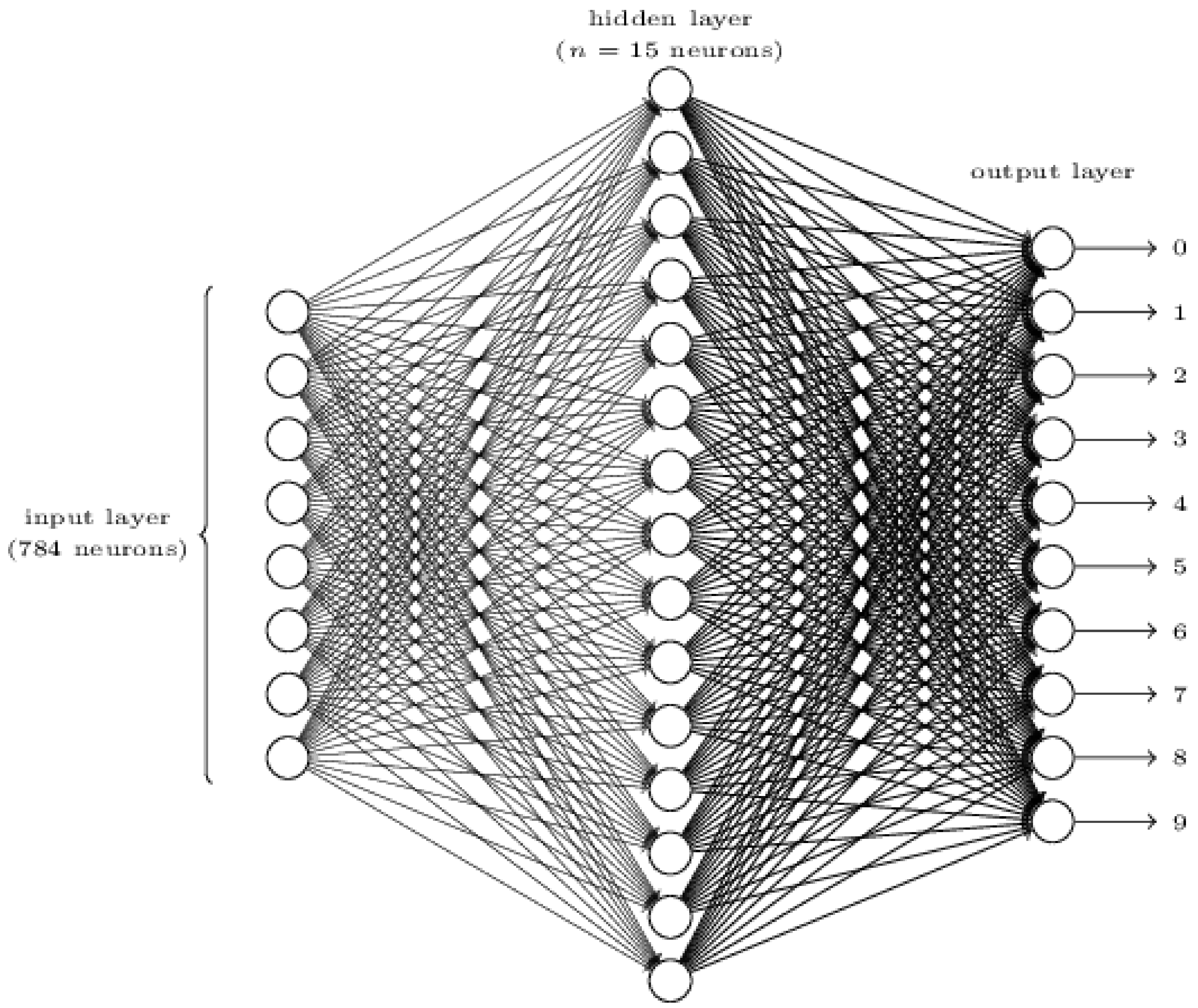
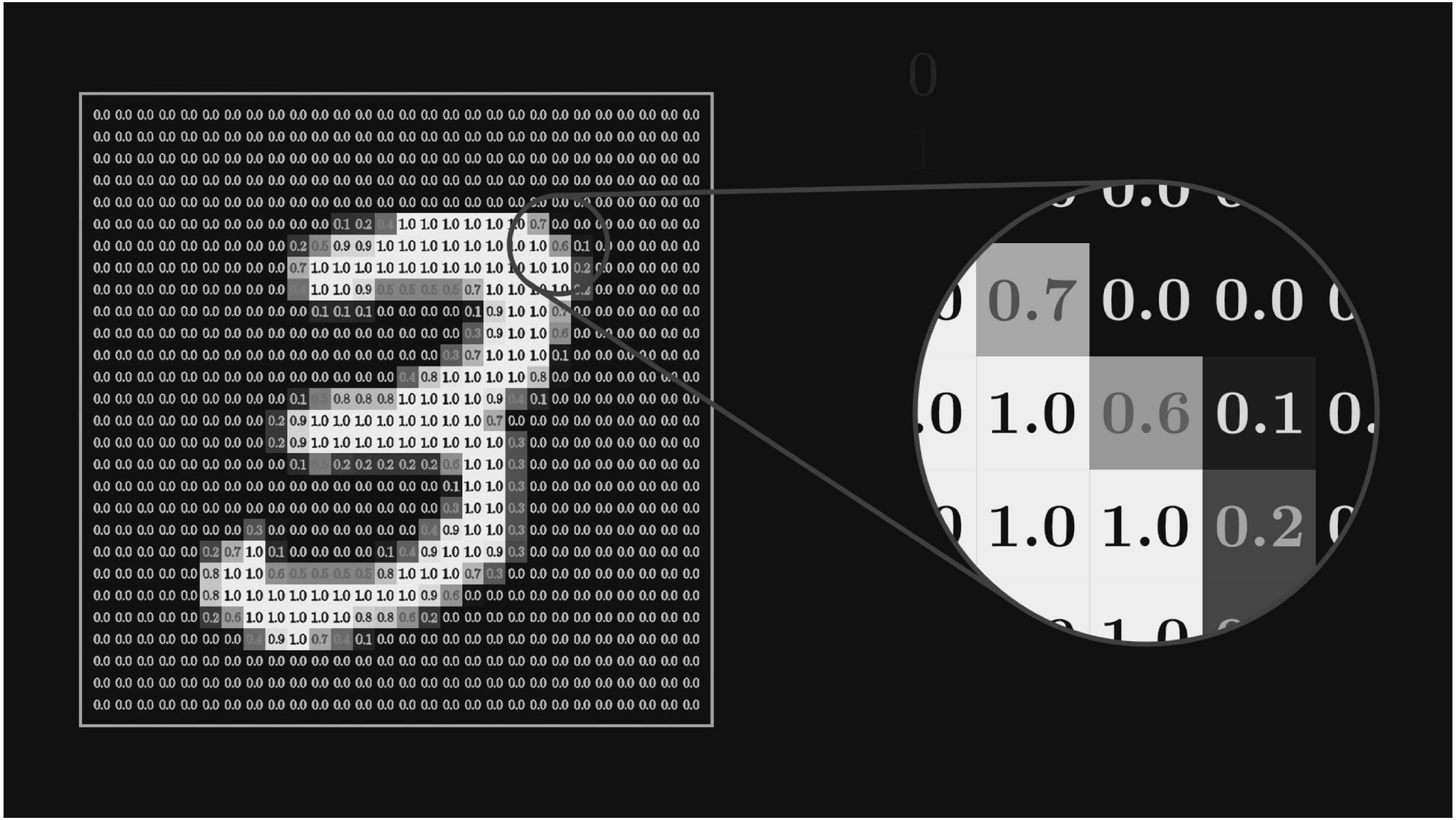
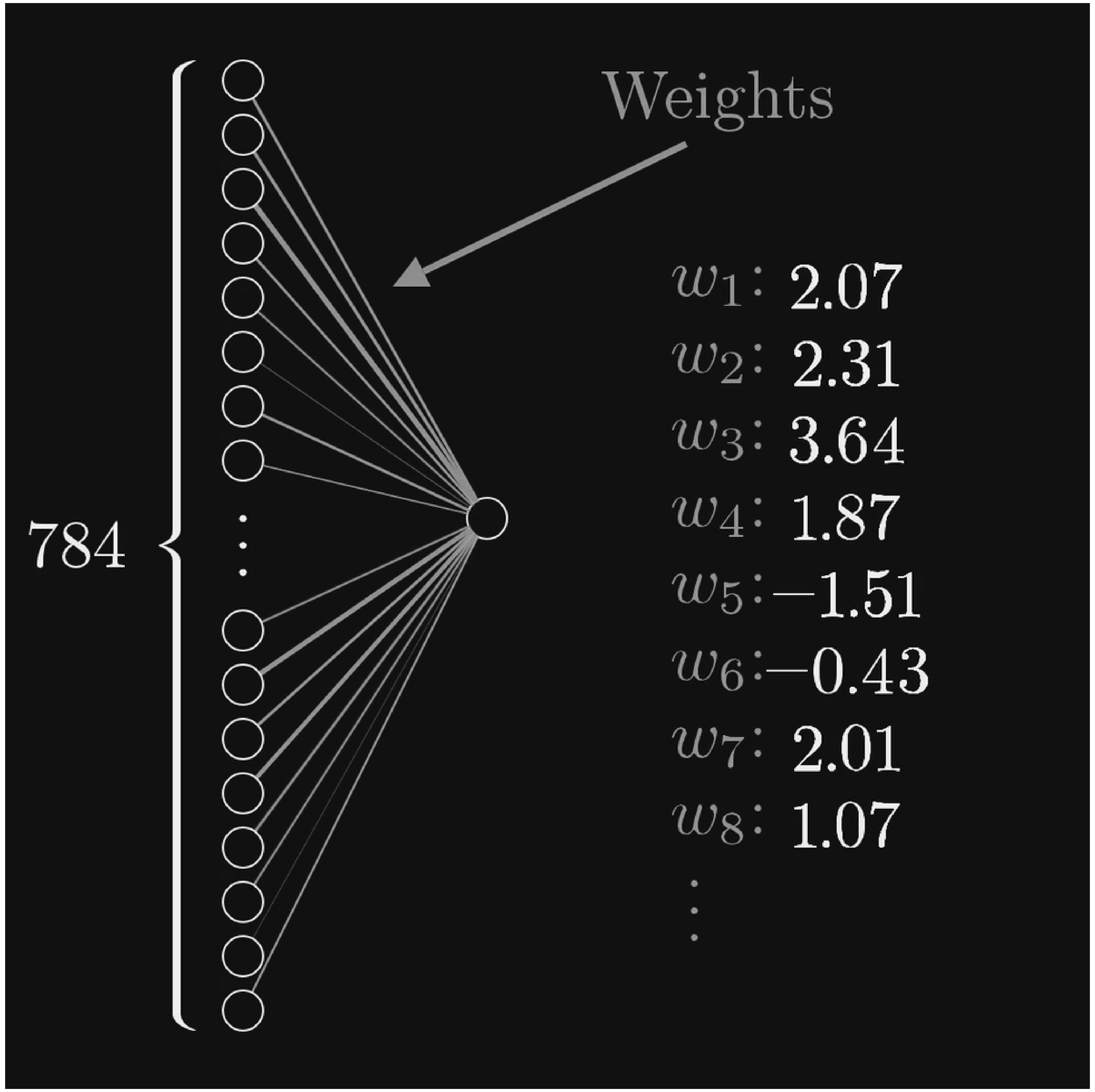


THE MULTI LAYER PERCEPTRON



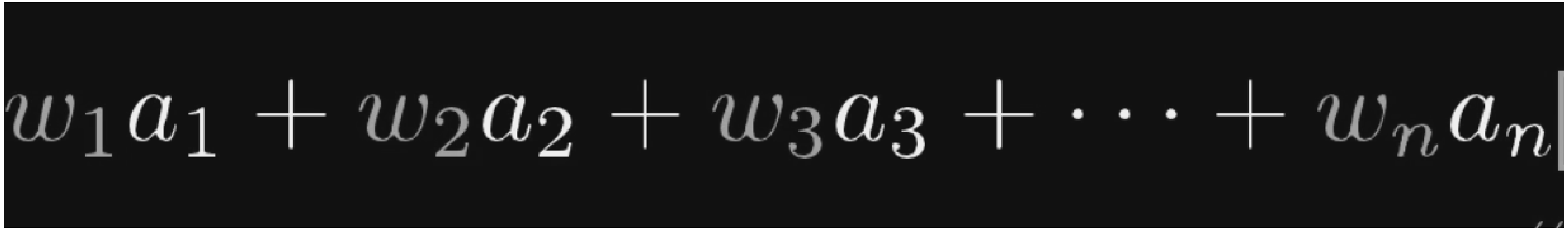
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.4	1.0	1.0	1.0	1.0	1.0	1.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.5	0.9	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.6	0.1	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.2	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.0	1.0	0.9	0.5	0.5	0.5	0.5	0.7	1.0	1.0	1.0	1.0	0.2	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.1	0.9	1.0	1.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.9	1.0	1.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.7	1.0	1.0	1.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.8	1.0	1.0	1.0	1.0	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.5	0.8	0.8	0.8	1.0	1.0	1.0	1.0	0.9	0.4	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.5	0.2	0.2	0.2	0.2	0.2	0.6	1.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	1.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.9	1.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.2	0.7	1.0	0.1	0.0	0.0	0.0	0.0	0.1	0.4	0.9	1.0	1.0	0.9	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.8	1.0	1.0	0.6	0.5	0.5	0.5	0.5	0.8	1.0	1.0	1.0	0.7	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.9	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.2	0.6	1.0	1.0	1.0	1.0	1.0	0.8	0.8	0.6	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.9	1.0	0.7	0.4	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

We have a 28*28 image of a number. We first flatten it to a 784-length vector. This essentially is our first input layer. The output layer comprises 10 neurons, each would output a probability that the NN thinks that that is the number in the image.



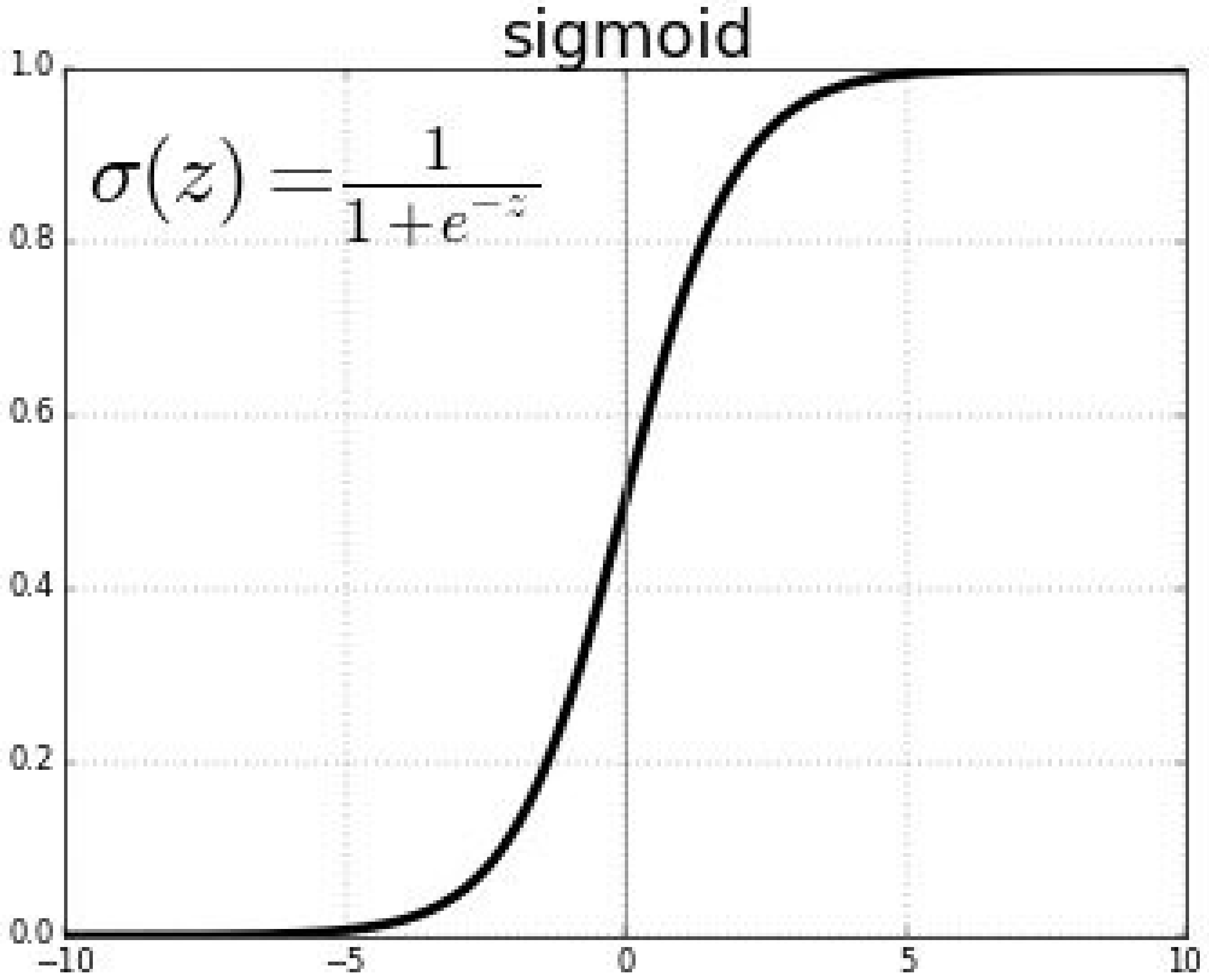
Just like in the case of a single perception, we have weights associated with all the neuron in the previous layer with neurons in the current layer. Have a look at the image.

Role of the hidden Neurons?

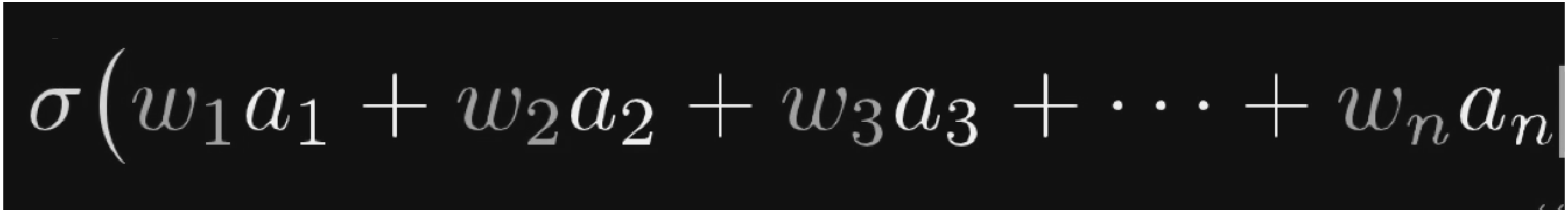


$$w_1 a_1 + w_2 a_2 + w_3 a_3 + \cdots + w_n a_n$$

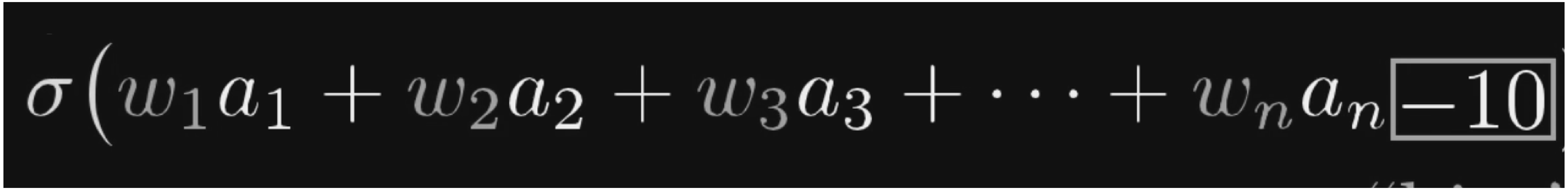
The Sigmoid Function



Our final function looks like this, the output lies b/w 0 & 1



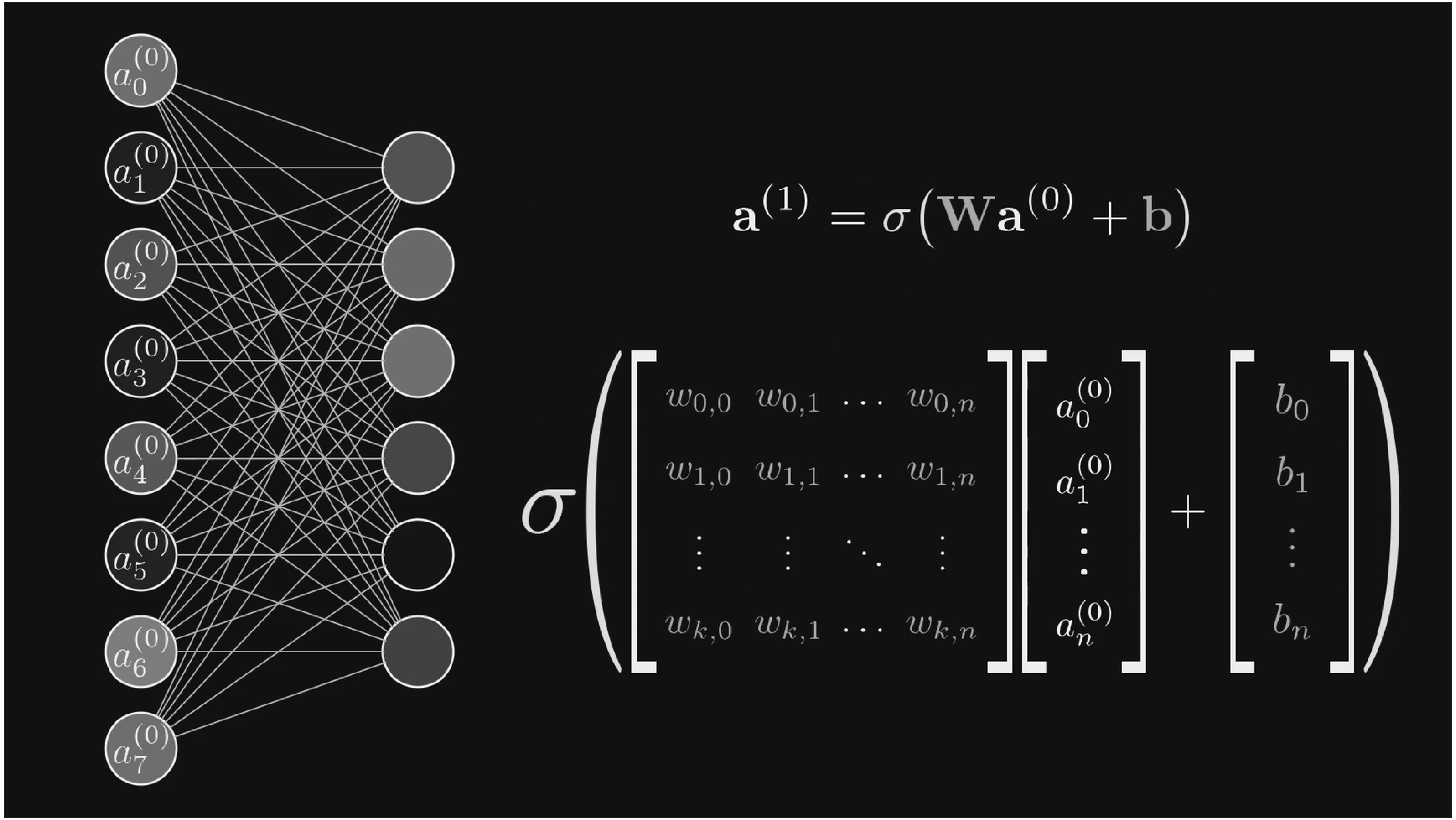
$$\sigma(w_1 a_1 + w_2 a_2 + w_3 a_3 + \cdots + w_n a_n$$



$$\sigma\left(w_1 a_1 + w_2 a_2 + w_3 a_3 + \cdots + w_n a_n \boxed{-10}\right.$$

The bias term allows us to control when a neuron activates. In this case, only if the weighted activations add up to or greater than 10 will it spit out a positive value.

An easier representation.



$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}\mathbf{a}^{(0)} + \mathbf{b}\right)$$

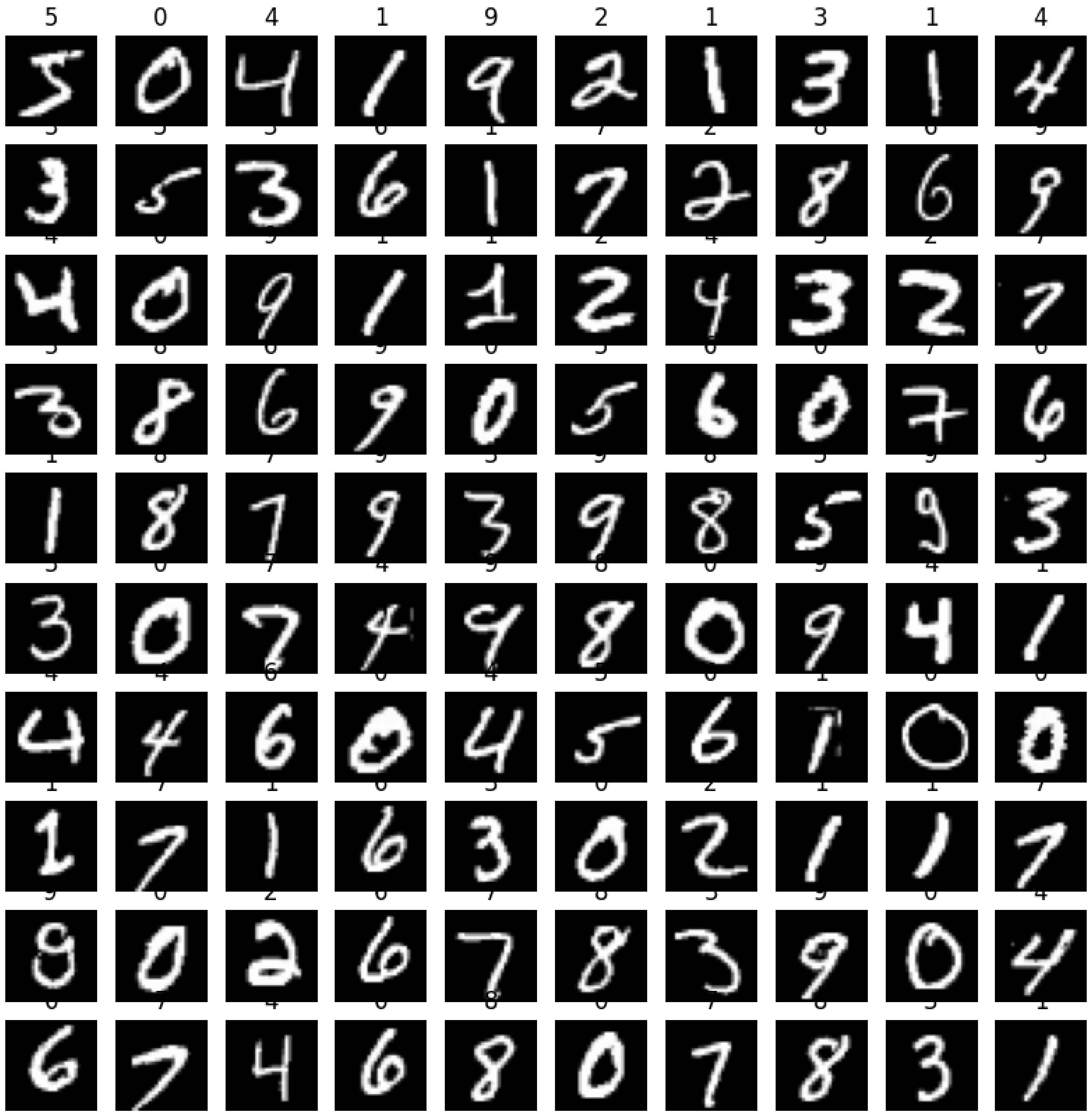
$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix}\begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

$$a_0^{(1)} = \sigma\left(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + \cdots + w_{0,n}a_n^{(0)} + b_0\right)$$

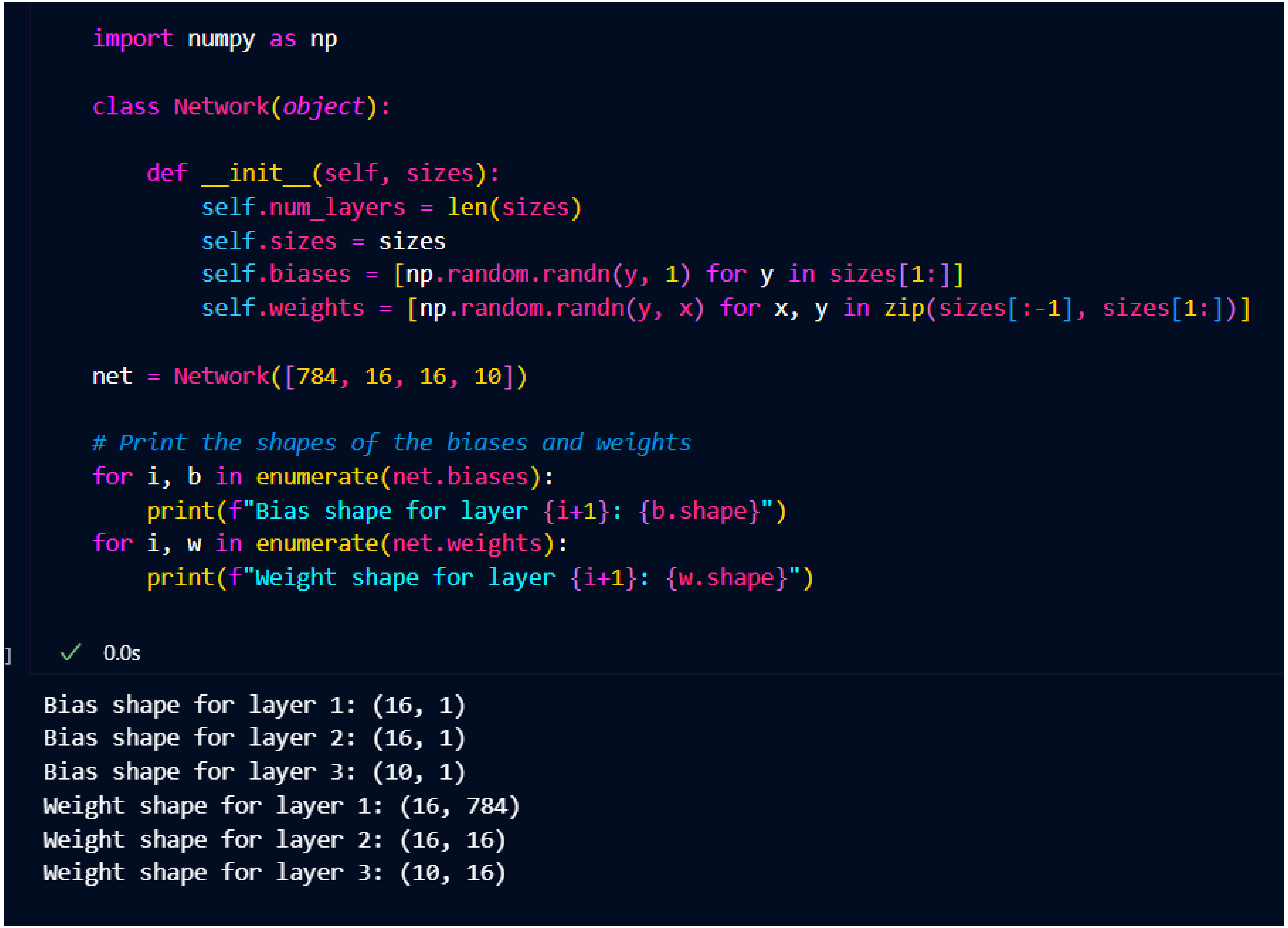
How do we make it "learn"?

The MNIST data set

MNIST Digit Data

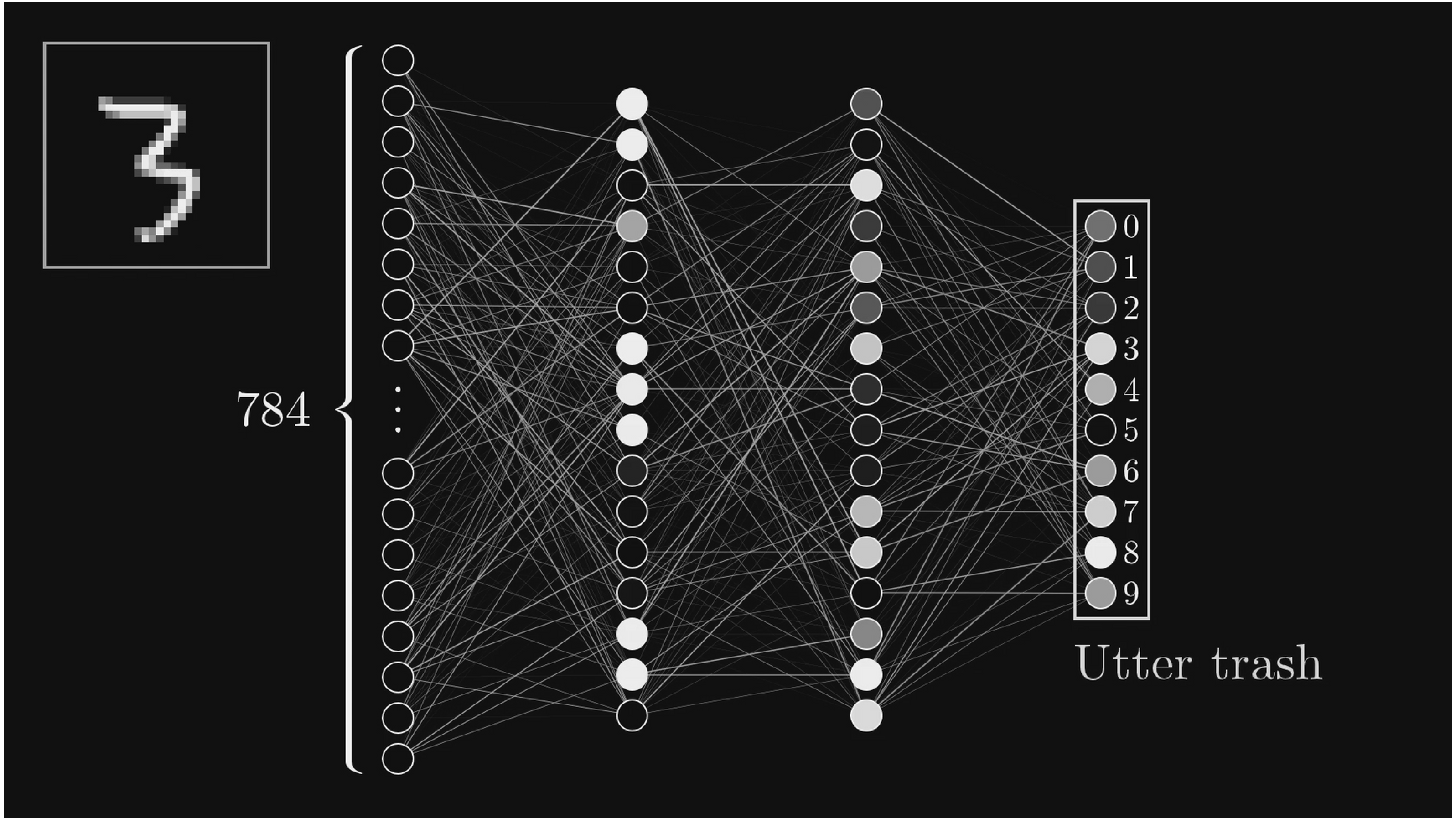


We begin by initializing random weights and biases. Pay attention to the sizes of the weight and bias matrix.

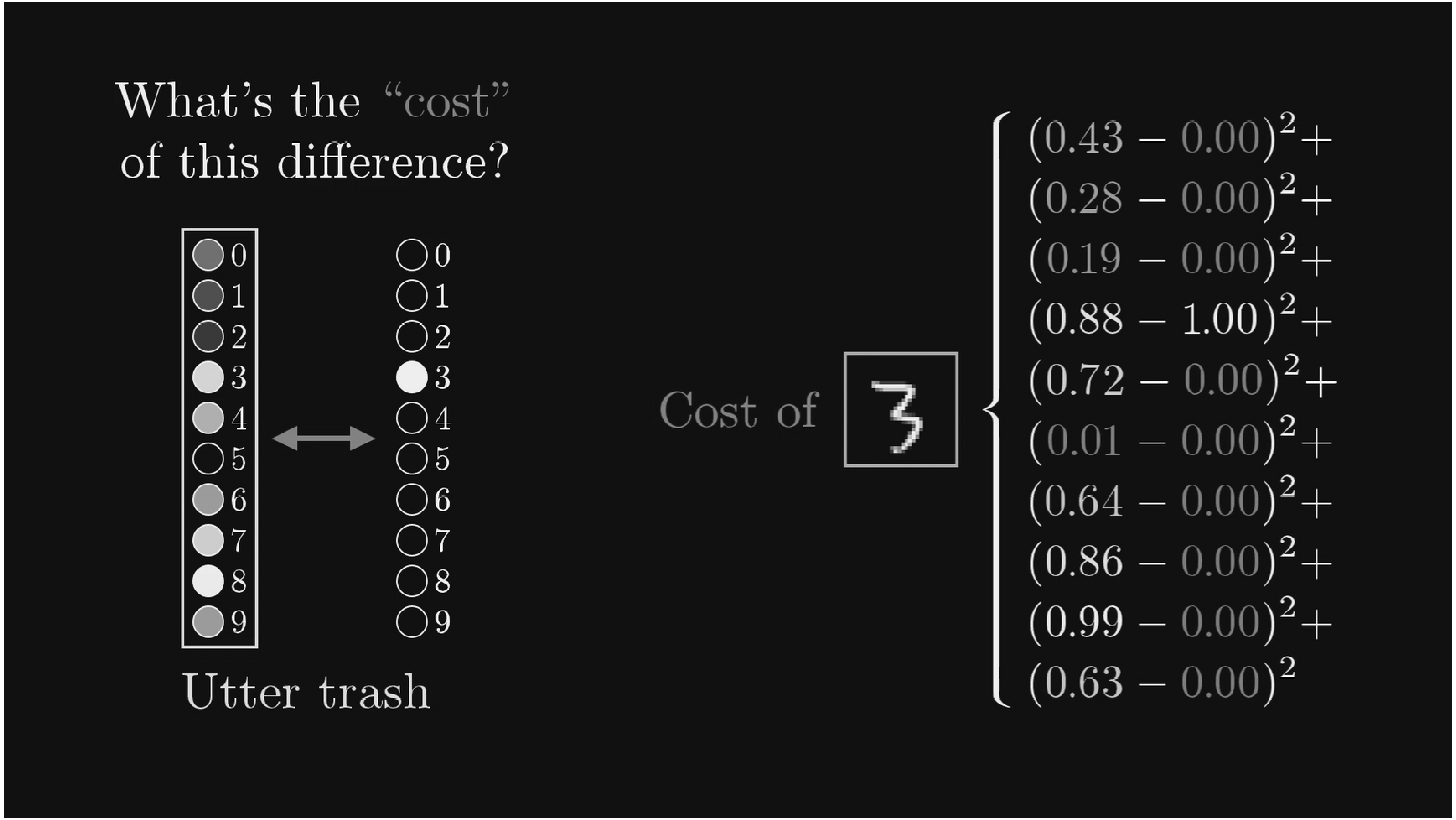


```python
import numpy as np

class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

net = Network([784, 16, 16, 10])

# Print the shapes of the biases and weights
for i, b in enumerate(net.biases):
    print(f"Bias shape for layer {i+1}: {b.shape}")
for i, w in enumerate(net.weights):
    print(f"Weight shape for layer {i+1}: {w.shape}")
```

```
Bias shape for layer 1: (16, 1)
Bias shape for layer 2: (16, 1)
Bias shape for layer 3: (10, 1)
Weight shape for layer 1: (16, 784)
Weight shape for layer 2: (16, 16)
Weight shape for layer 3: (10, 16)
```



The output would be **random with the initial random parameters.**



What's the "cost" of this difference?

Utter trash

Cost of $\boxed{3}$

$$\left\{\begin{array}{l}(0.43-0.00)^2+\\(0.28-0.00)^2+\\(0.19-0.00)^2+\\(0.88-1.00)^2+\\(0.72-0.00)^2+\\(0.01-0.00)^2+\\(0.64-0.00)^2+\\(0.86-0.00)^2+\\(0.99-0.00)^2+\\(0.63-0.00)^2\end{array}\right.$$

The "cost" is calculated by adding up the squares of the differences between what we got and what we want.

But we aren’t just interested in how the network performs on a single image. We need to consider the average cost over all the tens of thousands of training examples to measure its performance. This average is our measure of how lousy the network is and how bad the computer should feel.

This is, to put it lightly, a complicated function. Remember how the network itself is a function? It has 784 inputs (pixel values), 10 outputs, and **13,002** parameters.

How?

The Gradient Descent

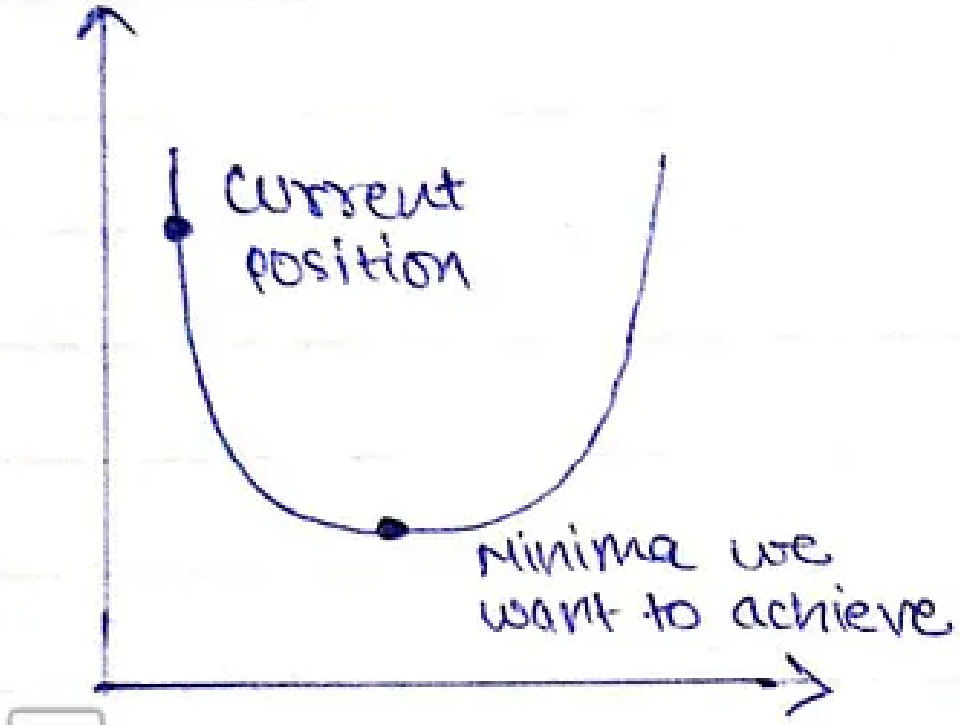
Gradient descent is an iterative method that starts with a random model parameter and uses the gradient of the cost function concerning the model parameter to determine the direction in which the model parameter should be updated.

Suppose we have a cost function $C(\theta)$, where θ represents the model parameters. We know the gradient of the cost function concerning θ gives us the direction of maximum increase of $C(\theta)$ in a linear sense at the value of θ at which the gradient is evaluated. So, to get the direction of the maximum decrease of $C(\theta)$ in a linear sense, one should use the negative of the gradient.

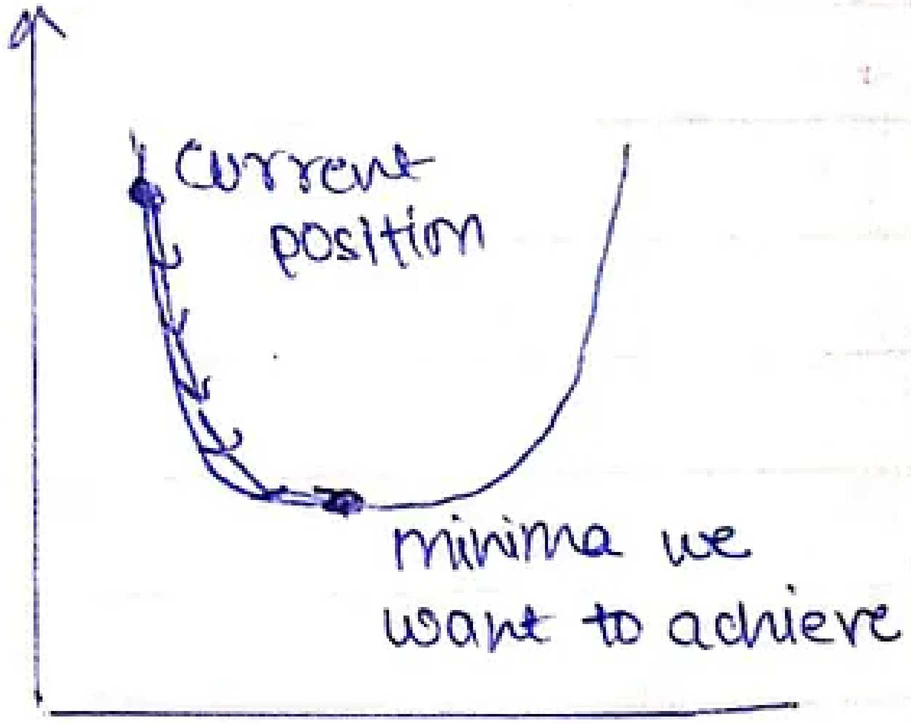
$$\theta_{\text{new}} = \theta_{\text{old}} - \eta\left(\frac{\partial C(\theta)}{\partial\theta_{(}old)}\right)$$

What is η?

The learning rate plays a vital role in converging the gradient descent to the minimum point. If the learning rate is large, the convergence might be faster but lead to severe oscillations around the minima point. A small learning rate might take longer to reach the minimum, but the convergence is generally oscillation free.

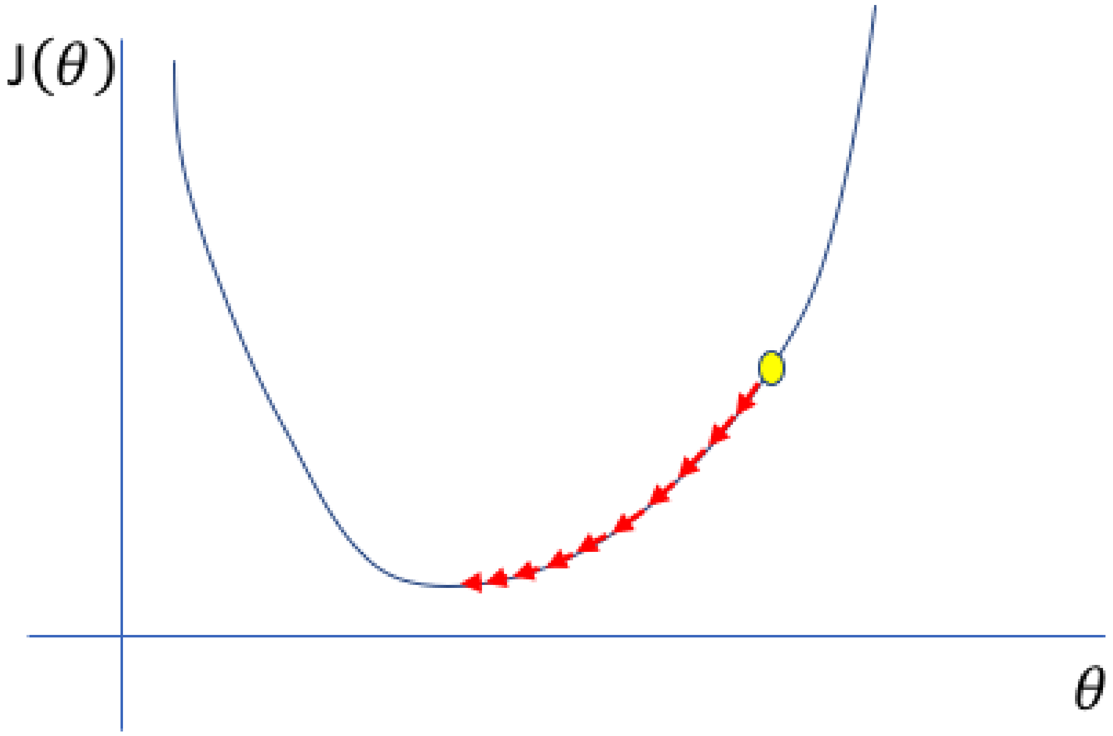


current position
minima we want to achieve



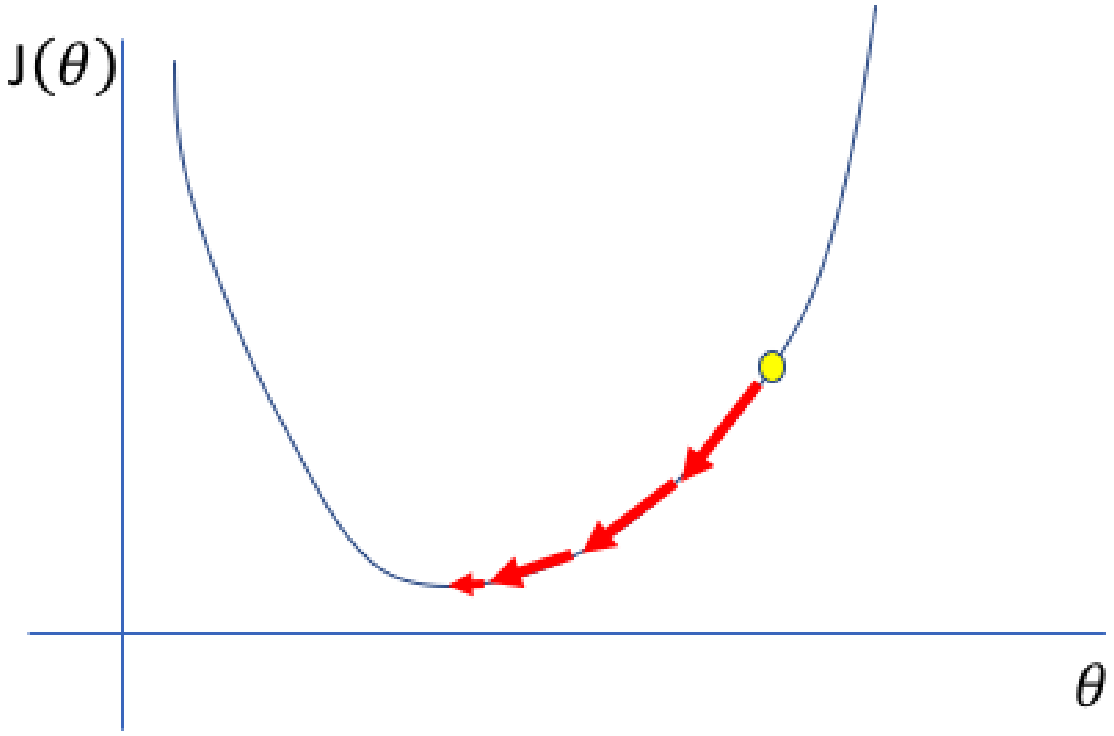
current position
minima we want to achieve

Too low



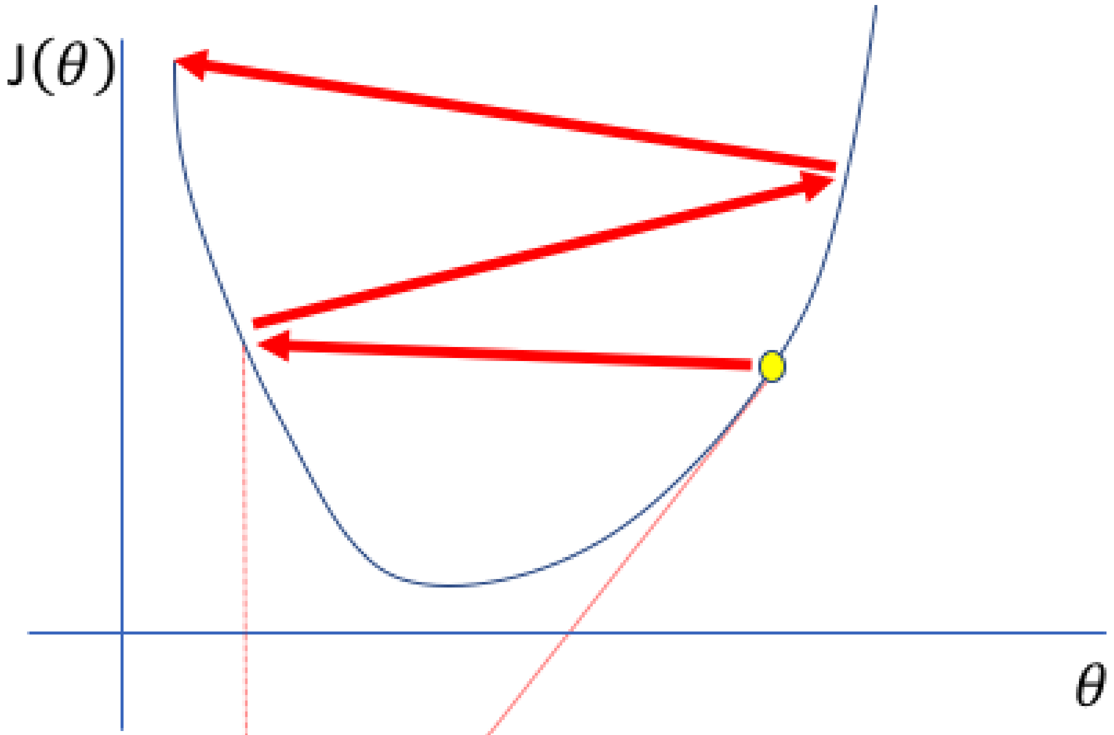
A small learning rate requires many updates before reaching the minimum point

Just right

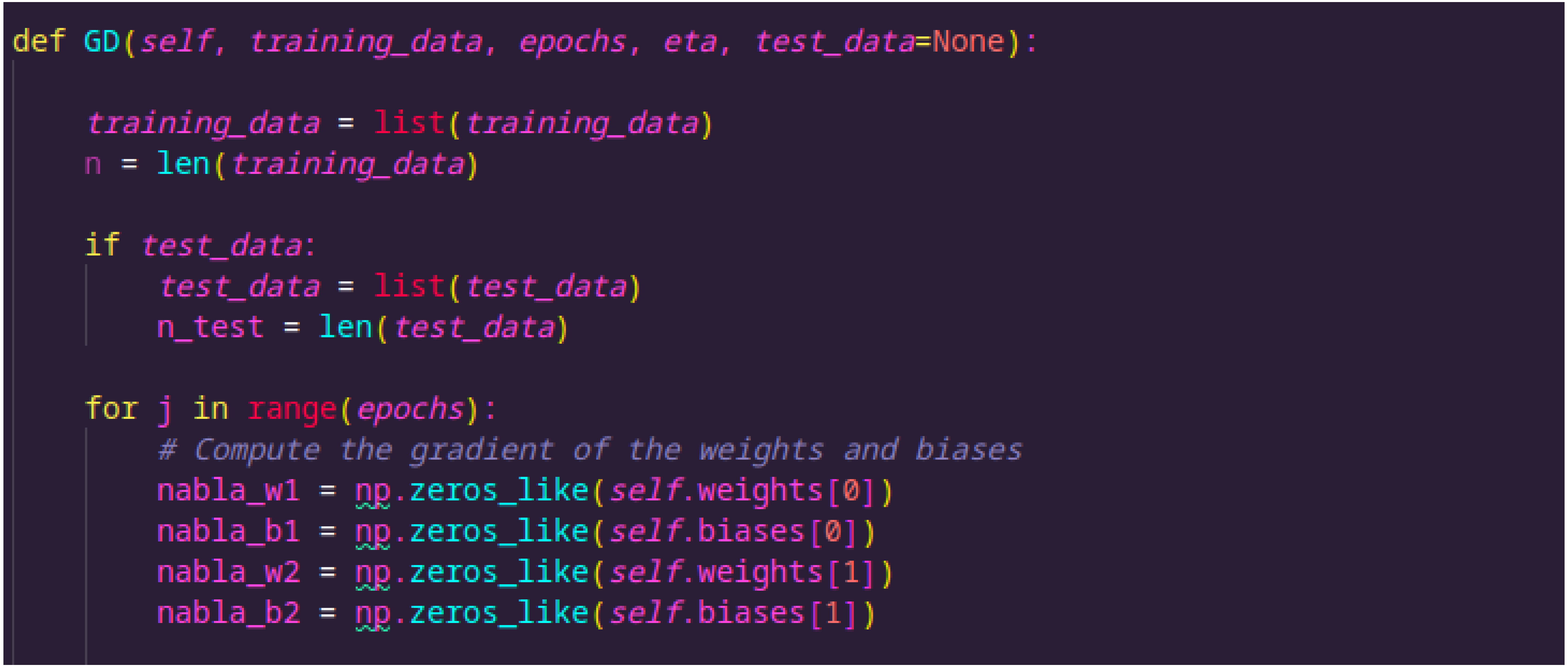


The optimal learning rate swiftly reaches the minimum point

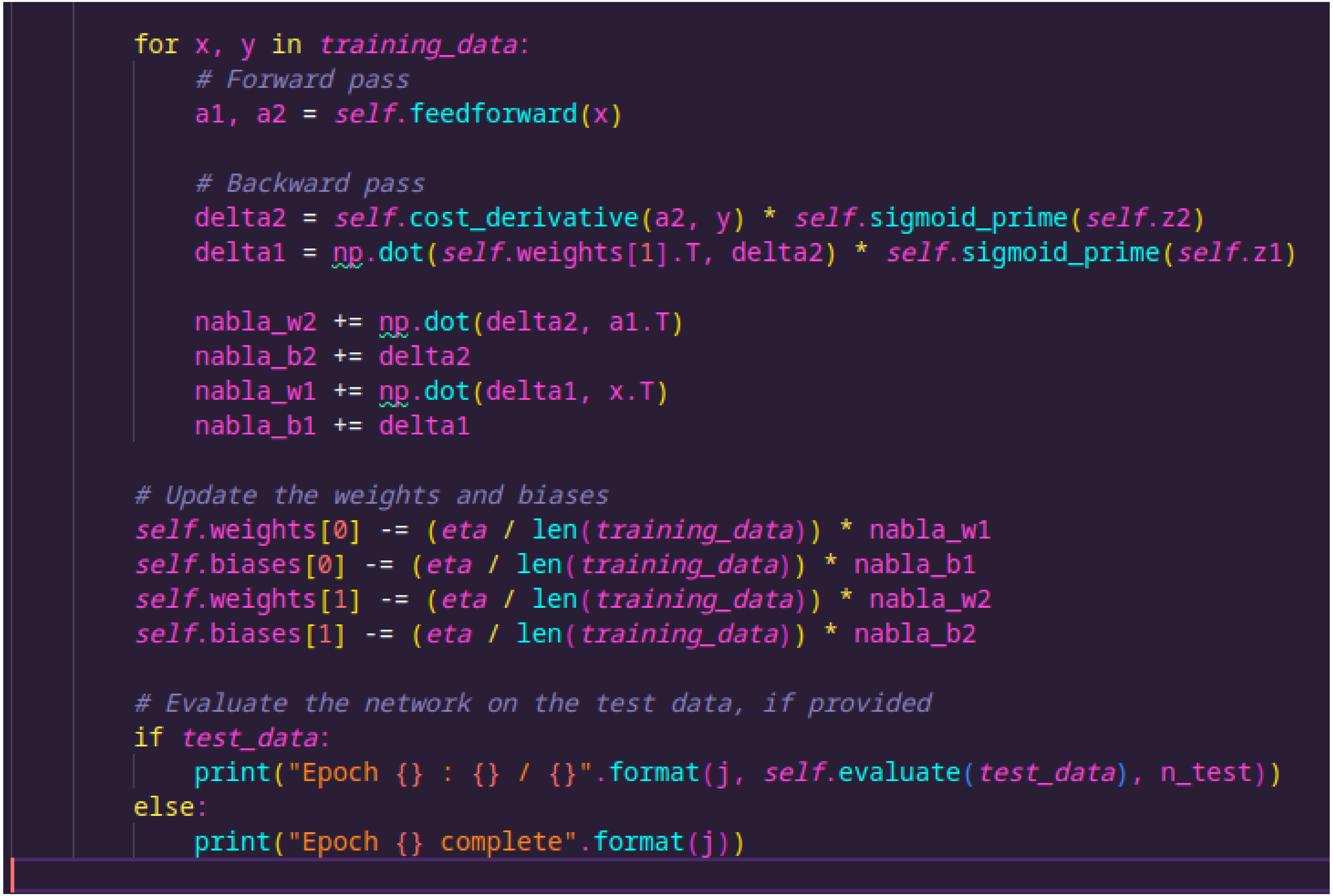
Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors



```python
def GD(self, training_data, epochs, eta, test_data=None):

    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)

    for j in range(epochs):
        # Compute the gradient of the weights and biases
        nabla_w1 = np.zeros_like(self.weights[0])
        nabla_b1 = np.zeros_like(self.biases[0])
        nabla_w2 = np.zeros_like(self.weights[1])
        nabla_b2 = np.zeros_like(self.biases[1])
```



```
        for x, y in training_data:
            # Forward pass
            a1, a2 = self.feedforward(x)

            # Backward pass
            delta2 = self.cost_derivative(a2, y) * self.sigmoid_prime(self.z2)
            delta1 = np.dot(self.weights[1].T, delta2) * self.sigmoid_prime(self.z1)

            nabla_w2 += np.dot(delta2, a1.T)
            nabla_b2 += delta2
            nabla_w1 += np.dot(delta1, x.T)
            nabla_b1 += delta1

        # Update the weights and biases
        self.weights[0] -= (eta / len(training_data)) * nabla_w1
        self.biases[0] -= (eta / len(training_data)) * nabla_b1
        self.weights[1] -= (eta / len(training_data)) * nabla_w2
        self.biases[1] -= (eta / len(training_data)) * nabla_b2

        # Evaluate the network on the test data, if provided
        if test_data:
            print("Epoch {} : {} / {}".format(j, self.evaluate(test_data), n_test))
        else:
            print("Epoch {} complete".format(j))
```

Stochastic Gradient Descent

In both gradient descent (GD) and stochastic gradient descent (SGD), you update a set of parameters in an iterative manner to minimize an error function.

The GD algorithm can be summarized as follows:-

1. Initialize the model parameters to random values.
2. Compute the cost function over the entire training dataset using the current model parameters.
3. Compute the gradient of the cost function with respect to the model parameters.
4. Update the model parameters in the opposite direction of the gradient, multiplied by a learning rate hyperparameter.
5. Repeat steps 2-4 until convergence or for a fixed number of iterations.

Stochastic gradient descent (SGD) is a variant of gradient descent that updates the model parameters based on the gradient of the cost function concerning the model parameters computed using a single training example at a time. The steps of SGD can be summarized as follows:

1. Initialize the model parameters to random values.
2. Shuffle the training dataset.
3. For each training example:

    a. Compute the cost function using the current model parameters and training example.

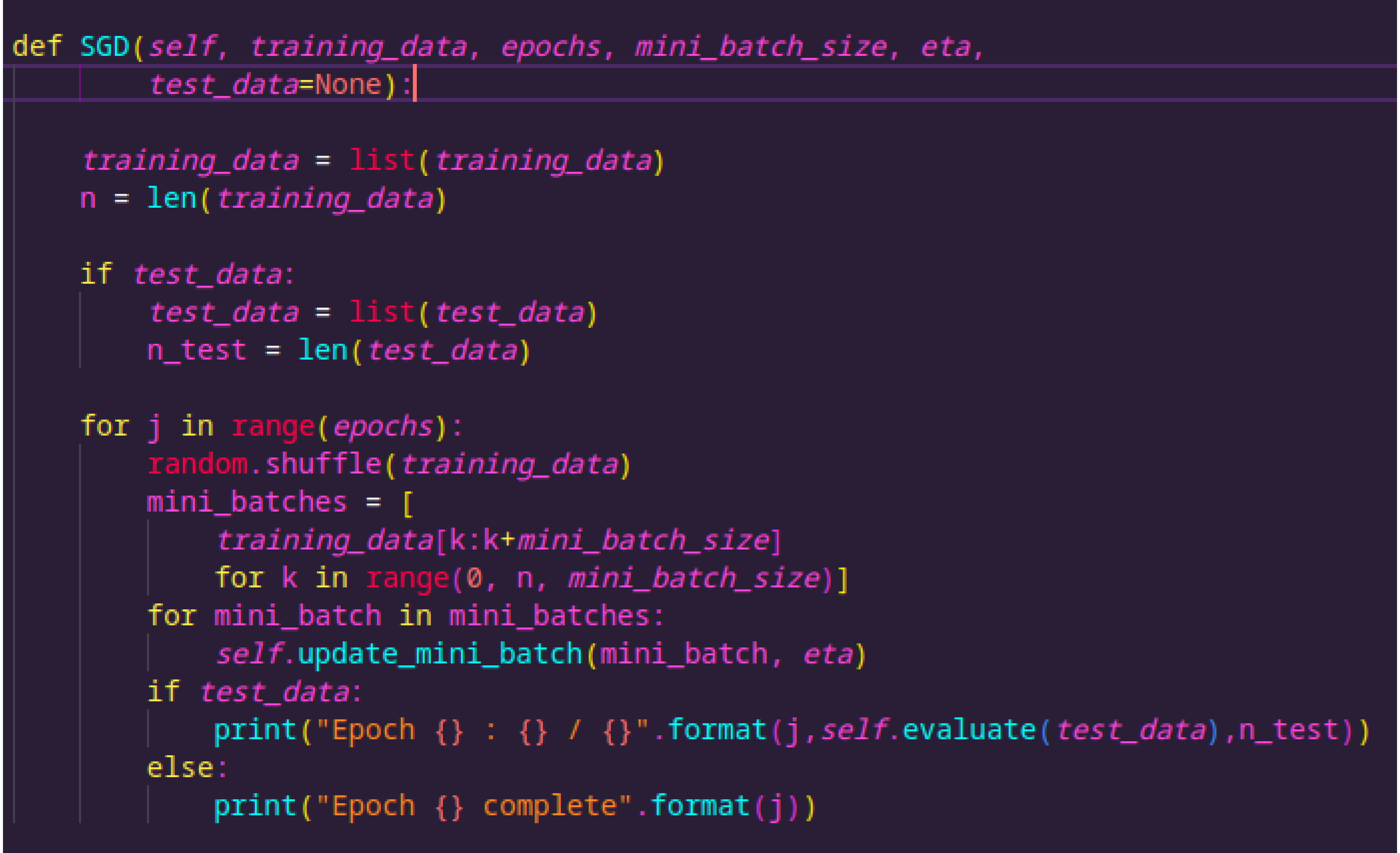
    b. Compute the gradient of the cost function concerning the model parameters using the current training example.

    c. Update the model parameters in the opposite direction of the gradient, multiplied by a learning rate hyperparameter.

1. Repeat steps 2–3 for a fixed number of epochs or until convergence.

Cost function
from a random
ith sample.

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta\left(\frac{\partial C\underline{\mathrm{i}}(\theta)}{\partial\theta_{(}old)}\right)$$

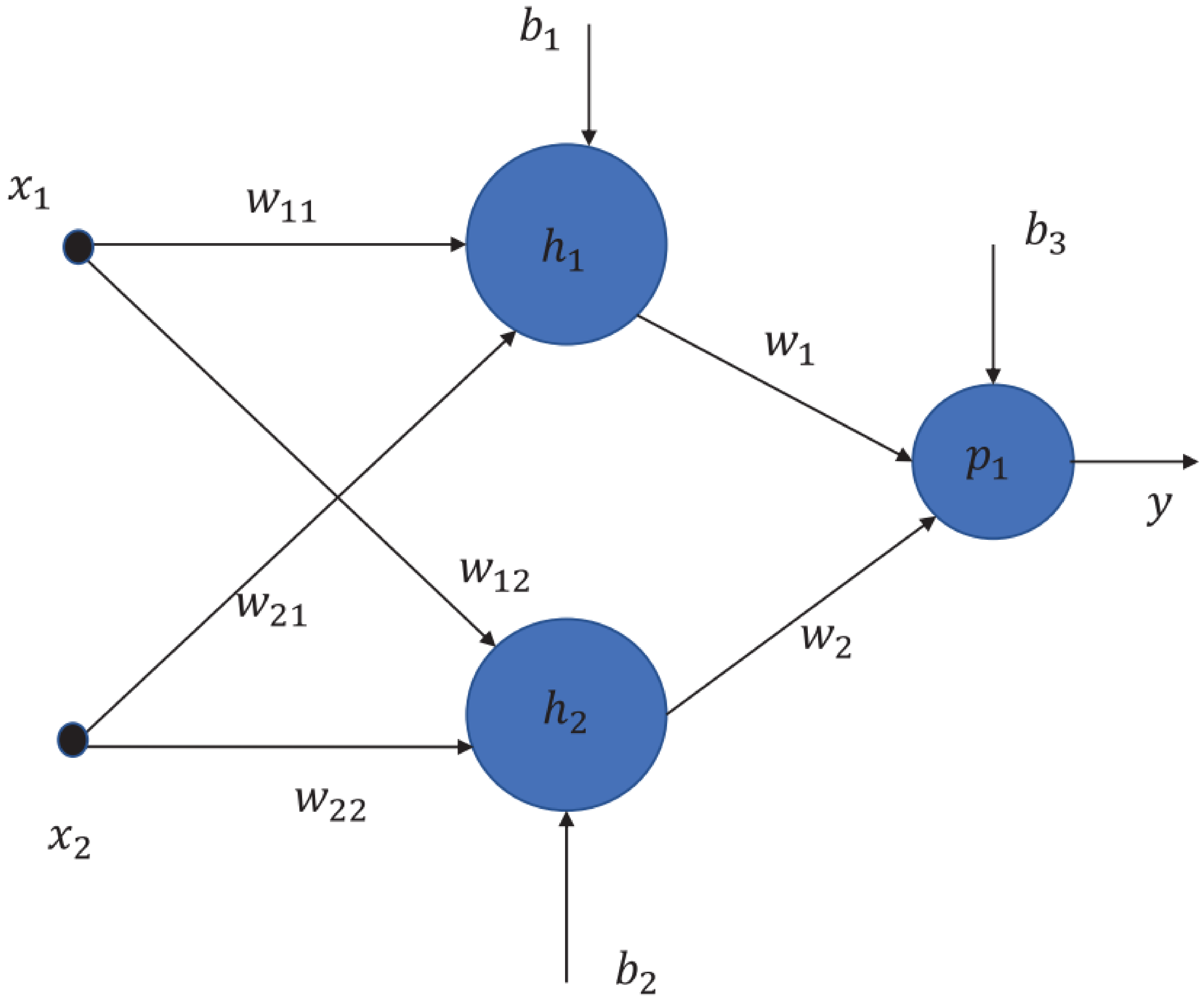


```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):

    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)

    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test))
        else:
            print("Epoch {} complete".format(j))
```

How will we compute the gradient of the Cost function vector?

Answer:- The Backpropagation Algorithm

Consider a simple Neural Network.



$$i_1 = w_{11}x_1 + w_{21}x_2 + b_1$$

$$i_2 = w_{12}x_1 + w_{22}x_2 + b_2$$

$$z_1 = 1/\left(1 + e^{-i_1}\right)$$

$$z_2 = 1/\left(1 + e^{-i_2}\right)$$

$$i_3 = w_1 z_1 + w_2 z_2 + b_3$$

$$z_3 = 1/\left(1 + e^{-i_3}\right)$$

To find the best values for w11, w12, w21, w22, w1, w2, b1, b2 & b3: We would be requiring the following gradients:

dC/dw1, dC/dw2, dC/db1, dC/db2, dC/db3, dC/dw11, dC/dw12, dC/dw21, dC/dw22

$$C = -ylogz_3 - (1-y)\log(1-z_3)$$

$$\frac{\partial C}{\partial w_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial w_1}$$

$$\frac{dC}{dz_3} = \frac{(z_3 - y)}{z_3(1-z_3)}$$

$$\frac{\partial i_3}{\partial w_1} = z_1$$

$$\frac{\partial C}{\partial w_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial w_1} = (z_3 - y)z_1$$

$$\frac{\partial C}{\partial w_2} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial w_2} = (z_3 - y)z_2$$

$$\frac{\partial C}{\partial b_3} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial b_3} = (z_3 - y)$$

Now $z_3 = 1/(1+e^{-z_3})$

$$\frac{dz_3}{di_3} = z_3\left(1-z_3\right)$$

$$\frac{dC}{di_3} = \frac{dC}{dz_3}\frac{dz_3}{di_3} = \frac{\left(z_3 - y\right)}{z_3\left(1-z_3\right)} z_3\left(1-z_3\right) = \left(z_3 - y\right)$$

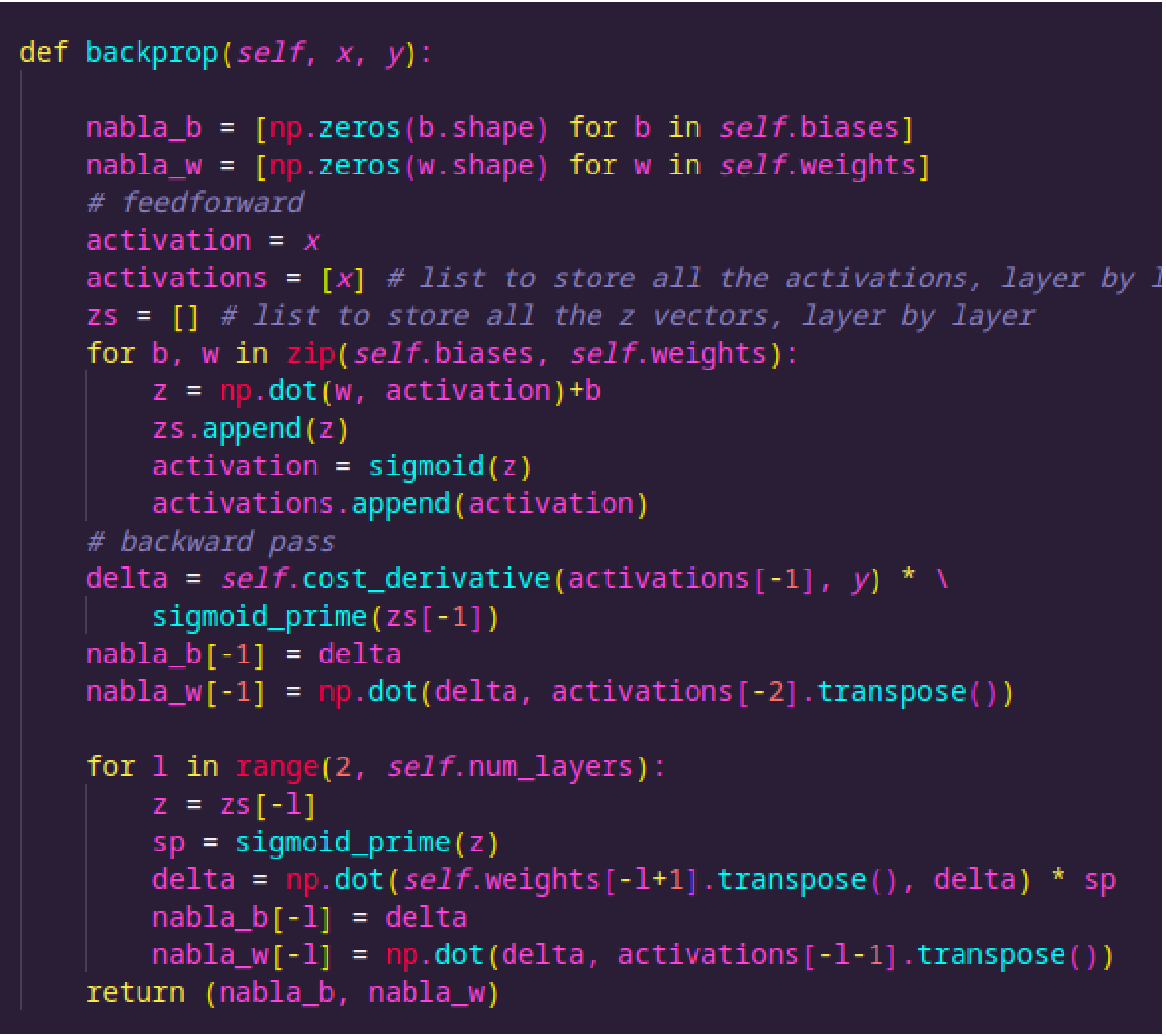
$$\frac{\partial C}{\partial z_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1} = (z_3 - y)w_1$$

$$\frac{\partial C}{\partial i_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1} = (z_3 - y)w_1 z_1(1 - z_1)$$

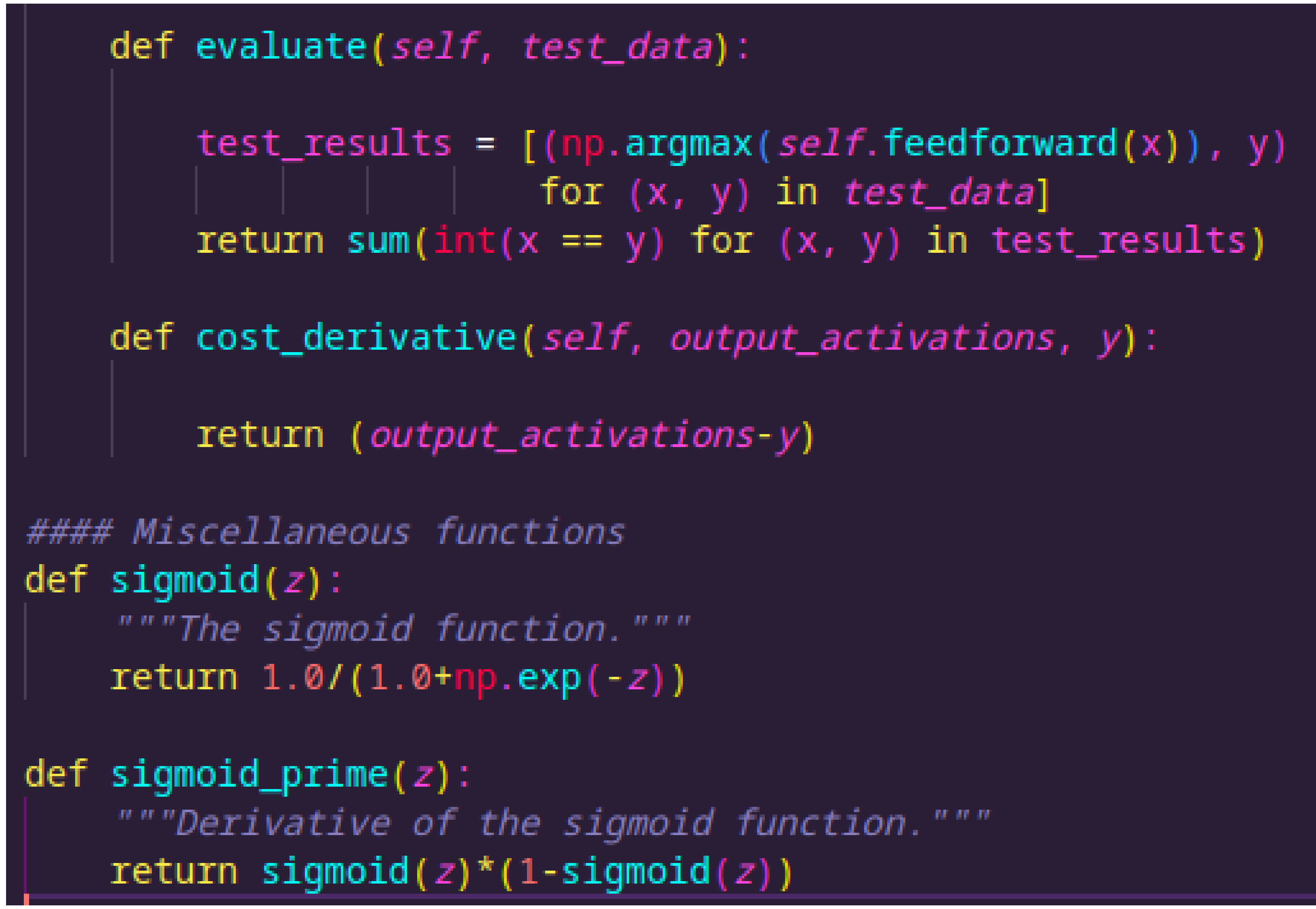
$$\frac{\partial C}{\partial w_{11}} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1}\frac{\partial i_i}{\partial w_{11}} = \left(z_3 - y\right) w_1 z_1 \left(1 - z_1\right) x_1$$

$$\frac{\partial C}{\partial w_{21}} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1}\frac{\partial i_i}{\partial w_{21}} = \left(z_3 - y\right) w_1 z_1 \left(1 - z_1\right) x_2$$

$$\frac{\partial C}{\partial b_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1}\frac{\partial i_i}{\partial w_{21}} = \left(z_3 - y\right) w_1 z_1 \left(1 - z_1\right)$$



```python
def backprop(self, x, y):

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by l
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```



```python
    def evaluate(self, test_data):

        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):

        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```