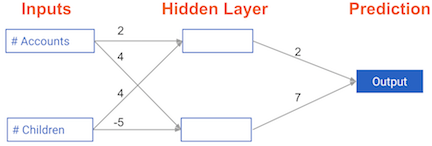
**Coding the forward propagation algorithm**

In this exercise, you'll write code to do forward propagation (prediction) for your first neural network:



Each data point is a customer. The first input is how many accounts they have, and the second input is how many children they have. The model will predict how many transactions the user makes in the next year. You will use this data throughout the first 2 chapters of this course.

The input data has been pre-loaded as input\_data, and the weights are available in a dictionary called weights. The array of weights for the first node in the hidden layer are in weights['node\_0'], and the array of weights for the second node in the hidden layer are in weights['node\_1'].

The weights feeding into the output node are available in weights['output'].

NumPy will be pre-imported for you as np in all exercises.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Calculate the value in node 0 by multiplying input\_data by its weights weights['node\_0'] and computing their sum. This is the 1st node in the hidden layer.
* Calculate the value in node 1 using input\_data and weights['node\_1']. This is the 2nd node in the hidden layer.
* Put the hidden layer values into an array. This has been done for you.
* Generate the prediction by multiplying hidden\_layer\_outputs by weights['output'] and computing their sum.
* Hit 'Submit Answer' to print the output!

[**Take Hint (-30 XP)**](javascript:void(0))

# Calculate node 0 value: node\_0\_value

node\_0\_value = (weights['node\_0'] \* input\_data).sum()

# Calculate node 1 value: node\_1\_value

node\_1\_value = (weights['node\_1'] \* input\_data).sum()

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_value, node\_1\_value])

# Calculate output: output

output = (hidden\_layer\_outputs \* weights['output']).sum()

# Print output

print(output)

In [1]: weights['node\_0']

Out[1]: array([2, 4])

In [2]: weights['node\_0'] \* input\_data

Out[2]: array([ 6, 20])

In [3]: (weights['node\_0'] \* input\_data).sum

Out[3]: <function ndarray.sum>

In [4]: (weights['node\_0'] \* input\_data).sum()

Out[4]: 26

In [5]: hidden\_layer\_outputs by weights['output']

File "<stdin>", line 1

hidden\_layer\_outputs by weights['output']

^

SyntaxError: invalid syntax

In [6]: input\_data

Out[6]: array([3, 5])

<script.py> output:

[ 52 -91]

<script.py> output:

-39

In [7]:

 +100 XP

Wonderful work! It looks like the network generated a prediction of -39.

**Exercise**

**The Rectified Linear Activation Function**

As Dan explained to you in the video, an "activation function" is a function applied at each node. It converts the node's input into some output.

The rectified linear activation function (called *ReLU*) has been shown to lead to very high-performance networks. This function takes a single number as an input, returning 0 if the input is negative, and the input if the input is positive.

Here are some examples:  
**relu(3) = 3**  
**relu(-3) = 0**

**Instructions**

**100 XP**

* Fill in the definition of the relu() function:
  + Use the max() function to calculate the value for the output of relu().
* Apply the relu() function to node\_0\_input to calculate node\_0\_output.
* Apply the relu() function to node\_1\_input to calculate node\_1\_output.

[**Take Hint (-30 XP)**](javascript:void(0))

def relu(input):

'''Define your relu activation function here'''

# Calculate the value for the output of the relu function: output

output = max(input, 0)

# Return the value just calculated

return(output)

# Calculate node 0 value: node\_0\_output

node\_0\_input = (input\_data \* weights['node\_0']).sum()

node\_0\_output = relu(node\_0\_input)

# Calculate node 1 value: node\_1\_output

node\_1\_input = (input\_data \* weights['node\_1']).sum()

node\_1\_output = relu(node\_1\_input)

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output (do not apply relu)

model\_output = (hidden\_layer\_outputs \* weights['output']).sum()

# Print model output

print(model\_output)

<script.py> output:

52

In [1]:

 +100 XP

Great work! You predicted 52 transactions. Without this activation function, you would have predicted a negative number! The real power of activation functions will come soon when you start tuning model weights.

**Exercise**

**Exercise**

**Applying the network to many observations/rows of data**

You'll now define a function called predict\_with\_network() which will generate predictions for multiple data observations, which are pre-loaded as input\_data. As before, weights are also pre-loaded. In addition, the relu() function you defined in the previous exercise has been pre-loaded.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Define a function called predict\_with\_network() that accepts two arguments - input\_data\_row and weights - and returns a prediction from the network as the output.
* Calculate the input and output values for each node, storing them as: node\_0\_input, node\_0\_output, node\_1\_input, and node\_1\_output.
  + To calculate the input value of a node, multiply the relevant arrays together and compute their sum.
  + To calculate the output value of a node, apply the relu() function to the input value of the node.
* Calculate the model output by calculating input\_to\_final\_layer and model\_output in the same way you calculated the input and output values for the nodes.
* Use a for loop to iterate over input\_data:
  + Use your predict\_with\_network() to generate predictions for each row of the input\_data - input\_data\_row. Append each prediction to results.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

The contents of node\_0\_input aren't correct. Did you multiply the correct arrays together and compute their sum?

Check your call of results.append(). Did you correctly specify the first argument? Have you specified the arguments for predict\_with\_network() using the right syntax?

Check your call of predict\_with\_network(). Did you correctly specify the first argument? Expected [3 5], but got 3.

# Define predict\_with\_network()

def predict\_with\_network(input\_data\_row, weights):

# Calculate node 0 value

node\_0\_input = (input\_data\_row \* weights['node\_0']).sum()

node\_0\_output = relu(node\_0\_input)

# Calculate node 1 value

node\_1\_input = (input\_data\_row \* weights['node\_1']).sum()

node\_1\_output = relu(node\_1\_input)

# Put node values into array: hidden\_layer\_outputs

hidden\_layer\_outputs = np.array([node\_0\_output, node\_1\_output])

# Calculate model output

input\_to\_final\_layer = np.array([node\_0\_output, node\_1\_output])

model\_output = (hidden\_layer\_outputs \* weights['output']).sum()

# Return model output

return(model\_output)

# Create empty list to store prediction results

results = []

for input\_data\_row in input\_data:

# Append prediction to results

results.append(predict\_with\_network(input\_data\_row, weights))

# Print results

print(results)

Traceback (most recent call last):

File "script.py", line 27, in <module>

results.append(predict\_with\_network(input\_data\_row))

TypeError: predict\_with\_network() missing 1 required positional argument: 'weights'

In [1]: input\_data

Out[1]: [array([3, 5]), array([ 1, -1]), array([0, 0]), array([8, 4])]

Traceback (most recent call last):

File "script.py", line 27, in <module>

results.append(predict\_with\_network(input\_data\_row))

TypeError: predict\_with\_network() missing 1 required positional argument: 'weights'

In [2]: input\_data

Out[2]: [array([3, 5]), array([ 1, -1]), array([0, 0]), array([8, 4])]

Traceback (most recent call last):

File "script.py", line 27, in <module>

results.append(predict\_with\_network(input\_data\_row[0], input\_data\_row[1]))

File "script.py", line 5, in predict\_with\_network

node\_0\_input = (input\_data\_row \* weights['node\_0']).sum()

IndexError: invalid index to scalar variable.

In [3]: input\_data\_row[0]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

input\_data\_row[0]

NameError: name 'input\_data\_row' is not defined

In [4]: weights

Out[4]: {'node\_0': array([2, 4]), 'node\_1': array([ 4, -5]), 'output': array([2, 7])}

In [5]:

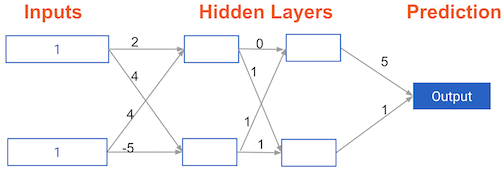
<script.py> output:

[52, 63, 0, 148]

**Forward propagation in a deeper network**

You now have a model with 2 hidden layers. The values for an input data point are shown inside the input nodes. The weights are shown on the edges/lines. What prediction would this model make on this data point?

Assume the activation function at each node is the *identity function*. That is, each node's output will be the same as its input. So the value of the bottom node in the first hidden layer is -1, and not 0, as it would be if the ReLU activation function was used.



**Answer the question**

**50 XP**

**Possible Answers**

0.

7.

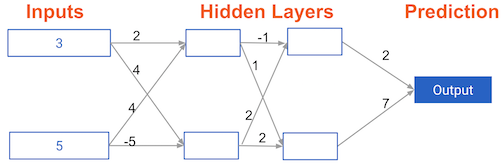
9.

**Multi-layer neural networks**

In this exercise, you'll write code to do forward propagation for a neural network with 2 hidden layers. Each hidden layer has two nodes. The input data has been preloaded as input\_data. The nodes in the first hidden layer are called node\_0\_0 and node\_0\_1. Their weights are pre-loaded as weights['node\_0\_0'] and weights['node\_0\_1'] respectively.

The nodes in the second hidden layer are called node\_1\_0 and node\_1\_1. Their weights are pre-loaded as weights['node\_1\_0'] and weights['node\_1\_1'] respectively.

We then create a model output from the hidden nodes using weights pre-loaded as weights['output'].



**Instructions**

**100 XP**

* Calculate node\_0\_0\_input using its weights weights['node\_0\_0'] and the given input\_data. Then apply the relu() function to get node\_0\_0\_output.
* Do the same as above for node\_0\_1\_input to get node\_0\_1\_output.
* Calculate node\_1\_0\_input using its weights weights['node\_1\_0'] and the outputs from the first hidden layer - hidden\_0\_outputs. Then apply the relu() function to get node\_1\_0\_output.
* Do the same as above for node\_1\_1\_input to get node\_1\_1\_output.
* Calculate model\_output using its weights weights['output'] and the outputs from the second hidden layer hidden\_1\_outputs array. Do not apply the relu() function to this output.

[**Take Hint (-30 XP)**](javascript:void(0))

def predict\_with\_network(input\_data):

# Calculate node 0 in the first hidden layer

node\_0\_0\_input = (weights['node\_0\_0'] \* input\_data).sum()

node\_0\_0\_output = relu(node\_0\_0\_input)

# Calculate node 1 in the first hidden layer

node\_0\_1\_input = (weights['node\_0\_1'] \* input\_data).sum()

node\_0\_1\_output = relu(node\_0\_1\_input)

# Put node values into array: hidden\_0\_outputs

hidden\_0\_outputs = np.array([node\_0\_0\_output, node\_0\_1\_output])

# Calculate node 0 in the second hidden layer

node\_1\_0\_input = (weights['node\_1\_0'] \* hidden\_0\_outputs).sum()

node\_1\_0\_output = relu(node\_1\_0\_input)

# Calculate node 1 in the second hidden layer

node\_1\_1\_input = (weights['node\_1\_1'] \* hidden\_0\_outputs).sum()

node\_1\_1\_output = relu(node\_1\_1\_input)

# Put node values into array: hidden\_1\_outputs

hidden\_1\_outputs = np.array([node\_1\_0\_output, node\_1\_1\_output])

# Calculate model output: model\_output

model\_output = (hidden\_1\_outputs\*weights['output']).sum()

# Return model\_output

return(model\_output)

output = predict\_with\_network(input\_data)

print(output)

In [1]: weights

Out[1]:

{'node\_0\_0': array([2, 4]),

'node\_0\_1': array([ 4, -5]),

'node\_1\_0': array([-1, 2]),

'node\_1\_1': array([1, 2]),

'output': array([2, 7])}

In [2]: relu

Out[2]: <function \_\_main\_\_.relu>

In [3]:

<script.py> output:

[ 0 182]

<script.py> output:

182

In [1]:

**Incorrect Submission**

The contents of node\_0\_1\_input aren't correct. Did you multiply the correct arrays together and compute their sum?

The contents of node\_1\_0\_input aren't correct. Did you multiply the correct arrays together and compute their sum?

The contents of model\_output aren't correct. Did you multiply the correct arrays and compute their sum?

 +100 XP

Wonderful work! The network generated a prediction of 182.

**Representations are learned**

How are the weights that determine the features/interactions in Neural Networks created?

**Answer the question**

**50 XP**

**Possible Answers**

* 

A user chooses them when creating the model.

press1

* 

The model training process sets them to optimize predictive accuracy.

press2

* 

The weights are random numbers.

press3

 +50 XP

Exactly! You will learn more about how Neural Networks optimize their weights in the next chapter!

**Levels of representation**

Which layers of a model capture more complex or "higher level" interactions?

**Answer the question**

**50 XP**

**Possible Answers**

The first layers capture the most complex interactions.

The last layers capture the most complex interactions.

All layers capture interactions of similar complexity.

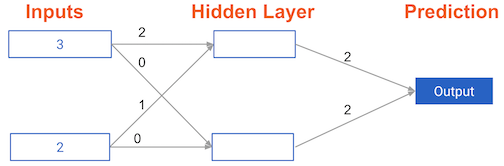
 +50 XP

Exactly! The last layers capture the most complex interactions.

**Calculating model errors**

For the exercises in this chapter, you'll continue working with the network to predict transactions for a bank.

What is the error (predicted - actual) for the following network using the ReLU activation function when the input data is [3, 2] and the actual value of the target (what you are trying to predict) is 5? It may be helpful to get out a pen and piece of paper to calculate these values.



**Answer the question**

**50 XP**

**Possible Answers**

5.

6.

11.

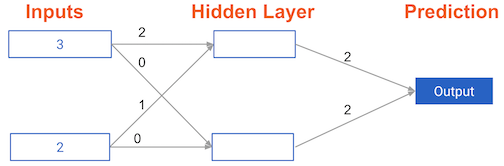
16.

 +50 XP

Well done! The network generates a prediction of 16, which results in an error of 11.

**Understanding how weights change model accuracy**

Imagine you have to make a prediction for a single data point. The actual value of the target is 7. The weight going from node\_0 to the output is 2, as shown below. If you increased it slightly, changing it to 2.01, would the predictions become more accurate, less accurate, or stay the same?



**Possible Answers**

More accurate.

Less accurate.

Stay the same.

 +50 XP

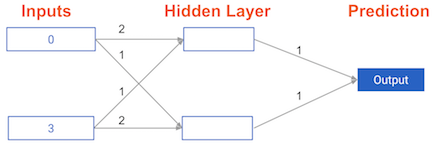
Exactly! Increasing the weight to 2.01 would increase the resulting error from 9 to 9.08, making the predictions *less* accurate.

**Exercise**

**Exercise**

**Coding how weight changes affect accuracy**

Now you'll get to change weights in a real network and see how they affect model accuracy!

Have a look at the following neural network: 

Its weights have been pre-loaded as weights\_0. Your task in this exercise is to update a **single** weight in weights\_0 to create weights\_1, which gives a perfect prediction (in which the predicted value is equal to target\_actual: 3).

Use a pen and paper if necessary to experiment with different combinations. You'll use the predict\_with\_network() function, which takes an array of data as the first argument, and weights as the second argument.

**Instructions**

**100 XP**

* Create a dictionary of weights called weights\_1 where you have changed **1** weight from weights\_0 (You only need to make 1 edit to weights\_0 to generate the perfect prediction).
* Obtain predictions with the new weights using the predict\_with\_network() function with input\_data and weights\_1.
* Calculate the error for the new weights by subtracting target\_actual from model\_output\_1.
* Hit 'Submit Answer' to see how the errors compare!

Ctrl+H

[**Take Hint (-30 XP)**](javascript:void(0))

# The data point you will make a prediction for

input\_data = np.array([0, 3])

# Sample weights

weights\_0 = {'node\_0': [2, 1],

'node\_1': [1, 2],

'output': [1, 1]

}

# The actual target value, used to calculate the error

target\_actual = 3

# Make prediction using original weights

model\_output\_0 = predict\_with\_network(input\_data, weights\_0)

# Calculate error: error\_0

error\_0 = model\_output\_0 - target\_actual

# Create weights that cause the network to make perfect prediction (3): weights\_1

weights\_1 = {'node\_0': [2, 1],

'node\_1': [1, 2],

'output': [1, 0]

}

# Make prediction using new weights: model\_output\_1

model\_output\_1 = predict\_with\_network(input\_data, weights\_1)

# Calculate error: error\_1

error\_1 = model\_output\_1 - target\_actual

# Print error\_0 and error\_1

print(error\_0)

print(error\_1)

Traceback (most recent call last):

File "<stdin>", line 20, in <module>

weights\_1 = {'node\_0': [\_\_\_\_, \_\_\_\_],

NameError: name '\_\_\_\_' is not defined

In [2]: # The data point you will make a prediction for

input\_data = np.array([0, 3])

# Sample weights

weights\_0 = {'node\_0': [2, 1],

'node\_1': [1, 2],

'output': [1, 1]

}

# The actual target value, used to calculate the error

target\_actual = 3

# Make prediction using original weights

model\_output\_0 = predict\_with\_network(input\_data, weights\_0)

# Calculate error: error\_0

error\_0 = model\_output\_0 - target\_actual

# Print error\_0 and error\_1

print(error\_0)

#print(error\_1)

6

In [3]: model\_output\_0

Out[3]: 9

<script.py> output:

6

0

In [4]:

 +100 XP

Fantastic! The network now generates a perfect prediction with an error of 0.