Neural Networks

1.Basic concepts on an example

In this example we want to take the RGB values from a pixel as input ajd output the decision if we should apply a light (0) or dark (1) font for that pixel.

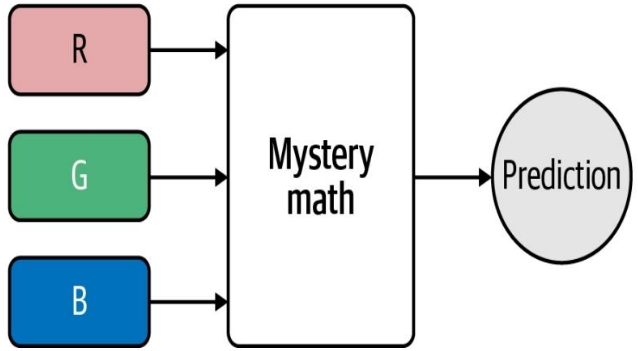


Figure 7-2. We have three numeric RGB values used to make a prediction for a light or dark font

This prediction output expresses a probability. Outputting probabilities is the most common model for classification with neural networks. Once we replace RGB with their numerical values, we see that less than 0.5 will suggest a dark font whereas greater than 0.5 will suggest a light font as demonstrated in Figure 7-3.

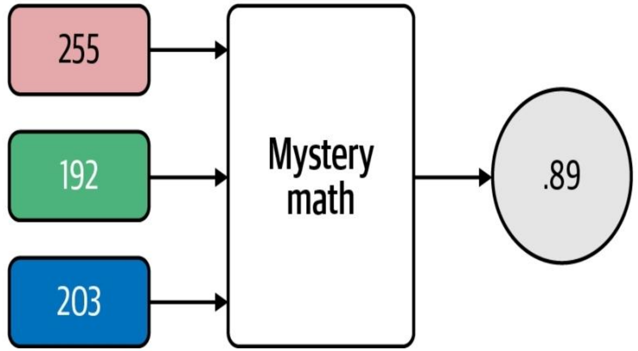
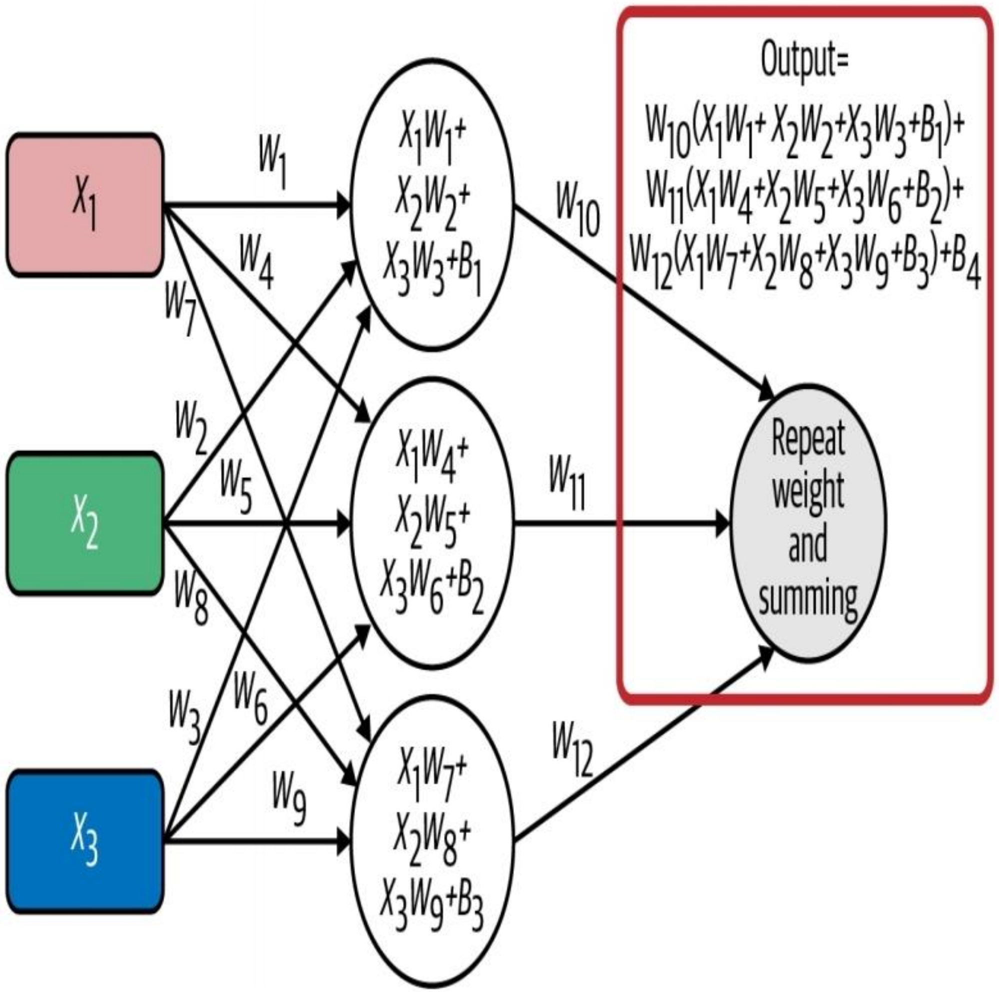


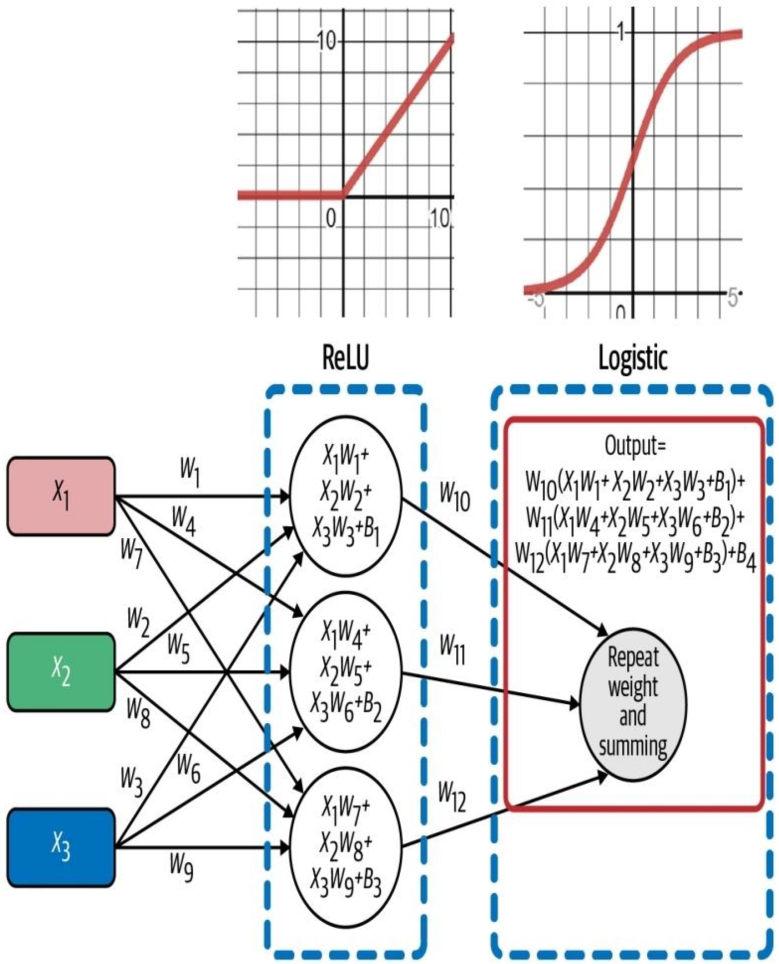
Figure 7-3. If we input a background color of pink (255,192,203), then the mystery math recommends a light font because the output probability 0.89 is greater than 0.5

So what is going on inside that mystery math black box? Let's take a look in Figure 7-4

In this simple example we have 3 layers. One input layer, one hidden layer and one output layer. The input layer is just the input RGB values for our pixel. The hidden layer adds weights and biases to the data (kind of a linear function where m is the weight and b is the bias). The output layers sums, weights and biases the output of the hidden layer and outputs an interpretable probability. In our case the output layer is just a logistic regression.


X_1
X_2
X_3
W_1
W_2
W_3
W_4
W_5
W_6
W_7
W_8
W_9
W_{10}
W_{11}
W_{12}
$X_1W_1+ X_2W_2+ X_3W_3+B_1$
$X_1W_4+ X_2W_5+ X_3W_6+B_2$
$X_1W_7+ X_2W_8+ X_3W_9+B_3$
Output=
$W_{10}(X_1W_1+X_2W_2+X_3W_3+B_1)+$
$W_{11}(X_1W_4+X_2W_5+X_3W_6+B_2)+$
$W_{12}(X_1W_7+X_2W_8+X_3W_9+B_3)+B_4$
Repeat weight and summing

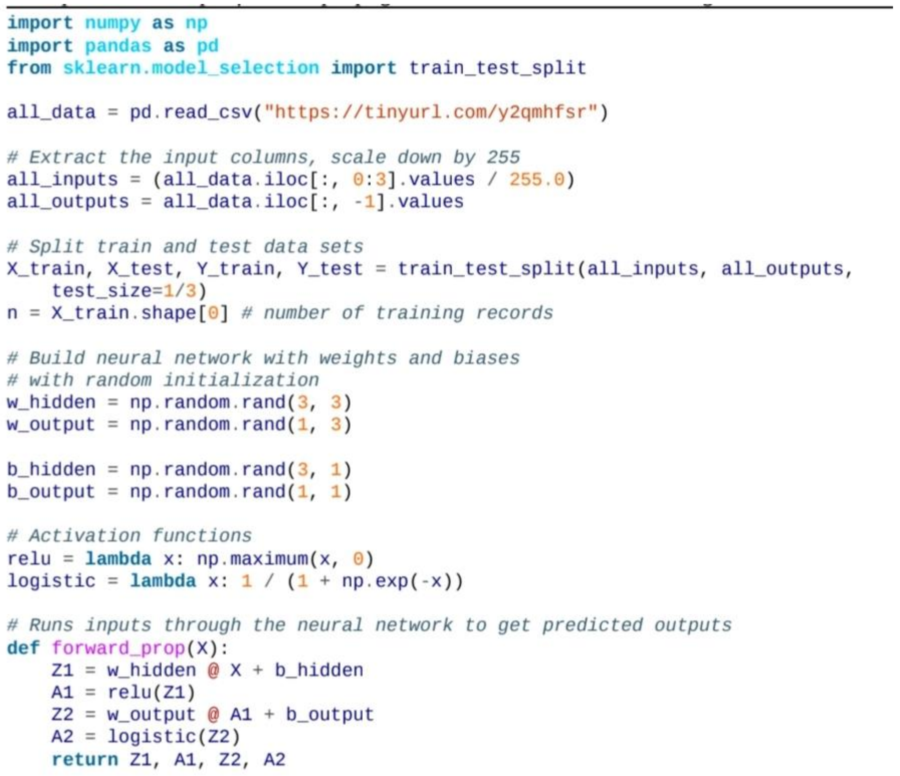
An activation function is a nonlinear finction that transforms the output of one layer so that the next layer can make a classification more easely. In this example the ReLu function is transforms the output of the hidden later and feeds it to the output layer. The ReLu function makes all negative values 0 and leaves all positive values unchanged. The sigmoid function after the output later, transforms its output into a binary 0,1 output. There are many types of activation functions.



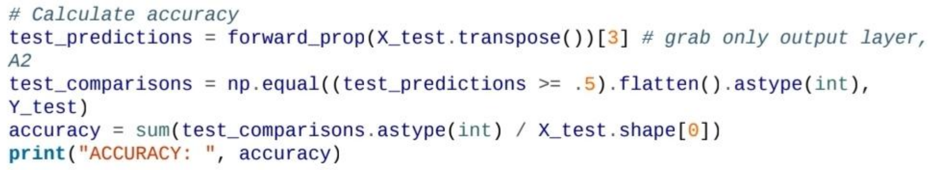
10
0
10
1
-5
0
5
ReLU
Logistic
X_1
X_2
X_3
W_1
W_4
W_7
W_2
W_5
W_8
W_3
W_6
W_9
$X_1W_1+X_2W_2+X_3W_3+B_1$
$X_1W_4+X_2W_5+X_3W_6+B_2$
$X_1W_7+X_2W_8+X_3W_9+B_3$
W_{10}
W_{11}
W_{12}
Output=
$W_{10}(X_1W_1+X_2W_2+X_3W_3+B_1)+$
$W_{11}(X_1W_4+X_2W_5+X_3W_6+B_2)+$
$W_{12}(X_1W_7+X_2W_8+X_3W_9+B_3)+B_4$
Repeat weight and summing

1.2. A simplified neural network in Python (forward propagation)

This is the entire code to make a simplified neural network. Here the weights and biases are assigned randomly. On the next pages we will look at the parts of this code.

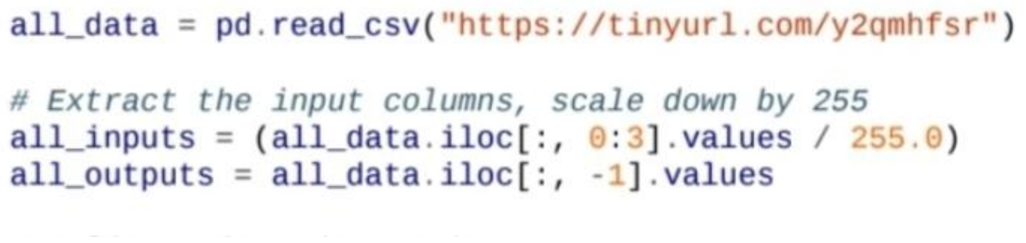


```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

all_data = pd.read_csv("https://tinyurl.com/y2qmhfsr")

# Extract the input columns, scale down by 255
all_inputs = (all_data.iloc[:, 0:3].values / 255.0)
all_outputs = all_data.iloc[:, -1].values

# Split train and test data sets
X_train, X_test, Y_train, Y_test = train_test_split(all_inputs, all_outputs,
    test_size=1/3)
n = X_train.shape[0] # number of training records

# Build neural network with weights and biases
# with random initialization
w_hidden = np.random.rand(3, 3)
w_output = np.random.rand(1, 3)

b_hidden = np.random.rand(3, 1)
b_output = np.random.rand(1, 1)

# Activation functions
relu = lambda x: np.maximum(x, 0)
logistic = lambda x: 1 / (1 + np.exp(-x))

# Runs inputs through the neural network to get predicted outputs
def forward_prop(X):
    Z1 = w_hidden @ X + b_hidden
    A1 = relu(Z1)
    Z2 = w_output @ A1 + b_output
    A2 = logistic(Z2)
    return Z1, A1, Z2, A2
```



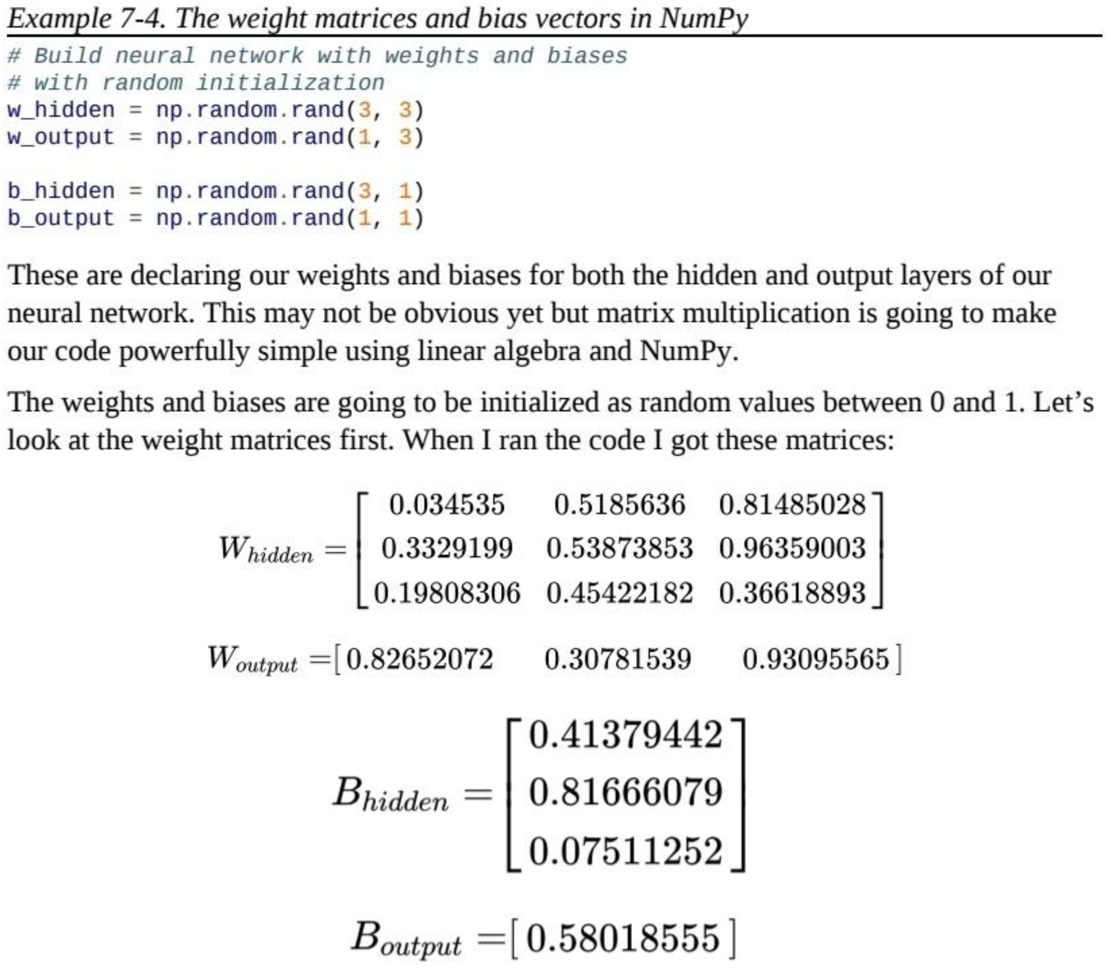
```python
# Calculate accuracy
test_predictions = forward_prop(X_test.transpose())[3] # grab only output layer,
A2
test_comparisons = np.equal((test_predictions >= .5).flatten().astype(int),
Y_test)
accuracy = sum(test_comparisons.astype(int) / X_test.shape[0])
print("ACCURACY: ", accuracy)
```

First the data is read. The inputs are divided by 255, this is to compact the 0-255 RGB value to 0-1. The outputs are not meddled with.



```
all_data = pd.read_csv("https://tinyurl.com/y2qmhfsr")

# Extract the input columns, scale down by 255
all_inputs = (all_data.iloc[:, 0:3].values / 255.0)
all_outputs = all_data.iloc[:, -1].values
```

Here we assign random weights and biases to our data. We have 3 input variables x1,x2,x3. Each of which has 3 weights. So the weight matrices are 3x3 matrices. Note that there are two weight matrices, one for the hidden layer and one for the output layer. Each of the three input variables has only.one bias, so the bias is a vector of lenght 3 for the hidden layer and a single value for the output layer (that single value is the probability from the logistic function)



Example 7-4. The weight matrices and bias vectors in NumPy

```
# Build neural network with weights and biases
# with random initialization
w_hidden = np.random.rand(3, 3)
w_output = np.random.rand(1, 3)

b_hidden = np.random.rand(3, 1)
b_output = np.random.rand(1, 1)
```

These are declaring our weights and biases for both the hidden and output layers of our neural network. This may not be obvious yet but matrix multiplication is going to make our code powerfully simple using linear algebra and NumPy.

The weights and biases are going to be initialized as random values between 0 and 1. Let's look at the weight matrices first. When I ran the code I got these matrices:

$$W_{hidden} = \begin{bmatrix} 0.034535 & 0.5185636 & 0.81485028 \\ 0.3329199 & 0.53873853 & 0.96359003 \\ 0.19808306 & 0.45422182 & 0.36618893 \end{bmatrix}$$

$$W_{output} = \begin{bmatrix} 0.82652072 & 0.30781539 & 0.93095565 \end{bmatrix}$$

$$B_{hidden} = \begin{bmatrix} 0.41379442 \\ 0.81666079 \\ 0.07511252 \end{bmatrix}$$

$$B_{output} = \begin{bmatrix} 0.58018555 \end{bmatrix}$$

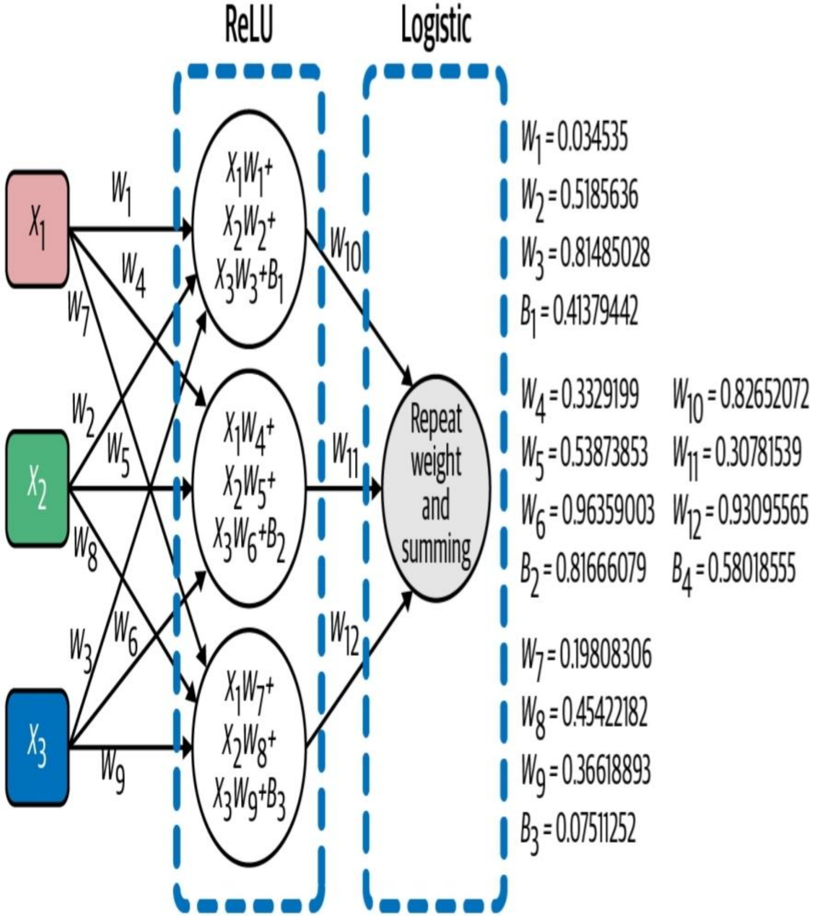
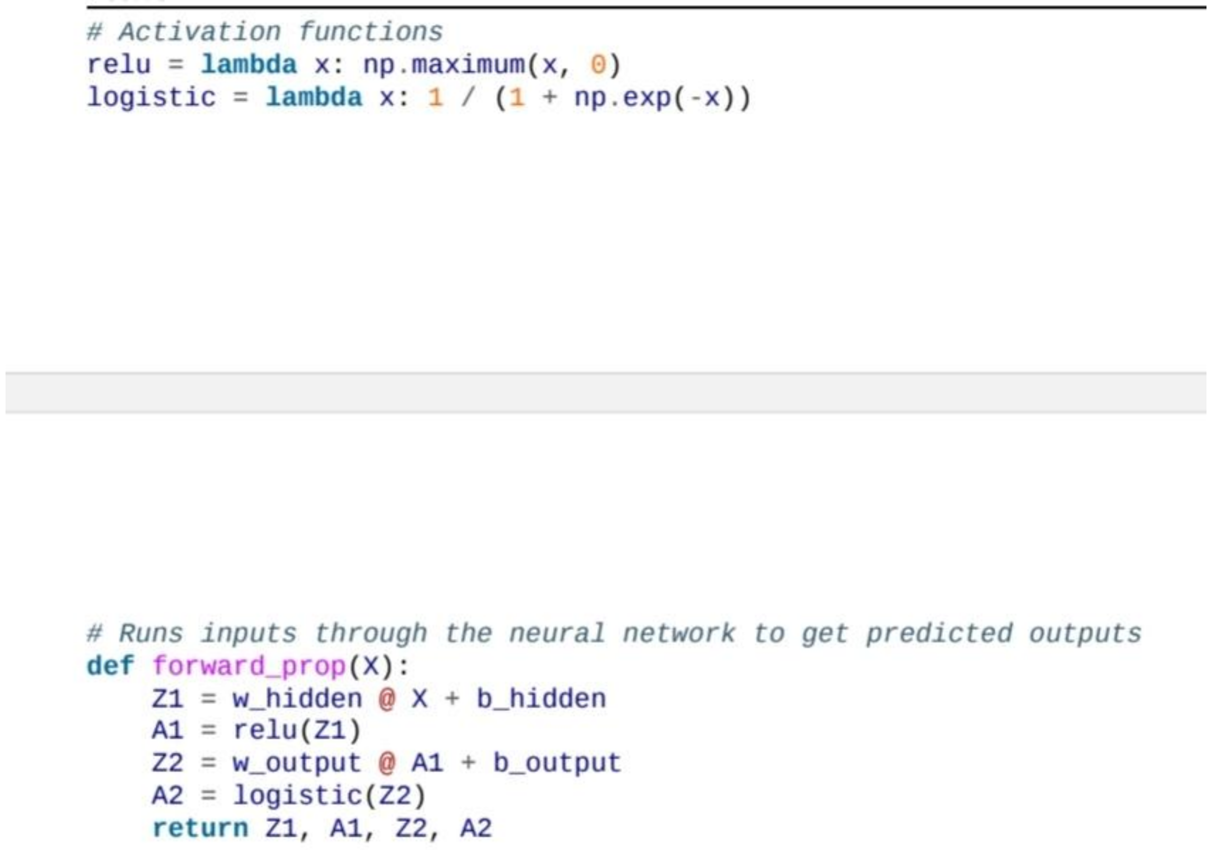


Figure 7-9. Visualizing our neural network against the weight and bias matrix values

Here we see the actual network. First Z1 takes the input and adds the weights and biases. The A1 applies the ReLu activation function to Z1. Z2 adds the weights and biases to A1 and A2 applies the logistic activation function to Z2 to produce the final output (a single probability).



```python
# Activation functions
relu = lambda x: np.maximum(x, 0)
logistic = lambda x: 1 / (1 + np.exp(-x))
```

```python
# Runs inputs through the neural network to get predicted outputs
def forward_prop(X):
    Z1 = w_hidden @ X + b_hidden
    A1 = relu(Z1)
    Z2 = w_output @ A1 + b_output
    A2 = logistic(Z2)
    return Z1, A1, Z2, A2
```

$$Z_1 = W_{hidden}X + B_{hidden}$$

$$Z_1 = \begin{bmatrix} 0.034535 & 0.5185636 & 0.81485028 \\ 0.3329199 & 0.53873853 & 0.96359003 \\ 0.19808306 & 0.45422182 & 0.36618893 \end{bmatrix} \begin{bmatrix} 0.82652072 \\ 0.30781539 \\ 0.93095565 \end{bmatrix} + \begin{bmatrix} 0.41379442 \\ 0.81666079 \\ 0.07511252 \end{bmatrix}$$

$$Z_1 = \begin{bmatrix} \boxed{0.946755221909086} \\ 1.33805678888247 \\ 0.644441873391768 \end{bmatrix} + \begin{bmatrix} \boxed{0.41379442} \\ 0.81666079 \\ 0.07511252 \end{bmatrix}$$

$$Z_1 = \begin{bmatrix} 1.36054964190909 \\ 2.15471757888247 \\ 0.719554393391768 \end{bmatrix}$$

Figure 7-10. Applying the hidden layer weights and biases to an input X using matrix-vector multiplication as well as vector addition

$$A_1 = ReLU\,(Z_1)$$

$$A_1 = \begin{bmatrix} ReLU(1.36054964190909) \\ ReLU(2.15471757888247) \\ ReLU(0.719554393391768) \end{bmatrix} = \begin{bmatrix} 1.36054964190909 \\ 2.15471757888247 \\ 0.719554393391768 \end{bmatrix}$$

Now let's take that hidden layer output A_1 and pass it through the final layer to get Z_2 and then A_2. A_1 becomes the input into the output layer.

$$Z_2 = W_{output}A_1 + B_{output}$$

$$Z_2 = \begin{bmatrix} 0.82652072 & 0.3078159 & 0.93095565 \end{bmatrix} \begin{bmatrix} 1.36054964190909 \\ 2.15471757888247 \\ 0.719554393391768 \end{bmatrix} + \begin{bmatrix} 0.58018555 \end{bmatrix}$$

$$Z_2 = \begin{bmatrix} 2.45765202842636 \end{bmatrix} + \begin{bmatrix} 0.58018555 \end{bmatrix}$$

$$Z_2 = \begin{bmatrix} 3.03783757842636 \end{bmatrix}$$

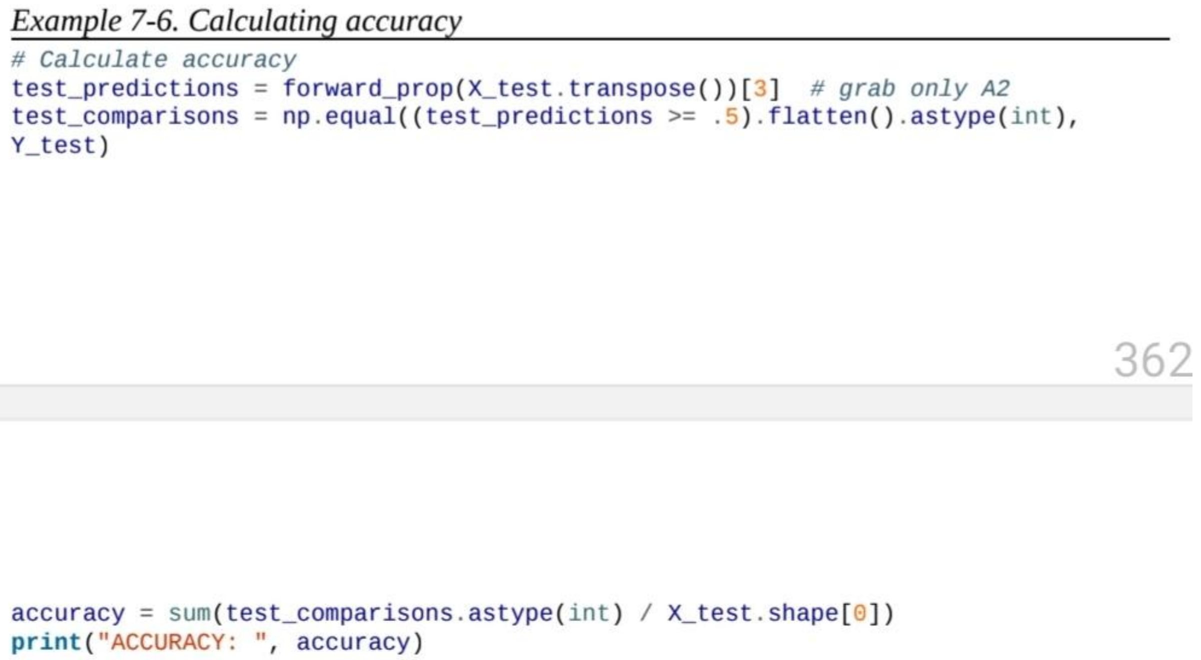
Finally, pass this single value in Z_2 through the activation function to get A_2. This will produce a prediction of approximately 0.95425:

$$A_2 = logistic\,(Z_2)$$

$$A_2 = logistic\,(\begin{bmatrix} 3.0378364795204 \end{bmatrix})$$

$$A_2 = 0.954254478103241$$

Testing the accuracy of our model.



Example 7-6. Calculating accuracy

```
# Calculate accuracy
test_predictions = forward_prop(X_test.transpose())[3]  # grab only A2
test_comparisons = np.equal((test_predictions >= .5).flatten().astype(int),
Y_test)
```

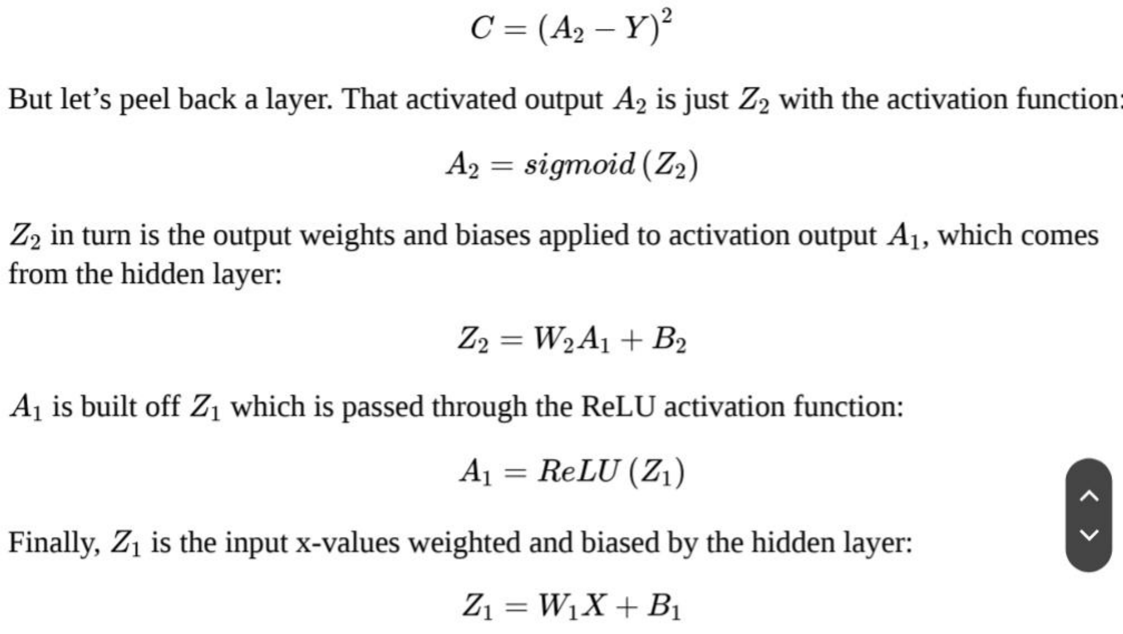
362

```
accuracy = sum(test_comparisons.astype(int) / X_test.shape[0])
print("ACCURACY: ", accuracy)
```

2. Backpropagation

The cost function

The cost function C is the square of the difference of the activated output later and the actual outputs. Our goal is to find such weights and biases that minimize our cost function C.



$$C = (A_2 - Y)^2$$

But let's peel back a layer. That activated output A_2 is just Z_2 with the activation function:

$$A_2 = sigmoid\,(Z_2)$$

Z_2 in turn is the output weights and biases applied to activation output A_1, which comes from the hidden layer:

$$Z_2 = W_2 A_1 + B_2$$

A_1 is built off Z_1 which is passed through the ReLU activation function:

$$A_1 = ReLU\,(Z_1)$$

Finally, Z_1 is the input x-values weighted and biased by the hidden layer:

$$Z_1 = W_1 X + B_1$$

Finding derivatives of the weights and biases

$$\frac{dC}{dw_2} = \frac{dZ_2}{dw_2}\frac{dA_2}{dZ_2}\frac{dC}{dA_2} = (A_1)\left(\frac{e^{-Z_2}}{(1+e^{-Z_2})^2}\right)(2A_2-2y)$$

To minimize the cost function we have to use stochastic gradient descent. To use stochastic gradient descent, we need to find the derivatives of the weights and biases. We find these derivatives using the chain function, which was described earlier. On the right you see the calculated derivatives for the weights and biases.

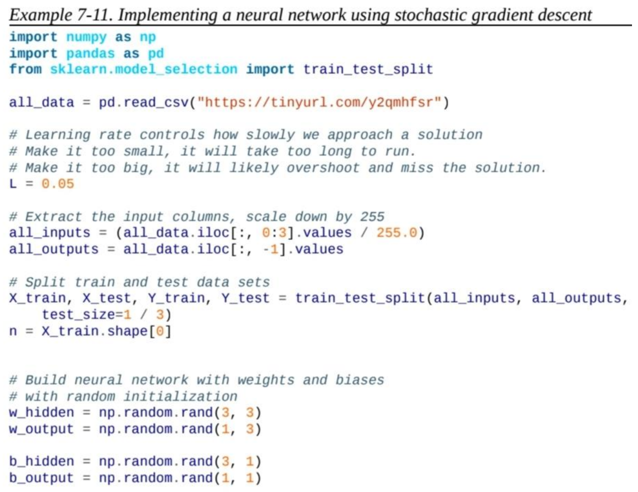
$$\frac{dC}{dW_2} = \frac{dZ_2}{dW_2}\frac{dA_2}{dZ_2}\frac{dC}{dA_2} = (A_1)\left(\frac{e^{-Z_2}}{(1+e^{-Z_2})^2}\right)(2A_2-2y)$$

$$\frac{dC}{dB_2} = \frac{dZ_2}{dB_2}\frac{dA_2}{dZ_2}\frac{dC}{dA_2} = (1)\left(\frac{e^{-Z_2}}{(1+e^{-Z_2})^2}\right)(2A_2-2y)$$

$$\frac{dC}{dW_1} = \frac{dC}{DA_2}\frac{DA_2}{dZ_2}\frac{dZ_2}{dA_1}\frac{dA_1}{dZ_1}\frac{dZ_1}{dW_1} = (2A_2-2y)\left(\frac{e^{-Z_2}}{(1+e^{-Z_2})^2}\right)(W_2)(Z_1>0)(X)$$

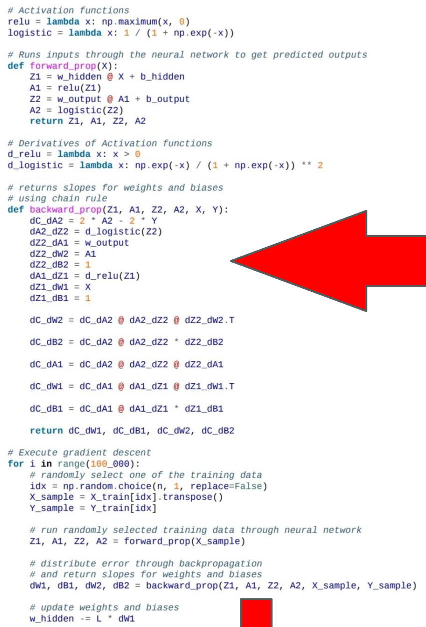
$$\frac{dC}{dB_1} = \frac{dC}{DA_2}\frac{DA_2}{dZ_2}\frac{dZ_2}{dA_1}\frac{dA_1}{dZ_1}\frac{dZ_1}{dB_1} = (2A_2-2y)\left(\frac{e^{-Z_2}}{(1+e^{-Z_2})^2}\right)(W_2)(Z_1>0)(1)$$

On the right you see the full code to create a simple neural network and find such weights and biases that our cost function is minimized. Note that the red arrows show the order in which the code blocks were written.

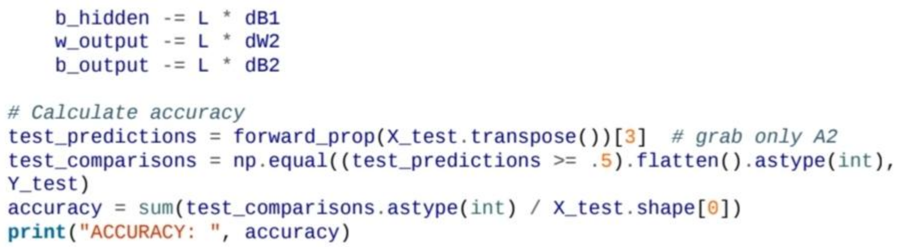


Example 7-11. Implementing a neural network using stochastic gradient descent

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

all_data = pd.read_csv("https://tinyurl.com/y2qmhfsr")

# Learning rate controls how slowly we approach a solution
# Make it too small, it will take too long to run.
# Make it too big, it will likely overshoot and miss the solution.
L = 0.05

# Extract the input columns, scale down by 255
all_inputs = (all_data.iloc[:, 0:3].values / 255.0)
all_outputs = all_data.iloc[:, -1].values

# Split train and test data sets
X_train, X_test, Y_train, Y_test = train_test_split(all_inputs, all_outputs,
    test_size=1 / 3)
n = X_train.shape[0]


# Build neural network with weights and biases
# with random initialization
w_hidden = np.random.rand(3, 3)
w_output = np.random.rand(1, 3)

b_hidden = np.random.rand(3, 1)
b_output = np.random.rand(1, 1)
```



```python
# Activation functions
relu = lambda x: np.maximum(x, 0)
logistic = lambda x: 1 / (1 + np.exp(-x))

# Runs inputs through the neural network to get predicted outputs
def forward_prop(X):
    Z1 = w_hidden @ X + b_hidden
    A1 = relu(Z1)
    Z2 = w_output @ A1 + b_output
    A2 = logistic(Z2)
    return Z1, A1, Z2, A2

# Derivatives of Activation functions
d_relu = lambda x: x > 0
d_logistic = lambda x: np.exp(-x) / (1 + np.exp(-x)) ** 2

# returns slopes for weights and biases
# using chain rule
def backward_prop(Z1, A1, Z2, A2, X, Y):
    dC_dA2 = 2 * A2 - 2 * Y
    dA2_dZ2 = d_logistic(Z2)
    dZ2_dA1 = w_output
    dZ2_dW2 = A1
    dZ2_dB2 = 1
    dA1_dZ1 = d_relu(Z1)
    dZ1_dW1 = X
    dZ1_dB1 = 1

    dC_dW2 = dC_dA2 @ dA2_dZ2 @ dZ2_dW2.T

    dC_dB2 = dC_dA2 @ dA2_dZ2 * dZ2_dB2

    dC_dA1 = dC_dA2 @ dA2_dZ2 @ dZ2_dA1

    dC_dW1 = dC_dA1 @ dA1_dZ1 @ dZ1_dW1.T

    dC_dB1 = dC_dA1 @ dA1_dZ1 * dZ1_dB1

    return dC_dW1, dC_dB1, dC_dW2, dC_dB2

# Execute gradient descent
for i in range(100_000):
    # randomly select one of the training data
    idx = np.random.choice(n, 1, replace=False)
    X_sample = X_train[idx].transpose()
    Y_sample = Y_train[idx]

    # run randomly selected training data through neural network
    Z1, A1, Z2, A2 = forward_prop(X_sample)

    # distribute error through backpropagation
    # and return slopes for weights and biases
    dW1, dB1, dW2, dB2 = backward_prop(Z1, A1, Z2, A2, X_sample, Y_sample)

    # update weights and biases
    w_hidden -= L * dW1
```



```python
    b_hidden -= L * dB1
    w_output -= L * dW2
    b_output -= L * dB2

# Calculate accuracy
test_predictions = forward_prop(X_test.transpose())[3]  # grab only A2
test_comparisons = np.equal((test_predictions >= .5).flatten().astype(int),
Y_test)
accuracy = sum(test_comparisons.astype(int) / X_test.shape[0])
print("ACCURACY: ", accuracy)
```