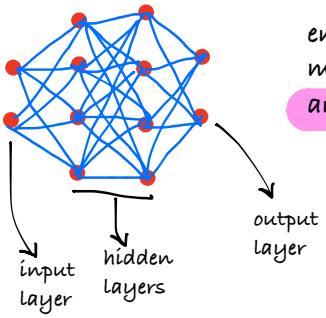
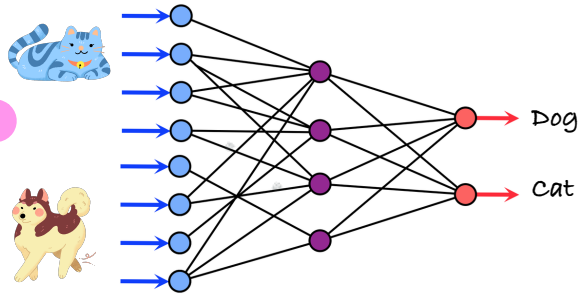
Neural Networks



end goal of a neural networks like most machine learnings is to take some input data and produce the output data that is desired

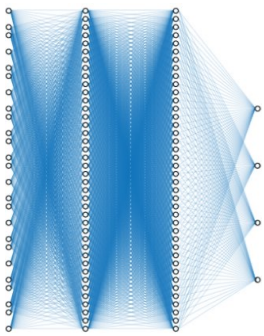


In this case, we've got images of cats and dogs, we pass it through in pixel form to our neural network, and if it's a dog, then the final output neuron on top is going to be the strongest. And if it's a cat then that final output neuron on the bottom is going to be strongest

And we can do this by tuning the weights and biases and that is the actual training process

- How and Why do neural networks work?

Well, if you look at them and really consider what's going on, every neuron is connected to the sub-sequent layer of neurons in full, so each of that connection is a unique weight and every neuron is a unique bias. So, what this ends up giving us is a huge number of uniquely tunable parameters that go into a gigantic function.



The hard part of neural networks and deep learning is figuring out how to tune such a thing

Let's code

```
"""
versions used:
Python: 3.7.7
Numpy: 1.18.2
Matplotlib: 3.2.1
"""

#We're creating a neuron yay!! (let's just say this neuron is getting input from 3
neurons)

inputs = [1.2,5.1,2.1] #outputs from 3 neurons
weights = [3.1,2.1,8.7] #every input has unique weight associated with it
bias = 3 #every unique neuron has a unique bias

#So, now, the 1st step for neuron is to addup all the inputs times the weights plus bias

output = 0
for i in range(len(inputs)):
    output = output + (inputs[i]*weights[i])

output = output +bias
print(output)
```

```
# What if we want to model three neurons with four inputs each XD
#this means, we're gonna have four inputs
#since there's three neurons there's going to be three unique weight sets and each weight
set is going to have four unique weights
#also, we're going to need 3 unique or separate biases

inputs = [1, 2, 3, 2.5]
weights1 = [0.2, 0.8, -0.5, 1.0]
weights2 = [0.5, -0.91, 0.26, -0.5]
weights3 = [-0.26, -0.27, 0.17, 0.87]

bias1 = 2
bias2 = 3
bias3 = 0.5

output = [inputs[0]*weights1[0] + inputs[1]*weights1[1] + inputs[2]*weights1[2] +
inputs[3]*weights1[3] + bias1,
inputs[0]*weights2[0] + inputs [1]*weights2[1] + inputs[2]*weights2[2] +
inputs[3]*weights2[3] + bias2,
inputs[0]*weights3[0] + inputs [1]*weights3[1] + inputs[2] *weights3[2] +
inputs[3]*weights3[3] + bias3]

print (output)



#let's switch to a cleaner and more dynamic approach

inputs = [1, 2, 3, 2.5]
weights = [[0.2, 0.8, -0.5, 1.0],[0.5, -0.91, 0.26, -0.5],[-0.26, -0.27, 0.17, 0.87]]
biases = [2,3,0.5]

layer_outputs = []
for neuron_weights, neuron_bias in zip(weights, biases):
    neuron_output = 0
    for n_weight, n_input in zip(neuron_weights,inputs):
        neuron_output += (n_weight*n_input)
    neuron_output += neuron_bias
    layer_outputs.append(neuron_output)

print(layer_outputs)
```

Weights & Biases

Weights and biases are these knobs that we tune but really down the line, it's gonna be the optimizer that tunes this values in an attempt to fit some data because we've got hundreds of thousands or millions of these tunable parameters. We can fit so many things but the way the weights and biases are impacting a neuron's output, it's a different way. So, these are just two different tools that help in two different ways and that's it. So, anyways, this will make more sense if we get into activation functions and rectified linear.

Shape

So the concept of shape is that basically it is at each dimension, what's the size of this dimension. Consider we got a list of four elements, this is a one-dimensional list and bcoz we have 4 elements the shape is just [4,] . This is also a 1d array and it is a vector so a list in python can be an array in numpy and if it's just a simple list, then it is a one dimensional array in numpy and in mathematics, a list is a vector. Now imagine a scenario where we have two lists contained in a list. In python, we call that a list of lists or lol for short in numpy, it is a 2 dimensional array because that 1st list has how many lists? 2 lists. Now, each list, how many elements does each list have? 4. These arrays have to be homologous. this means, they have to be of the same size for each dimension.

Tensor is an object that can be represented as an array. A tensor is not just an array, but in the context of deep learning and programming a tensor is represented as an array. So, you work with tensors in the array form

```
## Now, let's use dot product to implement the same thing

import numpy as np

layer_outputs = np.dot(weights,inputs) + biases

print(f"The output using numpy is {layer_outputs}")
```

Batches

To train, neural networks tend to receive data in batches. So far, the example input data have been only one sample (or observation) of various features called a feature set:

inputs = [1, 2, 3, 2.5]

Here, the [1, 2, 3, 2.5] data are somehow meaningful and descriptive to the output we desire. Imagine each number as a value from a different sensor all simultaneously. Each of these values is a feature observation datum, and together they form a feature set instance, also called an observation, or most commonly, a sample.

Input data :

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

Shape :

(8, 4)

Type :

2D Array, Matrix

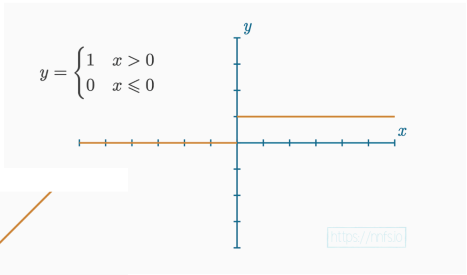
```
import numpy as np

np.random.seed(0)

X = [[1, 2, 3, 2.5],[2,5,-1,2],[-1.5,2.7,3.3,-0.8]]


#How do we initialize a layer in neural networks

# two ways:
# the first is you've got a trained model that you saved and you actually wanna load in that model , so what actually happens when you save the model is you're just saving the weights and biases
# In this case, we're gonna initialize weights and biases
# generally in nn, we want smaller values
# ranges between negative 1 and positive 1
# for we're gonna use random in numpy

class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.10 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))    #additional info: numpy.zeros(shape, dtype=float, order='C'), The 2nd parameter should be data type and not a number
    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) +self.biases

layer1 = Layer_Dense(4,5) # We specify here, the size of the inputs and how many neurons we wanna have # so the size of the inputs here, we know that is 4 (no. of features in each sample)
# And then for the number of neurons, is anything you want (any no. basically) let's say 5
# Similarly let's specify the layer 2
# The only requirement for layer 2 is that the output from layer 1 is gonna be the input to layer 2
# therefore, the no. of inputs's shape have to 5
# but again, the output can be of any shape that you want, let's say 2

layer2 = Layer_Dense(5,2)

layer1.forward(X)

print(layer1.output)

# layer 1 output becomes the input of layer 2 so,


layer2.forward(layer1.output)
print(layer2.output)
```

Activation Functions

The activation function is applied to the output of a neuron (or layer of neurons), which modifies outputs
We use activation functions because if the activation function itself is nonlinear, it allows for neural networks with usually two or more hidden layers to map nonlinear functions.

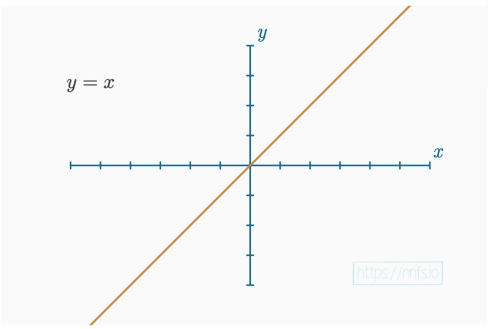
The Step Activation Function

In a single neuron, if the weights · inputs + bias results in a value greater than 0, the neuron will fire and output a 1; otherwise, it will output a 0.



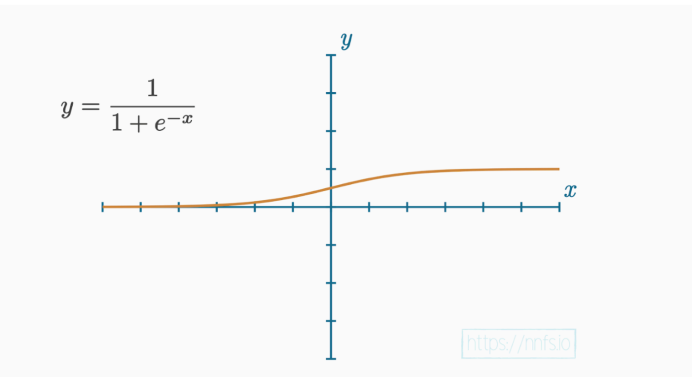
The Linear Activation Function

A linear function is simply the equation of a line.
It will appear as a straight line when graphed,
where y=x and the output value equals the input.



The Sigmoid Activation Function

This function returns a value in the range of 0 for negative infinity, through 0.5 for the input of 0, and to 1 for positive infinity.



The Sigmoid function, historically used in hidden layers, was eventually replaced by the Rectified Linear Units activation function (or ReLU)