    # diff should be very close to 0.
    run_check_grad(hyperparameters)


    # Begin learning with gradient descent
    cross_train = []
    cross_valid = []
    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.
        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):
            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)

        cross_train.append(cross_entropy_train)
        cross_valid.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}".format(
                  t+1, f / N, cross_entropy_train, frac_correct_train*100,
                  cross_entropy_valid, frac_correct_valid*100))

    plt.plot(cross_train)
    plt.plot(cross_valid)
    plt.legend(("cross_train", "cross_valid"), loc='upper right')
    plt.show()
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.random.rand(num_examples, 1)

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

    print ("diff =", diff)

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

logistic.py

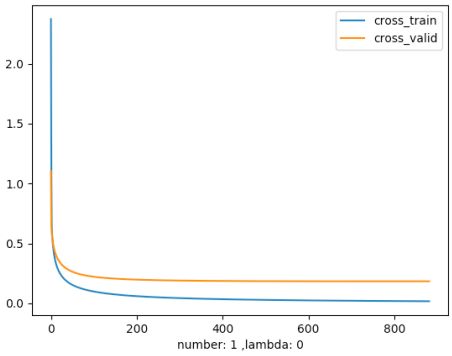
```
""" Methods for doing logistic regression."""

import numpy as np
from q2_materials.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.
    """

    # TODO: Finish this function
    data = np.concatenate((data, np.ones((data.shape[0], 1))), axis=1)
    z = np.dot(data, 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.
    """
    # TODO: Finish this function

    ce = -(targets * np.log(y) + (1 - targets) * np.log(1 - y)).mean()
    #ce = -np.dot(np.transpose(targets), np.log(y)) - np.dot(np.transpose(1 - targets), np.log(1 -
y))
    correct = 0
    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.
```

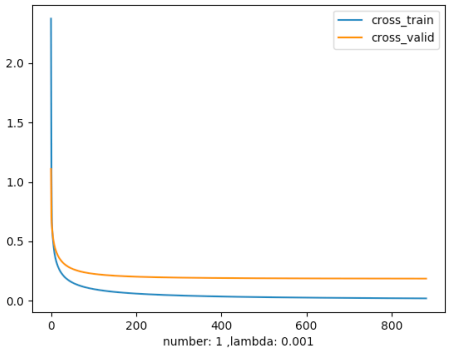
```
    Note: N is the number of examples and
          M is the number of features per example.

    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.
    """

    # TODO: Finish this function
    y = logistic_predict(weights, data)
    f = np.sum(-(targets * np.log(y) + (1 - targets) * np.log(1 - y)))
    data = np.concatenate((data, np.ones((data.shape[0], 1))), axis=1)

    loss = y - targets
    datat = data.transpose()
    df = np.dot(datat, loss)

    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.
    """

    # TODO: Finish this function
    y = logistic_predict(weights, data)
```

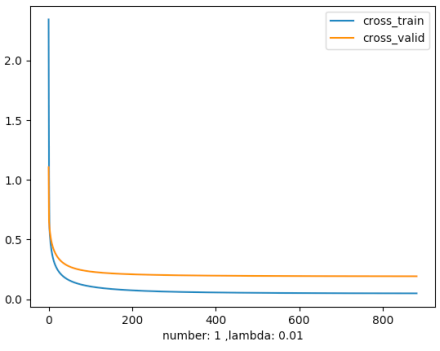
```
    f = np.sum(-(targets * np.log(y) + (1 - targets) * np.log(1 - y))) + np.sum(weights * weights) * \
        0.5 * hyperparameters["weight_regularization"]
    data = np.concatenate((data, np.ones((data.shape[0], 1))), axis=1)
    weight = np.zeros((data.shape[1], 1))
    weight[:-1,:] = weights[:-1,:] * data.shape[0]
    df = np.dot(data.transpose(), (y - targets)) + hyperparameters["weight_regularization"] *
weight
    return f, df, y
```

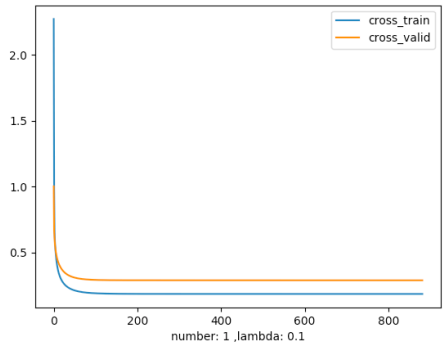
2.3

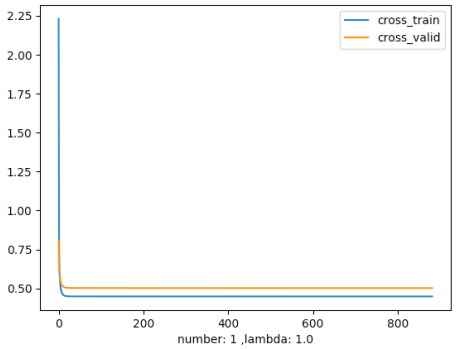
mnist_train:

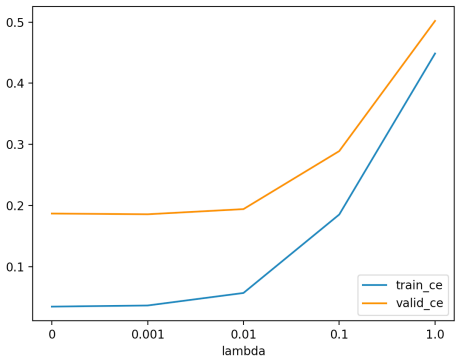


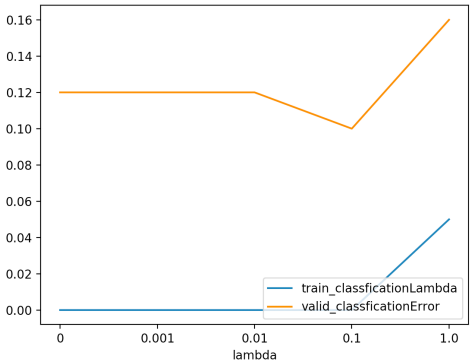








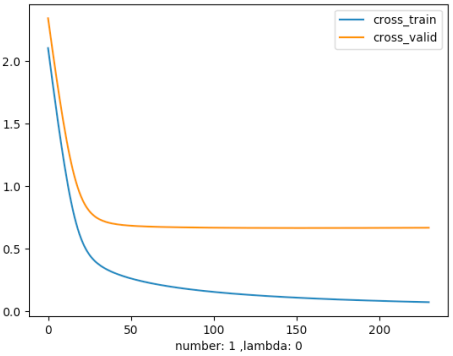


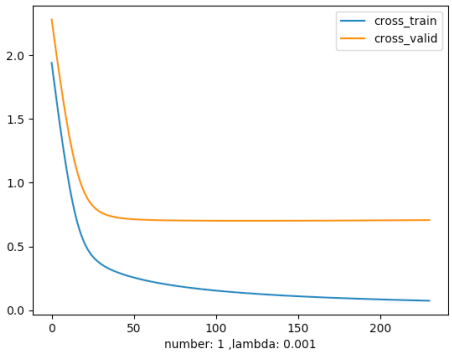


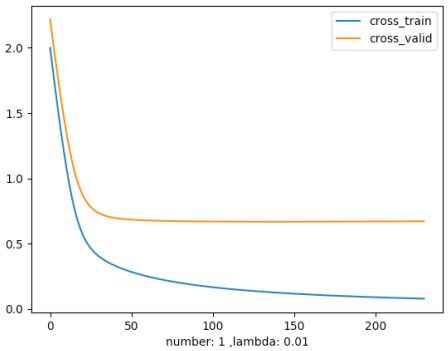
hyperparameters: `learning rate` : 0.13

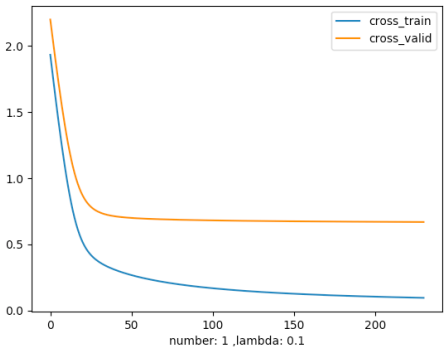
`num_iterations` : 882

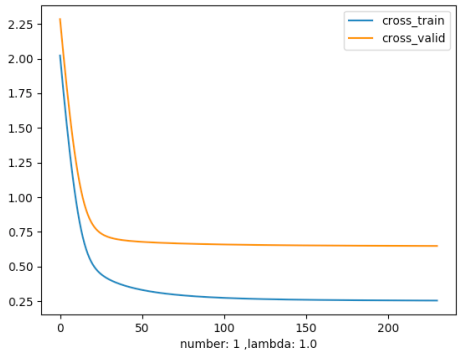
mnist_train_small:

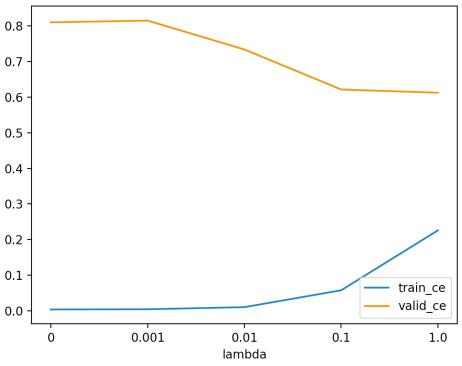


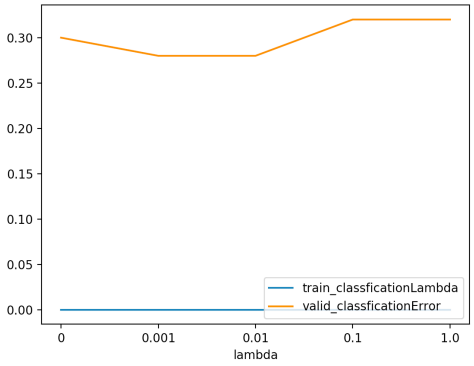












hyperparameters: `learning_rate`: 0.014

`num_iterations`: 231

In both mnist_train and mnist_train_small examples, the entropy increases as lambda increases for training set, this is because we are increasing the penalty on weights, so that with same weight,

Cross entropy must be higher is the penalty is higher. However, for the validation set, the cross entropy continue increases with mnist_train when lambda increases, but mnist_train_small's cross entropy has a decreasing trend when lambda increases. One possible explination is that for relatively smaller data sets, models are easier to become overfit without adding penalty, since there are less features and data. However, with the size of the data set increases, it is harder to find the best weight. Therefore, add the penalty will make the model more unsuitable for both training set and validation set, it decreases the speed to find best weight within a certain number of iterations. Then, the classification error of mnist_train follows the trend of cross entropy, which is, it increases when cross entropy increases, and vice versa. As for mnist_train_small, it decreases at first, and then increases. One possible reason is that as the penality increases from 0, it decreases since the model becomes better, but then the best lambda was missed, and the error increased as an result.

I think penalized regression works better with smaller data sets while the one without penalty works better with larger data sets, since with a small data set, the model is easier to become overfit to the training set, so adding the penalization can help to solve this problem. However, the learning process for larger data sets is already slow and adding penalty will decrease the speed even more.

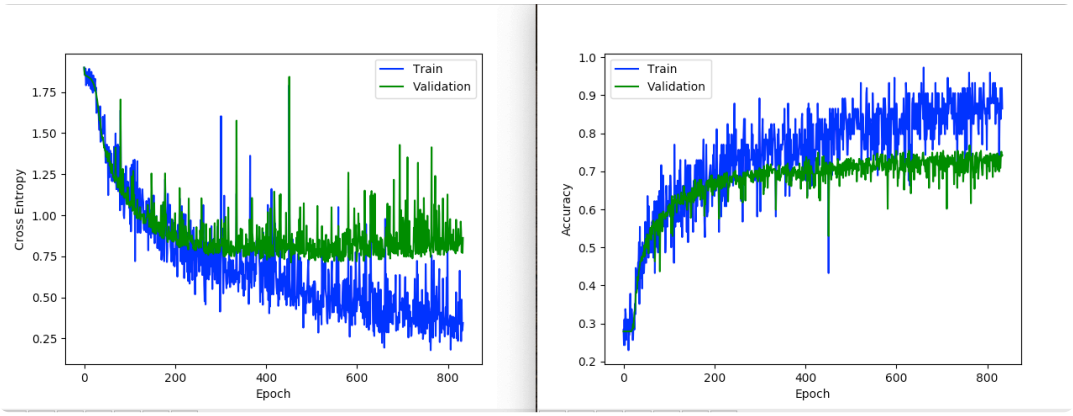
logistic_pen_regression_template.py

```
import statistics

import numpy as np
from scipy._lib.six import xrange
import matplotlib.pyplot as plt

from q2_materials.check_grad import check_grad
from q2_materials.utils import *
from q2_materials.logistic import *


def run_logistic_regression():
    #train_inputs, train_targets = load_train()
    train_inputs, train_targets = load_train_small()
    valid_inputs, valid_targets = load_valid()
    #valid_inputs, valid_targets = load_test()

    N, M = train_inputs.shape

    weight = np.random.rand(M + 1, 1) / 10

    # TODO: Set hyperparameters
    hyperparameters = {
        'learning_rate': 0.13,
        'weight_regularization': 0,
        'num_iterations': 231
    }

    l_list = [0, 0.001, 0.01, 0.1, 1.0]
    # Logistic regression weights

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

    # Begin learning with gradient descent

    train_ceLambda = [0, 0, 0, 0, 0]
    train_classficationLambda = [0, 0, 0, 0, 0]

    valid_ceLambda = [0, 0, 0, 0, 0]
    valid_classficationLambda = [0, 0, 0, 0, 0]
    for l in l_list:
        hyperparameters["weight_regularization"] = l
        for times in range(5):
            cross_train = []
            cross_valid = []
            classfication_train = []
            classfication_valid = []
            weights = weight
            plt.xlabel("number: " + str(times + 1) + " ,lambda: " + str(l))
            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.
            f, df, predictions = logistic_pen(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):
                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)

            cross_train.append(cross_entropy_train)
            classfication_train.append(frac_correct_train)
            cross_valid.append(cross_entropy_valid)
            classfication_valid.append(frac_correct_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}".format(
                t + 1, f / N, cross_entropy_train, frac_correct_train * 100,
                cross_entropy_valid, frac_correct_valid * 100))
            if t == hyperparameters['num_iterations'] - 1:
                train_ceLambda[l_list.index(l)] += cross_entropy_train / 5
                train_classficationLambda[l_list.index(l)] += (1 - frac_correct_train) / 5
                valid_ceLambda[l_list.index(l)] += cross_entropy_valid / 5
                valid_classficationLambda[l_list.index(l)] += (1 - frac_correct_valid) / 5
        # plt.plot(cross_train)
        # plt.plot(cross_valid)
        # plt.legend(("cross_train", "cross_valid"), loc='upper right')
        # plt.show()
    plt.xlabel('lambda')
    plt.xticks(list(range(len(l_list))), l_list)
    plt.plot(train_ceLambda)
    plt.plot(valid_ceLambda)
    plt.legend(("train_ce", "valid_ce"), loc='lower right')
    plt.show()
    plt.xlabel('lambda')
    plt.xticks(list(range(len(l_list))), l_list)
    plt.plot(train_classficationLambda)
    plt.plot(valid_classficationLambda)
    plt.legend(("train_classficationLambda", "valid_classficationError"), loc='lower right')
    plt.show()


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.random.rand(num_examples, 1)

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

    print("diff =", diff)


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

3.1 The cross entropy for training set has a decreasing trend over entire learning process, and infinately approaching to 0, while the cross entropy for validation set decreases alone with training set at first, and then the rate of decreasing decreases so that the cross entropy of training set becames slightly higher than the training set. Finally, it has an increasing trend. The error rate for both sets have similar behavior.

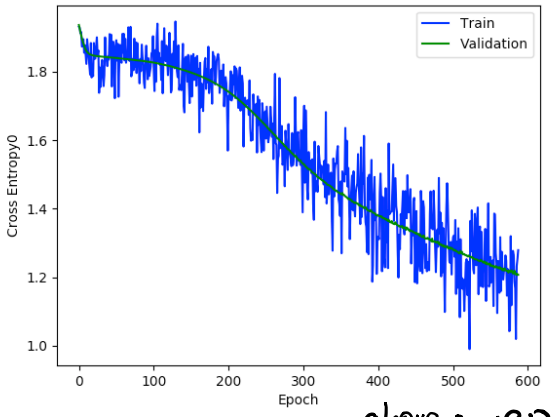


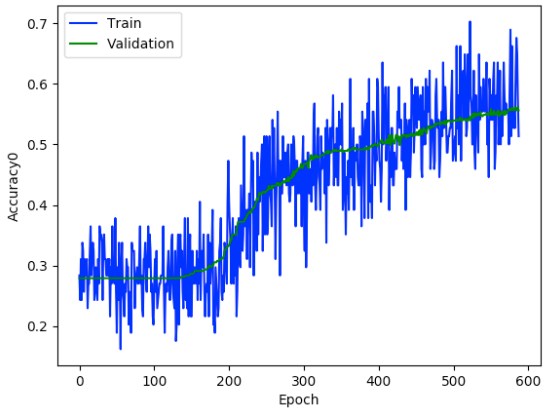
Cross Entropy

Accuracy = 1 - error rate

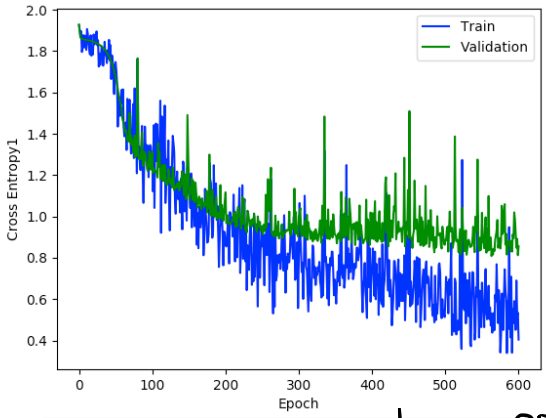
3.2 The covergence property that I observed for different learning rate is that: at a reasonable range, e.g. from 0.001 to 0.1, the convergence rate at very early stage moreases rapidly as learning rate increases, but as learning process keep going, the convergence rate for higher learning rate decreases early and approches to 0. As for relatively higher learning rates, such as 0.5 and 1.0, the entropy and correct rate can be expressed

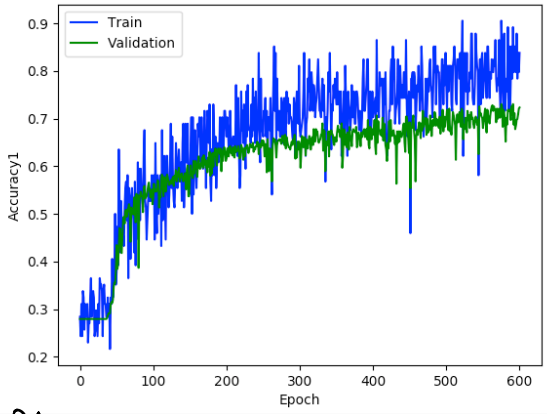
as nearly horizontal lines, since the learning rates are too large, it comes back to original points at every iteration, thus they are not convergent.



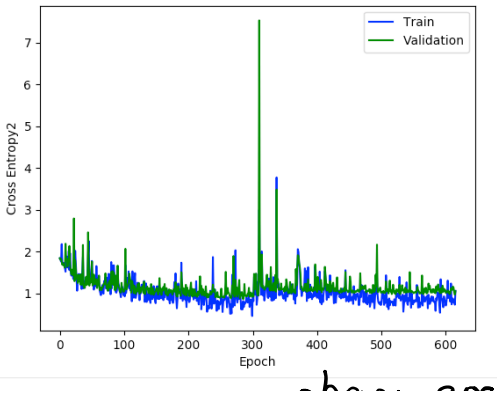


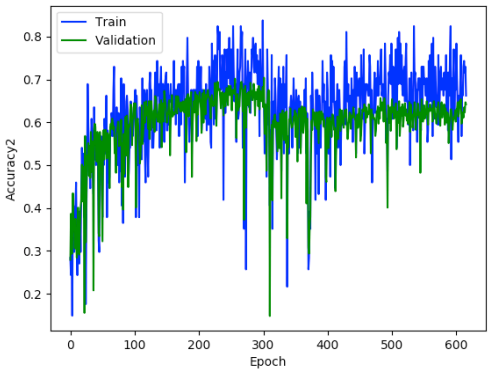
above : eps = 0.001





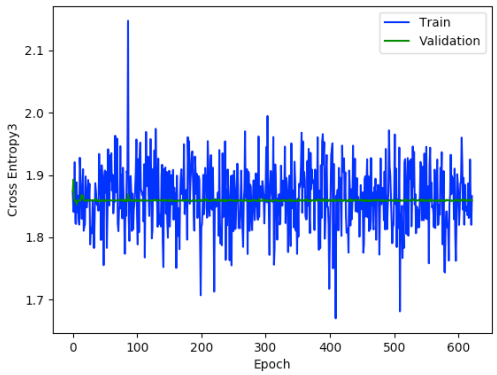
above : eps = 0.01

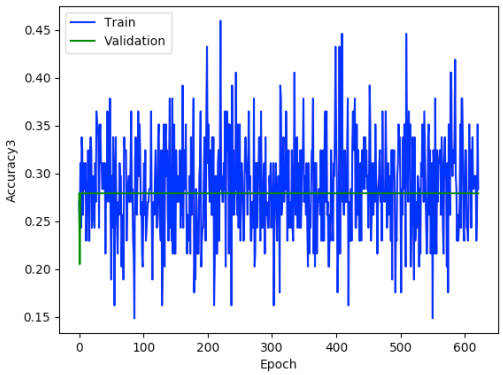




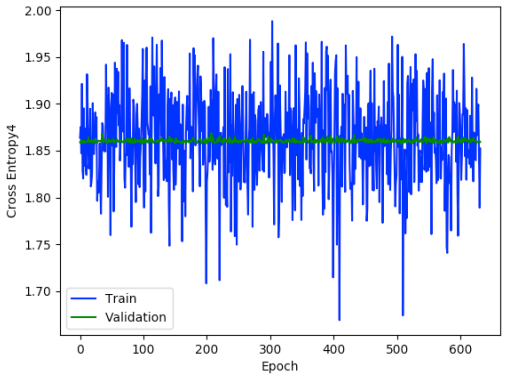
above : eps = 0.1

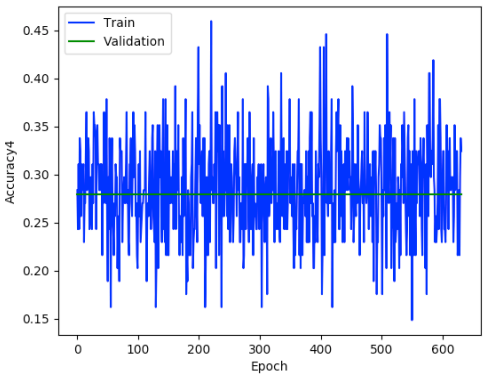
above: eps = 0.1





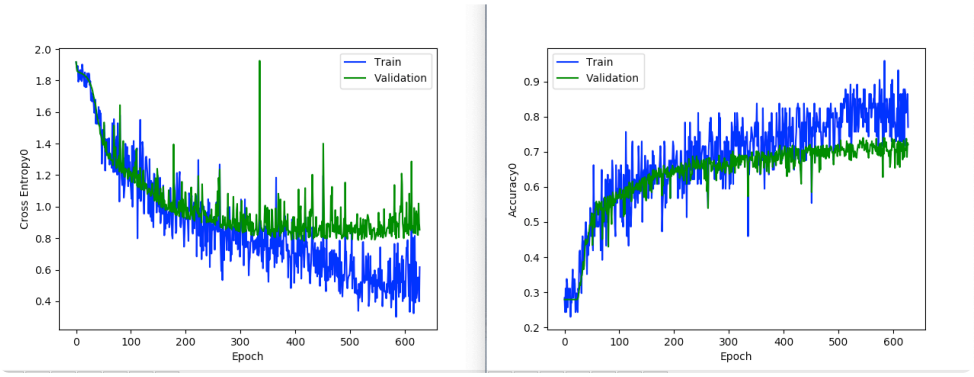
above: eps = 0.5



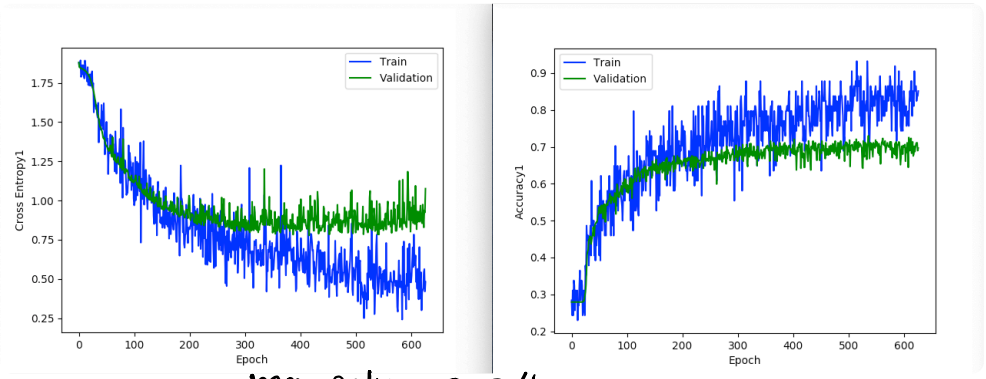


above: eps = 1.0

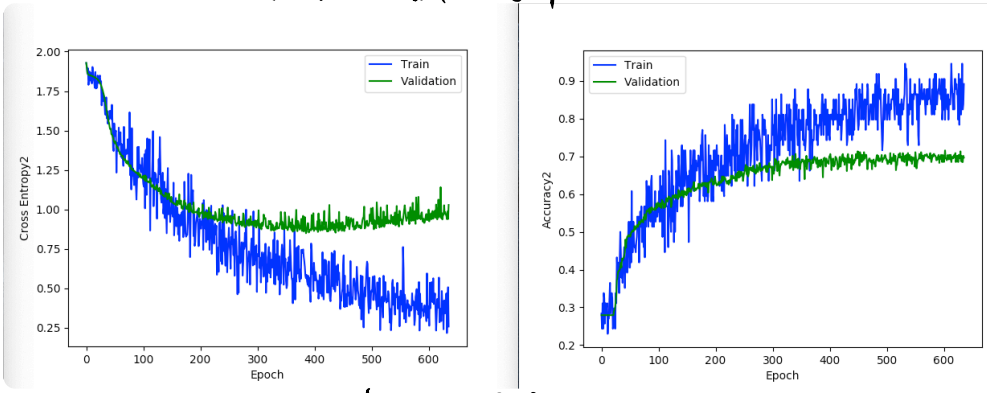
As for momentum, the convergence rates do not have any rapid changes through all tests that I ran (with 0.0, 0.4, 0.9). However, the convergence rate will still be larger at early stage with higher momentum, since it reaches lowest point first, and then decreases. The main difference is that the higher the momentum is, the smoother the curve will be.



momentum = 0.0

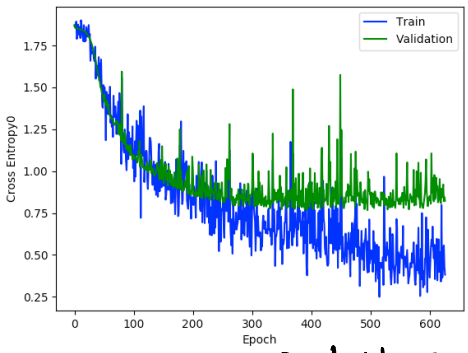


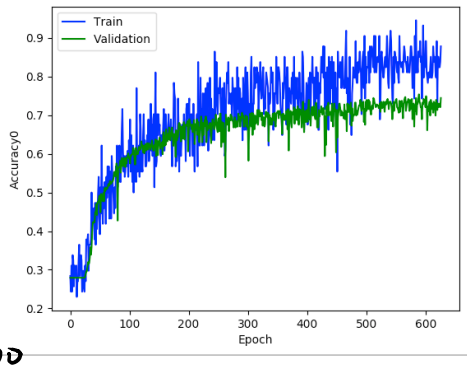
momentum = 0.4



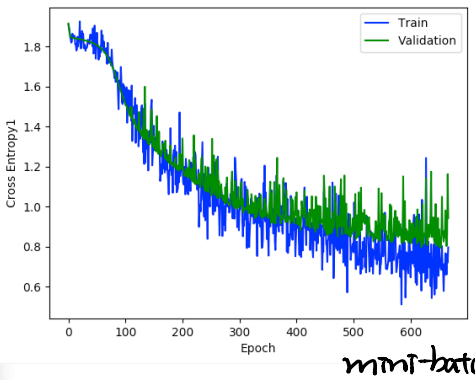
momentum = 0.9

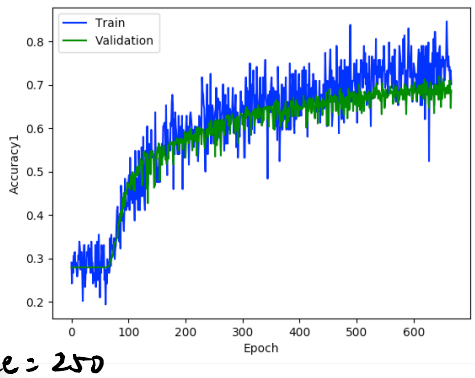
Lastly, for the mini-batch size, the rate of convergence decreases as mini-batch size increases at early stage. Therefore, with lower mini-batch size, both cross entropy and correct rate will approach to their limit first, and then keep getting closer to limit with smaller and smaller rate.



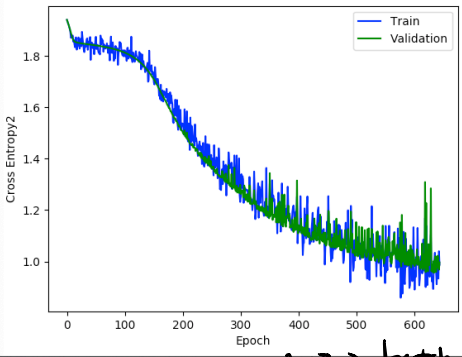


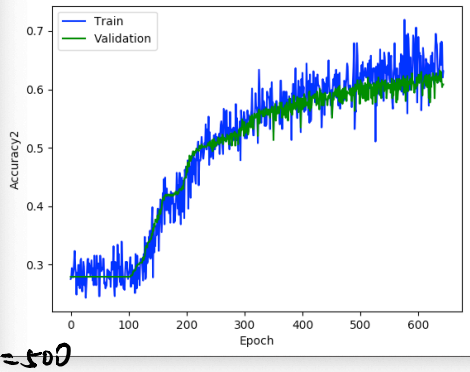
mini-batch size = 100



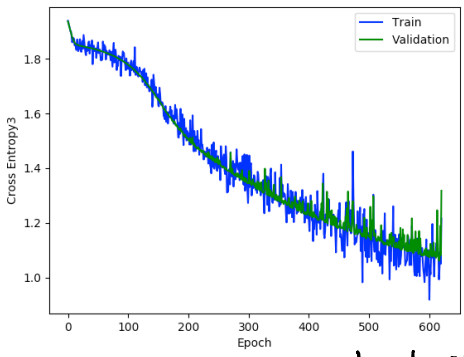


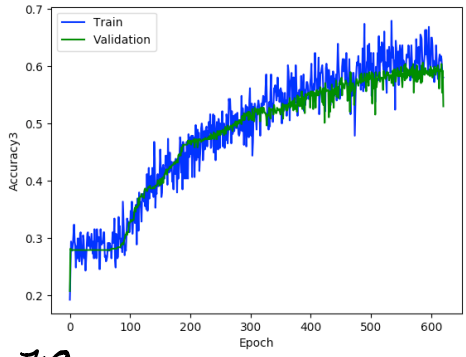
mini-batch size = 250



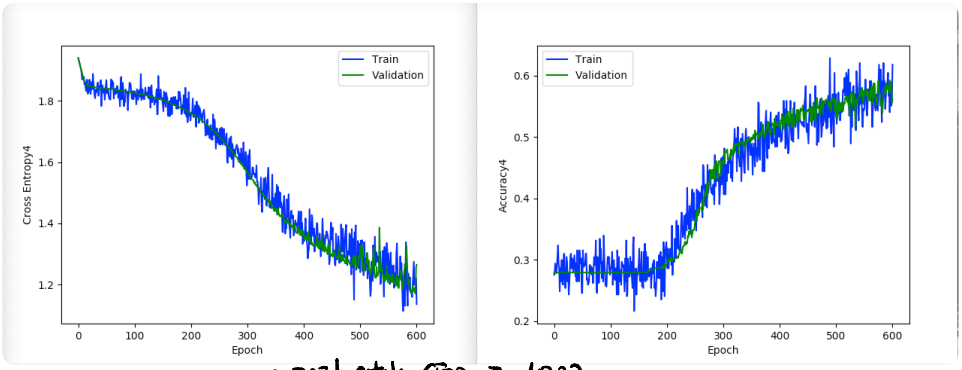


mini-batch size = 500





mini-batch size = 750

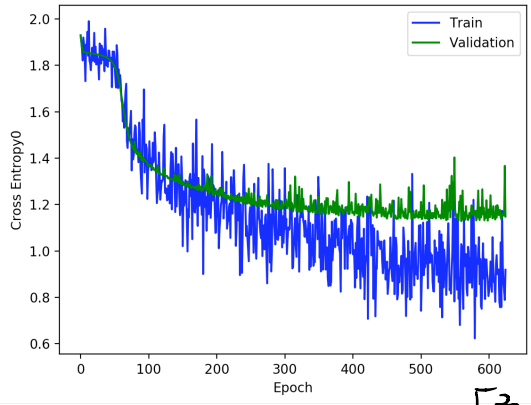


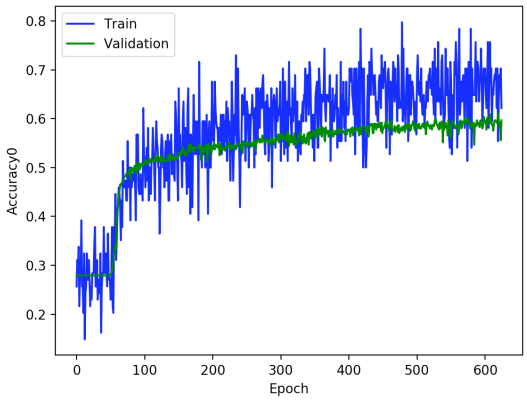
mini-batch size = 1000

I would choose as large as possible in a reasonable range e.g. 0.001 to 0.1 for learning rate, larger momentum, e.g. 0.9 and smaller mini-batch size, e.g. 100 in order to approach to the optimum value faster. I can not choose value outside of reasonable range, like 1.0 for learning rate, since otherwise the cross entroy and correct rate for validation set will stay the same (learning rate too large) or 1 for mini-batch size, since it will take too long to compute results.

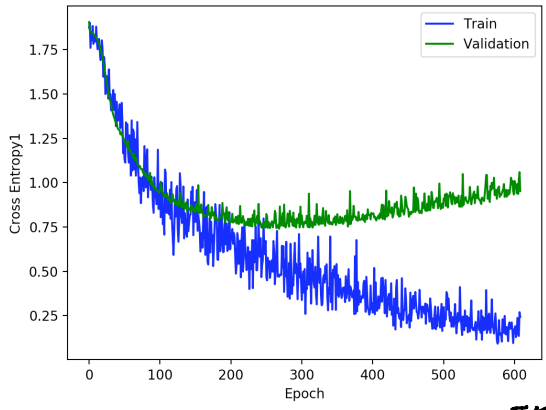
3.3 In this section, I examined [2, 32], [50, 32], [100, 32] and [16, 2], [16, 50], [16, 100], total of six different combination of amount of neuron network hidden units in each layer, and find out that with higher number of hidden units, the convergence rate at early stage is higher. However, the difference between train and validation set results are also getting larger as the number of hidden layers increases after it reaches

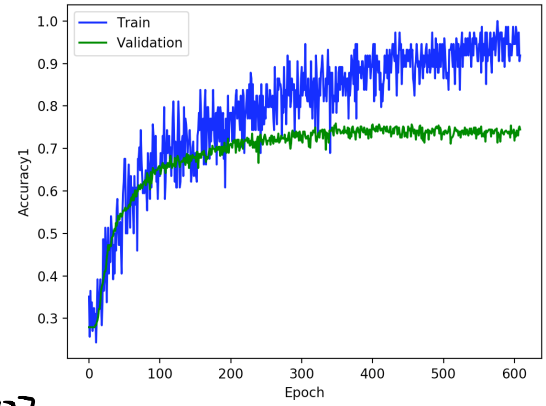
lowest value, which is, the model will become more overfit to the training set and less generalizable to other data sets, since there is an obviously increasing trend on cross entropy and error rate for validation set.



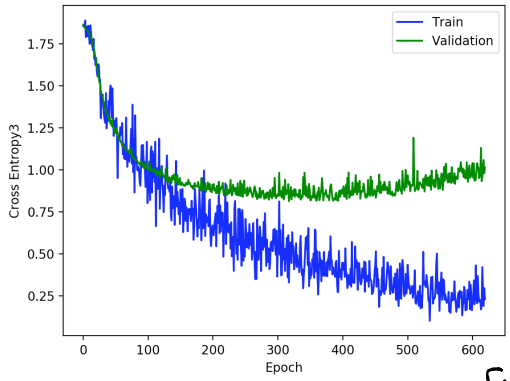


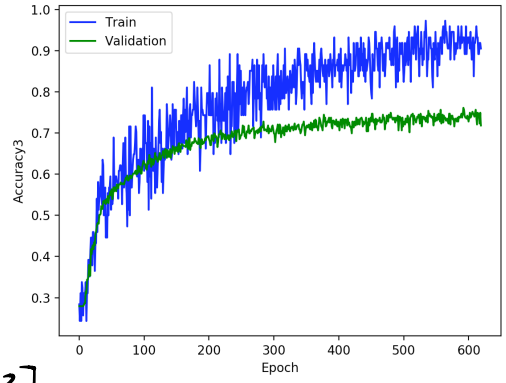
[2, 32]



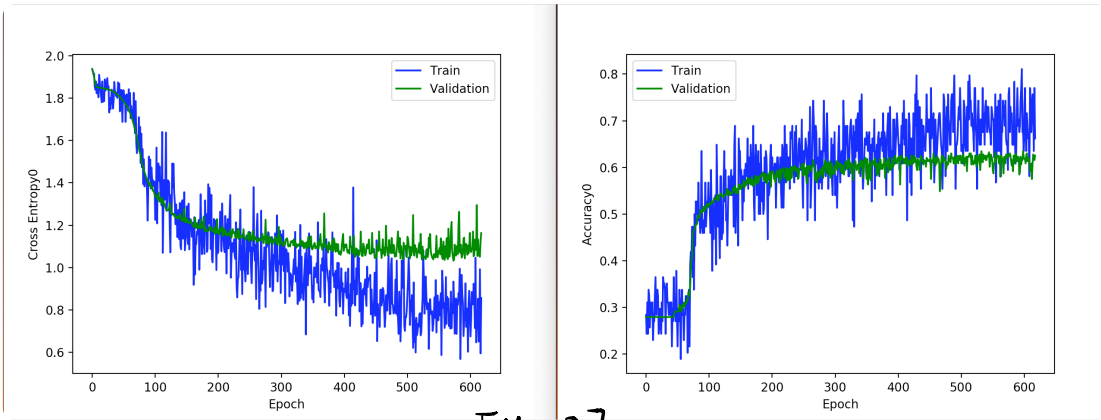


[50, 32]

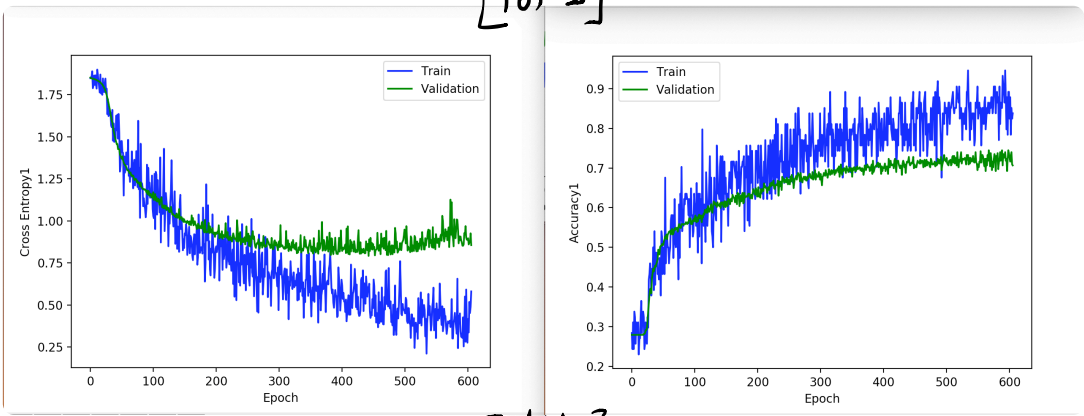




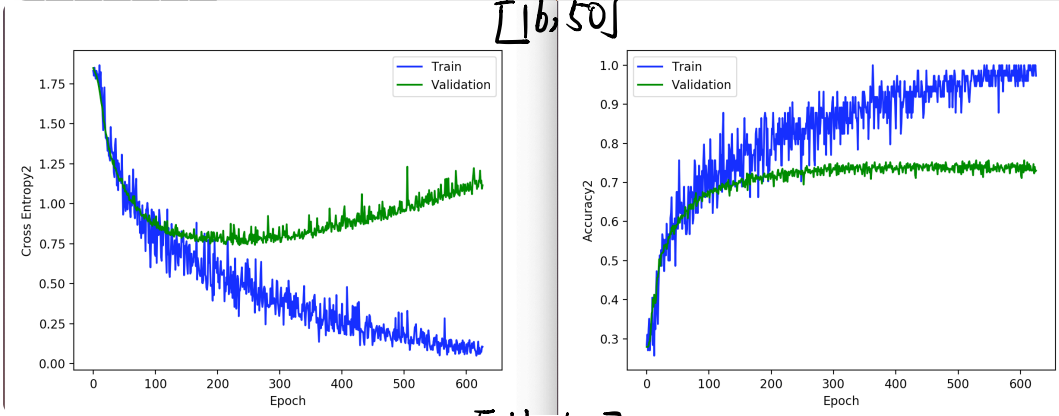
[100, 32]



[16, 2]



[16, 50]

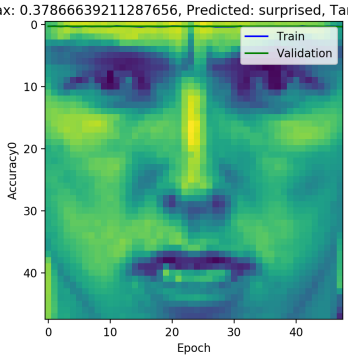


[16, 100]

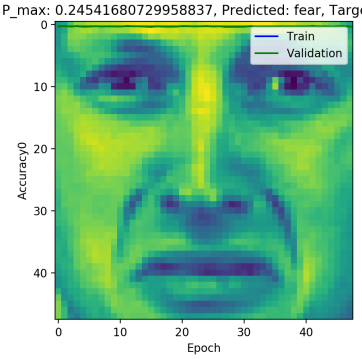
3.4 Among all uncertain examples with less trained model, almost all of them are either be classified as anger or neutral with the highest score being very low (around 0.1). One possible answer to that is the patterns of pixel intensity of these particular images are differed from any other images, they share some common patterns from other categories. Thus, the neuron network can not product valid answer. Moreover, the evidences show that simply classify them as the top scoring class is incorrect at majority of time.

However, as the training process of the neuron network keep going, I also examined some uncertain example after 1000 epoches, the P_max values of those examples become higher while the mismatch rate only becomes little bit lower. In some of the examples, it is even relatively difficult for human eyes to give a 100% correct prediction. Still, just classify them as top scoring class is still unsafe, the best solution is to continue training the neuron networks and add some penalties at late stage in order to avoid overfits.

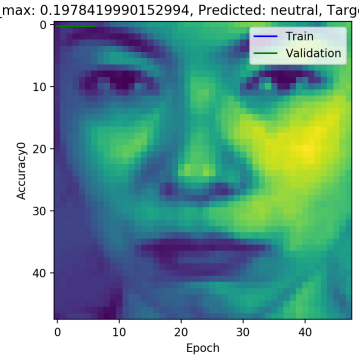
P_max: 0.37866639211287656, Predicted: surprised, Target: anger



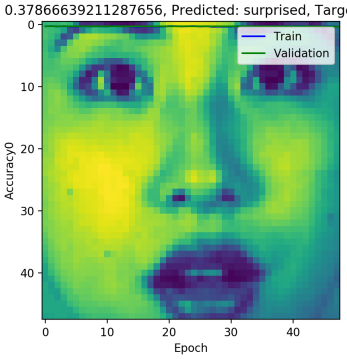
P_max: 0.24541680729958837, Predicted: fear, Target: sad



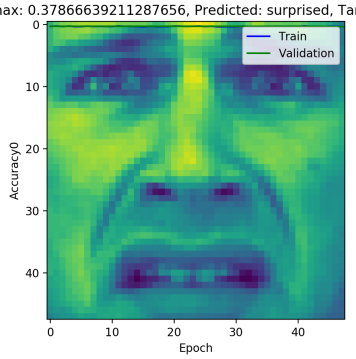
P_max: 0.1978419990152994, Predicted: neutral, Target: anger



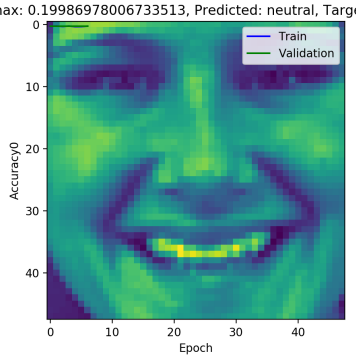
P_max: 0.37866639211287656, Predicted: surprised, Target: surprised



P_max: 0.37866639211287656, Predicted: surprised, Target: fear



P_max: 0.19986978006733513, Predicted: neutral, Target: happy



```
"""
Instruction:

In this section, you are asked to train a NN with different hyperparameters.
To start with training, you need to fill in the incomplete code. There are 3
places that you need to complete:
a) Backward pass equations for an affine layer (linear transformation + bias).
b) Backward pass equations for ReLU activation function.
c) Weight update equations with momentum.

After correctly fill in the code, modify the hyperparameters in "main()".
You can then run this file with the command: "python nn.py" in your terminal.
The program will automatically check your gradient implementation before start.
The program will print out the training progress, and it will display the
training curve by the end. You can optionally save the model by uncommenting
the lines in "main()".
"""

from __future__ import division
from __future__ import print_function

from q3_materials.util import LoadData, Load, Save, DisplayPlot
import sys
import numpy as np
import matplotlib.pyplot as plt


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.

    Returns:
        model:         Randomly initialized network weights.
    """
    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)
    b1 = np.zeros((num_hiddens[0]))
    b2 = np.zeros((num_hiddens[1]))
    b3 = np.zeros((num_outputs))
    model = {
        'W1': W1,
        'W2': W2,
        'W3': W3,
        'b1': b1,
        'b2': b2,
        'b3': b3
    }
    return model
```

```
def Affine(x, w, b):
    """Computes the affine transformation.

    Args:
        x: Inputs (or hidden layers)
        w: Weights
        b: Bias

    Returns:
        y: Outputs
    """
    # y = np.dot(w.T, x) + b
    y = x.dot(w) + b
    return y


def AffineBackward(grad_y, h, w):
    """Computes gradients of affine transformation.
    hint: you may need the matrix transpose np.dot(A,B).T = np.dot(B,A) and (A.T).T = A

    Args:
        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.
    """
    ############################
    # Insert your code here.
    grad_h = np.dot(grad_y, w.transpose())
    grad_w = np.dot(h.transpose(), grad_y)
    grad_b = np.sum(grad_y, axis=0)
    print(grad_b)
    return grad_h, grad_w, grad_b
    ############################
    #raise Exception('Not implemented')


def ReLU(z):
    """Computes the ReLU activation function.

    Args:
        z: Inputs

    Returns:
        h: Activation of z
    """
    return np.maximum(z, 0.0)
```

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

    Returns:
        grad_z: Gradients wrt. the hidden state prior to activation.
    """
    ############################
    # Insert your code here.
    grad_z = z
    for i in range(grad_z.shape[0]):
        for index in range(len(grad_z[i])):
            grad_z[i][index] = (z[i][index] > 0)
    grad_z = grad_z * grad_h
    return grad_z
    ############################
    #raise Exception('Not implemented')


def Softmax(x):
    """Computes the softmax activation function.

    Args:
        x: Inputs

    Returns:
        y: Activation
    """
    return np.exp(x) / np.exp(x).sum(axis=1, keepdims=True)


def NNForward(model, x):
    """Runs the forward pass.

    Args:
        model: Dictionary of all the weights.
        x:     Input to the network.

    Returns:
        var:   Dictionary of all intermediate variables.
    """
    z1 = Affine(x, model['W1'], model['b1'])
    h1 = ReLU(z1)
    z2 = Affine(h1, model['W2'], model['b2'])
    h2 = ReLU(z2)
    y = Affine(h2, model['W3'], model['b3'])
    var = {
        'x': x,
        'z1': z1,
        'h1': h1,
        'z2': z2,
        'h2': h2,
        'y': y
    }
    return var
```

```
def NNBackward(model, err, var):
    """Runs the backward pass.

    Args:
        model:    Dictionary of all the weights.
        err:      Gradients to the output of the network.
        var:      Intermediate variables from the forward pass.
    """
    dE_dh2, dE_dW3, dE_db3 = AffineBackward(err, var['h2'], model['W3'])
    dE_dz2 = ReLUBackward(dE_dh2, var['z2'])
    dE_dh1, dE_dW2, dE_db2 = AffineBackward(dE_dz2, var['h1'], model['W2'])
    dE_dz1 = ReLUBackward(dE_dh1, var['z1'])
    _, dE_dW1, dE_db1 = AffineBackward(dE_dz1, var['x'], model['W1'])
    model['dE_dW1'] = dE_dW1
    model['dE_dW2'] = dE_dW2
    model['dE_dW3'] = dE_dW3
    model['dE_db1'] = dE_db1
    model['dE_db2'] = dE_db2
    model['dE_db3'] = dE_db3
    pass


def NNUpdate(model, eps, momentum):
    """Update NN weights.

    Args:
        model:    Dictionary of all the weights.
        eps:      Learning rate.
        momentum: Momentum.
    """
    ############################
    if "vw1" not in model:
        model["vw1"] = np.zeros(model["W1"].shape)
    model["vw1"] = momentum * model["vw1"] + (1 - momentum) * model['dE_dW1']
    if "vw2" not in model:
        model["vw2"] = np.zeros(model["W2"].shape)
    model["vw2"] = momentum * model["vw2"] + (1 - momentum) * model['dE_dW2']
    if "vw3" not in model:
        model["vw3"] = np.zeros(model["W3"].shape)
    model["vw3"] = momentum * model["vw3"] + (1 - momentum) * model['dE_dW3']
    if "vb1" not in model:
        model["vb1"] = np.zeros(model["b1"].shape)
    model["vb1"] = momentum * model["vb1"] + (1 - momentum) * model['dE_db1']
    if "vb2" not in model:
        model["vb2"] = np.zeros(model["b2"].shape)
    model["vb2"] = momentum * model["vb2"] + (1 - momentum) * model['dE_db2']
    if "vb3" not in model:
        model["vb3"] = np.zeros(model["b3"].shape)
    model["vb3"] = momentum * model["vb3"] + (1 - momentum) * model['dE_db3']

    # Insert your code here.
    # Update the weights.
```

```
    model['W1'] -= eps * model['vw1']
    model['W2'] -= eps * model['vw2']
    model['W3'] -= eps * model['vw3']
    model['b1'] -= eps * model['vb1']
    model['b2'] -= eps * model['vb2']
    model['b3'] -= eps * model['vb3']
    ############################
    # raise Exception('Not implemented')


def Train(model, forward, backward, update, eps, momentum, num_epochs,
          batch_size):
    """Trains a simple MLP.

    Args:
        model:          Dictionary of model weights.
        forward:        Forward prop function.
        backward:       Backward prop function.
        update:         Update weights function.
        eps:            Learning rate.
        momentum:       Momentum.
        num_epochs:     Number of epochs to run training for.
        batch_size:     Mini-batch size, -1 for full batch.

    Returns:
        stats:          Dictionary of training statistics.
            - train_ce:       Training cross entropy.
            - valid_ce:       Validation cross entropy.
            - train_acc:      Training accuracy.
            - valid_acc:      Validation accuracy.
    """
    inputs_train, inputs_valid, inputs_test, target_train, target_valid, \
        target_test = LoadData('toronto_face.npz')
    rnd_idx = np.arange(inputs_train.shape[0])
    train_ce_list = []
    valid_ce_list = []
    train_acc_list = []
    valid_acc_list = []
    num_train_cases = inputs_train.shape[0]
    if batch_size == -1:
        batch_size = num_train_cases
    num_steps = int(np.ceil(num_train_cases / batch_size))
    for epoch in range(num_epochs):
        np.random.shuffle(rnd_idx)
        inputs_train = inputs_train[rnd_idx]
        target_train = target_train[rnd_idx]
        for step in range(num_steps):
            # Forward prop.
            start = step * batch_size
            end = min(num_train_cases, (step + 1) * batch_size)
            x = inputs_train[start: end]
            t = target_train[start: end]

            var = forward(model, x)
```

```
        prediction = Softmax(var['y'])
        train_ce = -np.sum(t * np.log(prediction)) / x.shape[0]
        train_acc = (np.argmax(prediction, axis=1) ==
                     np.argmax(t, axis=1)).astype('float').mean()
        print(('Epoch {:3d} Step {:2d} Train CE {:.5f} '
               'Train Acc {:.5f}').format(
            epoch, step, train_ce, train_acc))



        # Compute error.
        error = (prediction - t) / x.shape[0]

        # Backward prop.
        backward(model, error, var)

        # Update weights.
        update(model, eps, momentum)

    valid_ce, valid_acc = Evaluate(
        inputs_valid, target_valid, model, forward, batch_size=batch_size)
    print(('Epoch {:3d} '
           'Validation CE {:.5f} '
           'Validation Acc {:.5f}\n').format(
        epoch, valid_ce, valid_acc))
    train_ce_list.append((epoch, train_ce))
    train_acc_list.append((epoch, train_acc))
    valid_ce_list.append((epoch, valid_ce))
    valid_acc_list.append((epoch, valid_acc))
    DisplayPlot(train_ce_list, valid_ce_list, 'Cross Entropy0', number=0)
    DisplayPlot(train_acc_list, valid_acc_list, 'Accuracy0', number=1)


print()
train_ce, train_acc = Evaluate(
    inputs_train, target_train, model, forward, batch_size=batch_size)
valid_ce, valid_acc = Evaluate(
    inputs_valid, target_valid, model, forward, batch_size=batch_size)
test_ce, test_acc = Evaluate(
    inputs_test, target_test, model, forward, batch_size=batch_size)
print('CE: Train %.5f Validation %.5f Test %.5f' %
      (train_ce, valid_ce, test_ce))
print('Acc: Train {:.5f} Validation {:.5f} Test {:.5f}'.format(
    train_acc, valid_acc, test_acc))
low_index = np.max(prediction, axis=1) < 0.5
class_names = ['anger', 'disgust', 'fear', 'happy', 'sad', 'surprised', 'neutral']
if np.sum(low_index) > 0:
    for i in np.where(low_index > 0)[0][:20]:
        plt.imshow(x[i].reshape(int(np.sqrt(2304)), int(np.sqrt(2304))))
        plt.title('P_max: {}, Predicted: {}, Target: {}'.format(np.max(prediction[i]),
                                                    class_names[np.argmax(prediction[i])],
                                                    class_names[np.argmax(t[i])]))
        plt.savefig(str(i))
```

```
    stats = {
        'train_ce': train_ce_list,
        'valid_ce': valid_ce_list,
        'train_acc': train_acc_list,
        'valid_acc': valid_acc_list
    }

    return model, stats


def Evaluate(inputs, target, model, forward, batch_size=-1):
    """Evaluates the model on inputs and target.

    Args:
        inputs: Inputs to the network.
        target: Target of the inputs.
        model:  Dictionary of network weights.
    """
    num_cases = inputs.shape[0]
    if batch_size == -1:
        batch_size = num_cases
    num_steps = int(np.ceil(num_cases / batch_size))
    ce = 0.0
    acc = 0.0
    for step in range(num_steps):
        start = step * batch_size
        end = min(num_cases, (step + 1) * batch_size)
        x = inputs[start: end]
        t = target[start: end]
        prediction = Softmax(forward(model, x)['y'])
        ce += -np.sum(t * np.log(prediction))
        acc += (np.argmax(prediction, axis=1) == np.argmax(
            t, axis=1)).astype('float').sum()
    ce /= num_cases
    acc /= num_cases
    return ce, acc


def CheckGrad(model, forward, backward, name, x):
    """Check the gradients

    Args:
        model: Dictionary of network weights.
        name: Weights name to check.
        x: Fake input.
    """
    np.random.seed(0)
    var = forward(model, x)
    loss = lambda y: 0.5 * (y ** 2).sum()
    grad_y = var['y']
    backward(model, grad_y, var)
    grad_w = model['dE_d' + name].ravel()
    w_ = model[name].ravel()
    eps = 1e-7
```

```
    grad_w_2 = np.zeros(w_.shape)
    check_elem = np.arange(w_.size)
    np.random.shuffle(check_elem)
    # Randomly check 20 elements.
    check_elem = check_elem[:20]
    for ii in check_elem:
        w_[ii] += eps
        err_plus = loss(forward(model, x)['y'])
        w_[ii] -= 2 * eps
        err_minus = loss(forward(model, x)['y'])
        w_[ii] += eps
        grad_w_2[ii] = (err_plus - err_minus) / 2 / eps
    np.testing.assert_almost_equal(grad_w[check_elem], grad_w_2[check_elem],
                                   decimal=3)

def main():
    """Trains a NN."""
    model_fname = 'nn_model.npz'
    stats_fname = 'nn_stats.npz'

    # Hyper-parameters. Modify them if needed.
    num_hiddens = [16, 2]
    eps = 0.01
    momentum = 0.9
    num_epochs = 1000
    batch_size = 100

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

    # Initialize model.
    model = InitNN(num_inputs, num_hiddens, num_outputs)

    # Uncomment to reload trained model here.
    # model = Load(model_fname)

    # Check gradient implementation.
    print('Checking gradients...')
    x = np.random.rand(10, 48 * 48) * 0.1
    CheckGrad(model, NNForward, NNBackward, 'W3', x)
    CheckGrad(model, NNForward, NNBackward, 'b3', x)
    CheckGrad(model, NNForward, NNBackward, 'W2', x)
    CheckGrad(model, NNForward, NNBackward, 'b2', x)
    CheckGrad(model, NNForward, NNBackward, 'W1', x)
    CheckGrad(model, NNForward, NNBackward, 'b1', x)


    # Train model.
    stats = Train(model, NNForward, NNBackward, NNUpdate, eps,
                  momentum, num_epochs, batch_size)

    # Uncomment if you wish to save the model.
    # Save(model_fname, model)
```

```
    # Uncomment if you wish to save the training statistics.
    # Save(stats_fname, stats)

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