Neural Network

February 22, 2023

1 What is a Neural Network?

A neural network is a type of artificial intelligence that is modeled after the structure and function of the human brain. It is a collection of interconnected processing nodes, called neurons, that work together to process and analyze complex data. Neural networks are designed to recognize patterns, learn from data, and make predictions or decisions based on that learning.

In a neural network, each neuron receives input from other neurons and uses a mathematical function to combine and process that input. The output of one neuron then becomes the input for other neurons in the network, allowing information to flow through the network and be processed in a hierarchical manner. As data is processed through the network, the connections between neurons are strengthened or weakened based on the patterns in the data, allowing the network to learn and adapt over time.

Neural networks have a wide range of applications, from image and speech recognition to natural language processing and predictive modeling. They have proven to be very effective at solving complex problems that are difficult for traditional computer algorithms to handle.

A single neuron is a tiny, but powerful, building block of artificial intelligence that works just like a cell in your brain! Just like a cell in your brain, a single neuron takes in information from the world around it, processes that information, and then decides what to do next.

Let's think of it like this: imagine you're at a party, and you're trying to decide whether or not to dance. Your brain is like a bunch of little cells (neurons) that are working together to make that decision. First, your eyes and ears (inputs) take in information about the party - the music, the people, the vibe. Then, your brain cells (neurons) process that information, and based on what they're seeing and hearing, they make a decision about whether or not to dance (output).

In the same way, a single neuron in an artificial intelligence system takes in information (input) from the world around it - maybe it's looking at a picture or listening to a sound. It then processes that information using math (weighted sum and activation function), and based on the result, makes a decision (output).

In both cases, the input is sent through the neuron, processed in a way that involves math, and an output is produced. And just like the cells in your brain can work together to make decisions, multiple neurons in an artificial intelligence system can work together to solve complex problems.

So, to sum up, a single neuron is like a tiny robot that takes in information, processes

it, and then makes a decision. Just like a cell in your brain, a single neuron is a crucial building block for complex decision-making, and multiple neurons can work together to create advanced artificial intelligence systems.

1.0.1 Neural networks are like the brains of robots and computers! Just like how our brains help us see, hear, and think, neural networks help computers "see" and "hear" things too! They are made up of tiny "neurons" that work together to solve problems.

1.0.2 Imagine you have a big button that turns on a light. You want to teach the computer to turn on the light when it sees a red apple. First, you take a picture of a red apple and tell the computer "This is a red apple." Then, you take a picture of a banana and tell the computer "This is not a red apple." The computer will learn to recognize a red apple based on the differences between the apple and the banana.

1.0.3 Now, let's talk about a single neuron neural network. A single neuron is like a tiny robot that takes in information and makes a decision. It has inputs (like your eyes and ears) and outputs (like your hands and mouth). The neuron uses math to combine the inputs and produce an output.

1.0.4 For example, let's say we want the neuron to decide if it's sunny outside based on two inputs: temperature and cloudiness. The temperature input could be a number between 0 and 100, and the cloudiness input could be a number between 0 and 1. The neuron would combine these two numbers using math and output a number between 0 and 1, where 0 means "definitely not sunny" and 1 means "definitely sunny."

1.0.5 Now, let's look at the mathematical derivations. A single neuron combines its inputs using a formula called a weighted sum. The formula looks like this:

weighted sum = input 1 * weight 1 + input 2 * weight 2 + ... + input n * weight n

1.0.6 In our example, the weighted sum would be:

weighted sum = temperature * weight 1 + cloudiness * weight 2

1.0.7 The weights are like the importance of each input. If one input is more important, it will have a higher weight. The neuron then adds a bias to the weighted sum, which helps it make decisions. The formula looks like this:

output = activation function(weighted sum + bias)

1.0.8 The activation function is like a switch that decides if the neuron should "fire" (output 1) or "not fire" (output 0). In our example, we might use a sigmoid function as the activation function, which looks like this:

sigmoid(x) = 1 / (1 + e^-x)

1.0.9 The output of the neuron tells us the probability that it's sunny outside. If the output is close to 0, it's probably not sunny. If the output is close to 1, it's probably sunny.

Here's an example of a single neuron neural network, also known as a perceptron, in Python:

[ ]:
```
import numpy as np

class Perceptron:
    def __init__(self, n_inputs):
        self.weights = np.random.rand(n_inputs)
        self.bias = np.random.rand()

    def forward(self, inputs):
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        output = self.activate(weighted_sum)
        return output

    def activate(self, weighted_sum):
        if weighted_sum > 0:
            return 1
        else:
            return 0
```

The `Perceptron` class represents a single neuron in a neural network. It takes in an n_inputs parameter, which specifies the number of inputs that the neuron should expect.

When a set of inputs is passed to the neuron using the `forward` method, the neuron first calculates the weighted sum of the inputs by taking the dot product of the inputs and the neuron's weights, and then adding the neuron's bias term. This calculation is represented by the following equation:

```
weighted_sum = (input_1 * weight_1) + (input_2 * weight_2) + ... + bias
```

The neuron then applies an activation function to the weighted sum to produce the neuron's output. In this example, we are using a step function as the activation function, which returns 1 if the weighted sum is greater than 0, and 0 otherwise. The activation function is represented by the following equation:

```
output = 1 if weighted_sum > 0 else 0
```

This is a simplified example of a neural network, as it only contains a single neuron. However, more complex neural networks can be constructed by connecting multiple neurons together in layers, and applying different activation functions to the output of each neuron.

Here's an example code for creating a single neuron neural network using numpy to predict if it's sunny outside based on temperature and cloudiness inputs:

[ ]:
```
import numpy as np

# Define the inputs and labels
```

```
inputs = np.array([[65, 0.2], [70, 0.5], [80, 0.8], [72, 0.3], [68, 0.6], [75,␣
 ↪0.4]])
labels = np.array([[0], [1], [1], [0], [1], [1]])

# Define the weights and bias with random initial values
weights = np.random.rand(2, 1)
bias = np.random.rand()

# Define the sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Define the forward function
def forward(inputs, weights, bias):
    weighted_sum = np.dot(inputs, weights) + bias
    output = sigmoid(weighted_sum)
    return output

# Define the loss function
def loss(predictions, labels):
    loss = np.mean(-(labels * np.log(predictions) + (1 - labels) * np.log(1 -␣
 ↪predictions)))
    return loss

# Define the train function
def train(inputs, labels, weights, bias, learning_rate, epochs):
    for i in range(epochs):
        # Forward pass
        predictions = forward(inputs, weights, bias)

        # Calculate the loss
        current_loss = loss(predictions, labels)

        # Backward pass
        d_loss_d_output = (predictions - labels) / len(inputs)
        d_output_d_weight = inputs.T
        d_output_d_bias = 1

        d_loss_d_weight = np.dot(d_output_d_weight, d_loss_d_output *␣
 ↪predictions * (1 - predictions))
        d_loss_d_bias = np.sum(d_loss_d_output * predictions * (1 -␣
 ↪predictions))

        # Update the weights and bias
        weights -= learning_rate * d_loss_d_weight
        bias -= learning_rate * d_loss_d_bias
```

```
        # Print the loss every 100 epochs
        if i % 100 == 0:
            print(f"Epoch {i}, Loss: {current_loss:.4f}")

    return weights, bias

# Train the model
learning_rate = 0.1
epochs = 1000
weights, bias = train(inputs, labels, weights, bias, learning_rate, epochs)

# Test the model on new data
new_data = np.array([[70, 0.4], [75, 0.6]])
predictions = forward(new_data, weights, bias)
print(predictions)
```

In this code, we define the inputs as a numpy array with two columns (temperature and cloudiness) and the labels as a numpy array with one column (0 for not sunny and 1 for sunny). We also define the weights and bias with random initial values, and the sigmoid activation function. The forward function calculates the weighted sum and applies the sigmoid activation function to produce the output. The loss function calculates the binary cross-entropy loss between the predictions and labels. The train function uses the forward and loss functions to perform the forward and backward passes and update the weights and bias using gradient descent. Finally, we test the model on new data by passing it through the forward function.

Why we add bias, explain to grade 8 students

Imagine you are a student and you have a test. The teacher grades the test by adding up all of the points you earned for each question. But what if the test was too hard, or the teacher made a mistake in grading one of the questions? It wouldn't be fair to give you a bad grade just because the test was hard or the teacher made a mistake, right?

In the same way, when we use a neural network, we want to make sure that the network is fair and accurate. We don't want the network to always output a 0 or a 1 just because the inputs are too small or too large. We want the network to be able to adjust for differences in the inputs and make a decision based on the overall pattern of the data.

This is where the bias comes in. The bias is like an extra point that the neuron can use to adjust its decision. It's like saying, "Hey, I know that these inputs are a little bit tricky, so let me adjust my decision a little bit to make sure I'm being fair." By adding a bias, the neuron is able to make more accurate decisions and be more flexible when it comes to different types of inputs.

So, in short, we add a bias to make sure that the neural network is fair and accurate, even when the inputs are different or the data is a little bit tricky.

[2]:
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Generate some random data for training
```

```
x_train = np.linspace(-1, 1, 101)
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33

# Define the neural network architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(1, input_shape=(1,))
])

# Compile the model with an optimizer and loss function
model.compile(optimizer='sgd', loss='mse')

# Train the model on the data
model.fit(x_train, y_train, epochs=100)

# Use the trained model to make predictions on new data
x_test = np.linspace(-1, 1, 101)
y_test = model.predict(x_test)

# Visualize the data and model predictions
plt.scatter(x_train, y_train)
plt.plot(x_test, y_test, color='r')
plt.show()
```

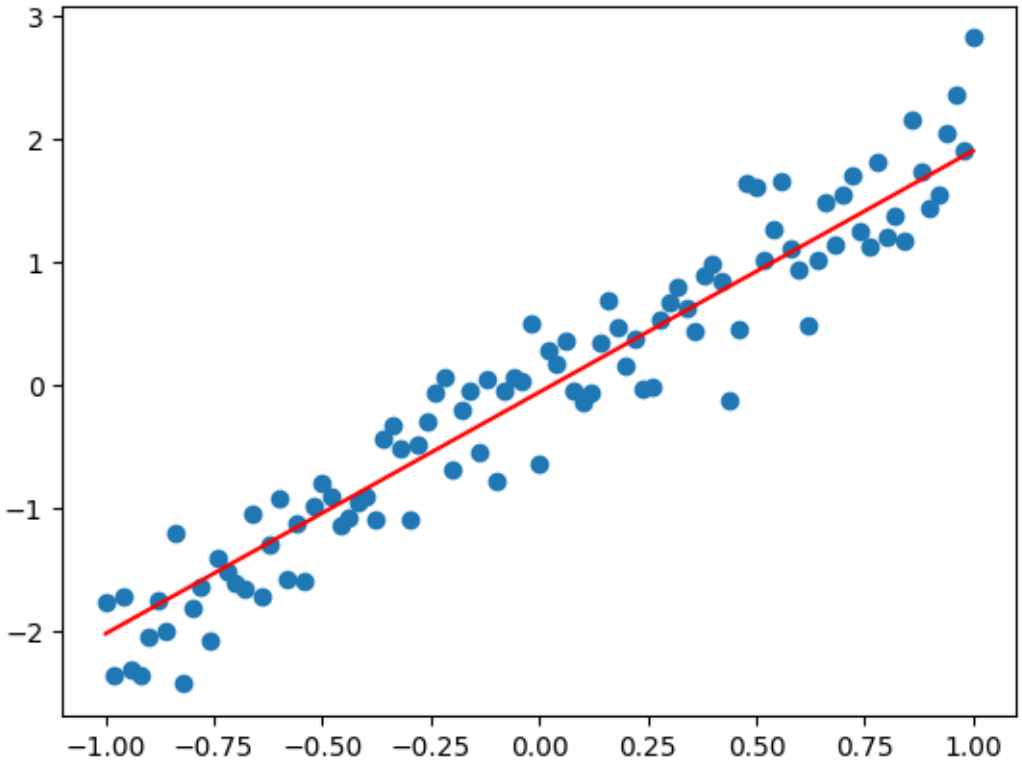
```
Epoch 1/100
4/4 [==============================] - 79s 9ms/step - loss: 0.9352
Epoch 2/100
4/4 [==============================] - 0s 6ms/step - loss: 0.8900
Epoch 3/100
4/4 [==============================] - 0s 3ms/step - loss: 0.8507
Epoch 4/100
4/4 [==============================] - 0s 3ms/step - loss: 0.8113
Epoch 5/100
4/4 [==============================] - 0s 4ms/step - loss: 0.7696
Epoch 6/100
4/4 [==============================] - 0s 3ms/step - loss: 0.7333
Epoch 7/100
4/4 [==============================] - 0s 3ms/step - loss: 0.6994
Epoch 8/100
4/4 [==============================] - 0s 3ms/step - loss: 0.6680
Epoch 9/100
4/4 [==============================] - 4s 1s/step - loss: 0.6331
Epoch 10/100
4/4 [==============================] - 0s 10ms/step - loss: 0.5984
Epoch 11/100
4/4 [==============================] - 0s 2ms/step - loss: 0.5725
Epoch 12/100
4/4 [==============================] - 0s 3ms/step - loss: 0.5510
Epoch 13/100
```

```
4/4 [==============================] - 0s 4ms/step - loss: 0.5270
Epoch 14/100
4/4 [==============================] - 0s 4ms/step - loss: 0.5082
Epoch 15/100
4/4 [==============================] - 0s 3ms/step - loss: 0.4898
Epoch 16/100
4/4 [==============================] - 0s 3ms/step - loss: 0.4723
Epoch 17/100
4/4 [==============================] - 0s 3ms/step - loss: 0.4549
Epoch 18/100
4/4 [==============================] - 0s 2ms/step - loss: 0.4381
Epoch 19/100
4/4 [==============================] - 0s 3ms/step - loss: 0.4227
Epoch 20/100
4/4 [==============================] - 0s 3ms/step - loss: 0.4052
Epoch 21/100
4/4 [==============================] - 0s 4ms/step - loss: 0.3904
Epoch 22/100
4/4 [==============================] - 0s 4ms/step - loss: 0.3771
Epoch 23/100
4/4 [==============================] - 0s 3ms/step - loss: 0.3603
Epoch 24/100
4/4 [==============================] - 0s 3ms/step - loss: 0.3479
Epoch 25/100
4/4 [==============================] - 0s 5ms/step - loss: 0.3376
Epoch 26/100
4/4 [==============================] - 0s 3ms/step - loss: 0.3265
Epoch 27/100
4/4 [==============================] - 0s 5ms/step - loss: 0.3176
Epoch 28/100
4/4 [==============================] - 0s 3ms/step - loss: 0.3081
Epoch 29/100
4/4 [==============================] - 0s 4ms/step - loss: 0.2988
Epoch 30/100
4/4 [==============================] - 0s 4ms/step - loss: 0.2903
Epoch 31/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2795
Epoch 32/100
4/4 [==============================] - 0s 4ms/step - loss: 0.2713
Epoch 33/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2643
Epoch 34/100
4/4 [==============================] - 0s 2ms/step - loss: 0.2572
Epoch 35/100
4/4 [==============================] - 0s 2ms/step - loss: 0.2493
Epoch 36/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2426
Epoch 37/100
```

```
4/4 [==============================] - 0s 3ms/step - loss: 0.2354
Epoch 38/100
4/4 [==============================] - 0s 6ms/step - loss: 0.2300
Epoch 39/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2246
Epoch 40/100
4/4 [==============================] - 0s 6ms/step - loss: 0.2202
Epoch 41/100
4/4 [==============================] - 0s 6ms/step - loss: 0.2144
Epoch 42/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2102
Epoch 43/100
4/4 [==============================] - 0s 2ms/step - loss: 0.2060
Epoch 44/100
4/4 [==============================] - 0s 2ms/step - loss: 0.2020
Epoch 45/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1977
Epoch 46/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1931
Epoch 47/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1888
Epoch 48/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1852
Epoch 49/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1815
Epoch 50/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1776
Epoch 51/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1739
Epoch 52/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1711
Epoch 53/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1671
Epoch 54/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1646
Epoch 55/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1625
Epoch 56/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1600
Epoch 57/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1573
Epoch 58/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1548
Epoch 59/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1525
Epoch 60/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1508
Epoch 61/100
```

```
4/4 [==============================] - 0s 3ms/step - loss: 0.1489
Epoch 62/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1474
Epoch 63/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1461
Epoch 64/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1440
Epoch 65/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1426
Epoch 66/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1420
Epoch 67/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1410
Epoch 68/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1398
Epoch 69/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1388
Epoch 70/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1376
Epoch 71/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1364
Epoch 72/100
4/4 [==============================] - 0s 6ms/step - loss: 0.1356
Epoch 73/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1346
Epoch 74/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1341
Epoch 75/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1330
Epoch 76/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1324
Epoch 77/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1318
Epoch 78/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1313
Epoch 79/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1307
Epoch 80/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1300
Epoch 81/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1295
Epoch 82/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1288
Epoch 83/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1282
Epoch 84/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1277
Epoch 85/100
```

```
4/4 [==============================] - 0s 3ms/step - loss: 0.1270
Epoch 86/100
4/4 [==============================] - 0s 3ms/step - loss: 0.1267
Epoch 87/100
4/4 [==============================] - 0s 5ms/step - loss: 0.1264
Epoch 88/100
4/4 [==============================] - 0s 8ms/step - loss: 0.1262
Epoch 89/100
4/4 [==============================] - 0s 4ms/step - loss: 0.1256
Epoch 90/100
4/4 [==============================] - 0s 5ms/step - loss: 0.1252
Epoch 91/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1247
Epoch 92/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1242
Epoch 93/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1238
Epoch 94/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1235
Epoch 95/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1231
Epoch 96/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1228
Epoch 97/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1228
Epoch 98/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1226
Epoch 99/100
4/4 [==============================] - 0s 2ms/step - loss: 0.1224
Epoch 100/100
4/4 [==============================] - 0s 4ms/step - loss: 0.1224
```


3
2
1
0
−1
−2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

[ ]: