Machine Learning: Mini Project II

By Colin Vande Vijvere

Kernel Perceptron

Description:

Kernelized perceptron (polynomial kernel) using ECOC to classify the mnist dataset.

Notes:

With this algorithm, I've noticed that the # of iterations is quite important. Unfortunately, those iterations ask a lot of computing power, which I do not have available at the time. Below you can see How my algorithm increased after raising the #of iterations.

Training and testing accuracies + confusion matrix:

with # of iterations = 10:

```
Training Accuracy = 71.0%
2522 test case predicted.
1651 are correct.
Testing Accuracy = 65.0%
[[194   0   4   0   0   0   0   0  39   0]
 [  0  89   6   3   1   0   0   0  10 177]
 [  1   0 224  14   0   0   6   1   2   1]
 [  1   0  25 211   0   1   1  12   1   2]
 [  7   0  15   1 129   6  22   3  62  18]
 [  8   2  14  14  32  68   4  22  46  26]
 [  4   0  22   0   1   0 208   0   1   0]
 [  0   0   7  65   1   6   9 179   4  14]
 [  0   0  43   8   0   0   1   0 168  10]
 [  0   0   4  20   0   6   0   4  31 181]]
```

with # of iterations = 30:

```
Training Accuracy = 83.0%
6000 test case predicted.
4732 are correct.
Testing Accuracy = 79.0%
[[594   0   5   2   1   0   2   0   5   0]
 [  4 615   1  24   0   2   1   1   1  22]
 [  9   0 499  53   0   1  12   2   5   0]
 [  1   6  14 538   0   2   2  20   2  19]
 [ 19  11  14   5 332 111  32  12  40  43]
 [ 22  67   9  47  12 283   6  45   6  27]
 [  8   0  17   3   4   2 563   4   2   0]
 [  7   7  11  72   6  11   9 492   2  13]
 [  9  40   5  23   1  26   0   8  13 469]]
```

Kernel Perceptron Code:

```
import numpy as np
from numpy import linalg
import matplotlib
import matplotlib.pyplot as plt
from libsvm.svmutil import svm_read_problem
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report


def decToBin(array):
    result = {
        0: [0, 0, 0, 0],
        1: [0, 0, 0, 1],
        2: [0, 0, 1, 0],
        3: [0, 0, 1, 1],
        4: [0, 1, 0, 0],
        5: [0, 1, 0, 1],
        6: [0, 1, 1, 0],
        7: [0, 1, 1, 1],
        8: [1, 0, 0, 0],
        9: [1, 0, 0, 1]
    }
    return np.array([result[number] for number in array])


def binToDec(array):
    result = np.array([num[0]*8 + num[1] * 4 + num[2]
                                * 2 + num[3] for num in array])
    if result > 9:
        return np.array([num[0]*0 + num[1] * 4 + num[2]
                                   * 2 + num[3] for num in array])
    else:
        return result


def kernel(a, x, z, p, lr):
    result = np.sum(a*(1+(lr*(np.dot(x, z)))**p))
    return result
```

```python
class perceptron:
    def __init__(self, learning_rate=0.001, n_iters=30):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
        self.alpha = None

    def fit(self, X, y):
        self.alpha = np.zeros(len(X))
        pred = []
        for i in range(self.n_iters):
            for j in range(len(X)):
                val = kernel(self.alpha, X, X[j], p=3, lr=0.001)
                result = np.sign(val)
                pred.append(result)
                if result != y[j]:
                    self.alpha[j] = self.alpha[j] + self.lr*y[j]

    def predict(self, X, Xi):
        val = kernel(self.alpha, X, Xi, p=5, lr=0.005)
        result = np.sign(val)
        if result <= 0:
            return 0
        else:
            return 1
```

```python
# reading in the data
y_raw, x_raw = svm_read_problem('mnist.scale')

y = np.array(y_raw)
x = np.zeros((len(y_raw), 780))
for i in range(len(y_raw)):
    line = x_raw[i]
    for k, v in line.items():
        x[i][k - 1] = v

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.5)

y_trainAcc = y_train
y_train = decToBin(y_train)
y_train = np.where(y_train == 0, -1, 1)

#splitting digits of binary number
y_train1 = np.copy(y_train[:, 0])
y_train2 = np.copy(y_train[:, 1])
y_train3 = np.copy(y_train[:, 2])
y_train4 = np.copy(y_train[:, 3])
```

```
perceptron1 = perceptron()
perceptron1.fit(x_train, y_train1)
perceptron2 = perceptron()
perceptron2.fit(x_train, y_train2)
perceptron3 = perceptron(learning_rate=0.1)
perceptron3.fit(x_train, y_train3)
perceptron4 = perceptron(learning_rate=0.05)
perceptron4.fit(x_train, y_train4)

# Calculating training accuracy
predicted = np.zeros(len(y_trainAcc))
for i in range(len(y_trainAcc)):
    pred = np.zeros(4)
    pred[0] = perceptron1.predict(x_train, x_train[i])
    pred[1] = perceptron2.predict(x_train, x_train[i])
    pred[2] = perceptron3.predict(x_train, x_train[i])
    pred[3] = perceptron4.predict(x_train, x_train[i])
    predicted[i] = binToDec([pred])
correct_num = np.sum(predicted == y_trainAcc)
print('Training Accuracy = ', np.round(
    correct_num * 100 / len(predicted)), '%', sep='')


# Calculating testing accuracy
predicted = np.zeros(len(y_test))
for i in range(len(y_test)):
    pred = np.zeros(4)
    pred[0] = perceptron1.predict(x_train, x_test[i])
    pred[1] = perceptron2.predict(x_train, x_test[i])
    pred[2] = perceptron3.predict(x_train, x_test[i])
    pred[3] = perceptron4.predict(x_train, x_test[i])
    predicted[i] = binToDec([pred])
correct_num = np.sum(predicted == y_test)
print('Testing Accuracy = ', np.round(
    correct_num * 100 / len(predicted)), '%', sep='')

# printing confusion matrix:
print(confusion_matrix(y_test, predicted))
```

Pegasos SVM

Description:

Pegasos SVM written with ECOC to classify the mnist dataset.

Training and testing accuracies + confusion matrix:

```
Training Accuracy = 74.0%
Testing Accuracy = 69.0%
[[47  4  3  0  3  1  0  1  1  0]
 [ 2 65  0  1  0  2  0  1  1  0]
 [14  1 46  4  1  0  5  4  0  0]
 [ 1 12  2 40  0  1  2  3  2  1]
 [ 3  1  0  0 53 12  2  0  1  1]
 [ 2 13  0  2  6 25  1  2  2  0]
 [ 7  0  4  0  4  0 48  1  0  0]
 [ 0  5  0  6  2 20  0 44  0  0]
 [ 5  1  1  2  3  0  0  0 43  4]
 [ 2  4  0  0  2  9  0  2  2 43]]
```

Note: this is the highest achieved accuracy. Average accuracy of 10 runs was 66.53%.

Code for pegasos_SVM:

```python
import numpy as np
from numpy import linalg
import math
from random import randint
import matplotlib
import matplotlib.pyplot as plt
from libsvm.svmutil import svm_read_problem
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix


# function to change decimal to binary array
def decToBin(array):
    result = {
        0: [0, 0, 0, 0],
        1: [0, 0, 0, 1],
        2: [0, 0, 1, 0],
        3: [0, 0, 1, 1],
        4: [0, 1, 0, 0],
        5: [0, 1, 0, 1],
        6: [0, 1, 1, 0],
        7: [0, 1, 1, 1],
        8: [1, 0, 0, 0],
        9: [1, 0, 0, 1]
    }
    return np.array([result[number] for number in array])
```

```python
# function to change binary array to decimal
def binToDec(array):
    result = np.array([num[0]*8 + num[1] * 4 + num[2]
                      * 2 + num[3] for num in array])
    if result > 9:
        return np.array([num[0]*0 + num[1] * 4 + num[2]
                        * 2 + num[3] for num in array])
    else:
        return result


# pegasos part of the algorithm
def pegasos(x, y, weights=None, iterations=2000, lam=0.001):
    if type(weights) == type(None):
        weights = np.zeros(x[0].shape)
    num_S = len(y)
    for i in range(iterations):
        it = randint(0, num_S-1)
        step = 1/(lam*(i+1))
        decision = y[it] * weights @ x[it].T
        if decision < 1:
            weights = (1 - step*lam) * weights + step*y[it]*x[it]
        else:
            weights = (1 - step*lam) * weights
    return weights
```

```python
class SVM:
    def __init__(self, learning_rate=0.0005, lambda_param=0.01, n_iters=100):
        self.lr = learning_rate
        self.lam = lambda_param
        self.n_iters = n_iters
        self.weights = None
        self.b = None

    def fit(self, X, y):
        self.weights = pegasos(x=X, y=y, iterations=self.n_iters)

    def predict(self, X):
        approx = np.dot(X, self.weights)
        sign_approx = np.sign(approx)
        if sign_approx <= 0:
            return 0
        else:
            return 1
```

```python
# reading in the data
y_raw, x_raw = svm_read_problem('mnist.scale')

y = np.array(y_raw)
x = np.zeros((len(y_raw), 780))
for i in range(len(y_raw)):
    line = x_raw[i]
    for k, v in line.items():
        x[i][k - 1] = v

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)

# converting label to binary and splitting them up
y_trainAcc = y_train
y_trainzeroOne = decToBin(y_train)
y_train = np.where(y_trainzeroOne == 0, 1, -1)

y_train1 = np.copy(y_train[:, 0])
y_train2 = np.copy(y_train[:, 1])
y_train3 = np.copy(y_train[:, 2])
y_train4 = np.copy(y_train[:, 3])
```

```python
#training 4 algorithms for each binary digit
svm1 = SVM()
svm1.fit(x_train, y_train1)
svm2 = SVM()
svm2.fit(x_train, y_train2)
svm3 = SVM()
svm3.fit(x_train, y_train3)
svm4 = SVM()
svm4.fit(x_train, y_train4)


# training accuracy
predicted = np.zeros(len(y_trainAcc))
for i in range(len(y_trainAcc)):
    pred = np.zeros(4)
    pred[0] = svm1.predict(x_train[i])
    pred[1] = svm2.predict(x_train[i])
    pred[2] = svm3.predict(x_train[i])
    pred[3] = svm4.predict(x_train[i])
    predicted[i] = binToDec([pred])
    # print("binary answer: ", pred, " | decimal prediction: ",
    #       predicted[i], " | correct prediction: ", y_train[i])
print(len(predicted), ' test case predicted.', sep='')
correct_num = np.sum(predicted == y_trainAcc)
```

```
# Testing accuracy
predicted = np.zeros(len(y_test))
for i in range(len(y_test)):
    pred = np.zeros(4)
    pred[0] = svm1.predict(x_test[i])
    pred[1] = svm2.predict(x_test[i])
    pred[2] = svm3.predict(x_test[i])
    pred[3] = svm4.predict(x_test[i])
    predicted[i] = binToDec([pred])
    print("binary answer: ", pred, " | decimal prediction: ",
            predicted[i], " | correct prediction: ", y_test[i])
print(len(predicted), ' test case predicted.', sep='')
correct_num = np.sum(predicted == y_test)
print(correct_num, ' are correct.', sep='')
print('Accuracy = ', np.round(correct_num * 100 / len(predicted)), '%', sep='')
```

Neural Network:

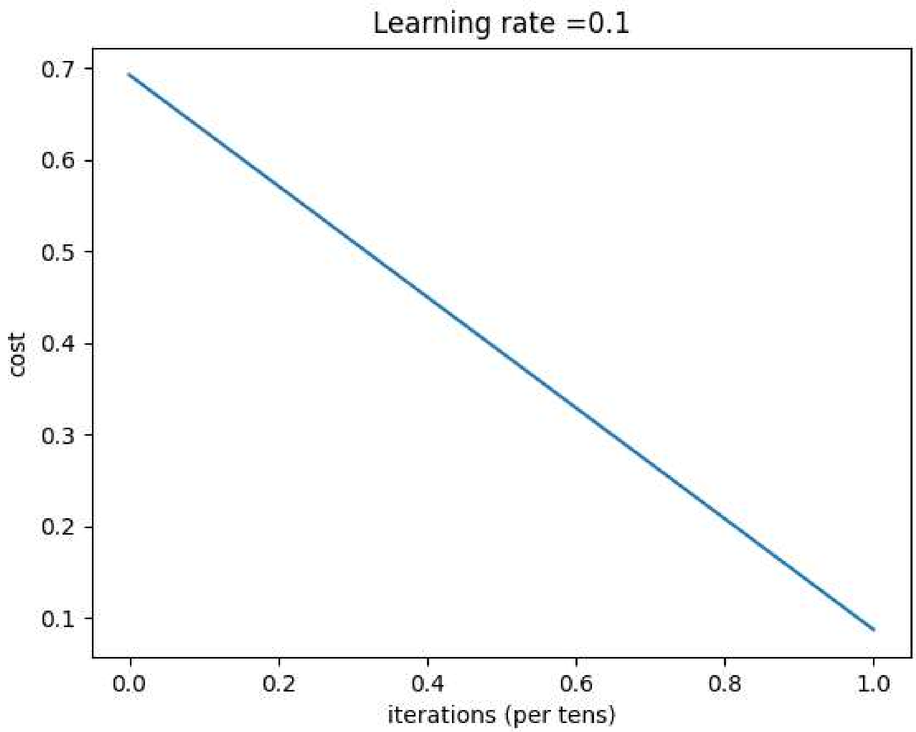
Description:

Neural Net to classify the mnist dataset.

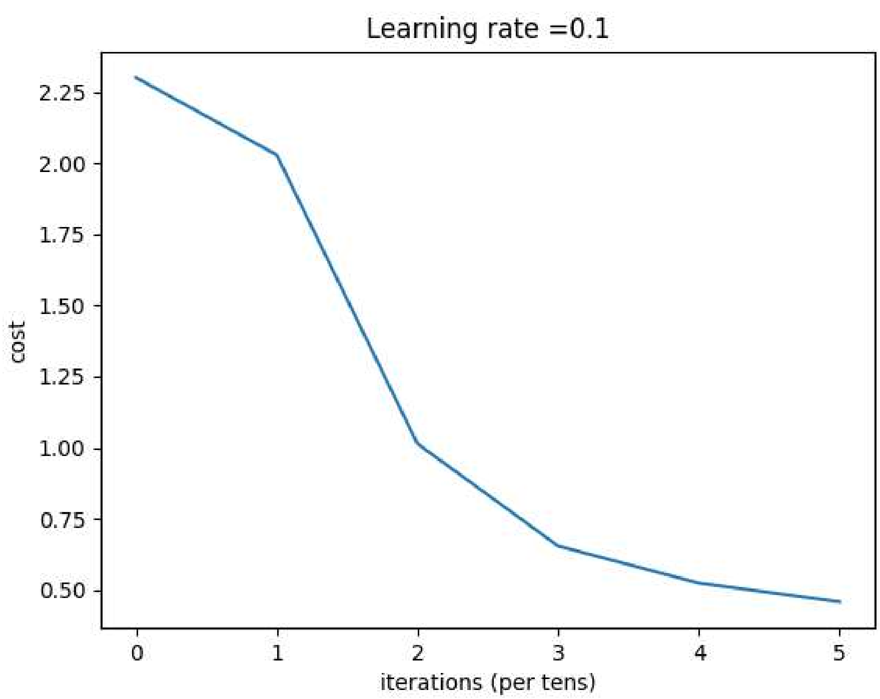
Training and testing accuracies + confusion matrix:

```
====== Binary (Digit 1 and 2) =======
Cost after iteration 0: 0.6926057785748851
Cost after iteration 50: 0.08490423769633489
Training Accuracy:
8890 test case predicted.
8890 are correct.
Accuracy = 1.0
Testing Accuracy:
3810 test case predicted.
3810 are correct.
Accuracy = 1.0
====== Multi-class =======
Cost after iteration 0: 2.302454060157147
Cost after iteration 50: 2.0319194764807524
Cost after iteration 100: 1.0125986456112632
Cost after iteration 150: 0.6517215378108929
Cost after iteration 200: 0.5212031076333735
Cost after iteration 250: 0.45508607556340214
Training Accuracy:
42000 Test case predicted.
41383 are correct.
Accuracy = 0.98530952381
Testing Accuracy:
18000 test case predicted.
15961 are correct.
Accuracy = 0.8867222222222222
[[1690    0   26    9    4   39   15   13    7    9]
 [   0 1881   41   14    6   15    7   34   36   13]
 [   9    9 1553   44   11   14   38   34   29   13]
 [  10    9   43 1549    0   69    1    5   68   30]
 [   5    2   36    2 1563   16   24   16    8   99]
 [  31    8    3   91    3 1358   28    2   63    8]
 [  13    4   54    4   26   37 1674    0   18    2]
 [   2    6   21   26    1   13    0 1676    3   80]
 [  12   29   54   50   12   64   11    7 1445   22]
 [   1    1    7   22  114   27    0   75   52 1572]]
```

Cost/Iteration for the Binary:



Cost/Iteration for the multiclass:



Code for the Neural Net:

```python
import numpy as np
from numpy import linalg
import matplotlib
import matplotlib.pyplot as plt
from libsvm.svmutil import svm_read_problem
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

def oneHot(array):
    result = np.zeros((len(array),10))
    for number in range(len(array)):
        result[number][int(array[number])] = 1
    return result

def softmax(Z):
    Z_shift = Z - np.max(Z, axis=0)
    A = np.exp(Z_shift) / np.sum(np.exp(Z_shift), axis=0)

    cache = Z_shift

    return A, cache

def relu(Z):
    A = np.maximum(0, Z)
    assert (A.shape == Z.shape)
    cache = Z
    return A, cache

def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(1)

    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}
    return parameters
```

```python
def linear_forward(A, W, b):
    Z = np.dot(W, A) + b

    assert (Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache

def linear_activation_forward(A_prev, W, b, activation):
    if activation == "softmax":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = softmax(Z)

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache

def compute_cost(AL, Y):
    m = Y.shape[1]
    cost = -(np.sum(Y * np.log(AL))) / float(m)
    # cost = np.squeeze(cost)
    assert (cost.shape == ())

    return cost

def linear_backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = np.dot(dZ, A_prev.T) / float(m)
    db = np.sum(dZ, axis=1, keepdims=True) / float(m)
    dA_prev = np.dot(W.T, dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

```python
def relu_backward(dA, cache):
    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def softmax_backward(Y, cache):
    Z = cache

    s = np.exp(Z) / np.sum(np.exp(Z), axis=0)
    dZ = s - Y

    assert (dZ.shape == Z.shape)

    return dZ

def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "softmax":
        dZ = softmax_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

```python
def update_parameters(parameters, grads, learning_rate):
    L = len(parameters) // 2  # number of layers in the neural network

    # Update rule for each parameter. Use a for loop.
    for l in range(1, L + 1):
        parameters['W' + str(l)] -= learning_rate * grads['dW' + str(l)]
        parameters['b' + str(l)] -= learning_rate * grads['db' + str(l)]

    return parameters
```

```python
def two_layer_model(X, Y, layers_dims, learning_rate=0.1, num_iterations=3000, print_cost=False):

    np.random.seed(1)
    grads = {}
    costs = []  # to keep track of the cost
    m = X.shape[1]  # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions you'd previously implemented
    parameters = initialize_parameters(n_x, n_h, n_y)

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        # Forward propagation
        A1, cache1 = linear_activation_forward(X, W1, b1, activation='relu')
        A2, cache2 = linear_activation_forward(A1, W2, b2, activation='softmax')

        # Compute cost
        cost = compute_cost(A2, Y)

        # Backward propagation
        dA1, dW2, db2 = linear_activation_backward(Y, cache2, activation='softmax')
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, activation='relu')

        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        # Update parameters.
        parameters = update_parameters(parameters, grads, learning_rate)

        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]
```

```python
        # Print the cost every 100 training example
        if i % 50 == 0:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
            costs.append(cost)



    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

```python
def predict(X, y, parameters):
    m = X.shape[1]

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    A1, _ = linear_activation_forward(X, W1, b1, activation='relu')
    probs, _ = linear_activation_forward(A1, W2, b2, activation='softmax')

    predicted = np.argmax(probs, axis=0)

    # print ("predictions: " + str(p))
    # print ("true labels: " + str(y))
    print(m, ' test case predicted.', sep='')
    correct_num = np.sum(predicted == y)
    print(correct_num, ' are correct.', sep='')
    #print('Accuracy = ', np.round(correct_num * 100 / len(predicted)), '%', sep='')
    print("Accuracy = " + str(correct_num / float(m)), sep='')

    return predicted

y_raw, x_raw = svm_read_problem('mnist.scale')

y = np.array(y_raw)
x = np.zeros((len(y_raw), 780))
for i in range(len(y_raw)):
    line = x_raw[i]
    for k, v in line.items():
        x[i][k - 1] = v

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)

y_train = oneHot(y_train)
```

```python
print('====== Binary (Digit 1 and 2) =======')

y_bin = np.concatenate((y[y==1],[0]*sum(y==2)))
temp = np.zeros((len(y_bin), 2))
for number in range(len(y_bin)):
    temp[number][int(y_bin[number])] = 1
y_bin = temp

x_bin = np.concatenate((x[y==1],x[y==2]))

x_train_bin, x_test_bin, y_train_bin, y_test_bin = train_test_split(x_bin, y_bin, test_size=0.3)

parameters_bin = two_layer_model(x_train_bin.T, y_train_bin.T, (780, 100, 2), 0.1, 100, True)
print("Training Accuracy:")
training_prediction = predict(x_train_bin.T, y_train_bin.T, parameters_bin)
print ("Testing Accuracy:")
prediction_bin = predict(x_test_bin.T, y_test_bin.T, parameters_bin)

print('====== Multi-class =======')
parameters = two_layer_model(x_train.T, y_train.T, (780, 100, 10), 0.1, 300, True)

print("Training Accuracy:")
training_pred = predict(x_train.T, y_train.T, parameters)

print("Testing Accuracy:")
prediction = predict(x_test.T, y_test.T, parameters)

print(confusion_matrix(prediction, y_test))
```