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 6

Introducing Optimization

Now that the neural network is built, able to have data passed through it, and capable of calculating loss, the next step is to determine how to adjust the weights and biases to decrease the loss. Finding an intelligent way to adjust the neurons' input's weights and biases to minimize loss is the main difficulty of neural networks.

The first option one might think of is randomly changing the weights, checking the loss, and repeating this until happy with the lowest loss found. To see this in action, we'll use a simpler dataset than we've been working with so far:

```
import matplotlib.pyplot as plt

import nnfs
from nnfs.datasets import vertical_data

nnfs.init()
```

```
X, y = vertical_data(samples=100, classes=3)

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap='brg')
plt.show()
```

Which looks like:

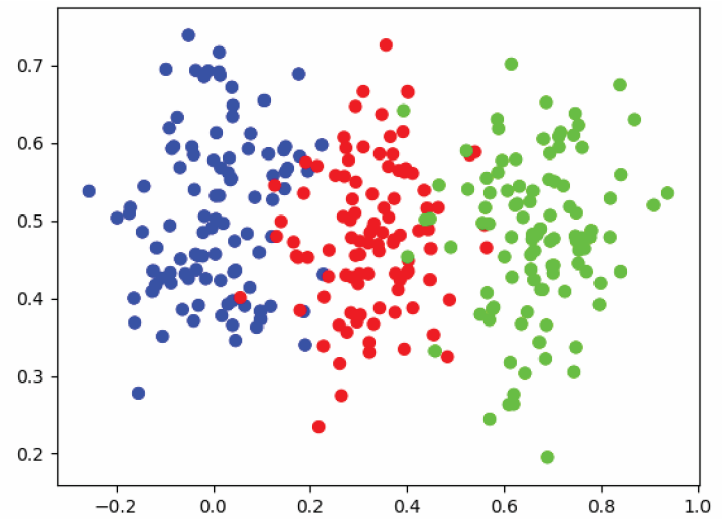

Fig 6.01: "Vertical data" graphed.

Using the previously created code up to this point, we can use this new dataset with a simple neural network:

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

# Create model
dense1 = Layer_Dense(2, 3)  # first dense layer, 2 inputs
activation1 = Activation_ReLU()
dense2 = Layer_Dense(3, 3)  # second dense layer, 3 inputs, 3 outputs
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()
```

Then create some variables to track the best loss and the associated weights and biases:

```
# Helper variables
lowest_loss = 9999999  # some initial value
best_dense1_weights = dense1.weights.copy()
best_dense1_biases = dense1.biases.copy()
best_dense2_weights = dense2.weights.copy()
best_dense2_biases = dense2.biases.copy()
```

We initialized the loss to a large value and will decrease it when a new, lower, loss is found. We are also copying weights and biases (`copy()` ensures a full copy instead of a reference to the object). Now we iterate as many times as desired, pick random values for weights and biases, and save the weights and biases if they generate the lowest-seen loss:

```
for iteration in range(10000):

    # Generate a new set of weights for iteration
    dense1.weights = 0.05 * np.random.randn(2, 3)
    dense1.biases = 0.05 * np.random.randn(1, 3)
    dense2.weights = 0.05 * np.random.randn(3, 3)
    dense2.biases = 0.05 * np.random.randn(1, 3)

    # Perform a forward pass of the training data through this layer
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)

    # Perform a forward pass through loss function
    # it takes the output of second activation function and returns loss
    loss = loss_function.calculate(activation2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    predictions = np.argmax(activation2.output, axis=1)
    accuracy = np.mean(predictions==y)

    # If loss is smaller - print and save weights and biases aside
    if loss < lowest_loss:
        print('New set of weights found, iteration:', iteration,
              'loss:', loss, 'acc:', accuracy)
        best_dense1_weights = dense1.weights.copy()
        best_dense1_biases = dense1.biases.copy()
        best_dense2_weights = dense2.weights.copy()
        best_dense2_biases = dense2.biases.copy()
        lowest_loss = loss
```

```
>>>
New set of weights found, iteration: 0 loss: 1.0986564 acc:
0.3333333333333333
New set of weights found, iteration: 3 loss: 1.098138 acc:
0.3333333333333333
New set of weights found, iteration: 117 loss: 1.0980115 acc:
0.3333333333333333
New set of weights found, iteration: 124 loss: 1.0977516 acc: 0.6
New set of weights found, iteration: 165 loss: 1.097571 acc:
0.3333333333333333
New set of weights found, iteration: 552 loss: 1.0974693 acc: 0.34
New set of weights found, iteration: 778 loss: 1.0968257 acc:
0.3333333333333333
New set of weights found, iteration: 4307 loss: 1.0965533 acc:
0.3333333333333333
New set of weights found, iteration: 4615 loss: 1.0964499 acc:
0.3333333333333333
New set of weights found, iteration: 9450 loss: 1.0964295 acc:
0.3333333333333333
```

Loss certainly falls, though not by much. Accuracy did not improve, except for a singular situation where the model randomly found a set of weights yielding better accuracy. Still, with a fairly large loss, this state is not stable. Running an additional 90,000 iterations for 100,000 in total:

```
New set of weights found, iteration: 13361 loss: 1.0963014 acc:
0.3333333333333333
New set of weights found, iteration: 14001 loss: 1.0959858 acc:
0.3333333333333333
New set of weights found, iteration: 24598 loss: 1.0947444 acc:
0.3333333333333333
```

Loss continued to drop, but accuracy did not change. This doesn't appear to be a reliable method for minimizing loss. After running for 1 billion iterations, the following was the best (lowest loss) result:

```
New set of weights found, iteration: 229865000 loss: 1.0911305 acc:
0.3333333333333333
```

Even with this basic dataset, we see that randomly searching for weight and bias combinations will take far too long to be an acceptable method. Another idea might be, instead of setting parameters with randomly-chosen values each iteration, apply a fraction of these values to parameters. With this, weights will be updated from what currently yields us the lowest loss instead of aimlessly randomly. If the adjustment decreases loss, we will make it the new point to adjust from. If loss instead increases due to the adjustment, then we will revert to the previous point. Using similar code from earlier, we will first change from randomly selecting weights and biases to randomly *adjusting* them:

```
# Update weights with some small random values
dense1.weights += 0.05 * np.random.randn(2, 3)
dense1.biases += 0.05 * np.random.randn(1, 3)
dense2.weights += 0.05 * np.random.randn(3, 3)
dense2.biases += 0.05 * np.random.randn(1, 3)
```

Then we will change our ending `if` statement to be:

```
# If loss is smaller - print and save weights and biases aside
if loss < lowest_loss:
    print('New set of weights found, iteration:', iteration,
          'loss:', loss, 'acc:', accuracy)
    best_dense1_weights = dense1.weights.copy()
    best_dense1_biases = dense1.biases.copy()
    best_dense2_weights = dense2.weights.copy()
    best_dense2_biases = dense2.biases.copy()
    lowest_loss = loss
# Revert weights and biases
else:
    dense1.weights = best_dense1_weights.copy()
    dense1.biases = best_dense1_biases.copy()
    dense2.weights = best_dense2_weights.copy()
    dense2.biases = best_dense2_biases.copy()
```

Modified code up to this point:

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

# Create model
dense1 = Layer_Dense(2, 3)  # first dense layer, 2 inputs
activation1 = Activation_ReLU()
dense2 = Layer_Dense(3, 3)  # second dense layer, 3 inputs, 3 outputs
activation2 = Activation_Softmax()

# Create loss function
loss_function = Loss_CategoricalCrossentropy()

# Helper variables
lowest_loss = 9999999  # some initial value
best_dense1_weights = dense1.weights.copy()
best_dense1_biases = dense1.biases.copy()
best_dense2_weights = dense2.weights.copy()
best_dense2_biases = dense2.biases.copy()

for iteration in range(10000):

    # Update weights with some small random values
    dense1.weights += 0.05 * np.random.randn(2, 3)
    dense1.biases += 0.05 * np.random.randn(1, 3)
    dense2.weights += 0.05 * np.random.randn(3, 3)
    dense2.biases += 0.05 * np.random.randn(1, 3)

    # Perform a forward pass of our training data through this layer
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)

    # Perform a forward pass through loss function
    # it takes the output of second activation function and returns loss
    loss = loss_function.calculate(activation2.output, y)
    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
```

```
    predictions = np.argmax(activation2.output, axis=1)
    accuracy = np.mean(predictions==y)

    # If loss is smaller - print and save weights and biases aside
    if loss < lowest_loss:
        print('New set of weights found, iteration:', iteration,
              'loss:', loss, 'acc:', accuracy)
        best_dense1_weights = dense1.weights.copy()
        best_dense1_biases = dense1.biases.copy()
        best_dense2_weights = dense2.weights.copy()
        best_dense2_biases = dense2.biases.copy()
        lowest_loss = loss
    # Revert weights and biases
    else:
        dense1.weights = best_dense1_weights.copy()
        dense1.biases = best_dense1_biases.copy()
        dense2.weights = best_dense2_weights.copy()
        dense2.biases = best_dense2_biases.copy()
```

```
>>>
New set of weights found, iteration: 0 loss: 1.0987684 acc:
0.3333333333333333
...
New set of weights found, iteration: 29 loss: 1.0725244 acc:
0.5266666666666666
New set of weights found, iteration: 30 loss: 1.0724432 acc:
0.3466666666666667
...
New set of weights found, iteration: 48 loss: 1.0303522 acc:
0.6666666666666666
New set of weights found, iteration: 49 loss: 1.0292586 acc:
0.6666666666666666
...
New set of weights found, iteration: 97 loss: 0.9277446 acc:
0.7333333333333333
...
New set of weights found, iteration: 152 loss: 0.73390484 acc:
0.8433333333333334
New set of weights found, iteration: 156 loss: 0.7235515 acc: 0.87
New set of weights found, iteration: 160 loss: 0.7049076 acc:
0.9066666666666666
...
New set of weights found, iteration: 7446 loss: 0.17280102 acc:
0.9333333333333333
New set of weights found, iteration: 9397 loss: 0.17279711 acc: 0.93
```

Loss descended by a decent amount this time, and accuracy raised significantly. Applying a fraction of random values actually lead to a result that we could almost call a solution. If you try 100,000 iterations, you will not progress much further:

```
>>>
...
New set of weights found, iteration: 14206 loss: 0.1727932 acc:
0.9333333333333333
New set of weights found, iteration: 63704 loss: 0.17278232 acc:
0.9333333333333333
```

Let's try this with the previously-seen spiral dataset instead:

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

>>>
New set of weights found, iteration: 0 loss: 1.1008677 acc:
0.3333333333333333 ...
New set of weights found, iteration: 31 loss: 1.0982264 acc:
0.37333333333333335 ...
New set of weights found, iteration: 65 loss: 1.0954362 acc:
0.38333333333333336
New set of weights found, iteration: 67 loss: 1.093989 acc:
0.4166666666666667 ...
New set of weights found, iteration: 129 loss: 1.0874122 acc:
0.42333333333333334 ...
New set of weights found, iteration: 5415 loss: 1.0790575
acc: 0.39
```

This training session ended with almost no progress. Loss decreased slightly and accuracy is barely above the initial value. Later, we'll learn that the most probable reason for this is called a local minimum of loss. The data complexity is also not irrelevant here. It turns out hard problems are hard for a reason, and we need to approach this problem more intelligently.

Supplementary Material: https://nnfs.io/ch6
Chapter code, further resources, and errata for this chapter.