

# NEURAL NETWORKS FROM SCRATCH IN PYTHON

By Harrison Kinsley & Daniel Kukiela

Feel free to post comments/questions by highlighting the text and clicking the comment button to the right. Looks like this:



*Do not resolve comments that are not yours.*

Links to chapters:

[Chapter 1 - Introducing Neural Networks](#)

[Chapter 2 - Coding Our First Neurons](#)

[Chapter 3 - Adding Layers](#)

[Chapter 4 - Activation Functions](#)

[Chapter 5 - Loss](#)

[Chapter 6 - Optimization](#)

[Chapter 7 - Derivatives](#)

[Chapter 8 - Gradients, Partial Derivatives, and the Chain Rule](#)

[Chapter 9 - Backpropagation](#)

[Chapter 10 - Optimizers](#)

[Chapter 11 - Testing Data](#)

[Chapter 12 - Validation Data](#)

[Chapter 13 - Training Dataset](#)

[Chapter 14 - L1 and L2 Regularization](#)

[Chapter 15 - Dropout](#)

[Chapter 16 - Binary Logistic Regression](#)

[Chapter 17 - Regression](#)

[Chapter 18 - Model Object](#)

[Chapter 19 - A Real Dataset](#)

[Chapter 20 - Model Evaluation](#)

[Chapter 21 - Saving and Loading Model Information](#)

[Chapter 22 - Model Predicting/Inference](#)

# *Neural Networks from Scratch in Python*

Harrison Kinsley & Daniel Kukiela

# *Copyright*

Copyright © 2020 Harrison Kinsley

Cover Design copyright © 2020 Harrison Kinsley

No part of this book may be reproduced in any form or by any electronic or mechanical means, with the following exceptions:

1. Brief quotations from the book.
2. Python Code/software (strings interpreted as logic with Python), which is housed under the MIT license, described on the next page.

## *License for Code*

The Python code/software in this book is contained under the following MIT License:

Copyright © 2020 Sentdex, Kinsley Enterprises Inc., <https://nnfs.io>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# *Readme*

The objective of this book is to break down an extremely complex topic, neural networks, into small pieces, consumable by anyone wishing to embark on this journey. Beyond breaking down this topic, the hope is to dramatically demystify neural networks. As you will soon see, this subject, when explored from scratch, can be an educational and engaging experience. This book is for anyone willing to put in the time to sit down and work through it. In return, you will gain a far deeper understanding than most when it comes to neural networks and deep learning.

This book will be easier to understand if you already have an understanding of Python or another programming language. Python is one of the most clear and understandable programming languages; we have no real interest in padding page counts and exhausting an entire first chapter with a basics of Python tutorial. If you need one, we suggest you start here: <https://pythonprogramming.net/python-fundamental-tutorials/> To cite this material:

*Harrison Kinsley & Daniel Kukiela Neural Networks from Scratch (NNFS) <https://nnfs.io>*

## Chapter 2

# *Coding Our First Neurons*

While we assume that we're all beyond beginner programmers here, we will still try to start slowly and explain things the first time we see them. To begin, we will be using **Python 3.7** (although any version of Python 3+ will likely work). We will also be using **NumPy** after showing the pure-Python methods and Matplotlib for some visualizations. It should be the case that a huge variety of versions should work, but you may wish to match ours exactly to rule out any version issues. Specifically, we are using:

*Python 3.7.5*

*NumPy 1.15.0*

*Matplotlib 3.1.1*

Since this is a *Neural Networks from Scratch in Python* book, we will demonstrate how to do things without NumPy as well, but NumPy is Python's all-things-numbers package. Building from scratch is the point of this book though ignoring NumPy would be a disservice since it is among the most, if not the most, important and useful packages for data science in Python.

## A Single Neuron

Let's say we have a single neuron, and there are three inputs to this neuron. As in most cases, when you initialize parameters in neural networks, our network will have weights initialized randomly, and biases set as zero to start. Why we do this will become apparent later on. The input will be either actual training data or the outputs of neurons from the previous layer in the neural network. We're just going to make up values to start with as input for now:

```
inputs = [1, 2, 3]
```

Each input also needs a weight associated with it. Inputs are the data that we pass into the model to get desired outputs, while the weights are the parameters that we'll tune later on to get these results. Weights are one of the types of values that change inside the model during the training phase, along with biases that also change during training. The values for weights and biases are what get "trained," and they are what make a model actually work (or not work). We'll start by making up weights for now. Let's say the first input, at index 0, which is a 1, has a weight of 0.2, the second input has a weight of 0.8, and the third input has a weight of -0.5. Our input and weights lists should now be:

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

Next, we need the bias. At the moment, we're modeling a single neuron with three inputs. Since we're modeling a single neuron, we only have one bias, as there's just one bias value per neuron. The bias is an additional tunable value but is not associated with any input in contrast to the weights. We'll randomly select a value of 2 as the bias for this example:

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

This neuron sums each input multiplied by that input's weight, then adds the bias. All the neuron does is take the fractions of inputs, where these fractions (weights) are the adjustable parameters, and adds another adjustable parameter — the bias — then outputs the result. Our output would be calculated up to this point like:

```
output = (inputs[0]*weights[0] +
          inputs[1]*weights[1] +
          inputs[2]*weights[2] + bias)
```

```
print(output)
```

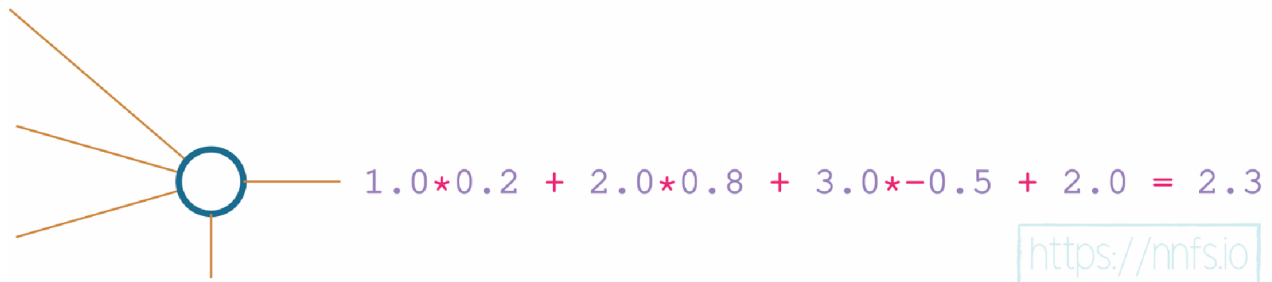
```
>>>
2.3
```

The output here should be **2.3**. We will use `>>>` to denote output in this book.

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

```
output = inputs[0]*weights[0] + inputs[1]*weights[1] + inputs[2]*weights[2] + bias
print(output)
```

```
>>> 2.3
```



**Fig 2.01:** Visualizing the code that makes up the math of a basic neuron.



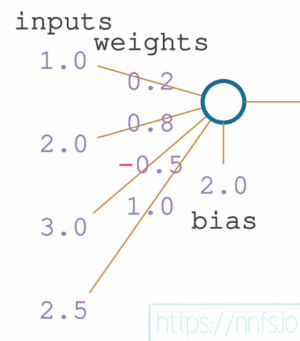
Anim 2.01: <https://nnfs.io/bkr>



What might we need to change if we have 4 inputs, rather than the 3 we've just shown? Next to the additional input, we need to add an associated weight, which this new input will be multiplied with. We'll make up a value for this new weight as well. Code for this data could be:

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

Which could be depicted visually as:



**Fig 2.02:** Visualizing how the inputs, weights, and biases from the code interact with the neuron.



**Anim 2.02:** <https://nnfs.io/djp>

All together in code, including the new input and weight, to produce output:

```
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)
```

```
>>>
```

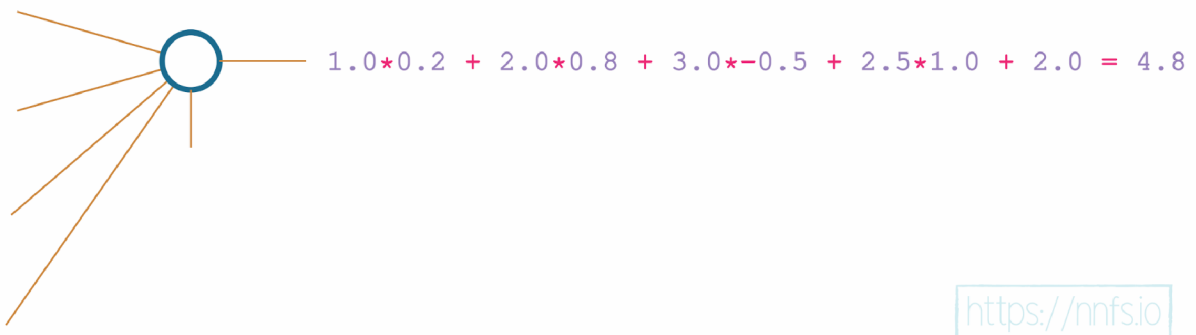
```
4.8
```

Visually:

```
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
```



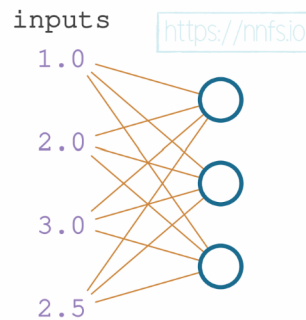
**Fig 2.03:** Visualizing the code that makes up a basic neuron, with 4 inputs this time.



**Anim 2.03:** <https://nnfs.io/djp>

## A Layer of Neurons

Neural networks typically have layers that consist of more than one neuron. Layers are nothing more than groups of neurons. Each neuron in a layer takes exactly the same input — the input given to the layer (which can be either the training data or the output from the previous layer), but contains its own set of weights and its own bias, producing its own unique output. The layer's output is a set of each of these outputs — one per each neuron. Let's say we have a scenario with 3 neurons in a layer and 4 inputs:



**Fig 2.04:** Visualizing a layer of neurons with common input.



**Anim 2.04:** <https://nnfs.io/mxo>

We'll keep the initial 4 inputs and set of weights for the first neuron the same as we've been using so far. We'll add 2 additional, made up, sets of weights and 2 additional biases to form 2 new neurons for a total of 3 in the layer. The layer's output is going to be a list of 3 values, not just a single value like for a single neuron.

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

weights1 = [0.2, 0.8, -0.5, 1]
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

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]
```

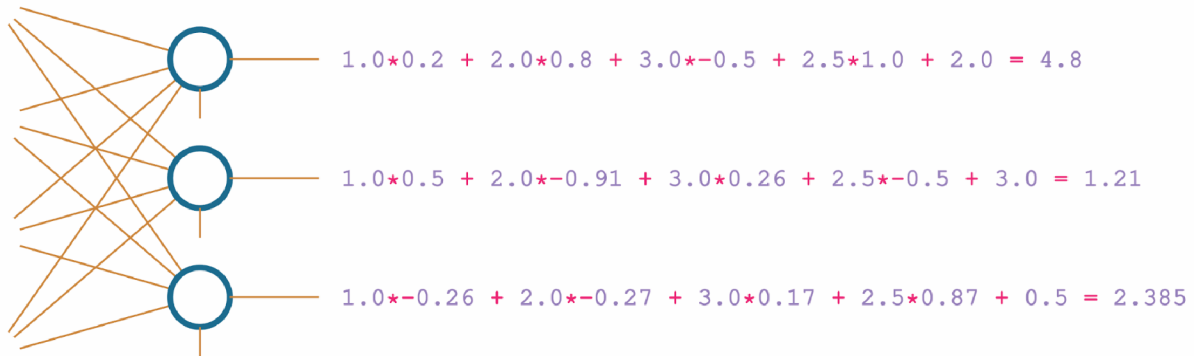
```

inputs = [1.0, 2.0, 3.0, 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 = 2.0
bias2 = 3.0
bias3 = 0.5

outputs = [
    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(outputs)

>>> [4.8, 1.21, 2.385]

```

<https://nnfs.io>


**Fig 2.04.2:** Code, math and visuals behind a layer of neurons.



**Anim 2.04:** <https://nnfs.io/mxo>

In this code, we have three sets of weights and three biases, which define three neurons. Each neuron is “connected” to the same inputs. The difference is in the separate weights and bias that each neuron applies to the input. This is called a **fully connected** neural network — every neuron in the current layer has connections to every neuron from the previous layer. This is a very common type of neural network, but it should be noted that there is no requirement to fully connect everything like this. At this point, we have only shown code for a single layer with very few neurons. Imagine coding many more layers and more neurons. This would get very challenging to code using our current methods. Instead, we could use a loop to scale and handle dynamically-sized inputs and layers. We’ve turned the separate weight variables into a list of weights so we can iterate over them, and we changed the code to use loops instead of the hardcoded operations.

```

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]]
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
    # Add bias
    neuron_output += neuron_bias
    # Put neuron's result to the layer's output list
    layer_outputs.append(neuron_output)

print(layer_outputs)

>>>
[4.8, 1.21, 2.385]

```

This does the same thing as before, just in a more dynamic and scalable way. If you find yourself confused at one of the steps, `print()` out the objects to see what they are and what's happening. The `zip` function lets us iterate over multiple iterables (lists in this case) simultaneously. Again, all we're doing is, for each neuron (the outer loop in the code above, over neuron weights and biases), taking each input value multiplied by the associated weight for that input (the inner loop in the code above, over inputs and weights), adding all of these together, then adding a bias at the end. Finally, sending the neuron's output to the layer's output list.

That's it! How do we know we have three neurons? Why do we have three? We can tell we have three neurons because there are 3 sets of weights and 3 biases. When you make a neural network of your own, you also get to decide how many neurons you want for each of the layers. You can combine however many inputs you are given with however many neurons that you desire. As you progress through this book, you will gain some intuition of how many neurons to try using. We will start by using trivial numbers of neurons to aid in understanding how neural networks work at their core.

With our above code that uses loops, we could modify our number of inputs or neurons in our layer to be whatever we wanted, and our loop would handle it. As we said earlier, it would be a disservice not to show NumPy here since Python alone doesn't do matrix/tensor/array math very efficiently. But first, the reason the most popular deep learning library in Python is called "TensorFlow" is that it's all about doing operations on **tensors**.

## Tensors, Arrays and Vectors

*What are "tensors?"*

Tensors are *closely-related to* arrays. If you interchange tensor/array/matrix when it comes to machine learning, people probably won't give you too hard of a time. But there are subtle differences, and they are primarily either the context or attributes of the tensor object. To understand a tensor, let's compare and describe some of the other data containers in Python (things that hold data). Let's start with a list. A Python list is defined by comma-separated objects contained in brackets. So far, we've been using lists.

This is an example of a simple list:

```
l = [1,5,6,2]
```

A list of lists:

```
l1l = [[1,5,6,2],  
       [3,2,1,3]]
```

A list of lists of lists!

```
l1l1l = [[[1,5,6,2],  
          [3,2,1,3]],  
         [[5,2,1,2],  
          [6,4,8,4]],  
         [[2,8,5,3],  
          [1,1,9,4]]]
```

Everything shown so far could also be an array or an array representation of a tensor. A list is just a list, and it can do pretty much whatever it wants, including:

```
another_list_of_lists = [[4,2,3],  
                        [5,1]]
```

The above list of lists cannot be an array because it is not **homologous**. A list of lists is homologous if each list along a dimension is identically long, and this must be true for each dimension. In the case of the list shown above, it's a 2-dimensional list. The first dimension's length is the number of sublists in the total list (2). The second dimension is the length of each of those sublists (3, then 2). In the above example, when reading across the “row” dimension (also called the second dimension), the first list is 3 elements long, and the second list is 2 elements long — this is not homologous and, therefore, cannot be an array. While failing to be consistent in one dimension is enough to show that this example is not homologous, we could also read down the “column” dimension (the first dimension); the first two columns are 2 elements long while the third column only contains 1 element. Note that every dimension does not necessarily need to be the same length; it is perfectly acceptable to have an array with 4 rows and 3 columns (i.e., 4x3).

A matrix is pretty simple. It's a rectangular array. It has columns and rows. It is two dimensional. So a matrix can be an array (a 2D array). Can all arrays be matrices? No. An array can be far more than just columns and rows, as it could have four dimensions, twenty dimensions, and so on.

```
list_matrix_array = [[4,2],  
                    [5,1],  
                    [8,2]]
```

The above list could also be a valid matrix (because of its columns and rows), which automatically means it could also be an array. The “shape” of this array would be 3x2, or more formally described as a shape of (3, 2) as it has 3 rows and 2 columns.

To denote a shape, we need to check every dimension. As we've already learned, a matrix is a 2-dimensional array. The first dimension is what's inside the most outer brackets, and if we look at the above matrix, we can see 3 lists there: [4,2], [5,1], and [8,2]; thus, the size in this dimension is 3 and each of those lists has to be the same shape to form an array (and matrix in this case). The next dimension's size is the number of elements inside this more inner pair of brackets, and we see that it's 2 as all of them contain 2 elements.



With 3-dimensional arrays, like in *lolol* below, we'll have a 3rd level of brackets:

```
lolol = [[[1,5,6,2],
           [3,2,1,3]],
          [[5,2,1,2],
           [6,4,8,4]],
          [[2,8,5,3],
           [1,1,9,4]]]
```

The first level of this array contains 3 matrices:

```
[[1,5,6,2],
 [3,2,1,3]]
```

```
[[5,2,1,2],
 [6,4,8,4]]
```

And

```
[[2,8,5,3],
 [1,1,9,4]]
```

That's what's inside the most outer brackets and the size of this dimension is then 3. If we look at the first matrix, we can see that it contains 2 lists — `[1,5,6,2]` and `[3,2,1,3]` so the size of this dimension is 2 — while each list of this inner matrix includes 4 elements. These 4 elements make up the 3rd and last dimension of this array since there are no more inner brackets. Therefore, the shape of this array is `(3, 2, 4)` and it's a 3-dimensional array since the shape contains 3 dimensions.

```
Array :                               Shape :
lolol = [[[1,5,6,2],                  (3, 2, 4)
           [3,2,1,3]],
          [[5,2,1,2],
           [6,4,8,4]],
          [[2,8,5,3],
           [1,1,9,4]]]
```

<https://nnfs.io> Type : 3D Array

**Fig 2.05:** Example of a 3-dimensional array.



**Anim 2.05:** <https://nnfs.io/jps>

Finally, what's a tensor? When it comes to the discussion of tensors versus arrays in the context of computer science, pages and pages of debate have ensued. This intense debate appears to be caused by the fact that people are arguing from entirely different places. There's no question that a tensor is not just an array, but the real question is: "What is a tensor, to a computer scientist, in the context of deep learning?" We believe that we can solve the debate in one line:

*A tensor object is an object that can be represented as an array.*

What this means is, as programmers, we can (and will) treat tensors as arrays in the context of deep learning, and that's really all the thought we have to put into it. Are all tensors *just* arrays? No, but they are represented as arrays in our code, so, to us, they're only arrays, and this is why there's so much argument and confusion.

Now, what is an array? In this book, we define an array as an ordered homologous container for numbers, and mostly use this term when working with the NumPy package since that's what the main data structure is called within it. A linear array, also called a 1-dimensional array, is the simplest example of an array, and in plain Python, this would be a list. Arrays can also consist of multi-dimensional data, and one of the best-known examples is what we call a matrix in mathematics, which we'll represent as a 2-dimensional array. Each element of the array can be accessed using a tuple of indices as a key, which means that we can retrieve any array element.

We need to learn one more notion — a vector. Put simply, a vector in math is what we call a list in Python or a 1-dimensional array in NumPy. Of course, lists and NumPy arrays do not have the same properties as a vector, but, just as we can write a matrix as a list of lists in Python, we can also write a vector as a list or an array! Additionally, we'll look at the vector algebraically (mathematically) as a set of numbers in brackets. This is in contrast to the physics perspective, where the vector's representation is usually seen as an arrow, characterized by a magnitude and a direction.

## Dot Product and Vector Addition

Let's now address vector multiplication, as that's one of the most important operations we'll perform on vectors. We can achieve the same result as in our pure Python implementation of multiplying each element in our inputs and weights vectors element-wise by using a **dot product**, which we'll explain shortly. Traditionally, we use dot products for **vectors** (yet another name for a container), and we can certainly refer to what we're doing here as working with vectors just as we can call them "tensors." Nevertheless, this seems to add to the mysticism of neural networks — like they're these objects out in a complex multi-dimensional vector space that we'll never understand. Keep thinking of vectors as arrays — a 1-dimensional array is just a vector (or a list in Python).

Because of the sheer number of variables and interconnections made, we can model very complex and non-linear relationships with non-linear activation functions, and truly feel like wizards, but this might do more harm than good. Yes, we will be using the "dot product," but we're doing this because it results in a clean way to perform the necessary calculations. It's nothing more in-depth than that — as you've already seen, we can do this math with far more rudimentary-sounding words. When multiplying vectors, you either perform a dot product or a cross product. A cross product results in a vector while a dot product results in a scalar (a single value/number).

First, let's explain what a dot product of two vectors is. Mathematicians would say:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

A dot product of two vectors is a sum of products of consecutive vector elements. Both vectors must be of the same size (have an equal number of elements).

Let's write out how a dot product is calculated in Python. For it, you have two vectors, which we can represent as lists in Python. We then multiply their elements from the same index values and then add all of the resulting products. Say we have two lists acting as our vectors:

```
a = [1, 2, 3]
b = [2, 3, 4]
```

To obtain the dot product:

```
dot_product = a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
print(dot_product)

>>>
20

a = [1, 2, 3]
b = [2, 3, 4]

dot_product = a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
>>> 20
```

$$\vec{a} \cdot \vec{b} = [1, 2, 3] \cdot [2, 3, 4] = 1 \cdot 2 + 2 \cdot 3 + 3 \cdot 4 = 20$$

<https://nnfs.io>

**Fig 2.06:** Math behind the dot product example.



**Anim 2.06:** <https://nnfs.io/xpo>

Now, what if we called  $a$  “inputs” and  $b$  “weights?” Suddenly, this dot product looks like a succinct way to perform the operations we need and have already performed in plain Python. We need to multiply our weights and inputs of the same index values and add the resulting values together. The dot product performs this exact type of operation; thus, it makes lots of sense to use here. Returning to the neural network code, let’s make use of this dot product. Plain Python does not contain methods or functions to perform such an operation, so we’ll use the NumPy package, which is capable of this, and many more operations that we’ll use in the future.

We’ll also need to perform a vector addition operation in the not-too-distant future. Fortunately, NumPy lets us perform this in a natural way — using the plus sign with the variables containing vectors of the data. The addition of the two vectors is an operation performed element-wise, which means that both vectors have to be of the same size, and the result will become a vector of this

size as well. The result is a vector calculated as a sum of the consecutive vector elements:

$$\vec{a} + \vec{b} = [a_1 + b_1, a_2 + b_2, \dots, a_n + b_n]$$

## A Single Neuron with NumPy

Let's code the solution, for a single neuron to start, using the dot product and the addition of the vectors with NumPy. This makes the code much simpler to read and write (and faster to run):

```
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

outputs = np.dot(weights, inputs) + bias

print(outputs)

>>>
4.8
```

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

outputs = np.dot(weights, inputs) + bias

np.dot([0.2, 0.8, -0.5, 1.0], [1.0, 2.0, 3.0, 2.5]) =
= 0.2*1.0 + 0.8*2.0 + -0.5*3.0 + 1.0*2.5 = 2.8
```

<https://nnfs.io>

**Fig 2.07:** Visualizing the math of the dot product of inputs and weights for a single neuron.

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

outputs = np.dot(weights, inputs) + bias
>>> 4.8
```

<https://nnfs.io>

$\text{np.dot}(\text{weights}, \text{inputs}) + \text{bias} = 2.8 + 2.0 = 4.8$

**Fig 2.08:** Visualizing the math summing the dot product and bias.



**Anim 2.07-2.08:** <https://nnfs.io/blq>

## A Layer of Neurons with NumPy

Now we're back to the point where we'd like to calculate the output of a layer of 3 neurons, which means the weights will be a matrix or list of weight vectors. 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. Previously with the 3-neuron example, we performed a multiplication of those weights with a list containing inputs, which resulted in a list of output values — one per neuron.

We also described the dot product of two vectors, but the weights are now a matrix, and we need to perform a dot product of them and the input vector. NumPy makes this very easy for us — treating this matrix as a list of vectors and performing the dot product one by one with the vector of inputs, returning a list of dot products.

The dot product's result, in our case, is a vector (or a list) of sums of the weight and input products for each of the neurons. From here, we still need to add corresponding biases to them. The biases can be easily added to the result of the dot product operation as they are a vector of the same size. We can also use the plain Python list directly here, as NumPy will convert it to an array internally.

Previously, we had calculated outputs of each neuron by performing a dot product and adding a bias, one by one. Now we have changed the order of those operations — we're performing dot product first as one operation on all neurons and inputs, and then we are adding a bias in the next operation. When we add two vectors using NumPy, each  $i$ -th element is added together, resulting in a new vector of the same size. This is both a simplification and an optimization, giving us simpler and faster code.

```
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]

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

print(layer_outputs)

>>>
array([4.8  1.21 2.385])
```

```
inputs = [1.0, 2.0, 3.0, 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 = [2.0, 3.0, 0.5]

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

np.dot(weights, inputs) = [np.dot(weights[0], inputs),
np.dot(weights[1], inputs), np.dot(weights[2], inputs)]
= [2.8, -1.79, 1.885]
```

<https://nnfs.io>

**Fig 2.09:** Code and visuals for the dot product applied to the layer of neurons.

```
inputs = [1.0, 2.0, 3.0, 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 = [2.0, 3.0, 0.5]

layer_outputs = np.dot(weights, inputs) + biases
>>> array([4.8  1.21 2.385])
```

<https://nnfs.io>

**Fig 2.10:** Code and visuals for the sum of the dot product and bias with a layer of neurons.



**Anim 2.09-2.10:** <https://nnfs.io/cyx>

This syntax involving the dot product of weights and inputs followed by the vector addition of bias is the most commonly used way to represent this calculation of  $inputs \cdot weights + bias$ . To explain the order of parameters we are passing into `np.dot()`, we should think of it as whatever comes first will decide the output shape. In our case, we are passing a list of neuron weights first and then the inputs, as our goal is to get a list of neuron outputs. As we mentioned, a dot product of a matrix and a vector results in a list of dot products. The `np.dot` method treats the matrix as a list of vectors and performs a dot product of each of those vectors with the other vector. In this example, we used that property to pass a matrix, which was a list of neuron weight vectors and a vector of inputs, and get a list of dot products — neuron outputs.




## A Batch of Data

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, from the example in chapter 1, 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**.

|   |                               |
|---|-------------------------------|
| <i>Input data :</i>   | <i>Shape :</i>                |
| sample = <code>[1, 5, 6, 2]</code>  | <code>(4,)</code>             |
| <br> | <i>Type :</i>                 |
|   | <code>1D array, Vector</code> |

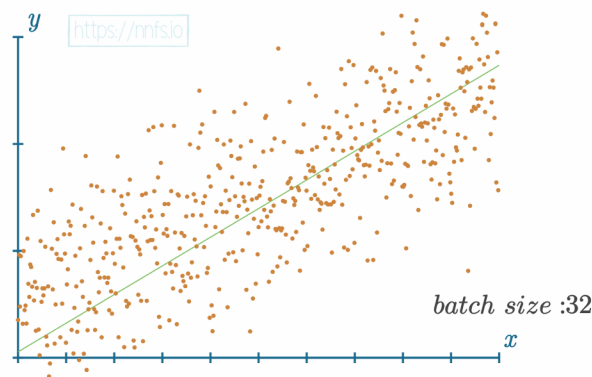
**Fig 2.11:** Visualizing a 1D array.



**Anim 2.11:** <https://nnfs.io/lqw>

Often, neural networks expect to take in many **samples** at a time for two reasons. One reason is that it's faster to train in batches in parallel processing, and the other reason is that batches

help with generalization during training. If you fit (perform a step of a training process) on one sample at a time, you're highly likely to keep fitting to that individual sample, rather than slowly producing general tweaks to weights and biases that fit the entire dataset. Fitting or training in batches gives you a higher chance of making more meaningful changes to weights and biases. For the concept of fitment in batches rather than one sample at a time, the following animation can help:



**Fig 2.12:** Example of a linear equation fitting batches of 32 chosen samples. See animation below for other sizes of samples at a time to see how much of a difference batch size can make.



**Anim 2.12:** <https://nnfs.io/vyu>

An example of a batch of data could look like:

|  |                         |
|--|-------------------------|
| <i>Input data :</i>  | <i>Shape :</i>          |
| <pre>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]]</pre> | <pre>(8, 4)</pre>       |
|  | <i>Type :</i>           |
|  | <i>2D Array, Matrix</i> |

<https://nnfs.io>

**Fig 2.13:** Example of a batch, its shape, and type.



Anim 2.13: <https://nnfs.io/lqw>

Recall that in Python, and in our case, lists are useful containers for holding a sample as well as multiple samples that make up a batch of observations. Such an example of a batch of observations, each with its own sample, looks like:

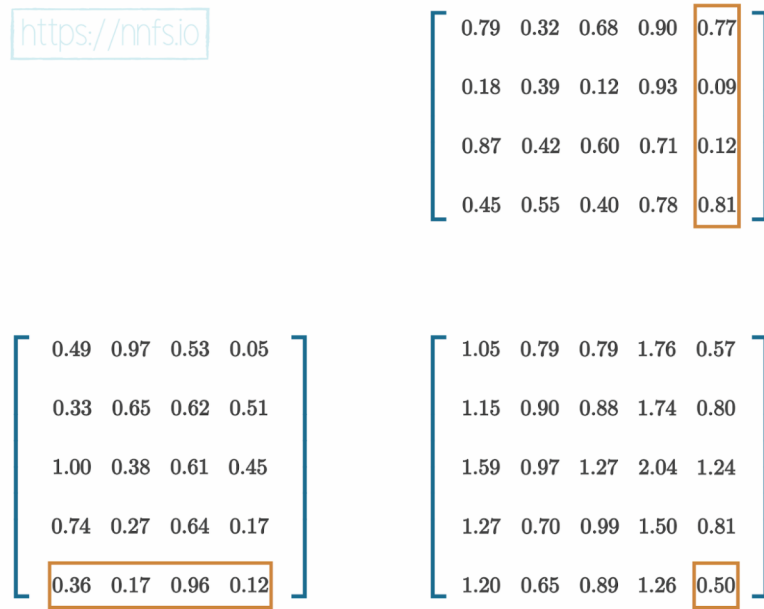
```
inputs = [[1, 2, 3, 2.5], [2, 5, -1, 2], [-1.5, 2.7, 3.3, -0.8]]
```

This list of lists could be made into an array since it is homologous. Note that each “list” in this larger list is a sample representing a feature set. `[1, 2, 3, 2.5]`, `[2, 5, -1, 2]`, and `[-1.5, 2.7, 3.3, -0.8]` are all **samples**, and are also referred to as **feature set instances** or **observations**.

We have a matrix of inputs and a matrix of weights now, and we need to perform the dot product on them somehow, but how and what will the result be? Similarly, as we performed a dot product on a matrix and a vector, we treated the matrix as a list of vectors, resulting in a list of dot products. In this example, we need to manage both matrices as lists of vectors and perform dot products on all of them in all combinations, resulting in a list of lists of outputs, or a matrix; this operation is called the **matrix product**.

## Matrix Product

The **matrix product** is an operation in which we have 2 matrices, and we are performing dot products of all combinations of rows from the first matrix and the columns of the 2nd matrix, resulting in a matrix of those atomic **dot products**:



**Fig 2.14:** Visualizing how a single element in the resulting matrix from matrix product is calculated. See animation for the full calculation of each element.



**Anim 2.14:** <https://nnfs.io/jei>

To perform a matrix product, the size of the second dimension of the left matrix must match the size of the first dimension of the right matrix. For example, if the left matrix has a shape of  $(5, 4)$  then the right matrix must match this 4 within the first shape value  $(4, 7)$ . The shape of the resulting array is always the first dimension of the left array and the second dimension of the right array,  $(5, 7)$ . In the above example, the left matrix has a shape of  $(5, 4)$ , and the upper-right matrix has a shape of  $(4, 5)$ . The second dimension of the left array and the first dimension of the second array are both 4, they match, and the resulting array has a shape of  $(5, 5)$ .

To elaborate, we can also show that we can perform the matrix product on vectors. In mathematics, we can have something called a column vector and row vector, which we'll explain better shortly. They're vectors, but represented as matrices with one of the dimensions having a size of 1:

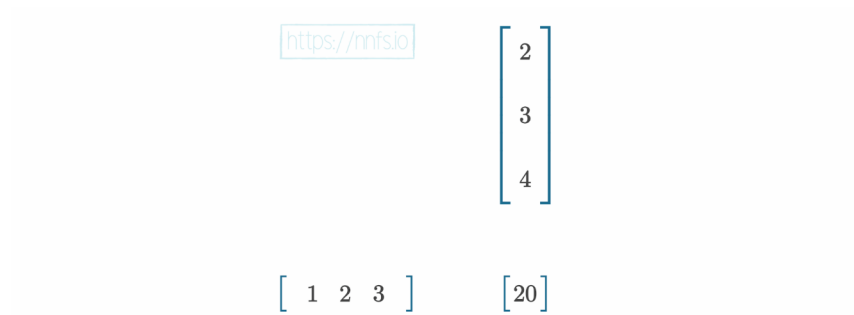
$$a = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$b = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

$a$  is a row vector. It looks very similar to a vector  $\vec{a}$  (with an arrow above it) described earlier along with the vector product. The difference in notation between a row vector and vector are commas between values and the arrow above symbol  $a$  is missing on a row vector. It's called a row vector as it's a vector of a row of a matrix.  $b$ , on the other hand, is called a column vector because it's a column of a matrix. As row and column vectors are technically matrices, we do not denote them with vector arrows anymore.

When we perform the matrix product on them, the result becomes a matrix as well, like in the previous example, but containing just a single value, the same value as in the dot product example we have discussed previously:

$$ab = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 20 \end{bmatrix}$$


$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = 20$$

**Fig 2.15:** Product of row and column vectors.



**Anim 2.15:** <https://nnfs.io/bkw>

In other words, row and column vectors are matrices with one of their dimensions being of a size of 1; and, we perform the **matrix product** on them instead of the **dot product**, which results in a matrix containing a single value. In this case, we performed a matrix multiplication of matrices with shapes  $(1, 3)$  and  $(3, 1)$ , then the resulting array has the shape  $(1, 1)$  or a size of  $1 \times 1$ .

## Transposition for the Matrix Product

How did we suddenly go from 2 vectors to row and column vectors? We used the relation of the dot product and matrix product saying that a dot product of two vectors equals a matrix product of a row and column vector (the arrows above the letters signify that they are vectors):

$$\vec{a} \cdot \vec{b} = ab^T$$

We also have temporarily used some simplification, not showing that column vector  $b$  is actually a **transposed** vector  $b$ . The proper equation, matching the dot product of vectors  $a$  and  $b$  written as matrix product should look like:

$$ab^T = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 20 \end{bmatrix}$$

Here we introduced one more new operation — **transposition**. Transposition simply modifies a matrix in a way that its rows become columns and columns become rows:

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 \\ 05 & 06 & 07 & 08 & 09 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \end{bmatrix}^T = \begin{bmatrix} 00 & 05 & 10 & 15 \\ 01 & 06 & 11 & 16 \\ 02 & 07 & 12 & 17 \\ 03 & 08 & 13 & 18 \\ 04 & 09 & 14 & 19 \end{bmatrix}$$

<https://nnfs.io>

**Fig 2.16:** Example of an array transposition.



Anim 2.16: <https://nnfs.io/qut>

$$\begin{bmatrix} 0.49 & 0.97 & 0.53 & 0.05 & 0.33 \\ 0.65 & 0.62 & 0.51 & 1.00 & 0.38 \\ 0.61 & 0.45 & 0.74 & 0.27 & 0.64 \\ 0.17 & 0.36 & 0.17 & 0.96 & 0.12 \\ 0.79 & 0.32 & 0.68 & 0.90 & 0.77 \end{bmatrix}^T = \begin{bmatrix} 0.49 & 0.65 & 0.61 & 0.17 & 0.79 \\ 0.97 & 0.62 & 0.45 & 0.36 & 0.32 \\ 0.53 & 0.51 & 0.74 & 0.17 & 0.68 \\ 0.05 & 1.00 & 0.27 & 0.96 & 0.90 \\ 0.33 & 0.38 & 0.64 & 0.12 & 0.77 \end{bmatrix}$$

<https://nnfs.io>

**Fig 2.17:** Another example of an array transposition.



**Anim 2.17:** <https://nnfs.io/pnq>

Now we need to get back to row and column vector definitions and update them with what we have just learned.

A row vector is a matrix whose first dimension's size (the number of rows) equals 1 and the second dimension's size (the number of columns) equals  $n$  — the vector size. In other words, it's a  $1 \times n$  array or array of shape (1,  $n$ ):

$$a = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_n \end{bmatrix}$$

With NumPy and with 3 values, we would define it as:

```
np.array([[1, 2, 3]])
```

Note the use of double brackets here. To transform a list into a matrix containing a single row (perform an equivalent operation of turning a vector into row vector), we can put it into a list and create numpy array:



```
a = [1, 2, 3]
np.array([a])

>>>
array([[1, 2, 3]])
```

Again, note that we encase `a` in brackets before converting to an array in this case.

Or we can turn it into a 1D array and expand dimensions using one of the NumPy abilities:

```
a = [1, 2, 3]
np.expand_dims(np.array(a), axis=0)

>>>
array([[1, 2, 3]])
```

Where `np.expand_dims()` adds a new dimension at the index of the `axis`.

A column vector is a matrix where the second dimension's size equals 1, in other words, it's an array of shape  $(n, 1)$ :

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}$$

With NumPy it can be created the same way as a row vector, but needs to be additionally transposed — transposition turns rows into columns and columns into rows:

$$\begin{bmatrix} b_1 & b_2 & b_3 & \dots & b_n \end{bmatrix}^T = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}$$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}^T = \begin{bmatrix} b_1 & b_2 & b_3 & \dots & b_n \end{bmatrix}$$

To turn vector  $b$  into row vector  $b$ , we'll use the same method that we used to turn vector  $a$  into row vector  $a$ , then we can perform a transposition on it to make it a column vector  $b$ :

$$b = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix}$$

$$b^T = \begin{bmatrix} 2 & 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

With NumPy code:

```
import numpy as np

a = [1, 2, 3]
b = [2, 3, 4]

a = np.array([a])
b = np.array([b]).T

np.dot(a, b)

>>>
array([[20]])
```

We have achieved the same result as the dot product of two vectors, but performed on matrices and returning a matrix — exactly what we expected and wanted. It's worth mentioning that NumPy does not have a dedicated method for performing matrix product — the dot product and matrix product are both implemented in a single method: `np.dot`.

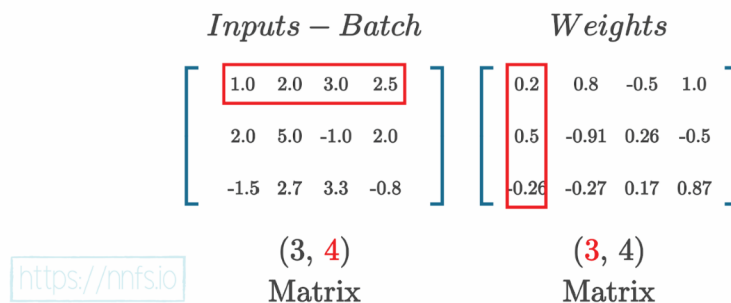
As we can see, to perform a matrix product on two vectors, we took one as is, transforming it into a row vector, and the second one using transposition on it to turn it into a column vector. That allowed us to perform a matrix product that returned a matrix containing a single value. We also performed the matrix product on two example arrays to learn how a matrix product works — it creates a matrix of dot products of all combinations of row and column vectors.

## A Layer of Neurons & Batch of Data w/ NumPy

Let's get back to our inputs and weights — when covering them, we mentioned that we need to perform dot products on all of the vectors that consist of both input and weight matrices. As we have just learned, that's the operation that the matrix product performs. We just need to perform transposition on its second argument, which is the weights matrix in our case, to turn the row vectors it currently consists of into column vectors.

Initially, we were able to perform the dot product on the inputs and the weights without a transposition because the weights were a matrix, but the inputs were just a vector. In this case, the dot product results in a vector of atomic dot products performed on each row from the matrix and this single vector. When inputs become a batch of inputs (a matrix), we need to perform the matrix product. It takes all of the combinations of rows from the left matrix and columns from the right matrix, performing the dot product on them and placing the results in an output array. Both arrays have the same shape, but, to perform the matrix product, the shape's value from the index 1 of the first matrix and the index 0 of the second matrix must match — they don't right now.

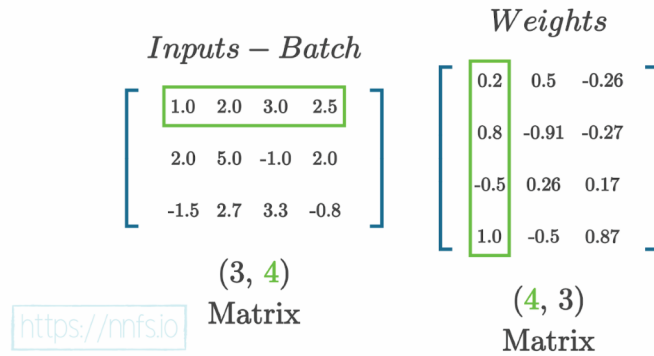
```
inputs = [[1.0, 2.0, 3.0, 2.5],
          [2.0, 5.0, -1.0, 2.0],
          [-1.5, 2.7, 3.3, -0.8]]
weights = [[0.2, 0.8, -0.5, 1.0],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
```



**Fig 2.18:** Depiction of why we need to transpose to perform the matrix product.

If we transpose the second array, values of its shape swap their positions.

```
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],
            [0.5, -0.91, 0.26, -0.5],
            [-0.26, -0.27, 0.17, 0.87]]
```



**Fig 2.19:** After transposition, we can perform the matrix product.



**Anim 2.18-2.19:** <https://nnfs.io/crq>

If we look at this from the perspective of the input and weights, we need to perform the dot product of each input and each weight set in all of their combinations. The dot product takes the row from the first array and the column from the second one, but currently the data in both arrays are row-aligned. Transposing the second array shapes the data to be column-aligned. The matrix product of inputs and transposed weights will result in a matrix containing all atomic dot products that we need to calculate. The resulting matrix consists of outputs of all neurons after operations performed on each input sample:

```

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],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

outputs = np.dot(inputs, np.array(weights).T) + biases

np.dot(inputs, np.array(weights).T)

```

<https://nnfs.io>

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 2.5 \\ 2.0 & 5.0 & -1.0 & 2.0 \\ -1.5 & 2.7 & 3.3 & -0.8 \end{bmatrix} \begin{bmatrix} 0.2 & 0.5 & -0.26 \\ 0.8 & -0.91 & -0.27 \\ -0.5 & 0.26 & 0.17 \\ 1.0 & -0.5 & 0.87 \end{bmatrix} = \begin{bmatrix} 2.8 & -1.79 & 1.885 \\ 6.9 & -4.81 & -0.3 \\ -0.59 & -1.949 & -0.474 \end{bmatrix}$$

**Fig 2.20:** Code and visuals depicting the dot product of inputs and transposed weights.



**Anim 2.20:** <https://nnfs.io/gjw>

We mentioned that the second argument for `np.dot()` is going to be our transposed weights, so first will be inputs, but previously weights were the first parameter. We changed that here. Before, we were modeling neuron output using a single sample of data, a vector, but now we are a step forward when we model layer behavior on a batch of data. We could retain the current parameter order, but, as we'll soon learn, it's more useful to have a result consisting of a list of layer outputs per each sample than a list of neurons and their outputs sample-wise. We want the resulting array to be sample-related and not neuron-related as we'll pass those samples further through the network, and the next layer will expect a batch of inputs.

We can code this solution using NumPy now. We can perform `np.dot()` on a plain Python list of lists as NumPy will convert them to matrices internally. We are converting weights ourselves though to perform transposition operation first, `T` in the code, as plain Python list of lists does not support it. Speaking of biases, we do not need to make it a NumPy array for the same reason — NumPy is going to do that internally.

Biases are a list, though, so they are a 1D array as a NumPy array. The addition of this bias vector to a matrix (of the dot products in this case) works similarly to the dot product of a matrix and vector that we described earlier; The bias vector will be added to each row vector of the matrix. Since each column of the matrix product result is an output of one neuron, and the vector is going to be added to each row vector, the first bias is going to be added to each first element of those vectors, second to second, etc. That's what we need — the bias of each neuron needs to be added to all of the results of this neuron performed on all input vectors (samples).

```
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],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

outputs = np.dot(inputs, np.array(weights).T) + biases
>>> array([[ 4.8   1.21  2.385],
          [ 8.9  -1.81  0.2 ],
          [ 1.41  1.051  0.026]])
```

<https://nnfs.io>

$$\begin{bmatrix} 2.8 & -1.79 & 1.885 \\ 6.9 & -4.81 & -0.3 \\ -0.59 & -1.949 & -0.474 \end{bmatrix} + \begin{bmatrix} 2.0 & 3.0 & 0.5 \end{bmatrix} = \begin{bmatrix} 4.8 & 1.21 & 2.385 \\ 8.9 & -1.81 & 0.2 \\ 1.41 & 1.051 & 0.026 \end{bmatrix}$$

**Fig 2.21:** Code and visuals for inputs multiplied by the weights, plus the bias.



**Anim 2.21:** <https://nnfs.io/qty>

Now we can implement what we have learned into code:

```
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],
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
biases = [2.0, 3.0, 0.5]

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

print(layer_outputs)

>>>
array([[ 4.8    1.21   2.385],
       [ 8.9   -1.81   0.2  ],
       [ 1.41   1.051  0.026]])
```

As you can see, our neural network takes in a group of samples (inputs) and outputs a group of predictions. If you've used any of the deep learning libraries, this is why you pass in a list of inputs (even if it's just one feature set) and are returned a list of predictions, even if there's only one prediction.



**Supplementary Material:** <https://nnfs.io/ch2>

Chapter code, further resources, and errata for this chapter.