Xiang Chen 1004704992
Instructor's Name: Chris Maddison
October 13, 2020

CSC311 Assignment 2

1. [4pts] Feature Maps.

(a) [2pts]

From the 1-D dataset, we can see there are only 3 elements and when $x = -1$ and $x = 3$, it should be in a positive region since it has value with 1. Let us follow page 8 of course note, Linear Classifiers.

Let assume this dataset is linearly separable and there exists λ_1 and λ_2 that satisfy $x_2 = \lambda_1 x_1 + \lambda_2 x_3$, and $\lambda_1 + \lambda_2 = 1$.

We can easily solve the two equations and get $\lambda_1 = \frac{1}{2}$ and $\lambda_2 = \frac{1}{2}$. Therefore, based on the page 8 of course note, Linear Classifiers, x_2 should also be in positive region. However, x_2 in the 1-D dataset is 0, which is a contradiction.

Therefore, this dataset is not linearly separable.

QED

(b) [2pts]

$\psi_1(x)$	ψ_2	t
-1	1	1
1	1	0
3	9	1

We could get the the following inequalities.

$$-w_1 + w_2 \geq 0$$
$$w_1 + w_2 < 0$$
$$w_1 + 3w_2 \geq 0$$

We can easily get one value based on the three inequalities.

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} -5 \\ 3 \end{pmatrix}$$

2. [22pts] kNN vs. Logistic Regression.

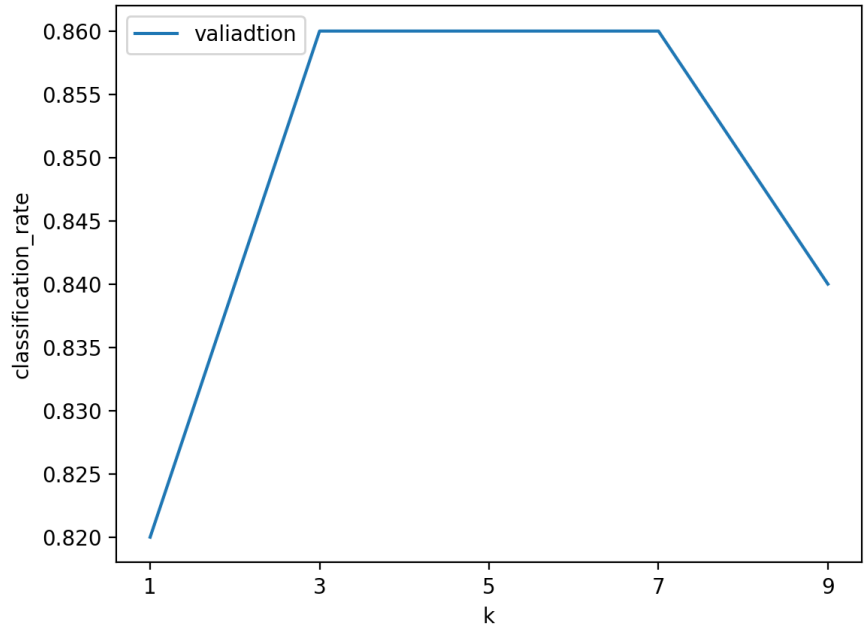
2.1. k-Nearest Neighbors.

(a) **[2pts]**

This is the function for run_knn.

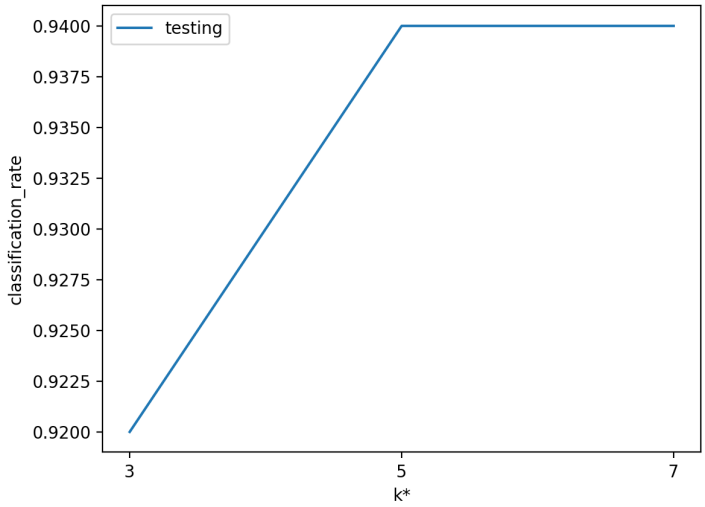
```
def run_knn():
    train_inputs, train_targets = load_train()
    valid_inputs, valid_targets = load_valid()
    test_inputs, test_targets = load_test()

    #####################################################################
    # TODO:                                                             #
    # Implement a function that runs kNN for different values of k,     #
    # plots the classification rate on the validation set, and etc.     #
    #####################################################################
    class_list = []
    k_sets = [1,3,5,7,9]
    for k in k_sets:
        prediction = knn(k, train_inputs, train_targets, valid_inputs)
        counter = 0
        for i in range(prediction.size):
            if(prediction[i]==valid_targets[i]): counter += 1
        class_rate = counter / valid_targets.size
        class_list.append(class_rate)

    plt.plot(k_sets,class_list,label="valiadtion")
    plt.xlabel("k")
    plt.ylabel("classification_rate")
    plt.xticks(k_sets)
    plt.legend()
    plt.show()
    k_test = [1,3,5]
```

This is the plot for $k \in \{1,3,5,7,9\}$ with kNN algorithms.



(b) [2pts]

I choose $k^* = 5$ since based on the graph above we could see 3 has the best performance , and compute the set of {3,5,7}. This is the graph for $k^* \in \{3,5,7\}$. We could see when $k^* = 3$, it has classification rate of 0.92; when $k^* = 5$, it has classification rate of 0.94; and when $k^* = 7$, it has classification rate of 0.94. For the performance, $k^* = 5$ & 7 actually perform better than $k^* = 3$, which is different from the validation set.



2.2. Logistic Regression.

(a) [4pts]

Here is the code for function logistic_predict.

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

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

    :param weights: A vector of weights with dimension (M + 1) x 1, where
    the last element corresponds to the bias (intercept).
    :param data: A matrix with dimension N x M, where each row corresponds to
    one data point.
    :return: A vector of probabilities with dimension N x 1, which is the output
    to the classifier.
    """
    ###########################################################################
    # TODO:                                                                   #
    # Given the weights and bias, compute the probabilities predicted   #
    # by the logistic classifier.                                             #
    ###########################################################################
    N, M = data.shape
    data_buff = np.concatenate((data, np.ones((N, 1))), axis=1)
    y = np.dot(data_buff, weights)
    y = sigmoid(y)
    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################
    return y
```

Here is the code for function evaluate.

```
def evaluate(targets, y):
    """ Compute evaluation metrics.

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

    :param targets: A vector of targets with dimension N x 1.
    :param y: A vector of probabilities with dimension N x 1.
    :return: A tuple (ce, frac_correct)
        WHERE
        ce: (float) Averaged cross entropy
        frac_correct: (float) Fraction of inputs classified correctly
    """
    ###########################################################################
    # TODO:                                                                   #
    # Given targets and probabilities predicted by the classifier,           #
    # return cross entropy and the fraction of inputs classified             #
    # correctly.                                                              #
    ###########################################################################
    counter = 0
    for i in range(targets.size):
        if (y[i]>=0.5 and targets[i]==1) or (y[i]<0.5 and targets[i]==0):
            counter += 1
    ce = (-np.dot(targets.T, np.log(y)) - (np.dot((1-targets).T, np.log(1-y)))) /
        float(targets.size)

    frac_correct = counter / float(len(targets))
    ###########################################################################
    #                           END OF YOUR CODE                              #
    ###########################################################################
    return ce[0], frac_correct
```

Here is the code for function logistic.

```

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

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

    :param weights: A vector of weights with dimension (M + 1) x 1, where
    the last element corresponds to the bias (intercept).
    :param data: A matrix with dimension N x M, where each row corresponds to
    one data point.
    :param targets: A vector of targets with dimension N x 1.
    :param hyperparameters: The hyperparameter dictionary.
    :returns: A tuple (f, df, y)
        WHERE
        f: The average of the loss over all data points.
           This is the same as averaged cross entropy.
           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.
    """
    y = logistic_predict(weights, data)

    ###########################################################################
    # TODO:                                                                   #
    # Given weights and data, return the averaged loss over all data          #
    # points, gradient of parameters, and the probabilities given by          #
    # logistic regression.                                                    #
    ###########################################################################
    # Hint: hyperparameters will not be used here.
    f, frac_correct = evaluate(targets, y)
    N, M = data.shape
    data_buff = np.concatenate((data, np.ones((N, 1))), axis=1)
    df = np.dot(data_buff.T, y - targets) / float(N)
    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################
    return f, df, y
```

(b) [5pts]

Here is the code for function run_logistic_regression.

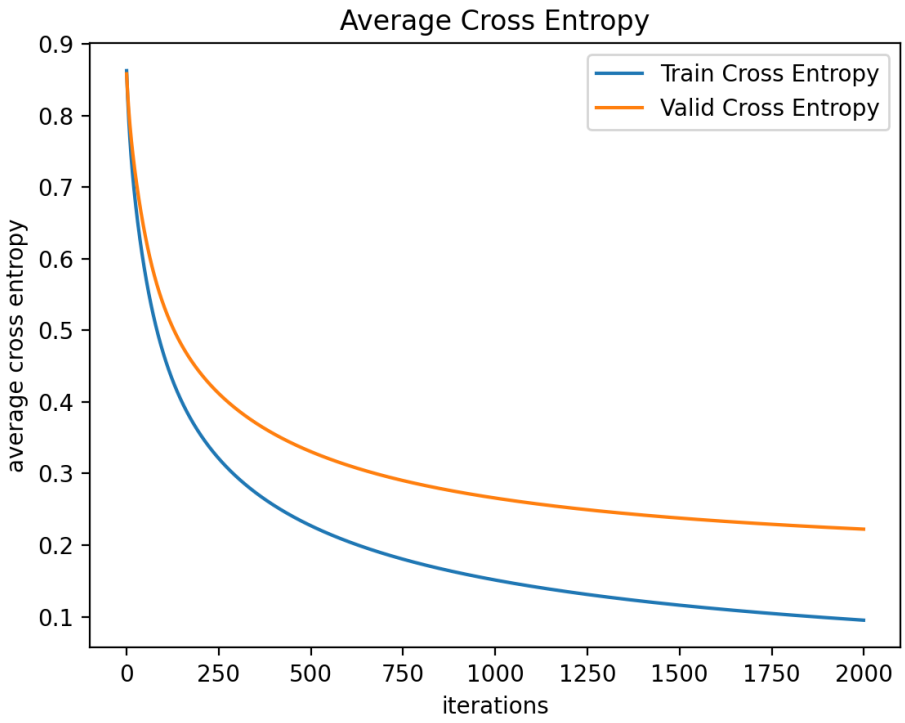
```
def run_logistic_regression():
    train_inputs, train_targets = load_train()
    #train_inputs, train_targets = load_train_small()
    valid_inputs, valid_targets = load_valid()

    N, M = train_inputs.shape

    #####################################################################
    # TODO:                                                             #
    # Set the hyperparameters for the learning rate, the number         #
    # of iterations, and the way in which you initialize the weights.   #
    #####################################################################
    hyperparameters = {
        "learning_rate": 0.001,
        "weight_regularization": 0.,
        "num_iterations": 2000
    }

    #weights = np.random.uniform(-1,1,(M + 1, 1))
    weights = np.random.randn(len(train_inputs[1])+1, 1) * 0.1
    #####################################################################
    #                       END OF YOUR CODE                            #
    #####################################################################

    # 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
    #####################################################################
    # TODO:                                                             #
    # Modify this section to perform gradient descent, create plots,    #
    # and compute test error.                                           #
    #####################################################################
    ce_trains = []
    ce_valids = []
    correct_trains = []
    correct_valids = []
    iterations = []
    for t in range(hyperparameters["num_iterations"]):
        f, df, y = logistic(weights, train_inputs, train_targets, hyperparameters)
        ce_train, frac_correct_train = evaluate(train_targets, y)
        ce_trains.append(ce_train)
        correct_trains.append(frac_correct_train)

        weights = weights - hyperparameters['learning_rate'] * df
        y_valid = logistic_predict(weights, valid_inputs)
        ce_valid, frac_correct_valid = evaluate(valid_targets, y_valid)
        ce_valids.append(ce_valid)
        correct_valids.append(frac_correct_valid)

        iterations.append(t)

    plt.plot(iterations,ce_trains,label="Train Cross Entropy")
    plt.plot(iterations,ce_valids,label="Valid Cross Entropy")
    plt.xlabel("iterations")
    plt.ylabel("average cross entropy")
    plt.title("Average Cross Entropy")
    plt.legend()
    plt.show()

    final_f, final_df, final_y = logistic(weights, train_inputs, train_targets, hyperparameters)
    final_ce_train, final_frac_correct_train = evaluate(train_targets, final_y)
    final_y_valid = logistic_predict(weights, valid_inputs)
    final_ce_valid, final_frac_correct_valid = evaluate(valid_targets, final_y_valid)

    print("CE: Train %.5f Validation %.5f" %
          (final_ce_train, final_ce_valid))
    print("Acc: Train {:.5f} Validation {:.5f}".format(
    1-final_frac_correct_train, 1-final_frac_correct_valid))
```

When $weights = np.random.randn(len(train_inputs[1]) + 1,1) * 0.1$, I have tried learning rate from 0.001 to 1.0, and I find for mnist_train set when learning rate is **0.008** and iterations is **2000**, my model has the relative small cross entry and high accuracy for the validation set. In this situation, it has train cross entropy 0.09859, validation cross entropy 0.22127, and **test cross entropy 0.21343**; train classification error 0.00000, validation classification error 0.08000 and **test classification error 0.08000**.

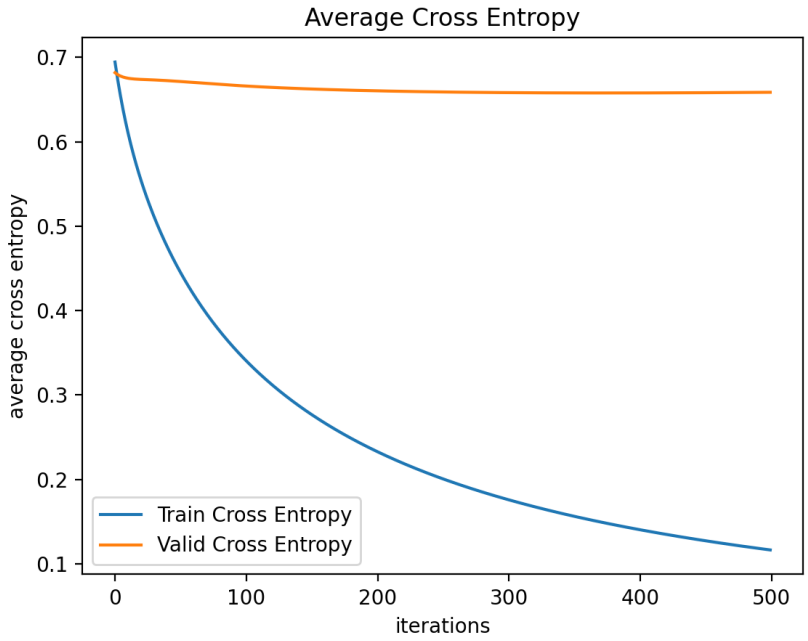
When $weights = np.random.randn(len(train_inputs[1]) + 1,1) * 0.1$, I have tried learning rate from 0.001 to 1.0, and I find for mnist_train_small set when learning rate is 0.004 and iterations is 500, my model has the relative small cross entry and high accuracy for the validation set. In this situation, it has train cross entropy 0.11595, validation cross entropy 0.67714, and **test cross entropy 0.54535**; train classification error 0.00000, validation classification error 0.38000 and **test classification error 0.24000**.

(c) [2pts]

Here is the Cross Entropy graph for mnist_train with learning rate 0.008 and num_iterations 2000.



Here is the Cross Entropy graph for mnist_train_small with learning rate 0.004 and num_iterations 500.



When I re-run for many times, I find the results actually have difference with others. I think this is may because I have the random weights as the input.

For best hyper-parameters, I think we can have a small learning rate and more iterations based on the reference from sample size, since we do not want to overfit and under-fit. We can also make a leaning model or a heuristic search algorithm to choose the best parameters for us. Since based on our observation, we should choose the learning rate not too small and not too big. Therefore, we could initialize an original value of λ, like 0.001 (enough small) and we make an upper bound value for λ, like 1 (enough big) and also the upper bound for iterations (based on the running time limitation). We use binary search method to break the interval and make our new λ. With everything else fixed, we can have our best lambda first by narrowing down the interval for λ, and when for the next update, the accuracy for validation set with the difference smaller than 0.0000001, which means we have get the best lambda. To use simple words, we build a new model to train or heuristic search to get the best hyper-parameters.

2.3. Penalized logistic regression.

(a) [4pts]

Here is the code for function logistic_pen.

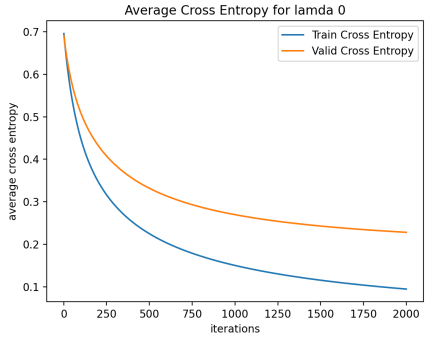
```
104
105 def logistic_pen(weights, data, targets, hyperparameters):
106     """ Calculate the cost of penalized logistic regression and its derivatives
107     with respect to weights. Also return the predictions.
108
109     Note: N is the number of examples
110           M is the number of features per example
111
112     :param weights: A vector of weights with dimension (M + 1) x 1, where
113     the last element corresponds to the bias (intercept).
114     :param data: A matrix with dimension N x M, where each row corresponds to
115     one data point.
116     :param targets: A vector of targets with dimension N x 1.
117     :param hyperparameters: The hyperparameter dictionary.
118     :returns: A tuple (f, df, y)
119         WHERE
120         f: The average of the loss over all data points, plus a penalty term.
121            This is the objective that we want to minimize.
122         df: (M+1) x 1 vector of derivative of f w.r.t. weights.
123         y: N x 1 vector of probabilities.
124     """
125     y = logistic_predict(weights, data)
126
127     ##################################################################
128     # TODO:                                                          #
129     # Given weights and data, return the averaged loss over all data    #
130     # points (plus a penalty term), gradient of parameters, and the     #
131     # probabilities given by penalized logistic regression.          #
132     ##################################################################
133     N, M = data.shape
134     lambd = hyperparameters["weight_decay"]
135     w = weights[:-1]
136     y = logistic_predict(weights, data)
137     ce, frac_correct = evaluate(targets, y)
138     f = ce + (lambd/2) * np.sum(w * w)
139     data_buff = np.concatenate((data, np.ones((N, 1))), axis=1)
140     weight_buff = np.copy(weights)
141     weight_buff[-1] = 0
142     df = np.dot(data_buff.T,y - targets) / float(N) + lambd * weight_buff
143     ##################################################################
144     #                       END OF YOUR CODE                         #
145     ##################################################################
146     return f, df, y
```

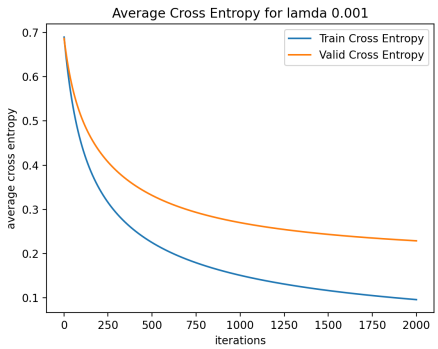
(b) [5pts]

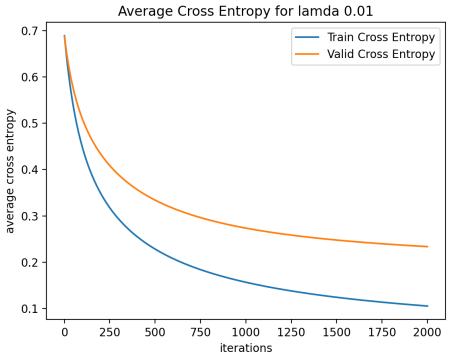
Here is the code for function run_pen_logistic_regression.

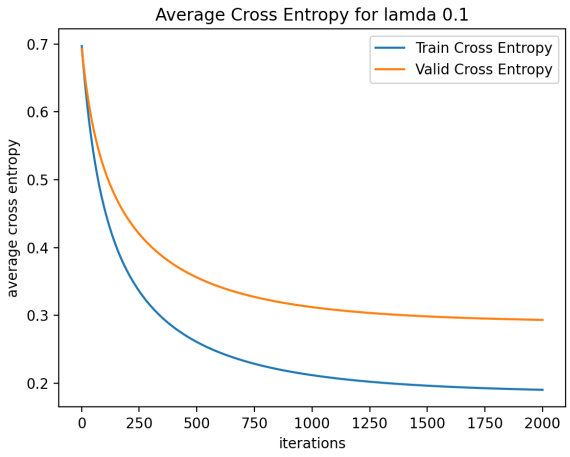
```
def run_pen_logistic_regression():
    train_inputs, train_targets = load_train()
    #train_inputs, train_targets = load_train_small()
    valid_inputs, valid_targets = load_valid()
    N, M = train_inputs.shape
    hyperparameters = {
        "learning_rate": 0.004,
        "weight_decay": 0.,
        "num_iterations": 500
    }
    lambd_list = [0, 0.001, 0.01, 0.1, 1.0]
    run_check_grad(hyperparameters)
    run_time = 5
    for l in range(len(lambd_list)):
        hyperparameters["weight_decay"] = lambd_list[l]
        ce_trains = np.zeros(hyperparameters["num_iterations"])
        ce_valids = np.zeros(hyperparameters["num_iterations"])
        correct_trains = np.zeros(hyperparameters["num_iterations"])
        correct_valids = np.zeros(hyperparameters["num_iterations"])
        iterations = [i for i in range(1,hyperparameters["num_iterations"]+1)]
        for r in range(run_time):
            weights = np.random.randn(len(train_inputs[1]) + 1, 1) * 0.01

            for t in range(hyperparameters["num_iterations"]):
                f, df, y = logistic_pen(weights, train_inputs, train_targets, hyperparameters)
                ce_train, frac_correct_train = evaluate(train_targets, y)
                ce_trains[t] += (ce_train / run_time)
                correct_trains[t] += (frac_correct_train / run_time)
                weights = weights - hyperparameters["learning_rate"] * df
                y_valid = logistic_predict(weights, valid_inputs)
                ce_valid, frac_correct_valid = evaluate(valid_targets, y_valid)
                ce_valids[t] += (ce_valid / run_time)
                correct_valids[t] += (frac_correct_valid / run_time)
        train_ce_avg = np.mean(ce_trains, axis=0)
        valid_ce_avg = np.mean(ce_valids, axis=0)
        train_crList_avg = np.mean(correct_trains, axis=0)
        valid_cr_avg = np.mean(correct_valids, axis=0)
        print("AVG_CE: Train %.5f Validation %.5f" %
              (train_ce_avg, valid_ce_avg))
        print("AVG_ERROR: Train {:.5f} Validation {:.5f}".format(
        1-train_crList_avg, 1-valid_cr_avg))
        plt.plot(iterations,ce_trains,label="Train Cross Entropy")
        plt.plot(iterations,ce_valids,label="Valid Cross Entropy")
        plt.xlabel("iterations")
        plt.ylabel("average cross entropy")
        plt.title("Average Cross Entropy for lamda {}".format(lambd_list[l]))
        plt.legend()
        plt.show()
```

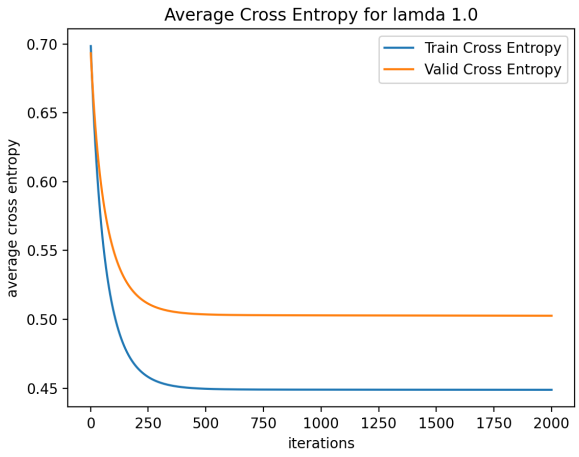
When $weights = np.random.randn(len(train_inputs[1]) + 1,1) * 0.01$, I have tried learning rate from 0.001 to 1.0, and I find for **mnist_train** set when learning rate is **0.008** and iterations is **2000**, my model has the relative small average cross entry and average high accuracy for the validation set. Here are the five graphs for **mnist_train**.



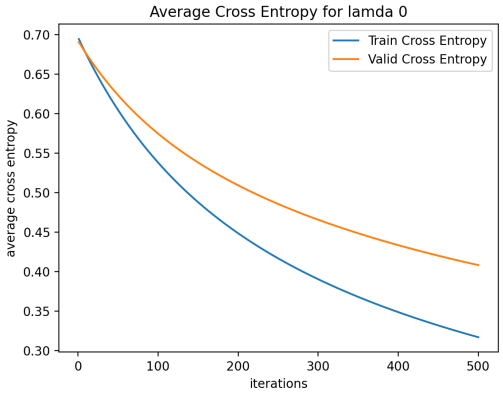


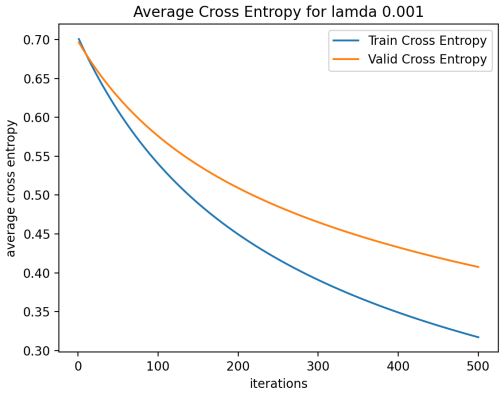


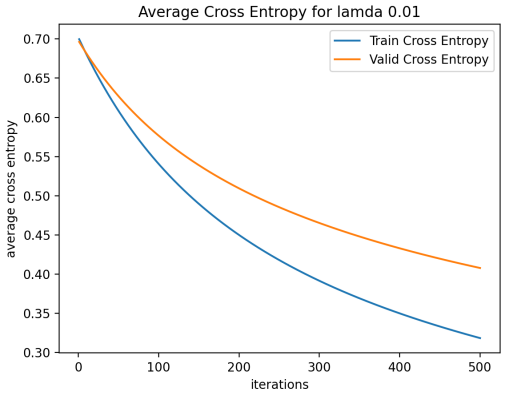


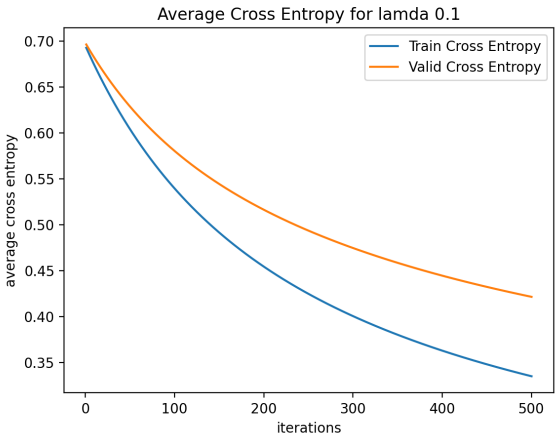


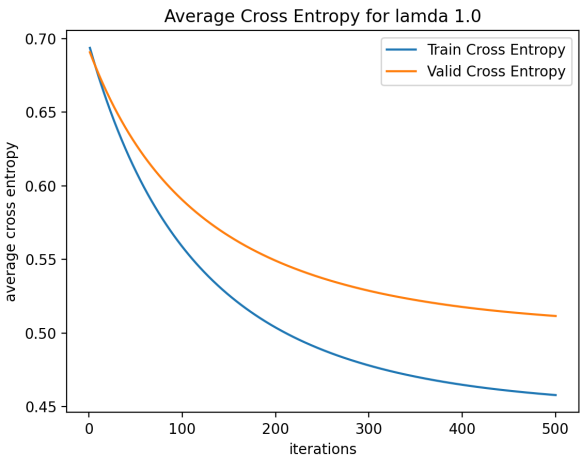
When $weights = np.random.randn(len(train_inputs[1]) + 1,1) * 0.1$, I have tried learning rate from 0.001 to 1.0, and I find for **mnist_train_small** set when learning rate is **0.004** and iterations is **500**, my model has the relative small average cross entry and average high accuracy for the validation set. Here are the five graphs for **mnist_train_small**.











For mnist_train set:

When $\lambda = 0$, AVG_CE: Train 0.19304 Validation 0.30552, AVG_ERROR: Train 0.02076 Validation 0.12610.

When $\lambda = 0.001$, AVG_CE: Train 0.19355 Validation 0.30526, AVG_ERROR: Train 0.02086 Validation 0.12752.

When $\lambda = 0.01$, AVG_CE: Train 0.19820 Validation 0.30720, AVG_ERROR: Train 0.02039 Validation 0.12503.

When $\lambda = 0.1$, AVG_CE: Train 0.24943 Validation 0.34385, AVG_ERROR: Train 0.02589 Validation 0.11812.

When $\lambda = 1.0$, AVG_CE: Train 0.45728 Validation 0.50975, AVG_ERROR: Train 0.06004 Validation 0.16590.

For mnist_train_small set:

When $\lambda = 0$, AVG_CE: Train 0.44554 Validation 0.50673, AVG_ERROR: Train 0.08954 Validation 0.20342.

When $\lambda = 0.001$, AVG_CE: Train 0.44201 Validation 0.50312, AVG_ERROR: Train 0.08196 Validation 0.18938.

When $\lambda = 0.01$, AVG_CE: Train 0.44539 Validation 0.50756, AVG_ERROR: Train 0.08659 Validation 0.20391.

When $\lambda = 0.1$, AVG_CE: Train 0.45131 Validation 0.51078, AVG_ERROR: Train 0.08669 Validation 0.20962.

When $\lambda = 1.0$, AVG_CE: Train 0.51341 Validation 0.55554, AVG_ERROR: Train 0.08876 Validation 0.21402.

(c) [2pts]

We can see when the lambda is greater , the graph reaches the coverage state faster. Therefore, I think the higher the lambda we use, the model will converge faster since for each time of gradient descent, we deduce more, which makes intuition for converge fast. We can also see regularization actually work better on large data set by comparing the performance on mnist_train set and mnist_train_small. I will choose $\lambda = 0.001$ based on my experiments.

When $\lambda = 0.001$, CE: Test 0.21977, ERROR: Test 0.08000, Classification Rate: Test 0.92000.

3. [15pts] Neural Networks.

(a) [4 pts]

Code for affine_backward.

```python
def affine_backward(grad_y, x, 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
    :param grad_y: Gradient from upper layer
    :param x: Inputs from the hidden layer
    :param w: Weights
    :return: A tuple of (grad_h, grad_w, grad_b)
        WHERE
        grad_x: Gradients wrt. the inputs/hidden layer.
        grad_w: Gradients wrt. the weights.
        grad_b: Gradients wrt. the biases.
    """
    ###########################################################################
    # TODO:                                                                   #
    # Complete the function to compute the gradients of affine          #
    # transformation.                                                         #
    ###########################################################################
    grad_x = grad_y.dot(w.T)
    grad_w = (x.T).dot(grad_y)
    grad_b = np.sum(grad_y, axis=0)
    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################
    return grad_x, grad_w, grad_b
```

Code for relu_backward.

```python
def relu_backward(grad_y, x):
    """ Computes gradients of the ReLU activation function wrt. the unactivated inputs.
    :param grad_y: Gradient of the activation.
    :param x: Inputs
    :return: Gradient wrt. x
    """
    ###########################################################################
    # TODO:                                                                   #
    # Complete the function to compute the gradients of relu.           #
    ###########################################################################
    grad_x = grad_y * (x > 0).astype(float)
    ###########################################################################
    #                       END OF YOUR CODE                                  #
    ###########################################################################
    return grad_x
```

Code for nn_update.

```
159
160 def nn_update(model, alpha):
161     """ Update NN weights.
162     :param model: Dictionary of all the weights.
163     :param alpha: Learning rate
164     :return: None
165     """
166     ###########################################################################
167     # TODO:                                                                   #
168     # Complete the function to update the neural network's parameters.        #
169     # Your code should look as follows                                        #
170     # model["W1"] = ...                                                       #
171     # model["W2"] = ...                                                       #
172     # ...                                                                     #
173     ###########################################################################
174     model["W1"] -= alpha * model["dE_dW1"]
175     model["W2"] -= alpha * model["dE_dW2"]
176     model["W3"] -= alpha * model["dE_dW3"]
177     model["b1"] -= alpha * model["dE_db1"]
178     model["b2"] -= alpha * model["dE_db2"]
179     model["b3"] -= alpha * model["dE_db3"]
180     ###########################################################################
181     #                         END OF YOUR CODE                                #
182     ###########################################################################
183     return
```

(b) [2 pts]

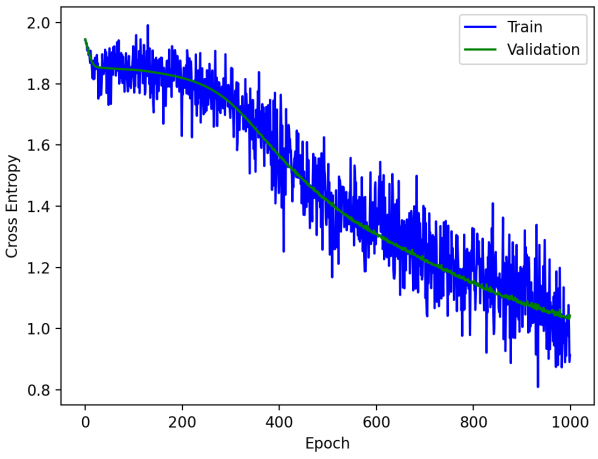
Graph for hyperparameters:

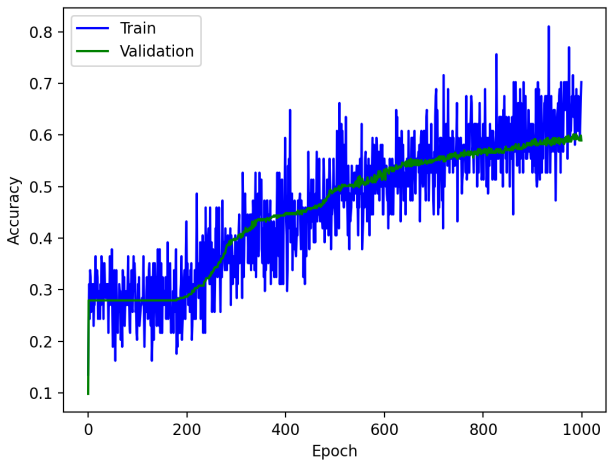
num_hiddens: [16, 32] alpha: 0.001

num_epochs: 1000 batch_size: 100

CE: Train 1.14061 Validation 1.18031 Test 1.16097

Acc: Train 0.60136 Validation 0.58711 Test 0.56883





(c) [3 pts]

5 different value for α:

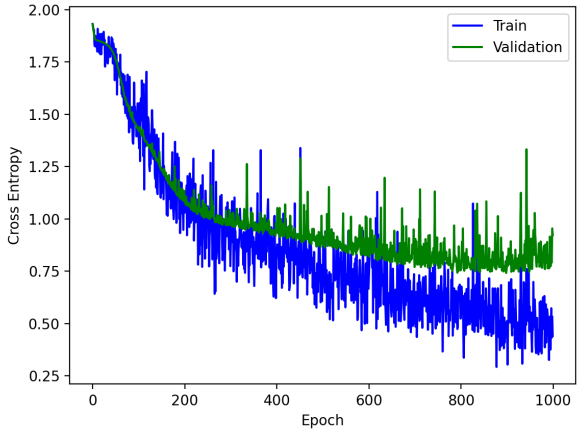
Graph for hyperparameters:

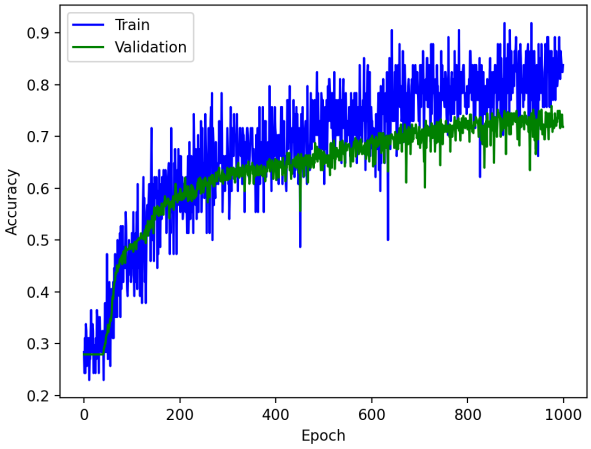
num_hiddens: [16, 32] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.58550 Validation 0.92216 Test 0.94877

Acc: Train 0.77830 Validation 0.71838 Test0.69610





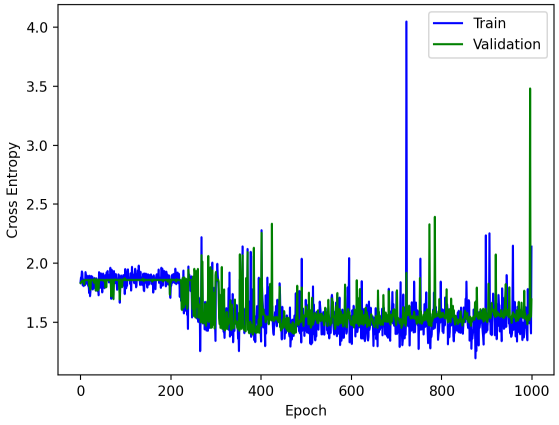
Graph for hyperparameters:

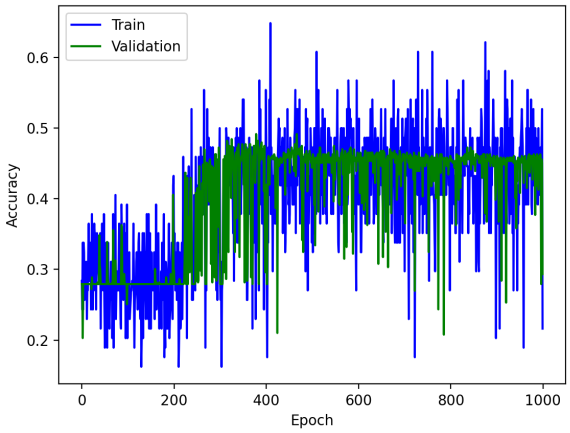
num_hiddens: [16, 32] alpha: 0.251

num_epochs: 1000 batch_size: 100

CE: Train 1.75893 Validation 1.69451 Test 1.75987

Acc: Train 0.28423 Validation 0.29356 Test 0.27273





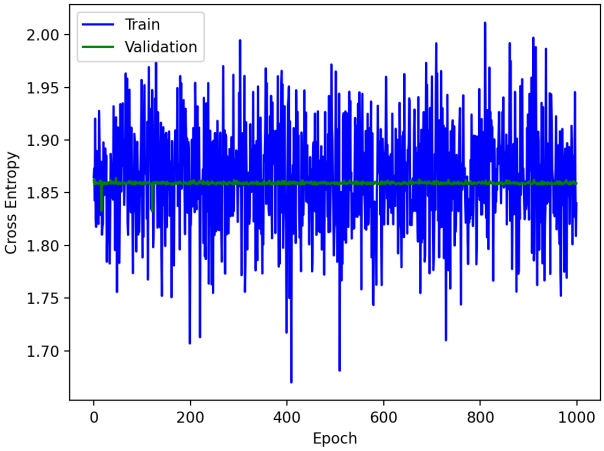
Graph for hyperparameters:

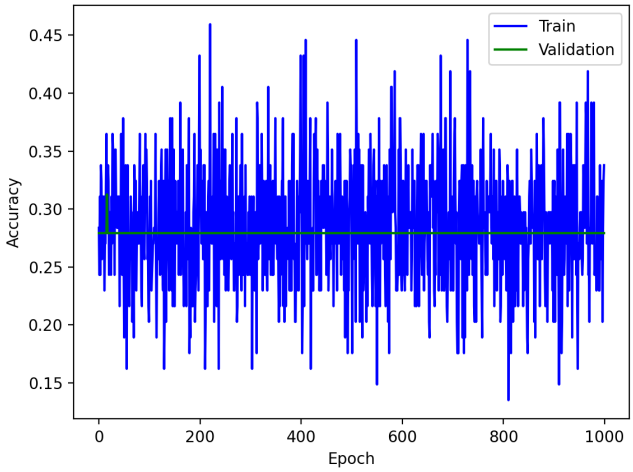
num_hiddens: [16, 32] alpha: 0.5

num_epochs: 1000 batch_size: 100

CE: Train 1.86108 Validation 1.85905 Test 1.83904

Acc: Train 0.28542 Validation 0.27924 Test 0.31688





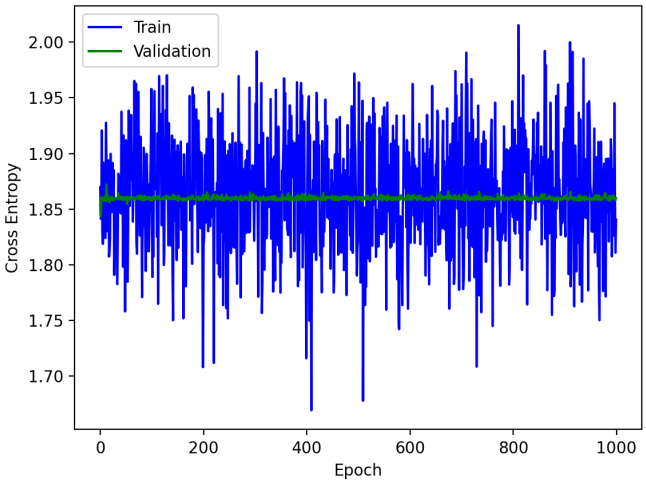
Graph for hyperparameters:

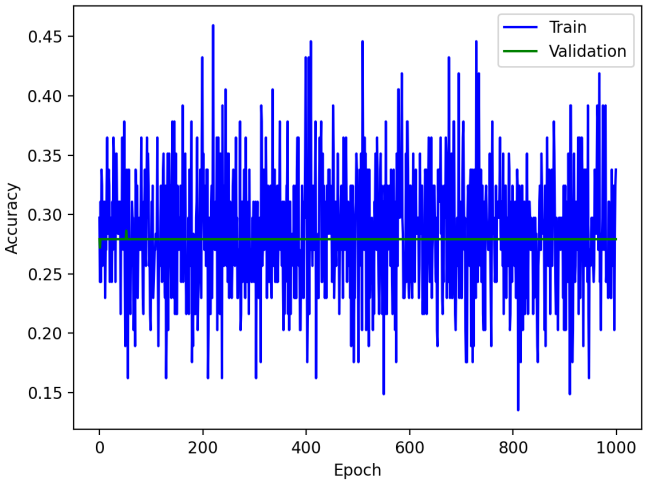
num_hiddens: [16, 32] alpha: 0.75

num_epochs: 1000 batch_size: 100

CE: Train 1.86168 Validation 1.85989 Test 1.83961

Acc: Train 0.28542 Validation 0.27924 Test 0.31688





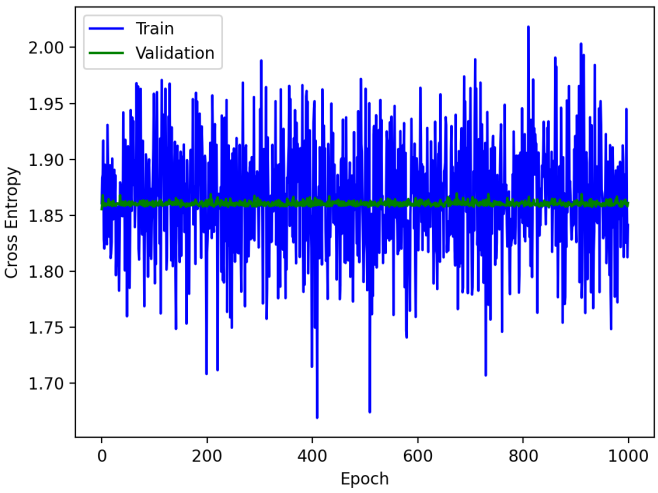
Graph for hyperparameters:

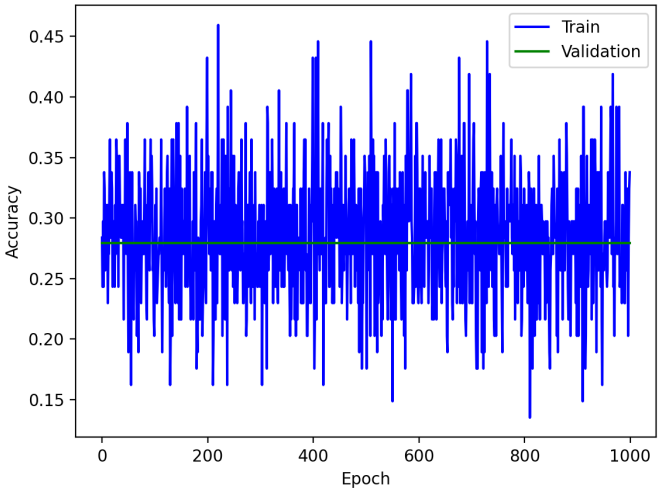
num_hiddens: [16, 32] alpha: 1.0

num_epochs: 1000 batch_size: 100

CE: Train 1.86245 Validation 1.86104 Test 1.84036

Acc: Train 0.28542 Validation 0.27924 Test 0.31688





When we increase the value of α, we can see both graphs become faster to converge at some level. We can also see that when we use α with large value, our model actually perform worse. We can see the cross entropy for train set when $\alpha = 0.005$ is 0.58550 and the cross entropy for train set when $\alpha = 1$ is 1.86245. This is because our model is too complex and cause the overfit, which will have less accuracy judgement for the new elements. We can make some intuitions on this. If you are training an image classifier, which is the same with this question, you want for similar image, they should have similar outputs and loss. However, if the loss changes rapidly, the model will have less accuracy for unseen inputs. We should choose a reasonable α, which is not too large or not too small.

5 different value for mini-batch:

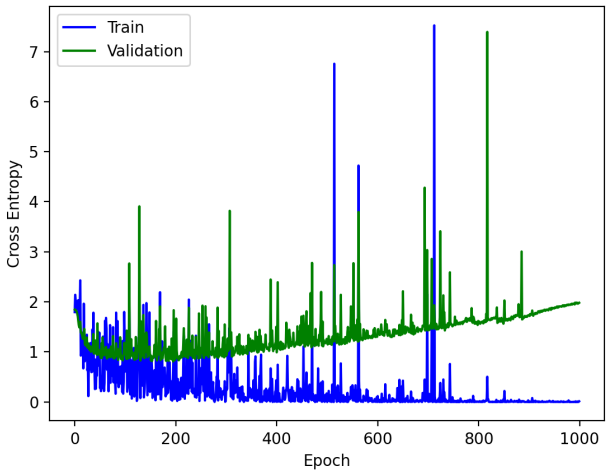
Graph for hyperparameters:

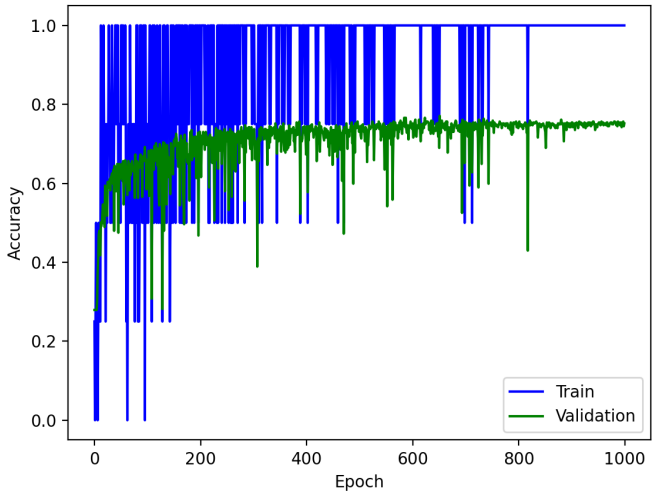
num_hiddens: [16, 32] alpha: 0.005

num_epochs: 1000 batch_size: 10

CE: Train 0.00392 Validation 1.98192 Test 1.60328

Acc: Train 1.00000 Validation 0.75179 Test 0.76623





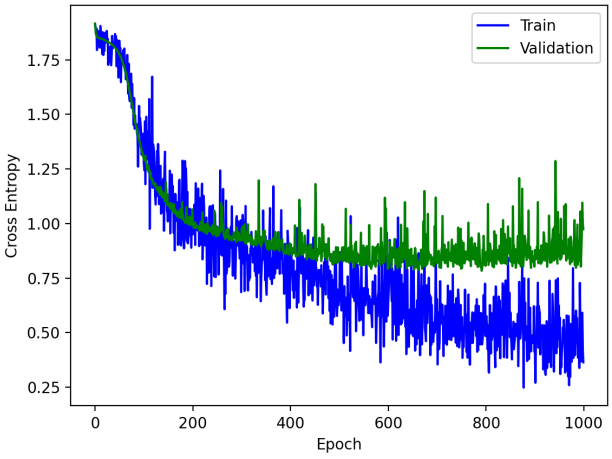
Graph for hyperparameters:

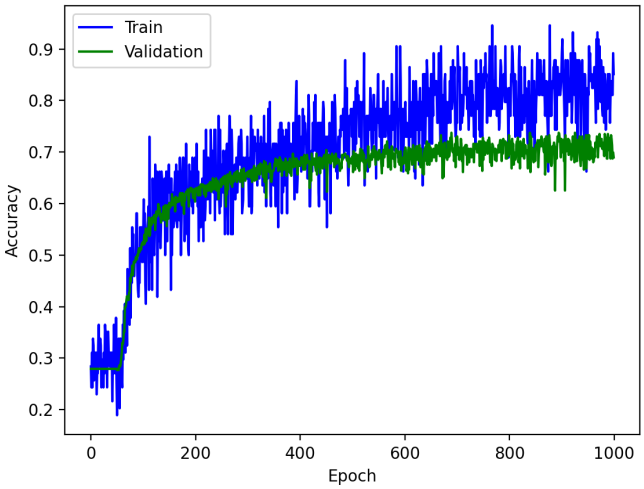
num_hiddens: [16, 32] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.50069 Validation 0.97403 Test 0.86342

Acc: Train 0.80854 Validation 0.68974 Test 0.70649





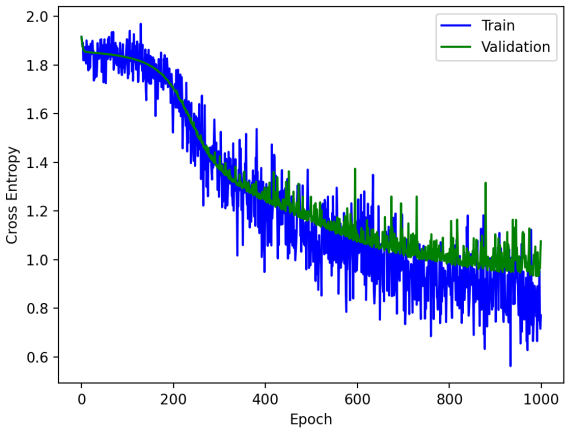
Graph for hyperparameters:

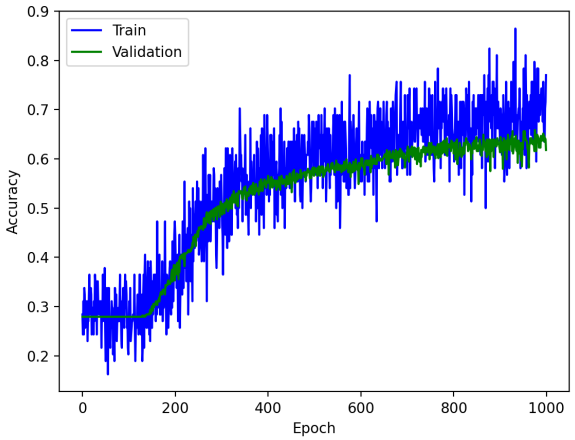
num_hiddens: [16, 32] alpha: 0.005

num_epochs: 1000 batch_size: 300

CE: Train 0.92200 Validation 1.07522 Test 0.98883

Acc: Train 0.65857 Validation 0.61814 Test 0.62597





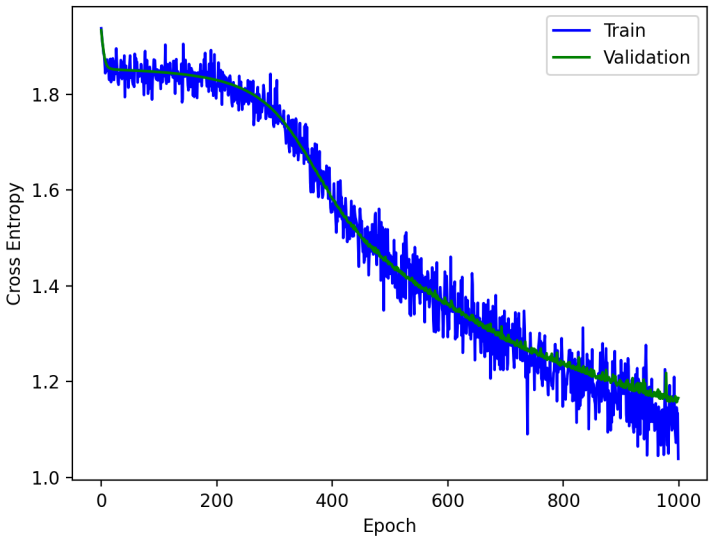
Graph for hyperparameters:

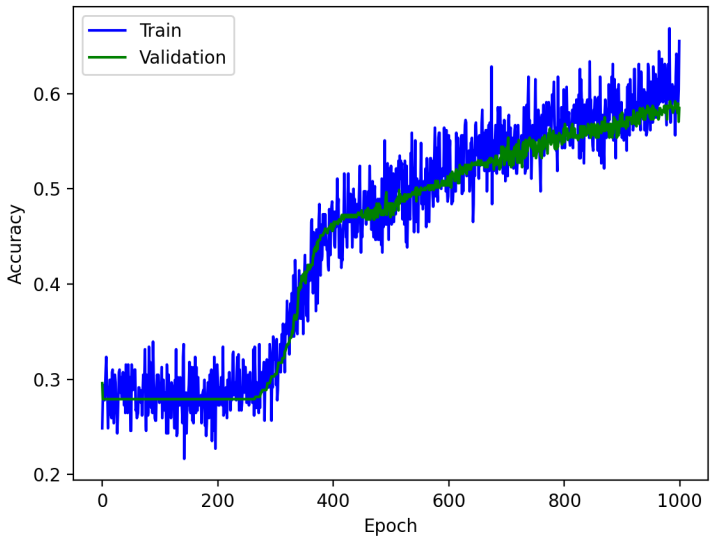
num_hiddens: [16, 32] alpha: 0.005

num_epochs: 1000 batch_size: 600

CE: Train 1.11538 Validation 1.16601 Test 1.13714

Acc: Train 0.59988 Validation 0.58473 Test 0.58182





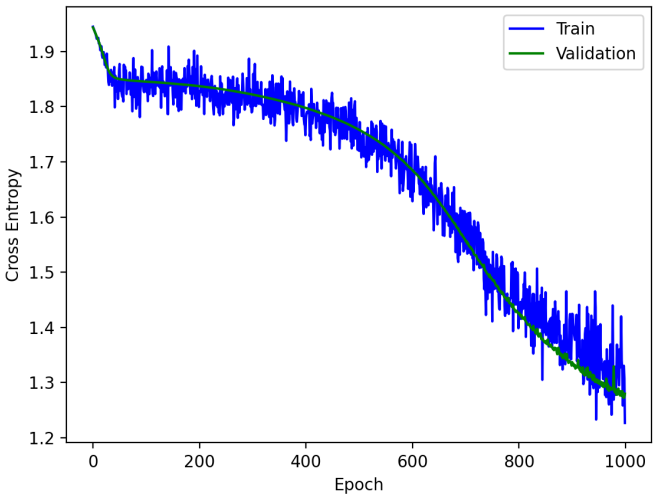
Graph for hyperparameters:

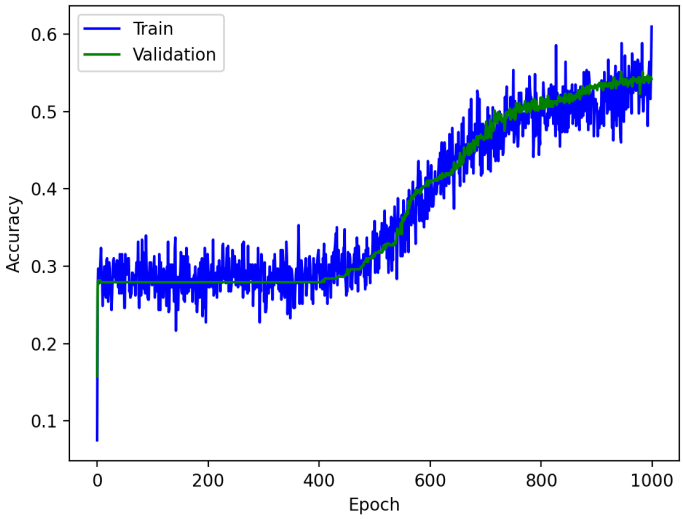
num_hiddens: [16, 32] alpha: 0.005

num_epochs: 1000 batch_size: 1000

CE: Train 1.31117 Validation 1.27990 Test 1.33751

Acc: Train 0.53468 Validation 0.54177 Test 0.53247





When we increase the size of mini-batch, we can see the slope for both graphs become steep. In general, when we have a larger mini-batch, we will get coverage slower, and when we have a really large mini-batch, it may need a huge amount of iterations to achieve converge. When we have a small batch size, which means for each time, we only train our model with limit inputs, we can see the graph with batch_size is 10, the graph even can't converge to some level, and we when have model with proper size, like graphs which have batch_size as 100, it can converge fast (1000 iterations).

For me, I personally will choose $\alpha = 0.01$ and $batch\_size = 64$, since I have tried over 50 α from 0.001 to 1.0 and over 20 $batch\_size$ from 4 to 128, and I get the best model in this situation, with CE: Train 0.07843 Validation 1.02213 Test 0.78184, Acc: Train 0.98637 Validation 0.75656 Test 0.78182.

Actually, we can have better parameters by writing a learning model or heuristic searching algorithms like what I have mentioned in 2.2 c.

(d) [3 pts]

3 different value for hidden units (6 graphs):

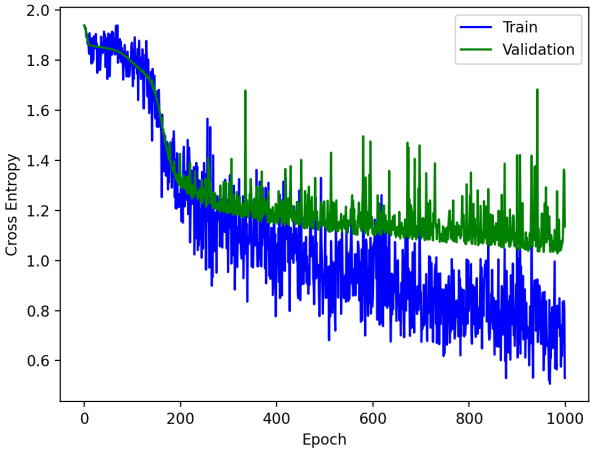
Graph for hyperparameters:

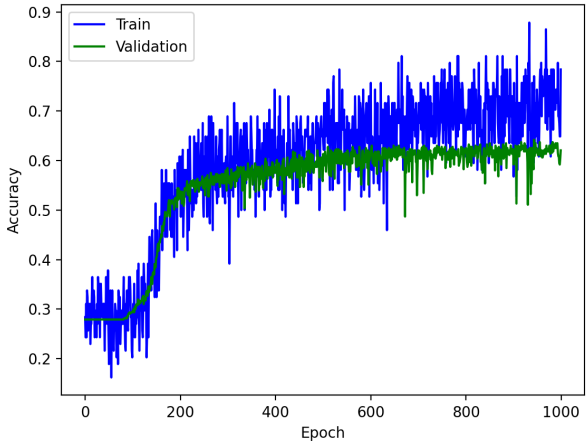
num_hiddens: [16, 4] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.72322 Validation 1.13538 Test 1.00634

Acc: Train 0.71606 Validation 0.62053 Test 0.61558





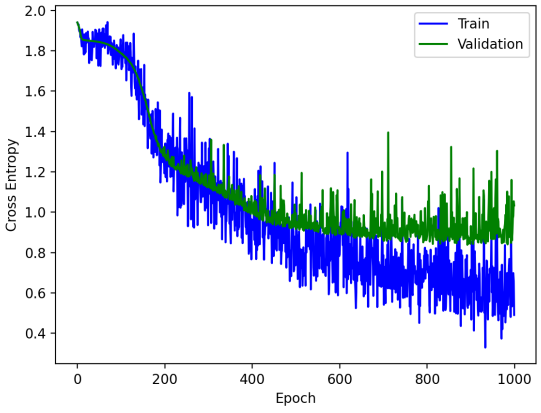
Graph for hyperparameters:

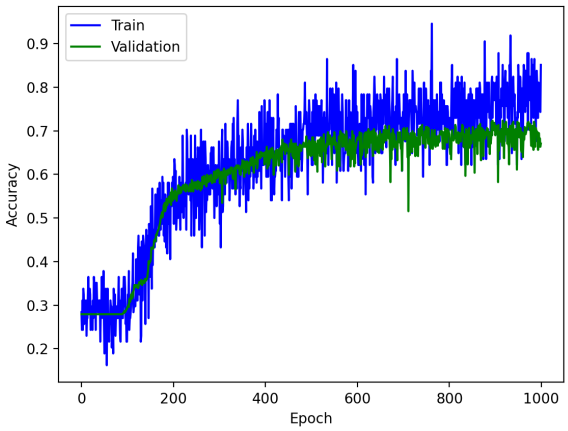
num_hiddens: [16, 8] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.66838 Validation 1.03362 Test 1.00633

Acc: Train 0.74481 Validation 0.67064 Test 0.67532





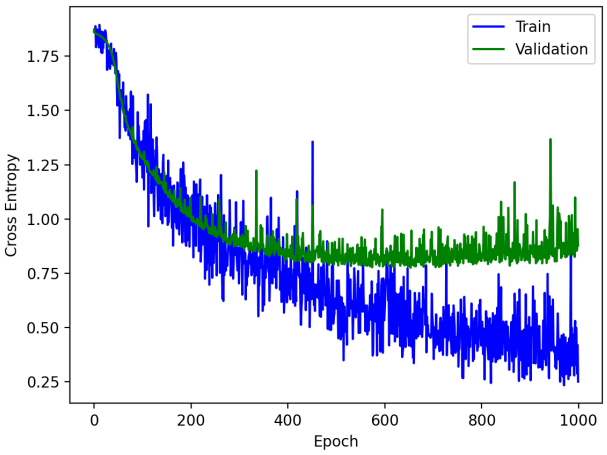
Graph for hyperparameters:

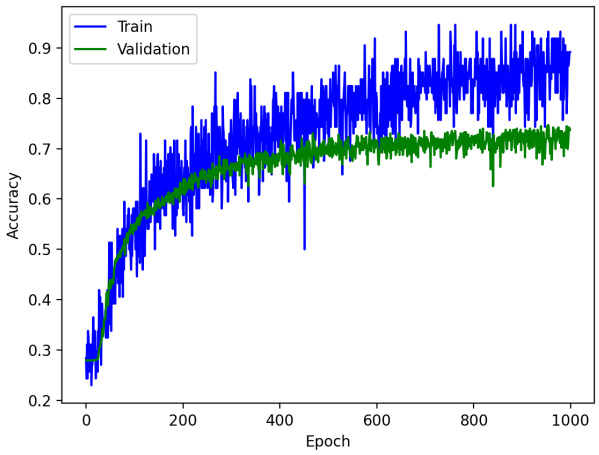
num_hiddens: [16, 64] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.37402 Validation 0.87759 Test 0.82770

Acc: Train 0.86100 Validation 0.73747 Test 0.74286





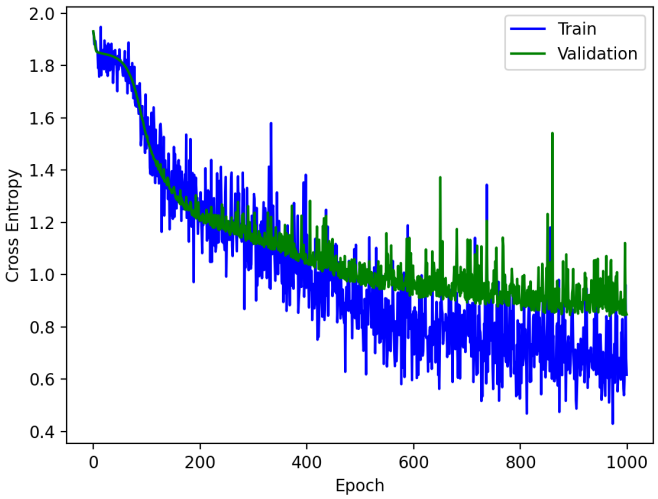
Graph for hyperparameters:

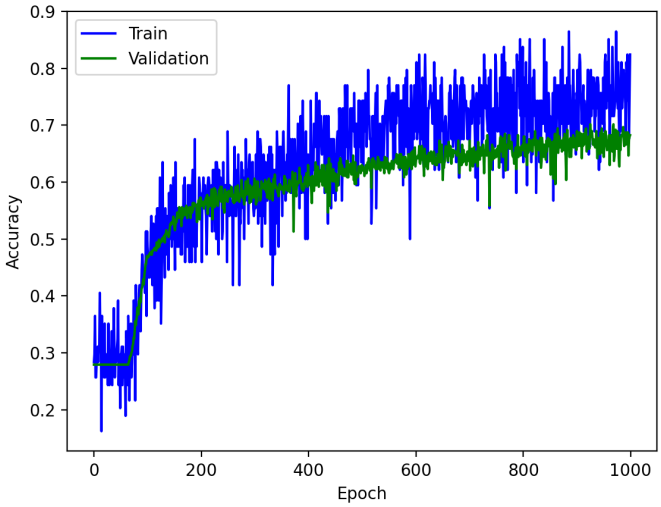
num_hiddens: [8, 16] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.61184 Validation 0.84747 Test 0.85185

Acc: Train 0.77356 Validation 0.68258 Test 0.68571





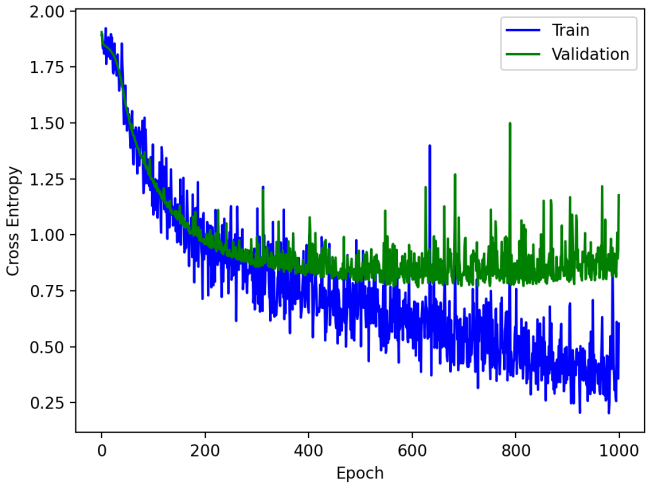
Graph for hyperparameters:

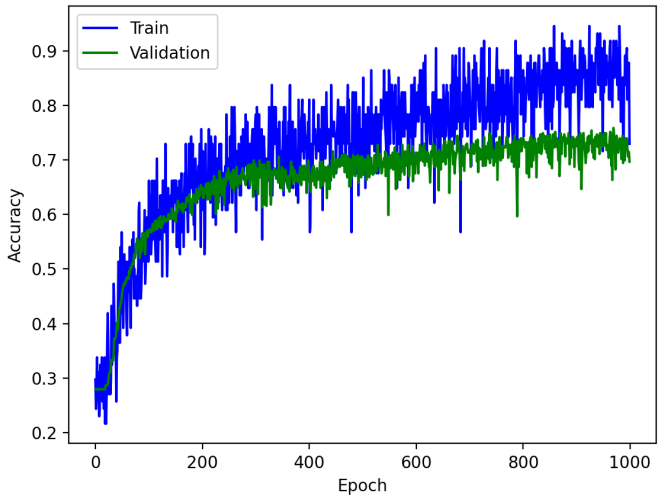
num_hiddens: [32, 16] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.59045 Validation 1.17669 Test 1.02425

Acc: Train 0.79787 Validation 0.69690 Test 0.68831





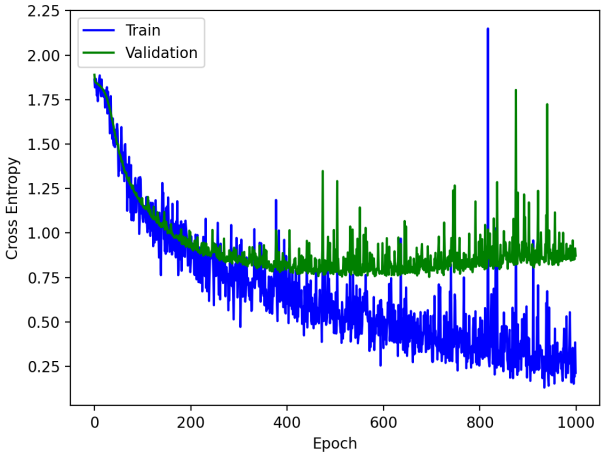
Graph for hyperparameters:

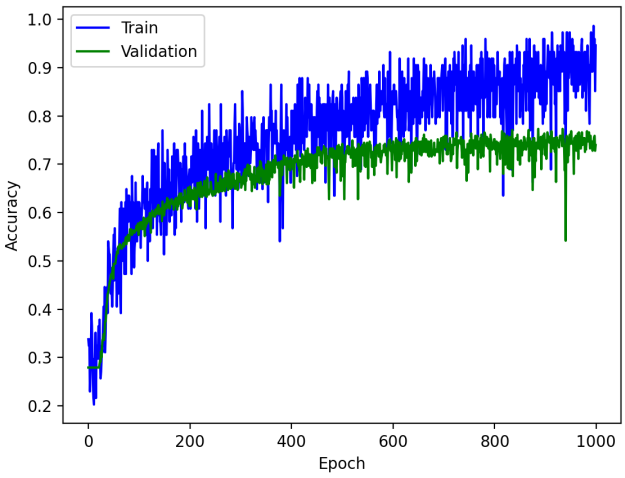
num_hiddens: [64, 16] alpha: 0.005

num_epochs: 1000 batch_size: 100

CE: Train 0.26999 Validation 0.87218 Test 0.86586

Acc: Train 0.90694 Validation 0.73986 Test 0.71429





Based on the above six graphs, we can see when the number of hidden units become larger, the graph can converge faster. We can compare the model has [16, 4] as num_hiddens and the model has [16, 64] as num_hiddens. We can see graphs with [16, 64] as num_hiddens converges much faster than graphs with [16, 4] as num_hidden. We can also compare the model has [8, 16] as num_hiddens and the model has [64, 16] as num_hiddens. We can see graphs with [64, 16] as num_hiddens converges much faster than graphs with [8, 16] as num_hidden. Therefore, we can get the conclusion that more hidden units will help our model converge faster. However, I think it is not the more hidden units you have the better model you will have. Since we can see graphs with [64, 16] as their num_hiddens actually have some problems of overfit due to the fact that our model is too complex. I think we can solve this problem with a regularization, since I think larger neural nets with proper regularization can become easier for us to get a local minimum/ absolute minimum for our loss function. If neural network is too simple, it can not be a good model.

(e) [3 pts]

Here is the code for function plot.

```
335
336 def plot(x,y,prediction):
337     target = np.max(prediction,axis=1)<0.5
338     names = ['anger','disgust','fear','happy','sad','surprise','neutral']
339     if np.sum(target)>0:
340         for i in np.where(target>0)[0]:
341             plt.figure()
342             width = 48
343             height = 48
344             plt.imshow(x[i].reshape(width,height))
345             print("Confidence_max is {}, Predicted is {}, Target is {}".format(np.max(prediction[i]), names[np.argmax(prediction[i])],names[np.argmax(y[i])]))
346             plt.show()
347     return
```

Code added in main.

```
382     # Train model.
383     stats = train(model, nn_forward, nn_backward, nn_update, alpha,
384                   num_epochs, batch_size)
385
386     inputs_train, inputs_valid, inputs_test, target_train, target_valid, target_test = load_data("data/toronto_face.npz")
387     var = nn_forward(stats[0], inputs_test[:])
388     prediction = softmax(var["y"])
389     plot(inputs_test,target_test,prediction)
```

Confidence<Threshold = 0.5:

Model for hyperparameters:

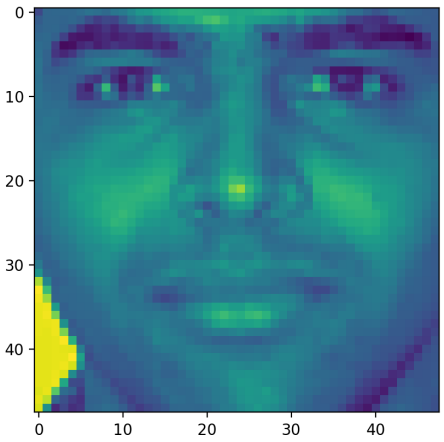
num_hiddens: [64, 32] alpha: 0.01

num_epochs: 1000 batch_size: 100

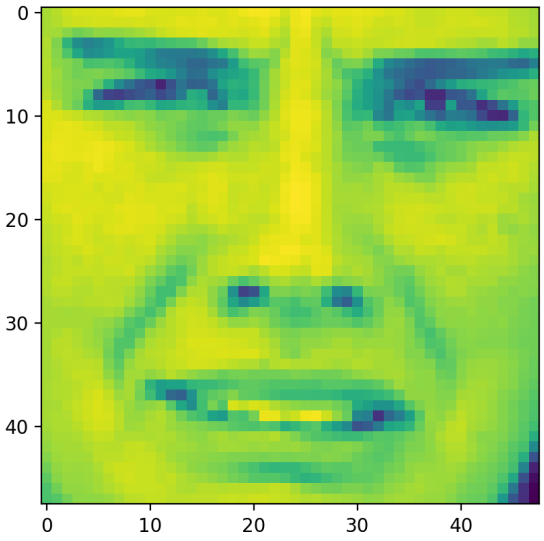
CE: Train 0.10575 Validation 1.00410 Test 0.64978

Acc: Train 0.97362 Validation 0.74940 Test 0.79221

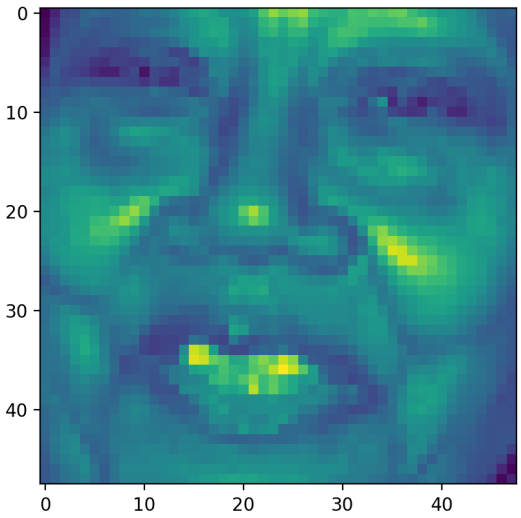
Confidence_max is 0.3878299635124929, Predicted is neutral, Target is neutral



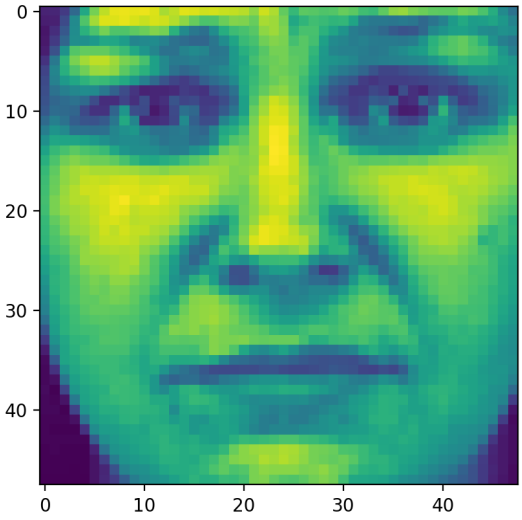
Confidence_max is 0.4151879960598142, Predicted is sad, Target is disgust



Confidence_max is 0.3664734595607387, Predicted is anger, Target is happy



Confidence_max is 0.3664734595607387, Predicted is anger, Target is happy



Above four graphs, my model has confidence less than 0.5. We can see if my model always choose the predict with the high score, it will not be really correct when the confidence is smaller than 0.5. Four of three are wrong, if my model choose the highest scoring class.