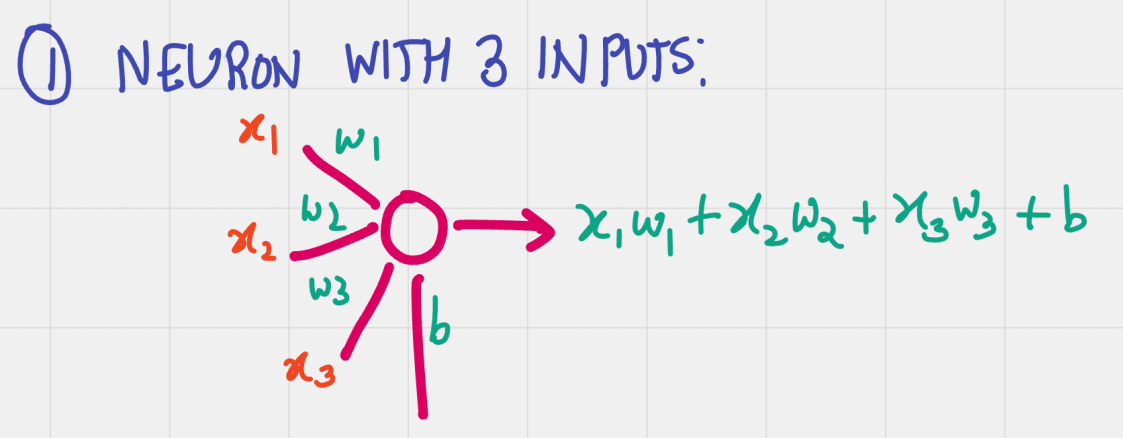
# handout_3

October 20, 2024

## 0.1 BUILDING NEURAL NETWORKS FROM SCRATCH PART 1: CODING NEURONS AND LAYERS

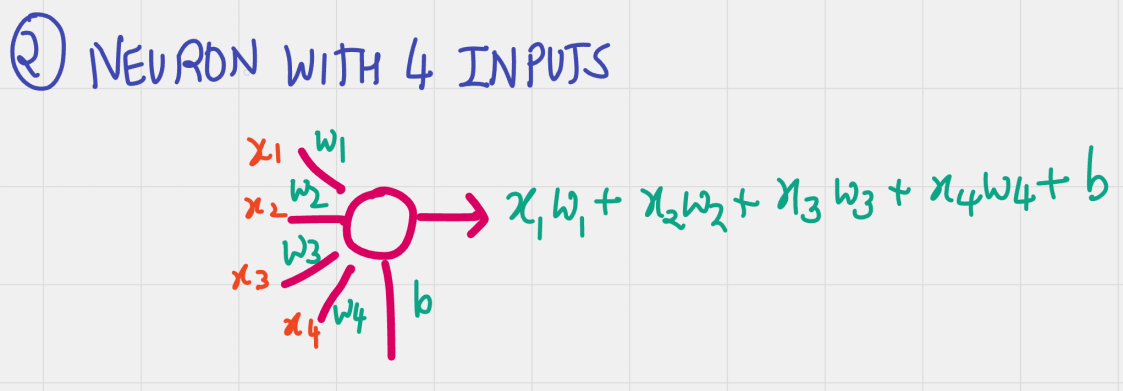
CODING OUR FIRST NEURON: 3 INPUTS




[111]:
```
inputs = [1, 2, 3]
weights = [0.2, 0.8, -0.5]
bias = 2

outputs = (inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] +␣
 ↪bias)

print(outputs)
```

```
2.3
```

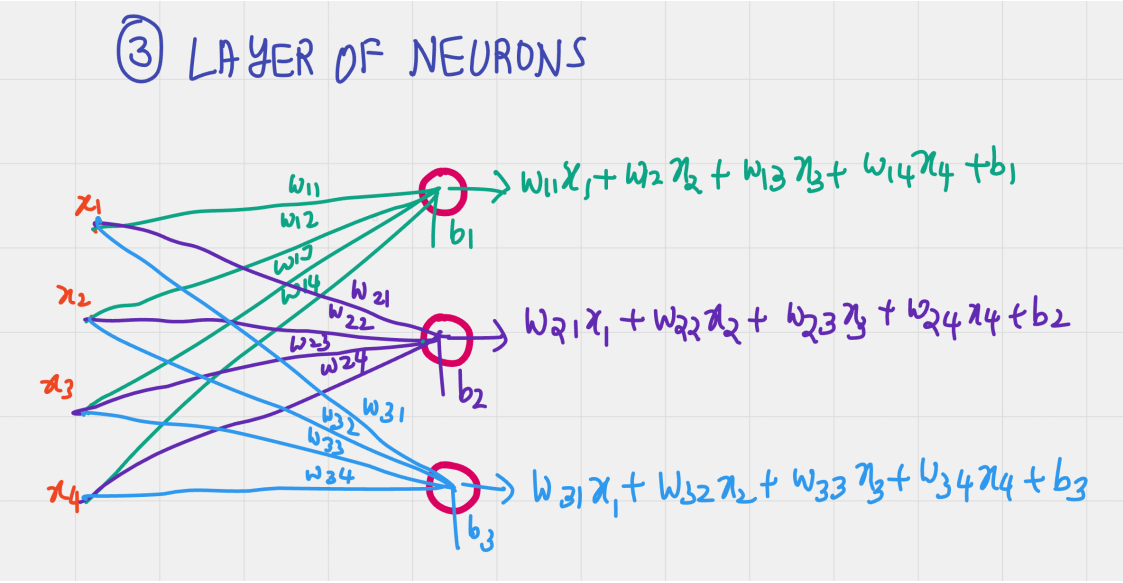
CODING OUR SECOND NEURON: 4 INPUTS




[112]:
```
inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0
output = (inputs[0]*weights[0] +
 inputs[1]*weights[1] +
 inputs[2]*weights[2] +
 inputs[3]*weights[3] + bias)

print(output)
```

4.8

CODING OUR FIRST LAYER




[113]:
```
inputs = [1, 2, 3, 2.5]

weights = [[0.2, 0.8, -0.5, 1],
 [0.5, -0.91, 0.26, -0.5],
```

```
 [-0.26, -0.27, 0.17, 0.87]]

weights1 = weights[0] #LIST OF WEIGHTS ASSOCIATED WITH 1ST NEURON : W11, W12,␣
 ↪W13, W14
weights2 = weights[1] #LIST OF WEIGHTS ASSOCIATED WITH 2ND NEURON : W21, W22,␣
 ↪W23, W24
weights3 = weights[2] #LIST OF WEIGHTS ASSOCIATED WITH 3RD NEURON : W31, W32,␣
 ↪W33, W34

biases = [2, 3, 0.5]

bias1 = 2
bias2 = 3
bias3 = 0.5

outputs = [
 # Neuron 1:
 inputs[0]*weights1[0] +
 inputs[1]*weights1[1] +
 inputs[2]*weights1[2] +
 inputs[3]*weights1[3] + bias1,
 # Neuron 2:
 inputs[0]*weights2[0] +
 inputs[1]*weights2[1] +
 inputs[2]*weights2[2] +
 inputs[3]*weights2[3] + bias2,
 # Neuron 3:
 inputs[0]*weights3[0] +
 inputs[1]*weights3[1] +
 inputs[2]*weights3[2] +
 inputs[3]*weights3[3] + bias3]

print(outputs)
```

```
[4.8, 1.21, 2.385]
```

USING LOOPS FOR BETTER AND EASIER CODING

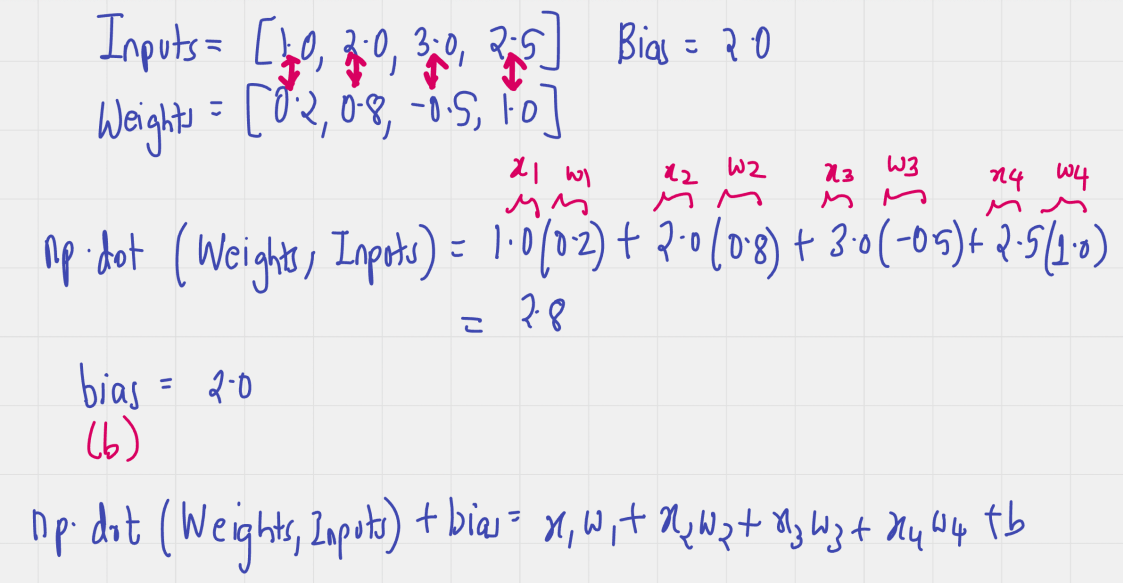
[114]:
```
inputs = [1, 2, 3, 2.5]

##LIST OF WEIGHTS
weights = [[0.2, 0.8, -0.5, 1],
 [0.5, -0.91, 0.26, -0.5],
 [-0.26, -0.27, 0.17, 0.87]]

##LIST OF BIASES
biases = [2, 3, 0.5]
```

```
# Output of current layer
layer_outputs = []

# For each neuron
for neuron_weights, neuron_bias in zip(weights, biases):
 # Zeroed output of given neuron
 neuron_output = 0
 # For each input and weight to the neuron
 for n_input, weight in zip(inputs, neuron_weights):
 # Multiply this input by associated weight
 # and add to the neuron's output variable
   neuron_output += n_input*weight ## W31*X1 + W32*X2 + W33*X3 + W34*X4
   # Add bias
 neuron_output += neuron_bias ## ## W31*X1 + W32*X2 + W33*X3 + W34*X4 + B3
 # Put neuron's result to the layer's output list
 layer_outputs.append(neuron_output)
print(layer_outputs)
```

```
[4.8, 1.21, 2.385]
```

USING NUMPY

SINGLE NEURON USING NUMPY

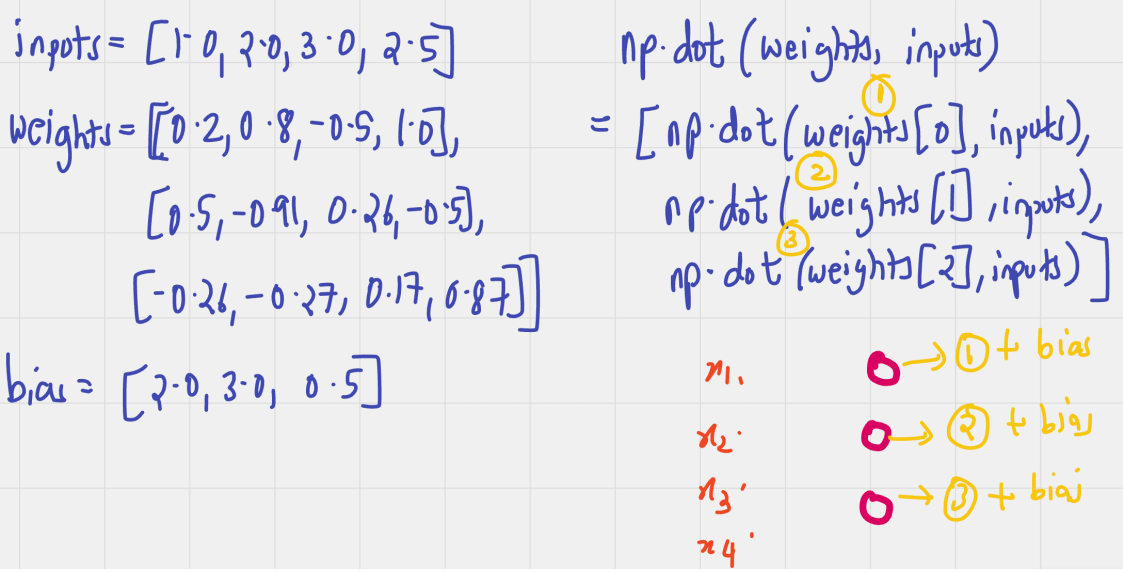



[118]:
```
import numpy as np

inputs = [1.0, 2.0, 3.0, 2.5]
weights = [0.2, 0.8, -0.5, 1.0]
bias = 2.0

# Convert lists to numpy arrays
```

```
inputs_array = np.array(inputs)
weights_array = np.array(weights)

# Calculate the dot product and add the bias
outputs = np.dot(weights_array, inputs_array) + bias

print(outputs)
```

```
4.799999952316284
```

LAYER OF NEURONS USING NUMPY

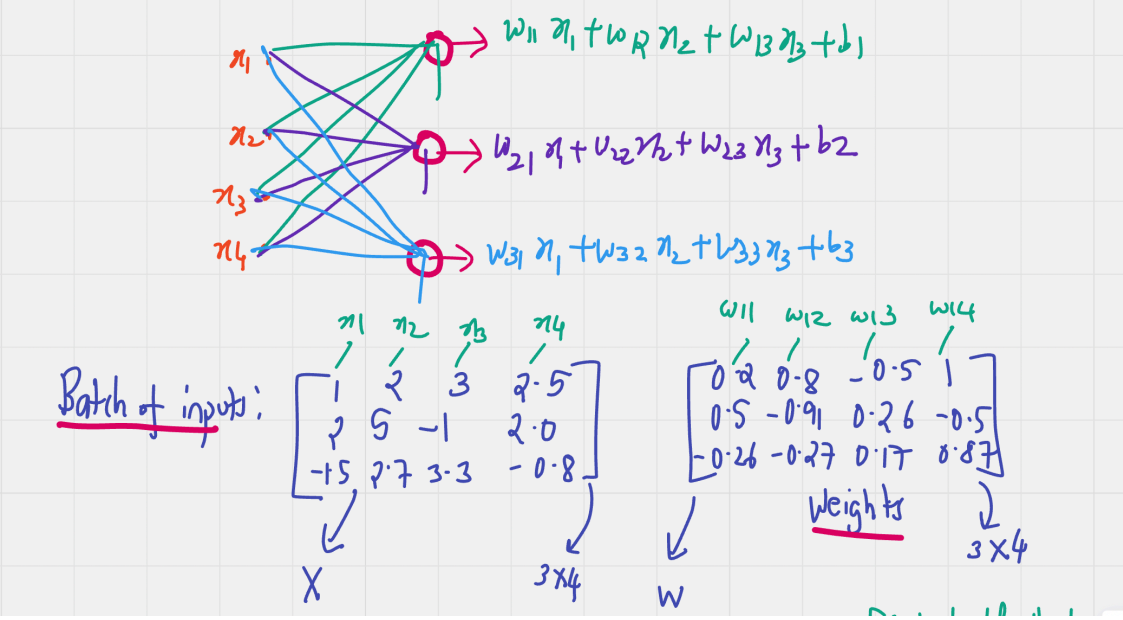



[119]:
```
##  In plain Python, we wrote this as a list of lists. With NumPy, this will be␣
↪a 2-dimensional array, which we'll call a matrix.
```

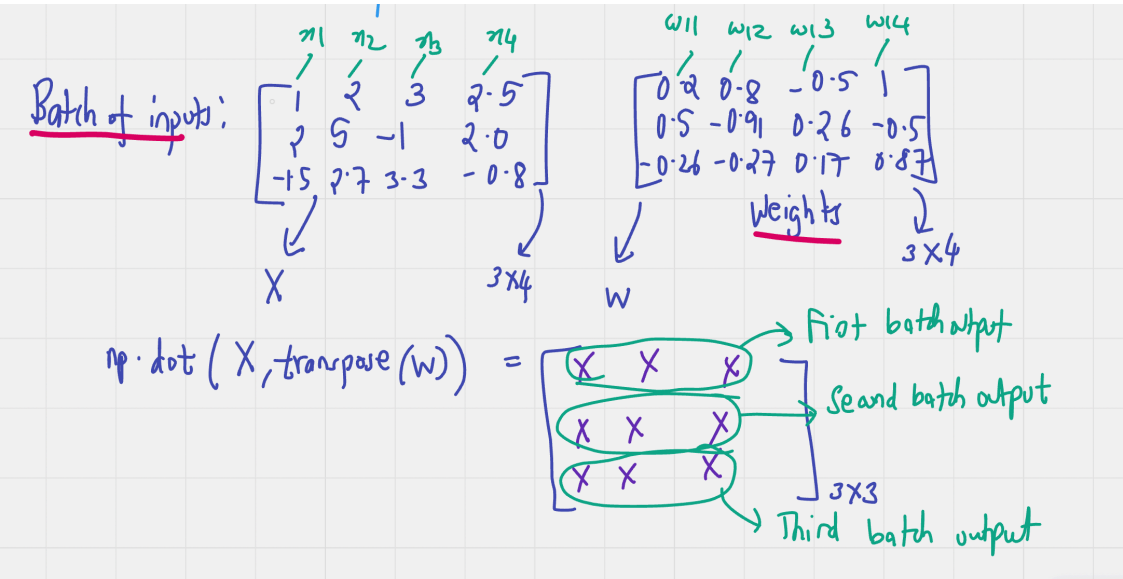
[140]:
```
import numpy as np

inputs = [1.0, 2.0, 3.0, 2.5]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

# Convert lists to numpy arrays
inputs_array = np.array(inputs)
weights_array = np.array(weights)
biases_array = np.array(biases)

# Calculate the dot product and add the biases
layer_outputs = np.dot(weights_array, inputs_array) + biases_array
print(layer_outputs)
```

```
[4.79999995 1.21000004 2.38499999]
```

LAYER OF NEURONS AND BATCH OF DATA USING NUMPY




NEED TO TAKE TRANSPOSE OF WEIGHT MATRIX




[141]:
```
import numpy as np

inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
```

```
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

# Convert lists to numpy arrays
inputs_array = np.array(inputs)
weights_array = np.array(weights)
biases_array = np.array(biases)

# Calculate the dot product and add the biases
outputs = np.dot(inputs_array, weights_array.T) + biases_array
print(outputs)
```

```
[[ 4.79999995  1.21000004  2.38499999]
 [ 8.9000001  -1.80999994  0.19999999]
 [ 1.41000003  1.051       0.02599999]]
```

2 LAYERS AND BATCH OF DATA USING NUMPY

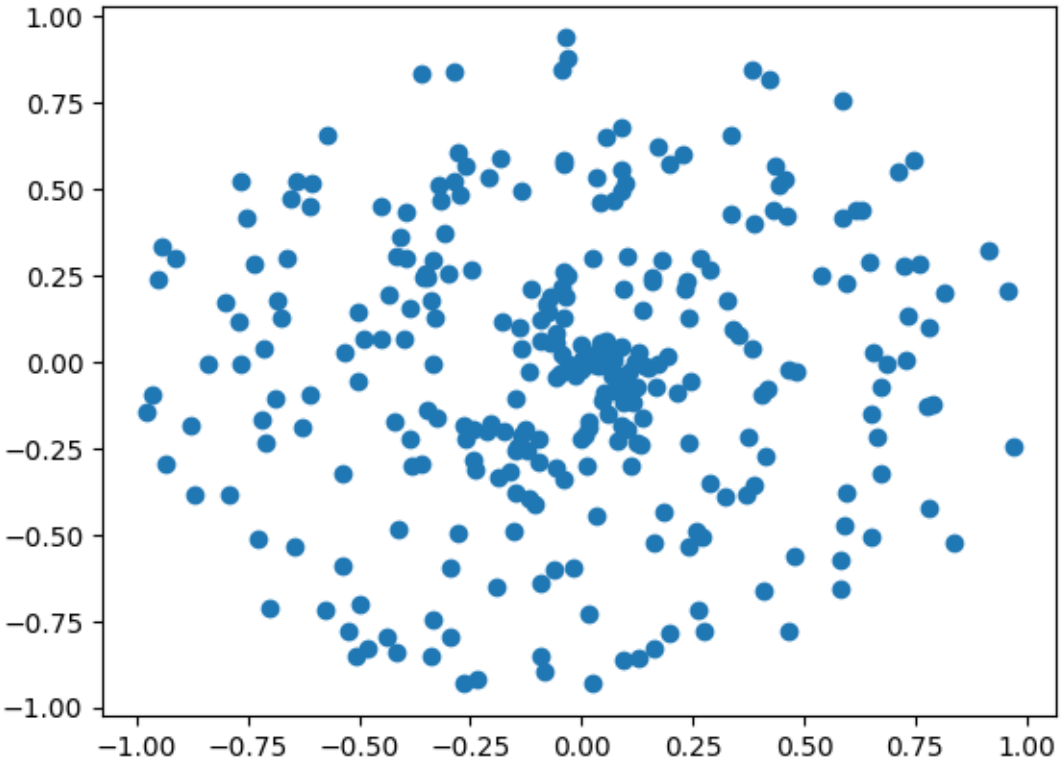
[143]:
```
import numpy as np

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

weights = [[0.2, 0.8, -0.5, 1],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]

biases = [2, 3, 0.5]

weights2 = [[0.1, -0.14, 0.5],
            [-0.5, 0.12, -0.33],
            [-0.44, 0.73, -0.13]]

biases2 = [-1, 2, -0.5]

# Convert lists to numpy arrays
inputs_array = np.array(inputs)
weights_array = np.array(weights)
biases_array = np.array(biases)
weights2_array = np.array(weights2)
biases2_array = np.array(biases2)

# Calculate the output of the first layer
layer1_outputs = np.dot(inputs_array, weights_array.T) + biases_array

# Calculate the output of the second layer
layer2_outputs = np.dot(layer1_outputs, weights2_array.T) + biases2_array
```

```
print(layer2_outputs)
```

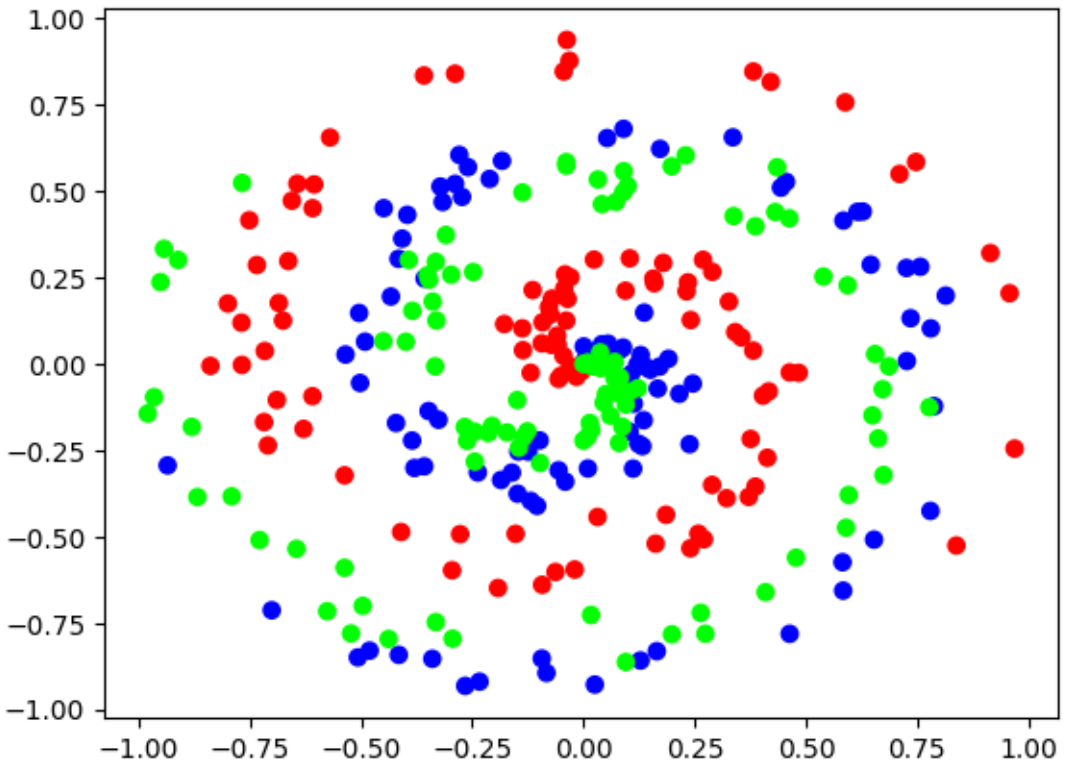
```
[[ 0.50310004 -1.04184985 -2.03874993]
 [ 0.24339998 -2.73320007 -5.76329994]
 [-0.99314     1.41254002 -0.35655001]]
```

GENERATING NON LINEAR TRAINING DATA

[3]:
```
from nnfs.datasets import spiral_data
import numpy as np
import nnfs
nnfs.init()
import matplotlib.pyplot as plt
X, y = spiral_data(samples=100, classes=3)
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```




[145]:
```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='brg')
plt.show()
```




The neural network will not be aware of the color differences as the data have no class encodings

DENSE LAYER CLASS

[5]:
```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data
nnfs.init()
# Dense layer
class Layer_Dense:
 # Layer initialization
 def __init__(self, n_inputs, n_neurons):
 # Initialize weights and biases
   self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
   self.biases = np.zeros((1, n_neurons))

 # Forward pass
 def forward(self, inputs):
 # Calculate output values from inputs, weights and biases
   self.output = np.dot(inputs, self.weights) + self.biases

# Create dataset
X, y = spiral_data(samples=100, classes=3)
```

```
# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)
# Perform a forward pass of our training data through this layer
dense1.forward(X)


# Let's see output of the first few samples:
print(dense1.output[:5])
```

```
[[ 0.0000000e+00  0.0000000e+00  0.0000000e+00]
 [-1.0475188e-04  1.1395361e-04 -4.7983500e-05]
 [-2.7414842e-04  3.1729150e-04 -8.6921798e-05]
 [-4.2188365e-04  5.2666257e-04 -5.5912682e-05]
 [-5.7707680e-04  7.1401405e-04 -8.9430439e-05]]
```

ACTIVATION FUNCTION: RELU

[22]:
```
import numpy as np
inputs = [0, 2, -1, 3.3, -2.7, 1.1, 2.2, -100]
output = np.maximum(0, inputs)
print(output)
```

```
[0.  2.  0.  3.3 0.  1.1 2.2 0. ]
```

[18]:
```
# ReLU activation
class Activation_ReLU:
 # Forward pass
 def forward(self, inputs):
 # Calculate output values from input
  self.output = np.maximum(0, inputs)
```

[19]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)
# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)
# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()
# Make a forward pass of our training data through this layer
dense1.forward(X)
# Forward pass through activation func.
# Takes in output from previous layer
activation1.forward(dense1.output)
# Let's see output of the first few samples:
print(activation1.output[:5])
```

```
[[0.         0.         0.        ]
 [0.00013767 0.         0.        ]
```

```
 [0.00022187 0.         0.        ]
 [0.0004077  0.         0.        ]
 [0.00054541 0.         0.        ]]
```

ACTIVATION FUNCTION: SOFTMAX

[4]:
```
### TRY THESE EXERCISES FOR YOURSELF!

A = [[1, 2, 3], [4, 5, 6], [7, 8,9]]
print(np.sum(A))

print(np.sum(A, axis = 0))
print(np.sum(A, axis = 0).shape)

print(np.sum(A, axis = 1))
print(np.sum(A, axis = 1).shape)

print(np.sum(A, axis = 0,keepdims = True))
print(np.sum(A, axis = 0,keepdims = True).shape)

print(np.sum(A, axis = 1,keepdims = True))
print(np.sum(A, axis = 1,keepdims = True).shape)

print(np.max(A, axis = 0))
print(np.max(A, axis = 1))
```

```
45
[12 15 18]
(3,)
[ 6 15 24]
(3,)
[[12 15 18]]
(1, 3)
[[ 6]
 [15]
 [24]]
(3, 1)
[7 8 9]
[3 6 9]
```

[5]:
```
inputs = [[1, 2, 3, 2.5],
 [2., 5., -1., 2],
 [-1.5, 2.7, 3.3, -0.8]]

# Get unnormalized probabilities
exp_values = np.exp(inputs - np.max(inputs, axis=1,keepdims=True))
 # Normalize them for each sample
probabilities = exp_values / np.sum(exp_values, axis=1,keepdims=True)
```

```
print(probabilities)
np.sum(probabilities, axis = 1)
```

```
[[0.06414769 0.17437149 0.47399085 0.28748998]
 [0.04517666 0.90739747 0.00224921 0.04517666]
 [0.00522984 0.34875873 0.63547983 0.0105316 ]]
```

[5]:
```
array([1., 1., 1.])
```

[6]:
```
# Softmax activation
class Activation_Softmax:
 # Forward pass
 def forward(self, inputs):
 # Get unnormalized probabilities
  exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
 # Normalize them for each sample
  probabilities = exp_values / np.sum(exp_values, axis=1,keepdims=True)
  self.output = probabilities
```

[20]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)
# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)
# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()
# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)
# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

# Make a forward pass of our training data through this layer
dense1.forward(X)

# Make a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)
# Make a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)
# Make a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)
# Let's see output of the first few samples:
print(activation2.output[:5])
```

```
[[0.33333334 0.33333334 0.33333334]
 [0.33333337 0.333333   0.3333336 ]
```

```
 [0.33333334 0.3333327  0.33333394]
 [0.33333334 0.333333   0.33333364]
 [0.3333334  0.33333305 0.33333358]]
```

CALCULATING NETWORK ERROR WITH LOSS

CROSS ENTROPY LOSS BUILDING BLOCKS IN PYTHON

[8]:
```
softmax_outputs = np.array([[0.7, 0.1, 0.2],
 [0.1, 0.5, 0.4],
 [0.02, 0.9, 0.08]])
class_targets = [0, 1, 1]
print(softmax_outputs[[0, 1, 2], class_targets])
```

```
[0.7 0.5 0.9]
```

[9]:
```
print(-np.log(softmax_outputs[
 range(len(softmax_outputs)), class_targets
]))
neg_log = -np.log(softmax_outputs[
 range(len(softmax_outputs)), class_targets
 ])
average_loss = np.mean(neg_log)
print(average_loss)
```

```
[0.35667494 0.69314718 0.10536052]
0.38506088005216804
```

IF DATA IS ONE HOT ENCODED, HOW TO EXTRACT THE RELEVANT PREDICTIONS

[10]:
```
y_true_check = np.array([
    [0, 1, 0],
    [1, 0, 0],
    [0, 0, 1]
])

y_pred_clipped_check = np.array([
    [0.2, 0.7, 0.1],
    [0.8, 0.1, 0.1],
    [0.1, 0.2, 0.7]
])

y_true_check*y_pred_clipped_check
```

[10]:
```
array([[0. , 0.7, 0. ],
       [0.8, 0. , 0. ],
       [0. , 0. , 0.7]])
```

IMPLEMENTING THE LOSS CLASS

[11]:
```
# Common loss class
class Loss:
 # Calculates the data and regularization losses
 # given model output and ground truth values
 def calculate(self, output, y):
  # Calculate sample losses
  sample_losses = self.forward(output, y)
  # Calculate mean loss
  data_loss = np.mean(sample_losses)
  # Return loss
  return data_loss
```

IMPLEMENTING THE CATEGORICAL CROSS ENTROPY CLASS

[12]:
```
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):
 # Forward pass
 def forward(self, y_pred, y_true):
  # Number of samples in a batch
  samples = len(y_pred)
  # Clip data to prevent division by 0
  # Clip both sides to not drag mean towards any value
  y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
  # Probabilities for target values -
  # only if categorical labels
  if len(y_true.shape) == 1:
   correct_confidences = y_pred_clipped[
   range(samples),
   y_true
   ]
  # Mask values - only for one-hot encoded labels
  elif len(y_true.shape) == 2:
   correct_confidences = np.sum(
   y_pred_clipped*y_true,
   axis=1
   )
 # Losses
  negative_log_likelihoods = -np.log(correct_confidences)
  return negative_log_likelihoods
```

[138]:
```
softmax_outputs = np.array([[0.7, 0.1, 0.2],
 [0.1, 0.5, 0.4],
 [0.02, 0.9, 0.08]])
class_targets = np.array([[1, 0, 0],
 [0, 1, 0],
 [0, 1, 0]])
loss_function = Loss_CategoricalCrossentropy()
```

```
loss = loss_function.calculate(softmax_outputs, class_targets)
print(loss)
```

```
0.38506088005216804
```

FULL CODE UPTO THIS POINT

[139]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)
# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 3)
# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()
# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)
# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()
# Create loss function
loss_function = Loss_CategoricalCrossentropy()
# Perform a forward pass of our training data through this layer
dense1.forward(X)
# Perform a forward pass through activation function
# it takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# it takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)
# Perform a forward pass through activation function
# it takes the output of second dense layer here
activation2.forward(dense2.output)
# Let's see output of the first few samples:
print(activation2.output[:5])
# Perform a forward pass through activation function
# it takes the output of second dense layer here and returns loss
loss = loss_function.calculate(activation2.output, y)
# Print loss value
print('loss:', loss)

# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(activation2.output, axis=1)
if len(y.shape) == 2:
 y = np.argmax(y, axis=1)
accuracy = np.mean(predictions == y)
# Print accuracy
print('acc:', accuracy)
```

```
[[0.33333334 0.33333334 0.33333334]
 [0.3333341  0.33333302 0.3333329 ]
 [0.3333341  0.33333302 0.33333296]
 [0.3333341  0.333333   0.33333293]
 [0.3333364  0.33333203 0.33333158]]
loss: 1.0986193
acc: 0.28
```

INTRODUCING ACCRUACY

[109]:
```
import numpy as np
# Probabilities of 3 samples
softmax_outputs = np.array([[0.7, 0.2, 0.1],
 [0.5, 0.1, 0.4],
 [0.02, 0.9, 0.08]])
# Target (ground-truth) labels for 3 samples
class_targets = np.array([0, 1, 1])
# Calculate values along second axis (axis of index 1)
predictions = np.argmax(softmax_outputs, axis=1)
# If targets are one-hot encoded - convert them
if len(class_targets.shape) == 2:
 class_targets = np.argmax(class_targets, axis=1)
# True evaluates to 1; False to 0
accuracy = np.mean(predictions == class_targets)
print('acc:', accuracy)
```

```
acc: 0.6666666666666666
```

[ ]:
```

```