Simple MNIST Neural Network from scratch

Author: Anshuman Sinha

In this notebook, I implemented a simple two-layer neural network and trained it on the MNIST digit recognizer dataset. It's meant to be an instructional example, through which you can understand the underlying math of neural networks better.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data=pd.read_csv(r"D:\Coding\Neural Network\train.csv")
```

This line reads the MNIST dataset from a CSV file named "train.csv" using Pandas and stores it in a DataFrame called data.

```
data.head()
```

```
   label  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6
pixel7  \
0      1       0       0       0       0       0       0       0
0
1      0       0       0       0       0       0       0       0
0
2      1       0       0       0       0       0       0       0
0
3      4       0       0       0       0       0       0       0
0
4      0       0       0       0       0       0       0       0
0

   pixel8  ...  pixel774  pixel775  pixel776  pixel777  pixel778
pixel779  \
0       0  ...         0         0         0         0         0
0
1       0  ...         0         0         0         0         0
0
2       0  ...         0         0         0         0         0
0
3       0  ...         0         0         0         0         0
0
4       0  ...         0         0         0         0         0
0

   pixel780  pixel781  pixel782  pixel783
0         0         0         0         0
```

```
1          0          0          0          0
2          0          0          0          0
3          0          0          0          0
4          0          0          0          0

[5 rows x 785 columns]
```

These lines convert the DataFrame into a NumPy array, shuffling the data to ensure randomness, and retrieve the number of rows m and the number of columns n.

```
data=np.array(data)
m, n= data.shape
np.random.shuffle(data)

data_dev=data[0:1000].T
Y_dev=data_dev[0]
X_dev=data_dev[1:n]
X_dev = X_dev / 255.

data_train=data[1000:m].T
Y_train=data_train[0]
X_train=data_train[1:n]
X_train = X_train / 255.
_,m_train = X_train.shape
```

Here, the data is split into a development set (data_dev) and a training set (data_train). The pixel values are normalized to a range between 0 and 1 by dividing by 255.

Our NN will have a simple two-layer architecture. Input layer $a^{[0]}$ will have 784 units corresponding to the 784 pixels in each 28x28 input image. A hidden layer $a^{[1]}$ will have 10 units with ReLU activation, and finally our output layer $a^{[2]}$ will have 10 units corresponding to the ten digit classes with softmax activation.

Forward propagation

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = g_{\text{ReLU}}\left(Z^{[1]}\right) ¿$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = g_{\text{softmax}}\left(Z^{[2]}\right)$$

Backward propagation

$$d Z^{[2]} = A^{[2]} - Y$$

$$d W^{[2]} = \frac{1}{m} d Z^{[2]} A^{[1]T}$$

$$dB^{[2]} = \frac{1}{m} \Sigma dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} .* g^{[1]'}\left(z^{[1]}\right)$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$$

$$dB^{[1]} = \frac{1}{m} \Sigma dZ^{[1]}$$

Parameter updates

$$W^{[2]} := W^{[2]} - \alpha\, dW^{[2]}$$

$$b^{[2]} := b^{[2]} - \alpha\, db^{[2]}$$

$$W^{[1]} := W^{[1]} - \alpha\, dW^{[1]}$$

$$b^{[1]} := b^{[1]} - \alpha\, db^{[1]}$$

Vars and shapes

Forward prop

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \sim A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]} A^{[0]} \sim Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \sim A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]} A^{[1]} \sim Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Backprop

- $dZ^{[2]}$: 10 x m ($A^{[2]}$)
- $dW^{[2]}$: 10 x 10
- $dB^{[2]}$: 10 x 1
- $dZ^{[1]}$: 10 x m ($A^{[1]}$)
- $dW^{[1]}$: 10 x 10
- $dB^{[1]}$: 10 x 1

```
def init_params():
    w1=np.random.rand(10,784)-0.5
    b1=np.random.rand(10,1)-0.5
    w2=np.random.rand(10,10)-0.5
    b2=np.random.rand(10,1)-0.5
    return w1, b1, w2, b2
```

This function initializes the parameters of the neural network: weights (w1, w2) and biases (b1, b2) for the input and hidden layers.

```
def ReLU(Z):
    return np.maximum(Z, 0)

def softmax(Z):
    A=np.exp(Z) / sum(np.exp(Z))
    return A
```

These functions define the activation functions used in the neural network: ReLU (Rectified Linear Unit) for the hidden layer and softmax for the output layer.

```
def forward_prop(w1, b1, w2, b2, X):
    z1=w1.dot(X)+b1
    a1=ReLU(z1)
    z2=w2.dot(a1)+b2
    a2=softmax(z2)
    return z1, a1, z2, a2
```

This function performs forward propagation, computing the activations of each layer sequentially until the final output is obtained.

```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y

def deriv_ReLU(Z):
    return Z>0

def back_prop(z1, a1, z2, a2, w2, Y, X):
    OneHot_Y=one_hot(Y)
    dZ2=a2-OneHot_Y
    dW2=1/m * dZ2.dot(a1.T)
    db2=1/m * np.sum(dZ2)
    dZ1=w2.T.dot(dZ2)*deriv_ReLU(z1)
    dW1=1/m * dZ1.dot(X.T)
    db1=1/m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

This function computes the gradients of the loss function with respect to the parameters of the network during backpropagation.

```
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1=W1-alpha*dW1
    b1=b1-alpha*db1
    W2=W2-alpha*dW2
```

```
    b2=b2-alpha*db2
    return W1, b1, W2, b2
```

This function updates the parameters of the neural network using gradient descent.

```
def get_predictions(a2):
    return np.argmax(a2,0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y)/Y.size

def gradient_descent(X, Y, iterations, alpha):
    w1, b1, w2, b2=init_params()
    for i in range(iterations):
        z1, a1, z2, a2 =forward_prop(w1, b1, w2, b2, X)
        dW1, db1, dW2, db2 = back_prop(z1, a1, z2, a2, w2, Y, X)
        w1, b1, w2, b2 = update_params(w1, b1, w2, b2, dW1, db1, dW2,
db2, alpha)
        if i%10==0:
            print("iterations: ",i)
            print("accuracy: ",get_accuracy(get_predictions(a2),Y))
    return w1, b1, w2, b2
```

This function performs gradient descent to train the neural network, iterating over a specified number of epochs (iterations) and updating the parameters.

```
w1, b1, w2, b2=gradient_descent(X_train, Y_train, 500, 0.1)
```

```
iterations:  0
[6 2 2 ... 6 6 2] [3 8 2 ... 7 0 0]
accuracy:  0.10451219512195122
iterations:  10
[1 0 1 ... 6 6 0] [3 8 2 ... 7 0 0]
accuracy:  0.1959268292682927
iterations:  20
[1 0 1 ... 6 6 0] [3 8 2 ... 7 0 0]
accuracy:  0.2781219512195122
iterations:  30
[1 0 1 ... 6 6 0] [3 8 2 ... 7 0 0]
accuracy:  0.3261707317073171
iterations:  40
[1 0 1 ... 2 6 0] [3 8 2 ... 7 0 0]
accuracy:  0.36973170731707317
iterations:  50
[6 2 1 ... 7 6 0] [3 8 2 ... 7 0 0]
accuracy:  0.4161219512195122
iterations:  60
[6 2 7 ... 7 6 0] [3 8 2 ... 7 0 0]
```

```
accuracy:  0.4642926829268293
iterations:  70
[5 2 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.5150975609756098
iterations:  80
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.5638780487804878
iterations:  90
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.6015853658536585
iterations:  100
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.6329268292682927
iterations:  110
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.6570243902439025
iterations:  120
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.6773170731707318
iterations:  130
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.6950731707317073
iterations:  140
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7100975609756097
iterations:  150
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7229756097560975
iterations:  160
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7338048780487805
iterations:  170
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7438536585365854
iterations:  180
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7532439024390244
iterations:  190
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7609512195121951
iterations:  200
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7678780487804878
iterations:  210
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7740731707317073
iterations:  220
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7801463414634147
```

```
iterations:  230
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.785219512195122
iterations:  240
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.789609756097561
iterations:  250
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7937317073170732
iterations:  260
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.7980731707317074
iterations:  270
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8018048780487805
iterations:  280
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8057804878048781
iterations:  290
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8096585365853659
iterations:  300
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.812560975609756
iterations:  310
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8154634146341463
iterations:  320
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8184878048780487
iterations:  330
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8220243902439024
iterations:  340
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8240243902439024
iterations:  350
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8262439024390243
iterations:  360
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8280975609756097
iterations:  370
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8305609756097561
iterations:  380
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8326341463414634
iterations:  390
```

```
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8344878048780487
iterations:  400
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.836780487804878
iterations:  410
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8385365853658536
iterations:  420
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8402926829268292
iterations:  430
[5 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8419268292682927
iterations:  440
[9 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8433170731707317
iterations:  450
[9 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8447073170731707
iterations:  460
[9 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8461951219512195
iterations:  470
[9 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8475609756097561
iterations:  480
[9 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8486585365853658
iterations:  490
[9 8 7 ... 7 0 0] [3 8 2 ... 7 0 0]
accuracy:  0.8499024390243902
```

~86% accuracy on training set.

```python
def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions
```

This function makes predictions using the trained neural network.

```python
def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2,
b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)
```

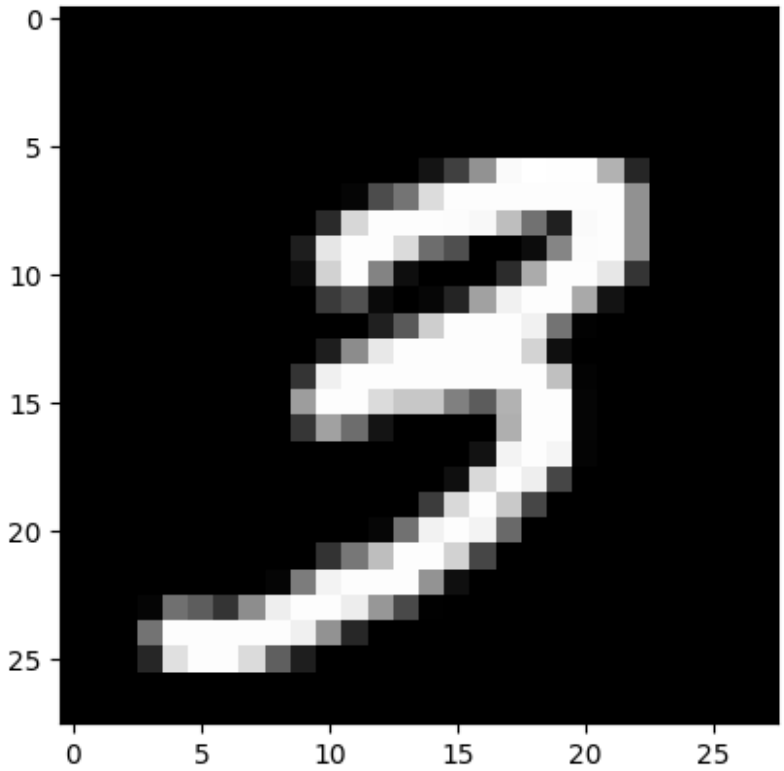
```
    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

These functions are for testing predictions and calculating accuracy on the validation set.

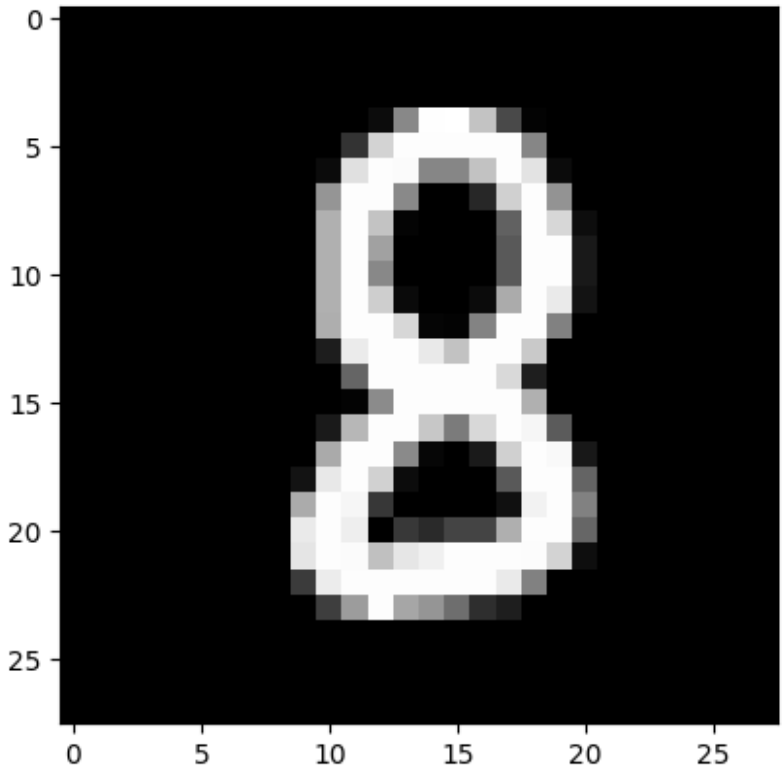
Let's look at a couple of examples:

```
test_prediction(0, w1, b1, w2, b2)
test_prediction(1, w1, b1, w2, b2)
test_prediction(2, w1, b1, w2, b2)
test_prediction(3, w1, b1, w2, b2)
```

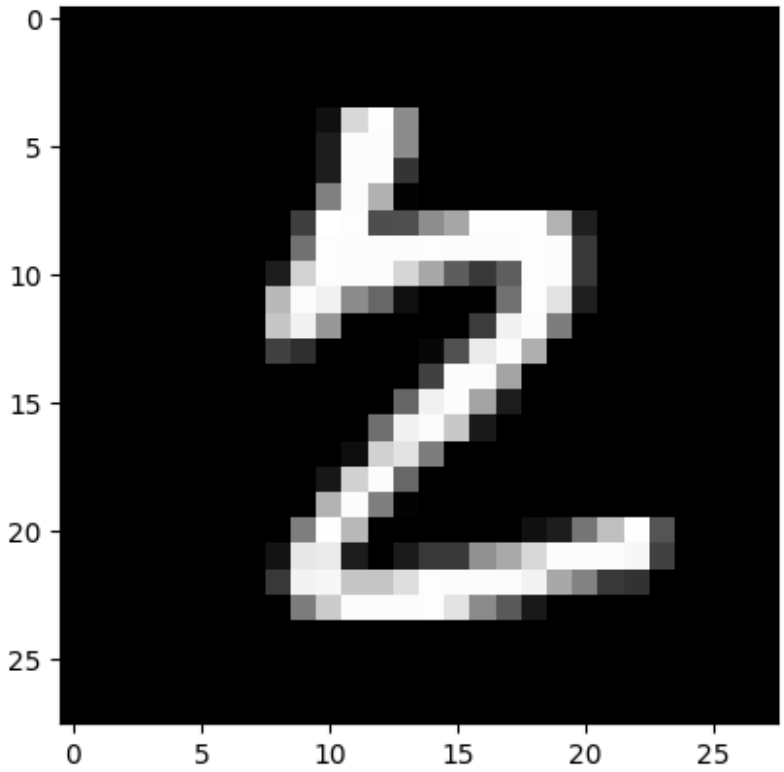
```
Prediction:  [9]
Label:  3
```



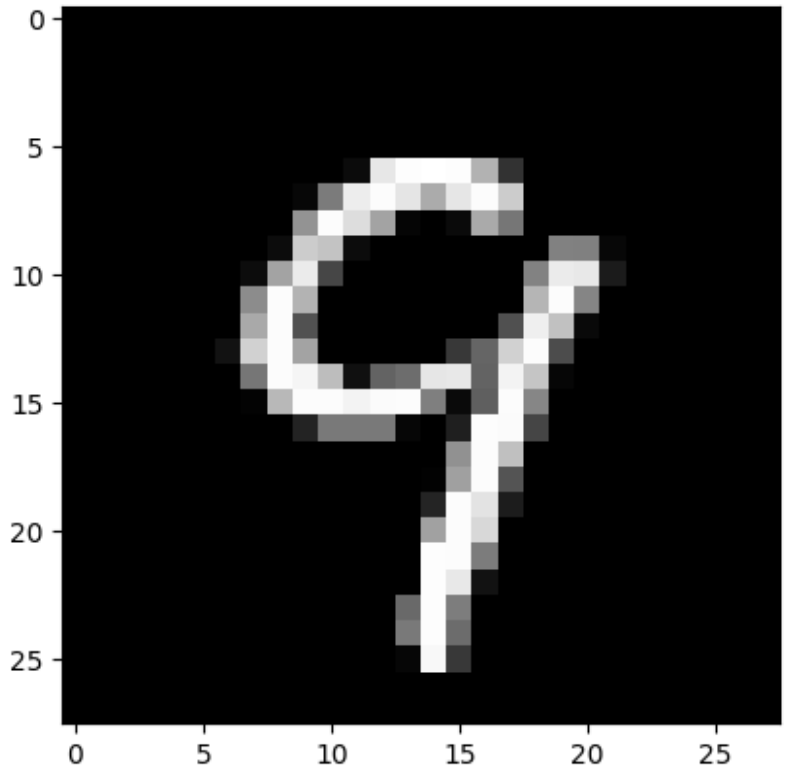

```
Prediction:  [8]
Label:  8
```




```
Prediction:  [7]
Label:  2
```




```
Prediction:  [9]
Label:  9
```



Finally, let's find the accuracy on the dev set:

```
dev_predictions = make_predictions(X_dev, w1, b1, w2, b2)
get_accuracy(dev_predictions, Y_dev)
```

```
[0 8 8 1 9 6 1 7 4 6 5 5 4 4 5 3 8 6 2 3 7 9 6 9 5 1 6 1 3 3 2 1 4 7 5
1 6
 1 9 2 7 6 9 1 0 5 7 3 0 0 1 6 3 6 4 8 2 5 5 2 2 2 9 1 9 6 2 3 1 8 5 1
1 2
 5 8 2 4 2 1 4 7 8 9 2 6 4 6 1 5 9 0 2 7 5 9 8 9 4 3 4 3 0 5 3 5 8 3 6
7 9
 9 9 6 6 2 4 4 9 1 9 2 2 6 9 6 2 2 1 6 5 7 2 1 9 8 0 9 6 7 5 3 5 7 7 1
8 2
 6 2 8 6 1 5 1 3 6 4 2 6 9 8 2 0 9 9 7 4 0 9 6 3 7 9 2 6 9 8 5 9 2 3 1
3 6
 1 2 1 3 9 2 4 0 1 0 4 6 7 4 7 8 8 5 9 3 2 7 6 8 0 3 0 1 9 0 8 4 2 6 7
7 4
 8 4 0 2 6 9 1 6 0 9 1 1 8 1 1 3 5 4 9 5 7 2 2 3 9 9 0 9 1 9 9 7 1 7 5
4 1
 6 2 2 6 3 9 8 1 4 9 8 4 3 4 3 6 9 9 0 6 9 0 1 7 1 2 0 4 9 2 7 6 8 3 8
8 1
 4 9 9 0 3 1 2 5 2 6 9 1 8 2 3 4 6 6 2 9 8 0 3 9 1 1 7 4 5 1 0 5 8 1 7
9 4
 8 1 5 1 5 8 9 8 2 9 6 1 4 3 4 1 5 9 8 9 9 5 6 3 0 7 8 0 4 2 0 4 8 3 8
7 6
 8 0 2 1 8 8 5 3 6 6 0 6 3 9 9 5 4 5 7 0 7 7 1 1 9 9 9 6 4 9 9 1 3 7 1
```

```
7 2
 6 3 8 4 3 3 6 4 0 0 3 7 9 2 1 8 7 8 6 1 5 0 2 1 0 0 6 0 2 8 1 2 2 1 9
2 4
 1 0 4 1 0 4 0 0 9 4 9 9 7 0 2 0 6 3 9 0 0 5 7 6 2 8 9 8 5 5 5 9 0 5 3
1 4
 0 6 1 4 4 8 8 8 5 6 3 8 2 2 6 6 0 0 3 6 7 4 7 7 4 5 0 8 6 0 2 0 6 0 9
4 3
 7 6 9 5 0 1 0 2 6 9 2 6 3 7 1 2 6 8 0 3 1 0 5 5 9 6 2 0 2 8 1 7 3 2 2
0 4
 9 5 8 2 0 8 4 2 6 1 1 8 5 7 4 2 3 8 8 0 9 2 3 4 5 5 8 7 9 1 4 8 1 9 3
6 8
 0 6 1 7 4 8 5 4 2 3 3 6 9 6 9 8 9 5 7 0 8 8 6 4 7 4 1 7 2 4 0 1 5 8 1
7 4
 3 5 6 3 3 3 0 9 3 0 9 7 5 1 6 3 4 8 9 4 1 5 4 6 4 2 9 7 5 7 7 1 8 4 1
4 7
 1 3 7 7 0 9 7 2 7 0 8 0 2 7 8 1 9 3 7 6 9 1 5 8 8 8 8 4 5 7 8 7 1 8 6
5 0
 2 1 8 9 7 7 3 5 1 6 1 1 2 7 2 0 7 0 6 2 7 7 4 8 4 7 5 1 7 7 7 6 7 1 4
4 6
 4 9 0 4 9 6 8 7 5 4 6 2 0 4 7 0 9 8 4 1 4 2 5 7 4 0 6 7 2 6 9 8 7 5 0
7 9
 5 2 3 5 0 7 4 6 6 5 4 0 6 2 6 2 4 8 7 8 8 6 0 9 0 6 0 3 6 7 8 7 0 3 7
6 4
 5 9 6 8 4 6 6 9 4 1 0 2 8 2 4 3 9 1 0 9 1 6 1 9 8 0 1 9 7 0 0 6 8 1 8
8 3
 6 9 6 3 2 7 2 9 0 2 3 3 3 1 0 2 5 0 1 9 0 7 0 0 3 4 2 0 1 6 0 3 0 2 1
0 2
 5 3 0 9 9 2 3 1 5 4 1 5 9 0 5 9 6 7 4 7 8 2 0 7 3 1 3 1 2 3 3 5 8 9 1
6 7
 6 3 1 9 8 3 5 6 8 8 6 3 3 8 3 3 6 9 5 7 7 9 8 7 2 1 3 3 4 1 6 4 2 1 0
1 9
 2 9 2 0 0 8 9 6 0 0 3 0 3 8 1 0 2 6 8 3 9 9 7 3 0 6 4 1 4 3 1 4 1 7 2
7 5
 2] [0 8 8 1 9 6 1 7 4 8 5 5 4 4 5 3 8 6 2 3 7 9 2 4 5 1 6 1 3 5 2 1 4
7 3 1 6
 8 4 2 8 6 9 1 0 5 7 3 0 0 1 2 3 6 4 8 2 5 5 2 2 2 9 1 4 6 2 3 1 8 5 4
1 2
 5 8 2 4 2 1 4 7 8 9 2 6 4 6 1 3 9 0 2 7 5 9 5 9 4 3 9 3 0 6 3 8 4 3 6
7 9
 9 9 6 6 2 4 4 9 1 9 2 7 8 9 6 2 3 1 6 8 7 2 1 9 3 0 9 6 7 5 3 5 7 7 1
1 2
 5 2 8 6 1 5 1 5 5 4 3 6 9 8 2 0 9 9 7 4 0 9 6 3 7 9 2 6 5 8 5 9 2 8 1
3 6
 1 2 1 8 4 2 4 0 1 0 4 6 7 4 7 8 4 5 9 3 2 7 6 8 0 3 0 1 9 0 8 9 2 6 7
7 4
 8 4 0 2 6 9 1 6 0 9 1 1 8 8 1 3 5 4 9 5 7 2 2 3 9 4 0 9 1 9 9 7 1 7 5
4 1
 6 2 2 6 8 9 8 1 4 9 8 4 5 4 8 2 9 9 8 6 9 0 8 7 1 2 0 4 9 6 9 6 8 3 8
8 1
```

```
 9 9 9 0 3 1 2 5 2 6 4 1 8 2 3 4 6 6 9 8 8 0 3 9 1 1 7 4 5 1 0 5 8 1 7
8 9
 8 1 5 1 5 8 9 1 2 9 6 1 4 3 4 1 6 9 8 9 9 5 5 9 2 7 8 0 4 2 0 4 8 3 8
7 6
 8 0 2 1 8 8 5 3 6 6 0 4 3 9 9 5 9 5 7 0 7 8 1 1 9 9 4 6 4 9 9 1 3 7 1
7 7
 6 3 8 4 3 9 6 4 0 0 3 7 4 2 1 9 7 8 6 1 8 0 2 1 0 0 5 0 5 8 1 2 2 1 7
2 4
 1 0 4 1 0 4 0 0 9 4 7 4 7 0 2 0 6 3 9 0 0 5 7 6 2 8 7 8 5 3 5 9 0 5 3
1 4
 0 2 1 4 0 3 8 8 5 2 3 8 2 6 5 6 6 0 3 6 7 4 7 7 4 3 0 8 4 0 2 0 3 0 9
4 3
 7 6 3 5 0 1 0 2 6 9 2 6 3 7 1 5 2 9 0 3 1 0 5 5 9 6 2 0 2 8 1 7 3 2 2
0 4
 9 5 8 2 0 8 4 2 6 1 8 8 5 7 5 7 3 8 8 0 9 2 3 4 5 9 8 7 9 1 9 8 1 9 3
0 8
 0 6 8 7 4 3 3 4 2 3 3 6 7 6 9 8 9 3 8 0 8 8 6 4 7 5 1 7 2 9 0 1 5 8 1
7 4
 3 5 6 3 3 3 0 5 3 9 9 7 5 1 6 3 3 8 9 9 1 5 4 6 4 2 4 7 8 7 7 1 7 4 1
4 7
 5 3 7 7 0 9 7 2 7 0 8 0 2 7 8 1 9 3 7 6 9 1 5 3 8 8 8 4 9 7 8 7 1 8 6
3 0
 2 6 8 9 7 9 3 5 1 6 1 1 2 7 2 0 7 0 6 2 7 7 4 8 9 0 5 1 7 7 7 6 7 1 4
4 6
 4 3 5 4 9 6 8 7 5 1 6 2 0 4 9 0 9 8 2 1 4 2 5 7 9 0 6 7 2 4 4 8 7 5 2
7 9
 5 2 3 5 0 7 4 6 6 5 4 0 6 2 6 2 4 4 7 8 8 4 4 4 0 3 0 3 6 7 5 7 2 3 7
6 4
 5 9 8 8 4 6 6 9 4 1 0 2 8 2 5 3 9 8 0 9 1 6 1 7 8 0 1 9 7 0 0 6 8 1 8
8 3
 6 9 6 3 2 7 2 9 0 2 3 3 3 3 0 2 5 0 1 9 0 7 0 0 3 5 2 0 1 6 0 7 0 2 1
0 2
 5 8 0 9 9 2 8 1 5 4 1 5 9 0 5 9 6 9 4 7 8 2 0 7 3 1 3 2 2 3 3 5 8 9 1
6 7
 6 3 1 9 8 3 5 2 8 0 6 3 3 3 3 3 6 9 5 7 7 9 8 7 2 1 3 3 4 1 6 4 2 1 0
1 9
 2 9 2 0 0 3 9 6 0 0 3 0 3 8 1 0 2 6 8 3 9 7 3 3 0 6 4 8 4 3 1 4 1 7 2
7 5
 2]

0.853
```

Still 85% accuracy, so our model generalized from the training data pretty well.