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:

Neural Networks from Scratch in Python

Harrison Kinsley & Daniel Kukieła

Copyright

License for Code

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://pythonprogrammingnet/python-fundamental-tutorials/* To cite this material:

Harrison Kinsley & Daniel Kukieła Neural Networks from Scratch (NNFS) https://nnfs.io

Chapter 3

Adding Layers

The neural network we've built is becoming more respectable, but at the moment, we have only one layer. Neural networks become "deep" when they have 2 or more **hidden layers**. At the moment, we have just one layer, which is effectively an output layer. Why we want two or more **hidden** layers will become apparent in a later chapter. Currently, we have no hidden layers. A hidden layer isn't an input or output layer; as the scientist, you see data as they are handed to the input layer and the resulting data from the output layer. Layers between these endpoints have values that we don't necessarily deal with, hence the name "hidden." Don't let this name convince you that you can't access these values, though. You will often use them to diagnose issues or improve your neural network. To explore this concept, let's add another layer to this neural network, and, for now, let's assume these two layers that we're going to have will be the hidden layers, and we just have not coded our output layer yet.

Before we add another layer, let's think about what will be coming. In the case of the first layer, we can see that we have an input with 4 features.

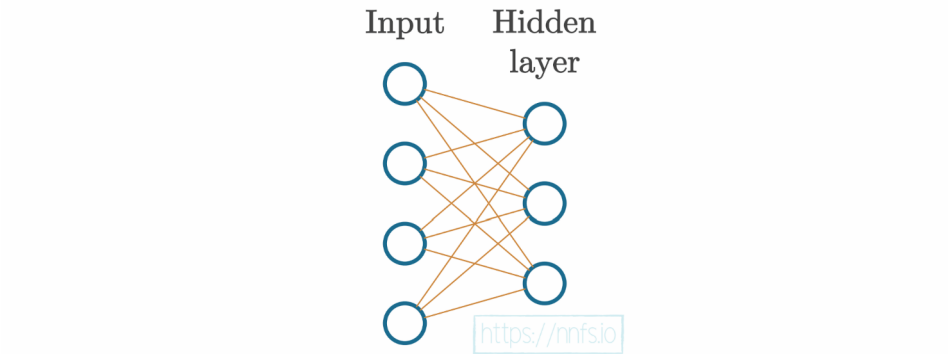

Fig 3.01: Input layer with 4 features into a hidden layer with 3 neurons.

Samples (feature set data) get fed through the input, which does not change it in any way, to our first hidden layer, which we can see has 3 sets of weights, with 4 values each.

Each of those 3 unique weight sets is associated with its distinct neuron. Thus, since we have 3 weight sets, we have 3 neurons in this first hidden layer. Each neuron has a unique set of weights, of which we have 4 (as there are 4 inputs to this layer), which is why our initial weights have a shape of *(3,4)*.

Now, we wish to add another layer. To do that, we must make sure that the expected input to that layer matches the previous layer's output. We have set the number of neurons in a layer by setting how many weight sets and biases we have. The previous layer's influence on weight sets for the current layer is that each weight set needs to have a separate weight per input. This means a distinct weight per neuron from the previous layer (or feature if we're talking the input). The previous layer has 3 weight sets and 3 biases, so we know it has 3 neurons. This then means, for the next layer, we can have as many weight sets as we want (because this is how many neurons this new layer will have), but each of those weight sets must have 3 discrete weights.

To create this new layer, we are going to copy and paste our `weights` and `biases` to `weights2` and `biases2`, and change their values to new made up sets. Here's an example:

```
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]
```

Next, we will now call *outputs* "*layer1_outputs*":

```
layer1_outputs = np.dot(inputs, np.array(weights).T) + biases
```

As previously stated, inputs to layers are either inputs from the actual dataset you're training with or outputs from a previous layer. That's why we defined 2 versions of *weights* and *biases* but only 1 of *inputs* — because the inputs for layer 2 will be the outputs from the previous layer:

```
layer2_outputs = np.dot(layer1_outputs, np.array(weights2).T) + biases2
```

All together now:

```
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]

layer1_outputs = np.dot(inputs, np.array(weights).T) + biases
layer2_outputs = np.dot(layer1_outputs, np.array(weights2).T) + biases2

print(layer2_outputs)


>>>
array([[ 0.5031  -1.04185 -2.03875],
       [ 0.2434  -2.7332  -5.7633 ],
       [-0.99314  1.41254 -0.35655]])
```

At this point, our neural network could be visually represented as:

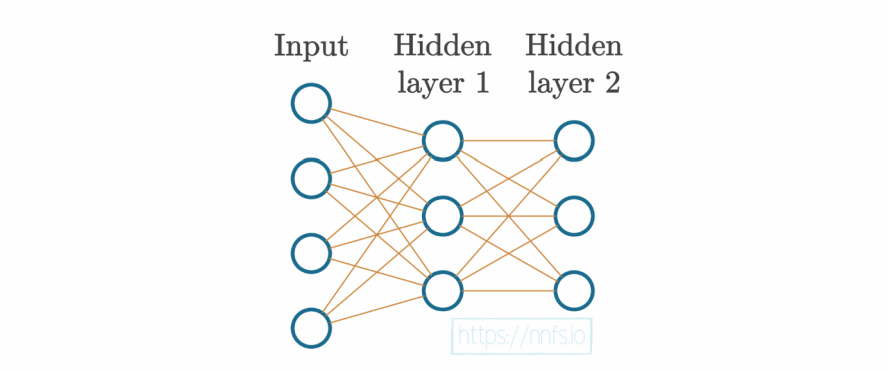

Fig 3.02: 4 features input into 2 hidden layers of 3 neurons each.

Training Data

Next, rather than hand-typing in random data, we'll use a function that can create non-linear data. What do we mean by non-linear? Linear data can be fit with or represented by a straight line.

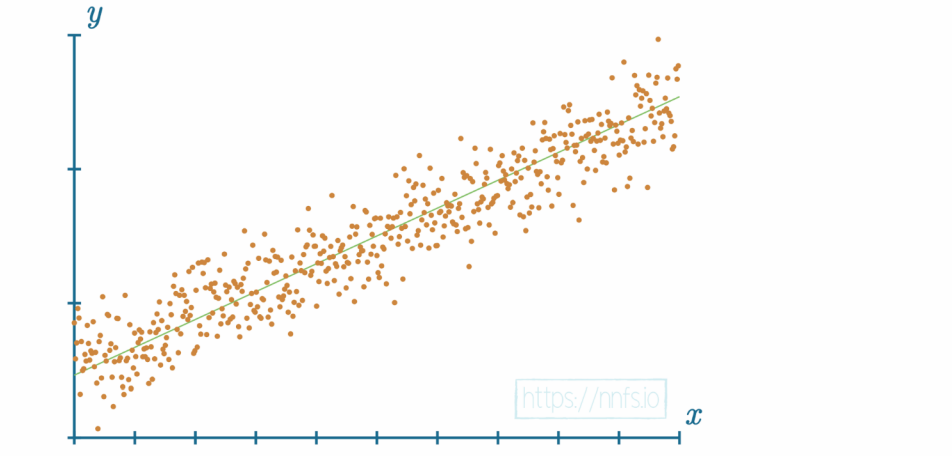

Fig 3.03: Example of data (orange dots) that can be represented (fit) by a straight line (green line).

Non-linear data cannot be represented well by a straight line.

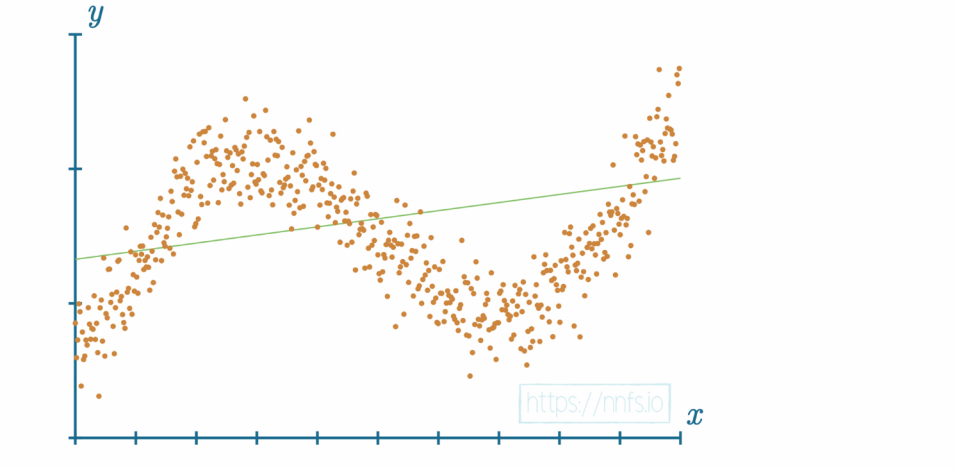

Fig 3.04: Example of data (orange dots) that is not well fit by a straight line.

If you were to graph data points of the form *(x, y)* where *y = f(x)*, and it looks to be a line with a clear trend or slope, then chances are, they're linear data! Linear data are very easily approximated by far simpler machine learning models than neural networks. What other machine learning algorithms cannot approximate so easily are non-linear datasets. To simplify this, we've created a Python package that you can install with pip, called *nnfs*:

```
pip install nnfs
```

The nnfs package contains functions that we can use to create data. For example:

```
from nnfs.datasets import spiral_data
```

The `spiral_data` function was slightly modified from https://cs231n.github.io/neural-networks-case-study/, which is a great supplementary resource for this topic.

You will typically not be generating training data from a function for your neural networks. You will have an actual dataset. Generating a dataset this way is purely for convenience at this stage. We will also use this package to ensure repeatability for everyone, using `nnfs.init()`, after importing NumPy:

```
import numpy as np
import nnfs

nnfs.init()
```

The `nnfs.init()` does three things: it sets the random seed to 0 (by default), creates a *float32* dtype default, and overrides the original dot product from NumPy. All of these are meant to ensure repeatable results for following along.

The `spiral_data` function allows us to create a dataset with as many classes as we want. The function has parameters to choose the number of classes and the number of points/observations per class in the resulting non-linear dataset. For example:

```
import matplotlib.pyplot as plt

X, y = spiral_data(samples=100, classes=3)

plt.scatter(X[:,0], X[:,1])
plt.show()
```

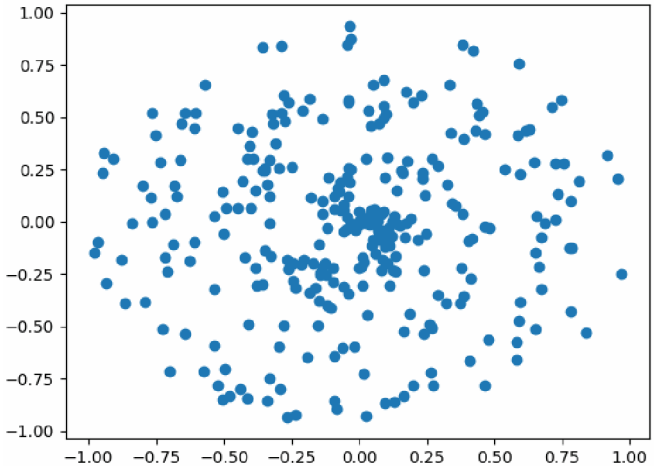

Fig 3.05: Uncolored spiral dataset.

If you trace from the center, you can determine all 3 classes separately, but this is a very challenging problem for a machine learning classifier to solve. Adding color to the chart makes this more clear:

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

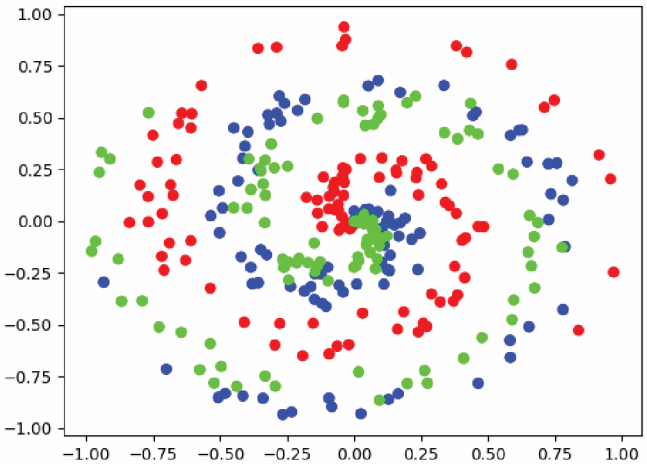

Fig 3.06: Spiral dataset colored by class.

Keep in mind that the neural network will not be aware of the color differences as the data have no class encodings. This is only made as an instruction for the reader. In the data above, each of the dot's coordinate values represents a feature, so each dot is a representation of the featureset. The “classification” for that dot has to do with which spiral it is a part of, depicted by blue, green, or red color in the previous image. These colors would then be assigned a class number for the model to fit to, like 0, 1, and 2.

Dense Layer Class

Now that we no longer need to hand-type our data, we should create something similar for our various types of neural network layers. So far, we've only used what's called a **dense** or **fully-connected** layer. These layers are commonly referred to as "dense" layers in papers, literature, and code, but you will occasionally see them called fully-connected or "fc" for short in code. Our dense layer class will begin with two methods.

```
# Dense layer
class Layer_Dense:

    # Layer initialization
    def __init__(self, n_inputs, n_neurons):
        # Initialize weights and biases
        pass  # using pass statement as a placeholder

    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs, weights and biases
        pass  # using pass statement as a placeholder
```

As previously stated, weights are often initialized randomly for a model, but not always. If you wish to load a pre-trained model, you will initialize the parameters to whatever that pretrained model finished with. It's also possible that, even for a new model, you have some other initialization rules besides random. For now, we'll stick with random initialization. Next, we have the *forward* method. When we pass data through a model from beginning to end, this is called a **forward pass**. Just like everything else, however, this is not the only way to do things. You can have the data loop back around and do other interesting things. We'll keep it usual and perform a regular forward pass.

To continue the `Layer_Dense` class' code let's add the random initialization of weights and biases:

```
# Layer initialization
def __init__(self, n_inputs, n_neurons):
    self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
    self.biases = np.zeros((1, n_neurons))
```

Here, we're setting weights to be random and biases to be 0. Note that we're initializing weights to be *(inputs, neurons),* rather than (*neurons, inputs)*. We're doing this ahead instead of transposing every time we perform a forward pass, as explained in the previous chapter. Why zero biases? In specific scenarios, like with many samples containing values of 0, a bias can ensure that a neuron fires initially. It sometimes may be appropriate to initialize the biases to some non-zero number, but the most common initialization for biases is 0. However, in these scenarios, you may find success in doing things another way. This will vary depending on your use-case and is just one of many things you can tweak when trying to improve results. One situation where you might want to try something else is with what's called **dead neurons**. We haven't yet covered activation functions in practice, but imagine our step function again.

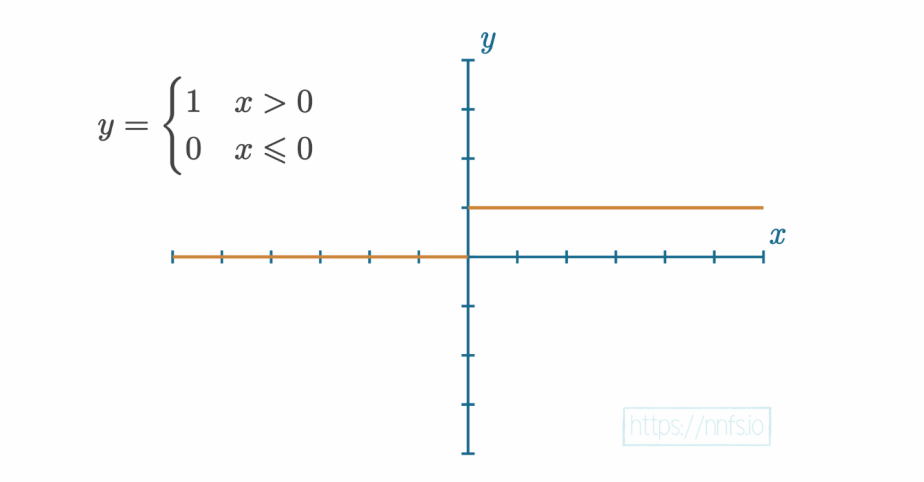

Fig 3.07: Graph of a step function.

It's possible for *weights · inputs + biases* not to meet the threshold of the step function, which means the neuron will output a 0. Alone, this is not a big issue, but it becomes a problem if this happens to this neuron for every one of the input samples (it'll become clear why once we cover backpropagation). So then this neuron's 0 output is the input to another neuron. Any weight multiplied by zero will be zero. With an increasing number of neurons outputting 0, more inputs to the next neurons will receive these 0s rendering the network essentially non-trainable, or "dead."

Next, let's explore `np.random.randn()` and `np.zeros()` in more detail. These methods are convenient ways to initialize arrays. `np.random.randn()` produces a Gaussian distribution with a mean of 0 and a variance of 1, which means that it'll generate random numbers, positive and negative, centered at 0 and with the mean value close to 0. In general, neural networks work best with values between -1 and +1, which we'll discuss in an upcoming chapter. So this `np.random.randn()` generates values around those numbers. We're going to multiply this Gaussian distribution for the weights by *0.01* to generate numbers that are a couple of magnitudes smaller. Otherwise, the model will take more time to fit the data during the training process as starting values will be disproportionately large compared to the updates being made

during training. The idea here is to start a model with non-zero values small enough that they won't affect training. This way, we have a bunch of values to begin working with, but hopefully none too large or as zeros. You can experiment with values other than *0.01* if you like.

Finally, the `np.random.randn` function takes dimension sizes as parameters and creates the output array with this shape. The weights here will be the number of inputs for the first dimension and the number of neurons for the 2nd dimension. This is similar to our previous made up array of weights, just randomly generated. Whenever there's a function or block of code that you're not sure about, you can always print it out. For example:

```
import numpy as np
import nnfs

nnfs.init()

print(np.random.randn(2,5))
```

```
>>>
[[ 1.7640524   0.4001572   0.978738    2.2408931   1.867558  ]
 [-0.9772779   0.95008844 -0.1513572  -0.10321885  0.41059852]]
```

The example function call has returned a 2x5 array (which we can also say is "*with a shape of (2,5)*") with data randomly sampled from a Gaussian distribution with a mean of 0.

Next, the `np.zeros` function takes a desired array shape as an argument and returns an array of that shape filled with zeros.

```
print(np.zeros((2,5)))
```

```
>>>
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

We'll initialize the biases with the shape of *(1, n_neurons)*, as a row vector, which will let us easily add it to the result of the dot product later, without additional operations like transposition.

To see an example of how our method initializes weights and biases:

```
import numpy as np
import nnfs

nnfs.init()

n_inputs = 2
n_neurons = 4

weights = 0.01 * np.random.randn(n_inputs, n_neurons)
biases = np.zeros((1, n_neurons))

print(weights)
print(biases)


>>>
[[ 0.01764052  0.00400157  0.00978738  0.02240893]
 [ 0.01867558 -0.00977278  0.00950088 -0.00151357]]
[[0. 0. 0. 0.]]
```

On to our forward method — we need to update it with the dot product+biases calculation:

```
def forward(self, inputs):
    self.output = np.dot(inputs, self.weights) + self.biases
```

Nothing new here, just turning the previous code into a method. Our full `Layer_Dense` class so far:

```
class Layer_Dense:

    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))

    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) + self.biases
```

We're ready to make use of this new class instead of hardcoded calculations, so let's generate some data using the discussed dataset creation method and use our new layer to perform a forward pass:

```
# 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])
```

Go ahead and run everything.

Full code up to this point:

```
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]]
```

In the output, you can see we have 5 rows of data that have 3 values each. Each of those 3 values is the value from the 3 neurons in the `dense1` layer after passing in each of the samples. Great! We have a network of neurons, so our neural network model is almost deserving of its name, but we're still missing the activation functions, so let's do those next!

Supplementary Material: https://nnfs.io/ch3
Chapter code, further resources, and errata for this chapter.

Chapter 4 - Activation Functions