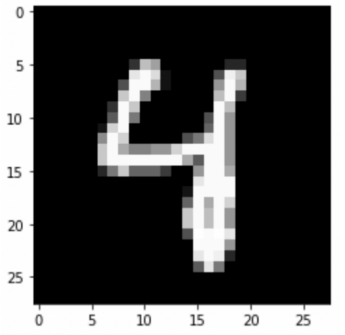
# Neural Network Example

## Problem Statement




The dataset we're working with is the famous MNIST handwritten digit dataset, commonly used for instructive ML and computer vision projects. It contains 28 x 28 grayscale images of handwritten digits that look like this :

Each image is accompanied by a label of what digit it belongs to, from 0 to 9. Our task is to build a network that takes in an image like this and predicts what digit is written in it.

## Neural Network Overview

Our network will have three layers total: an input layer and two layers with parameters. Because the input layer has no parameters, this network would be referred to as a **two-layer neural network.**

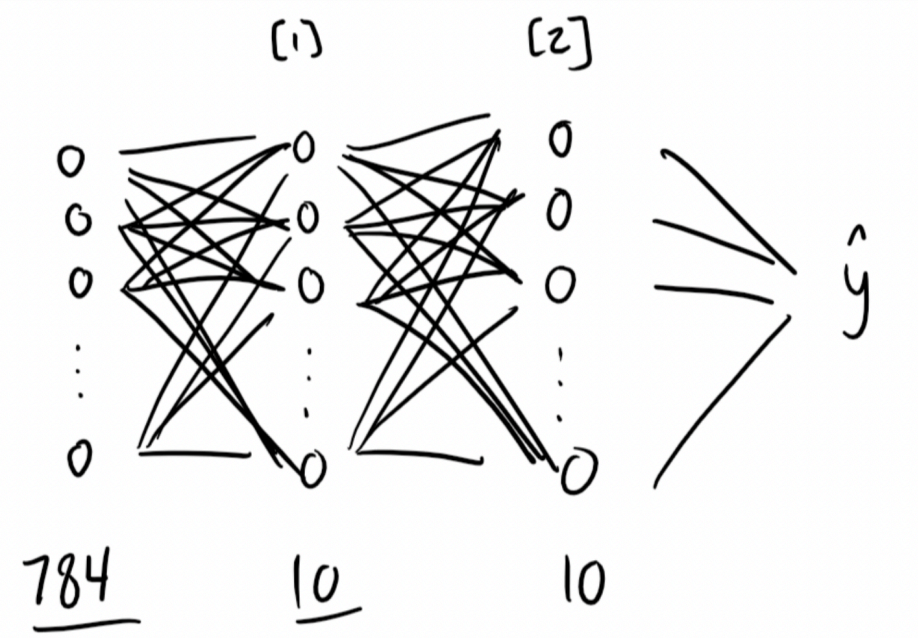
The input layer has 784 nodes, corresponding to each of the 784 pixels in the 28x28 input image. Each pixel has a value between 0 and 255, with 0 being black and 255 being white. It's common to **normalize** these values — getting all values between 0 and 1, here by simply dividing by 255 — before feeding them in the network.

The second layer, or hidden layer, could have any amount of nodes, but we've made it really simple here with just 10 nodes. The value of each of these nodes is calculated based on weights and biases applied to the

value of the 784 nodes in the input layer. After this calculation, a ReLU activation is applied to all nodes in the layer (more on this later).

In a deeper network, there may be multiple hidden layers back to back before the output layer. In this network, we'll only have one hidden layer before going to the output.

The output layer also has 10 nodes, corresponding to each of the output classes (digits 0 to 9). The value of each of these nodes will again be calculated from weights and biases applied to the value of the 10 nodes in the hidden layer, with a softmax activation applied to them to get the final output.




The process of taking an image input and running through the neural network to get a prediction is called **forward propagation.** The prediction that is made from a given image depends on the **weights and biases, or parameters,** of the network.

To train a neural network, then, we need to update these weights and biases to produce accurate predictions. We do this through **gradient descent.**

In a neural network, gradient descent is carried out via a process called backward propagation, or backprop. In backprop, instead of taking an input image and running it forwards through the network to get a prediction, we take the previously made prediction, calculate an error of how off it was from the actual value, then run this error backwards through the network to find out how much each weight and bias parameter contributed to this error. Once we have these error derivative terms, we can nudge our weights and biases accordingly to improve our model. Do it enough times, and we'll have a neural network that can recognize handwritten digits accurately.

## The Math

That's the high level overview — now let's get into the math. This section contains quite a lot of matrix/vector operations, and just a little calculus, so be prepared!

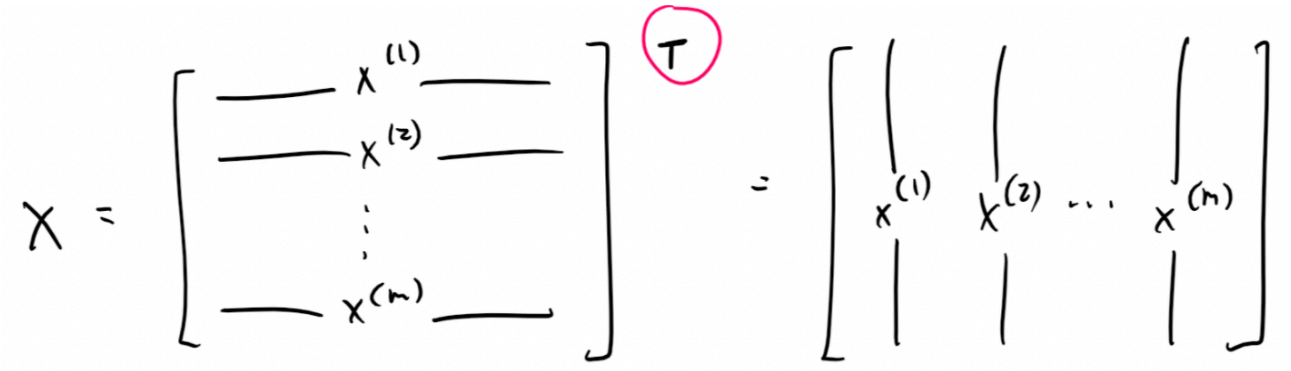
### Representing our data

As mentioned earlier, each training example can be represented by a vector with 784 elements, corresponding to each of the image's 784 pixels.

These vectors can be stacked together in a matrix to carry out vectorized calculations. That is, instead of using a for loop to go over all training examples, we can calculate error from all examples at once with matrix operations.

In most contexts, including for machine learning, the convention is to stack these vectors as rows of the matrix.

To make our math easier, we're going to transpose this matrix, with each column corresponding to a training example and each row a training feature.

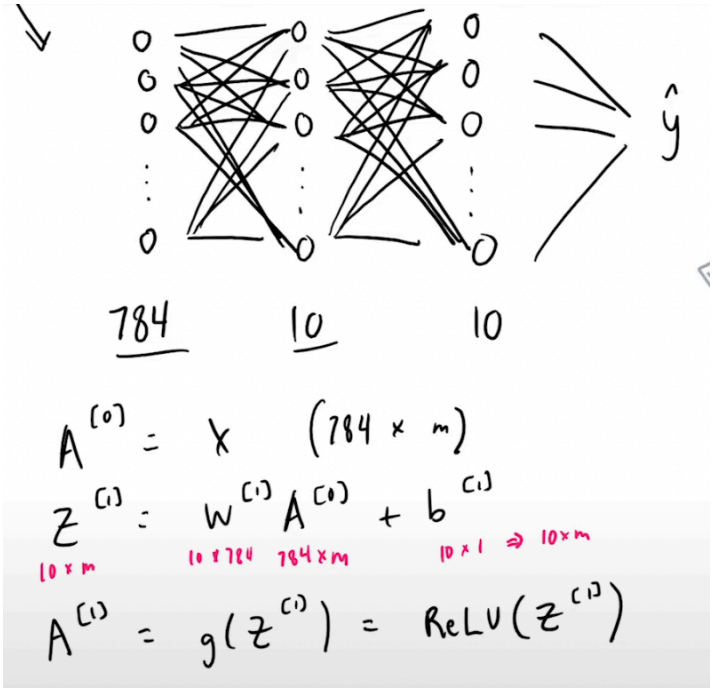


## Representing weights and biases

Let's look at our neural network now. Between every two layers is a set of connections between every node in the previous layer and every node in the following one.

## Forward propagation

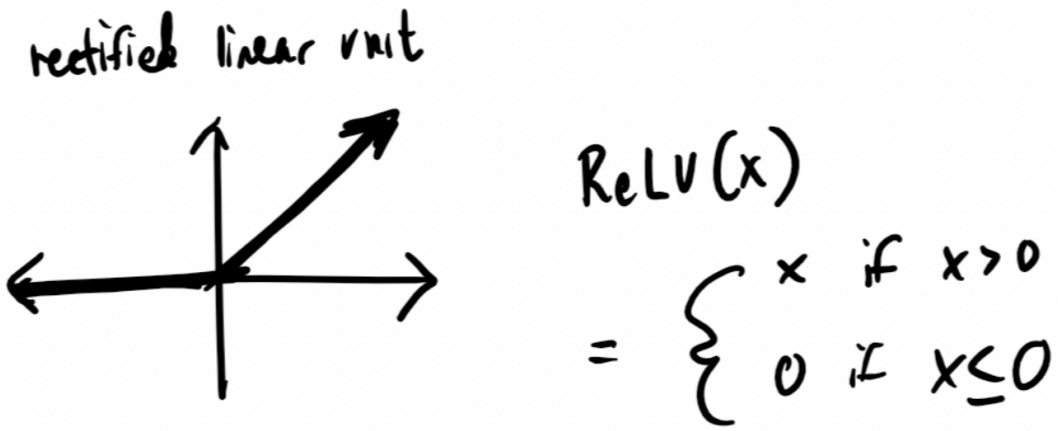
With these representations in mind, we can now write the equations for forward propagation.In forward propagation we take image to input then check for the outcome.




'ReLU is activation function (like sigmoid)to apply non-linearity.'
'A(0) is  is the first layer'
'w(1) weight from 1 to 2(same as theta)'
'b(1) is bias(same as x0)'

Imagine that we didn't do anything to Z(1) now, and multiplied it by W(2)] and added b(2) to get the value for the next layer. Z(1) is a linear combination of the input features, and the second layer would be a linear combination of Z(1) making it still a linear combination of the input features. That means that our hidden layer is essentially useless, and we're just building a linear regression model.

To prevent this reduction and actually add complexity with our layers, we'll run Z(1) through a non-linear activation function before passing it off to the next layer. In this case, we'll be using a function called a rectified linear unit, or ReLU:




From Z(1), we'll calculate a value A(1) for the values of the nodes in the hidden layer of our neural network after applying our activation function to it:

$$A(1) = Relu(Z(1))$$

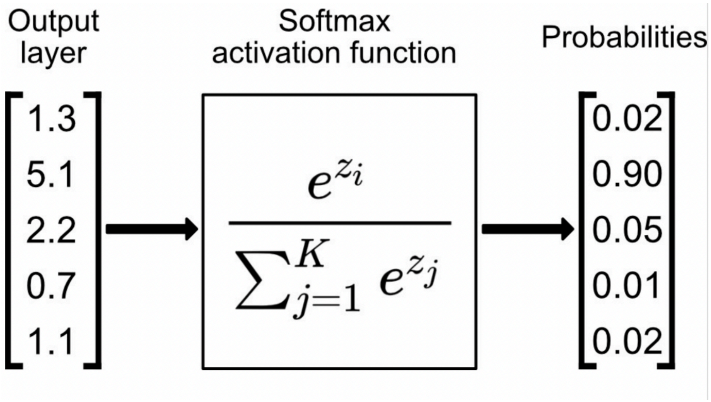
More generally, you might see this written as A(1)= g(Z(1)), with g referring to an arbitrary activation function that may be something other than ReLU.Like g can be sigmoid.

Once we have A(1), we can proceed to calculating the values for our second layer, which is also our output layer. First, we calculate Z(2):

$$Z(2) = W(2)A(1)+b(2)$$

Then, we'll apply an activation function to Z(2) to get our final output.

If this second layer were just another hidden layer, with more hidden layers or an output layer after it, we would apply ReLU again. But since it's the output layer, we'll apply a special activation function called softmax:




Softmax takes a column of data at a time, taking each element in the column and outputting the exponential of that element divided by the sum of the exponentials of each of the elements in the input column. The end result is a column of probabilities between 0 and 1.

The value of using softmax for our output layer is that we can read the output as probabilities for certain predictions. In the diagram above, for example, we might read the output as a prediction that the second class has a 90% probability of being the correct label, the third a 5% probability, the fourth a 1% probability, and so on.

Let's find our A(2) :

$$A(2)=\text{softmax}(z(2))$$

With that, we've run through the entire neural network, going from our input X containing all of our training examples to an output matrix A(2) containing prediction probabilities for each example.

## Backward propagation

Now, we'll go the opposite way and calculate how to nudge our parameters to carry out gradient descent.

Mathematically, what we're actually computing is the derivative of the loss function with respect to each weight and bias parameter. For a softmax classifier, we'll use a cross-entropy loss function:

$$dZ^{[2]} = A^{[2]} - Y \qquad (10 \times m) \quad (10 \times m) \quad (10 \times m)$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \qquad (10 \times 10) \quad (10 \times m) \quad (m \times 10)$$

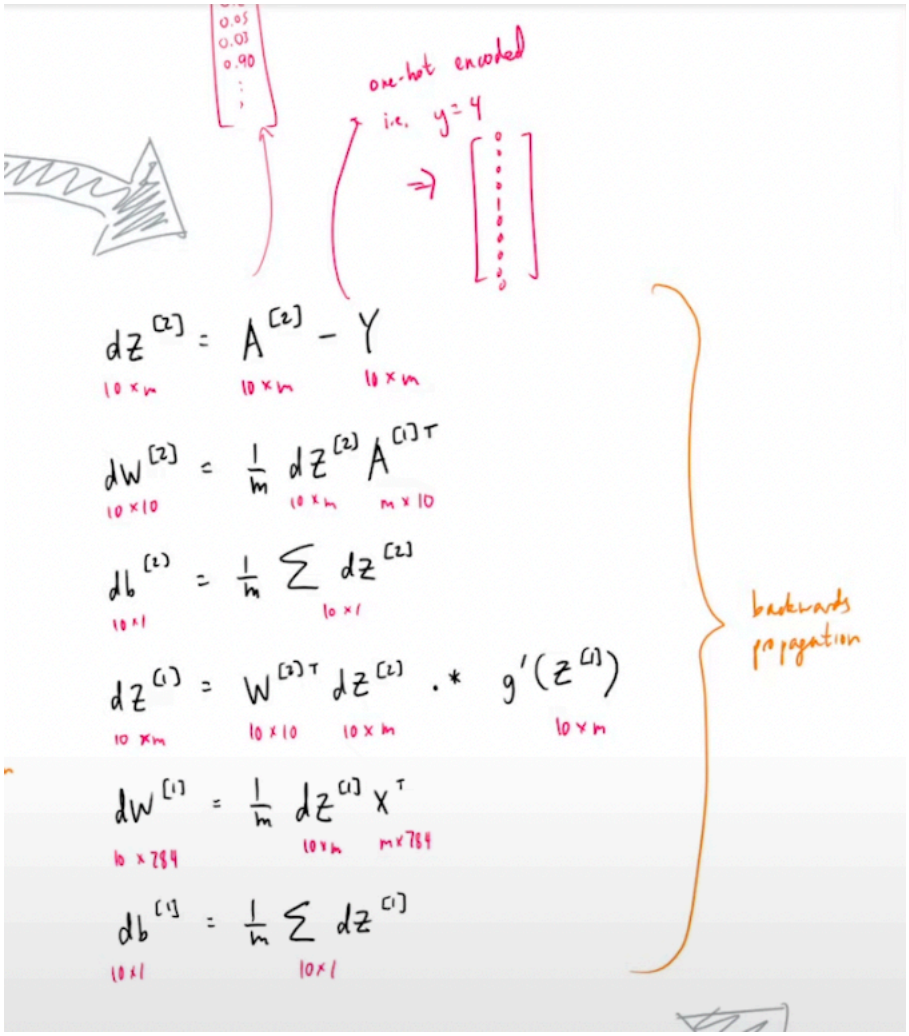
$$db^{[2]} = \frac{1}{m} \sum dZ^{[2]} \qquad (10 \times 1) \quad (10 \times 1)$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} .* \qquad (10 \times m) \quad (10 \times 10) \quad (10 \times m)$$

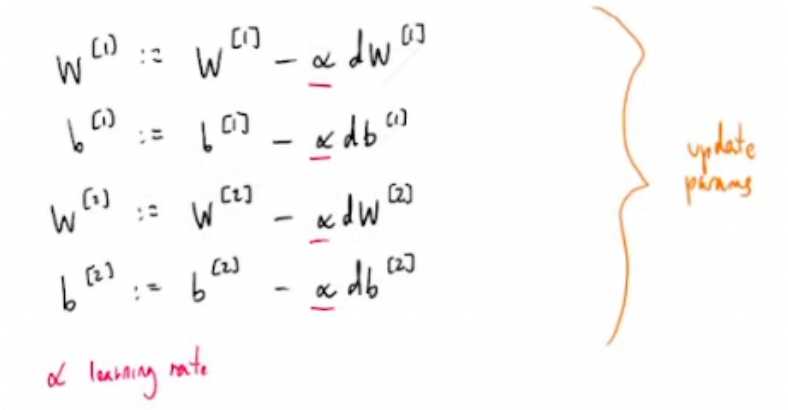
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} . \qquad (10 \times m) \quad (10 \times 10) \quad (10 \times m)$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T} \qquad (10 \times 784) \quad (10 \times m) \quad (m \times 784)$$

$$db^{[2]} = \frac{1}{m} \sum dZ^{[1]} \qquad (10 \times 1) \quad (10 \times 1)$$




By these calculation we are finding how much is the error contributed by bias term then we update our parameter/weights accordingly :



# The code

[ ]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

data = pd.read_csv('/kaggle/input/digit-recognizer/train.csv')
```

[ ]:
```
data = np.array(data)
m, n = data.shape
np.random.shuffle(data) # shuffle before splitting into dev and training sets

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
```

[ ]:
```
Y_train
```

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**

$$
\begin{aligned}
Z^{[1]} &= W^{[1]} X + b^{[1]} \\
A^{[1]} &= g_{\text{ReLU}}(Z^{[1]}) \\
Z^{[2]} &= W^{[2]} A^{[1]} + b^{[2]} \\
A^{[2]} &= g_{\text{softmax}}(Z^{[2]})
\end{aligned}
$$

**Backward propagation**

$$
\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\
dB^{[2]} &= \frac{1}{m} \Sigma dZ^{[2]} \\
dZ^{[1]} &= W^{[2]T} dZ^{[2]} .* g^{[1]\prime}(z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} A^{[0]T} \\
dB^{[1]} &= \frac{1}{m} \Sigma dZ^{[1]}
\end{aligned}
$$

Link for code: https://www.kaggle.com/nipunrajeev/simple-mnist-nn-from-scratch-numpy-no-tf-keras

**Parameter updates**

$$
\begin{aligned}
W^{[2]} &:= W^{[2]} - \alpha dW^{[2]} \\
b^{[2]} &:= b^{[2]} - \alpha db^{[2]} \\
W^{[1]} &:= W^{[1]} - \alpha dW^{[1]} \\
b^{[1]} &:= b^{[1]} - \alpha db^{[1]}
\end{aligned}
$$

**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

```python
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

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

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

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

def ReLU_deriv(Z):
    return Z > 0

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 backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2

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
```

```
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, alpha, iterations):
    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 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2
```

```
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.10, 500)
```

~85% accuracy on training set.

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

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()
```

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)
```

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)
```

Still 84% accuracy, so our model generalized from the training data pretty well.