Artificial Neural Networks

Dr. Muhammad Aqib

University Institute of Information Technology

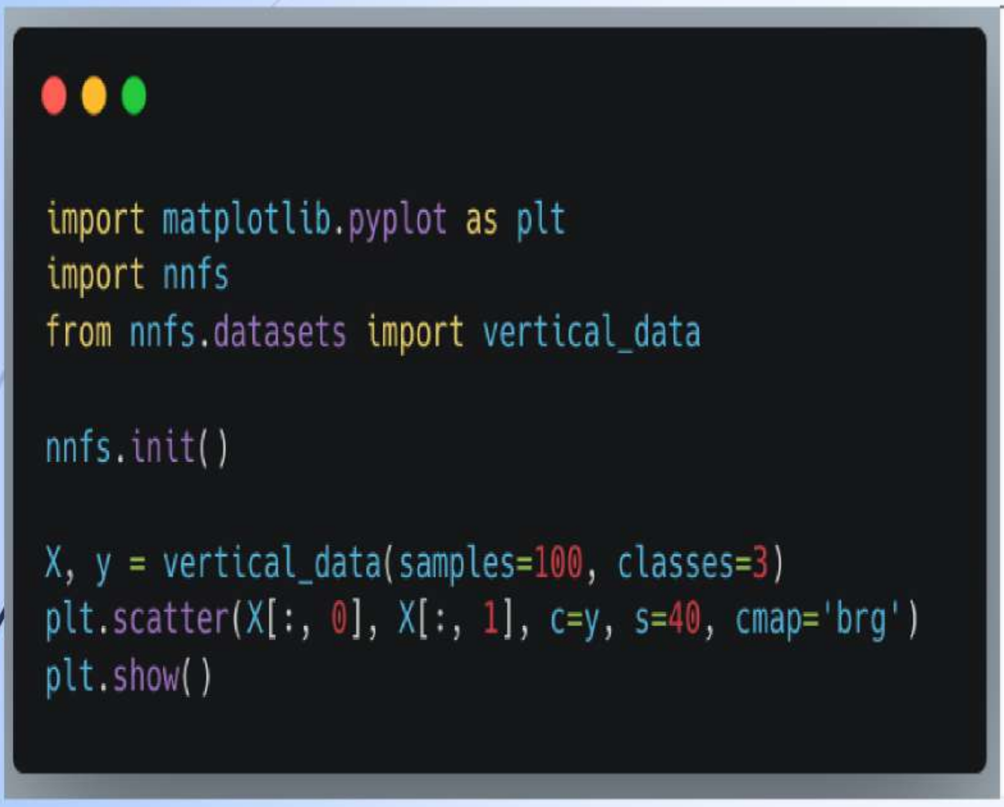
PMAS-Arid Agriculture University Rawalpindi

NN Optimization

Introduction

- 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.
- Let's see it through example by utilizing the nnfs package for creating dummy data and playing with that.
- A total of 100 random samples consisting of 3 classes are generated for experiment - shown next.

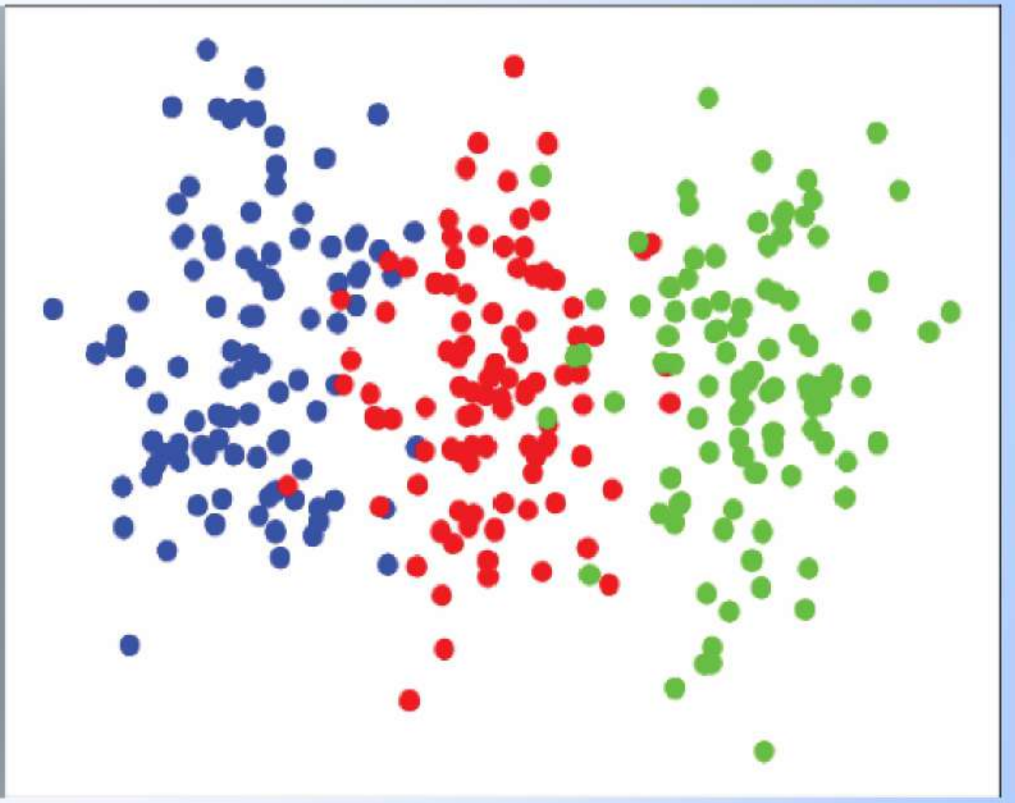
Experimentation



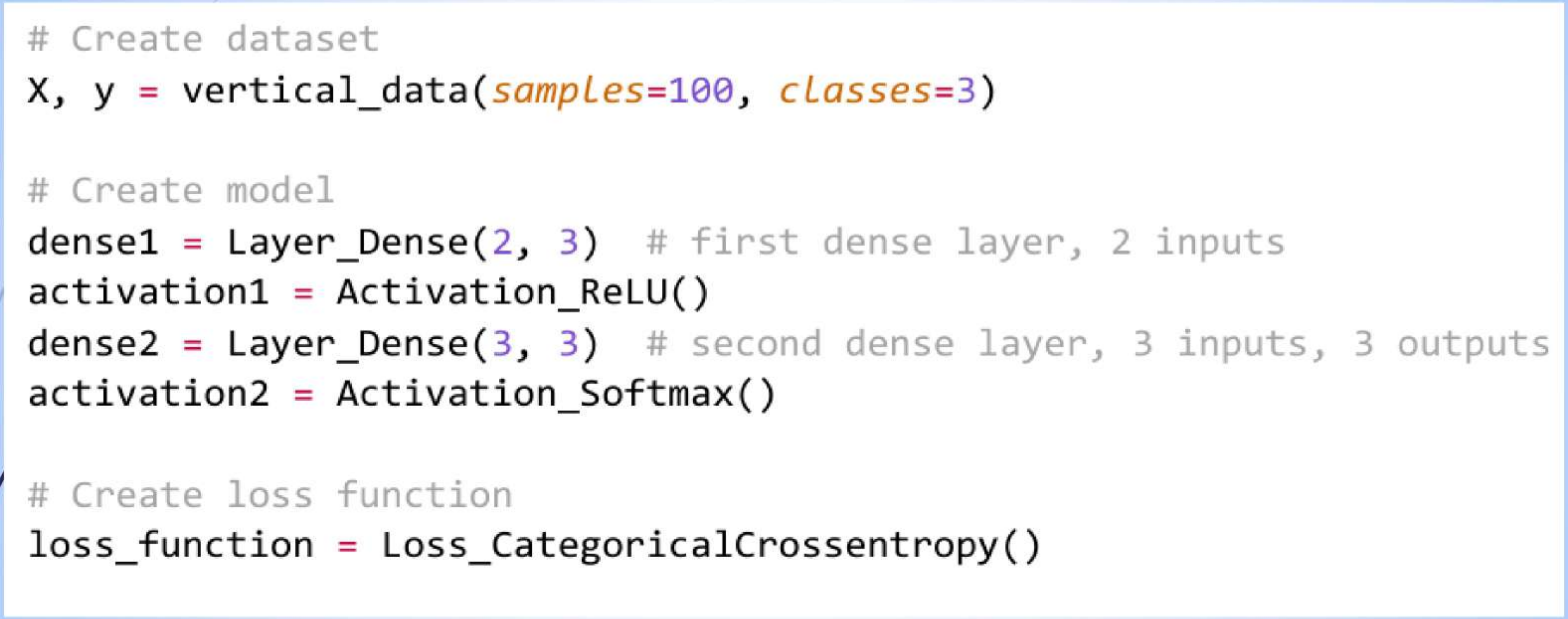
```
import matplotlib.pyplot as plt
import nnfs
from nnfs.datasets import vertical_data

nnfs.init()

X, y = vertical_data(samples=100, classes=3)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap='brg')
plt.show()
```



Experimentation



```
# 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()
```

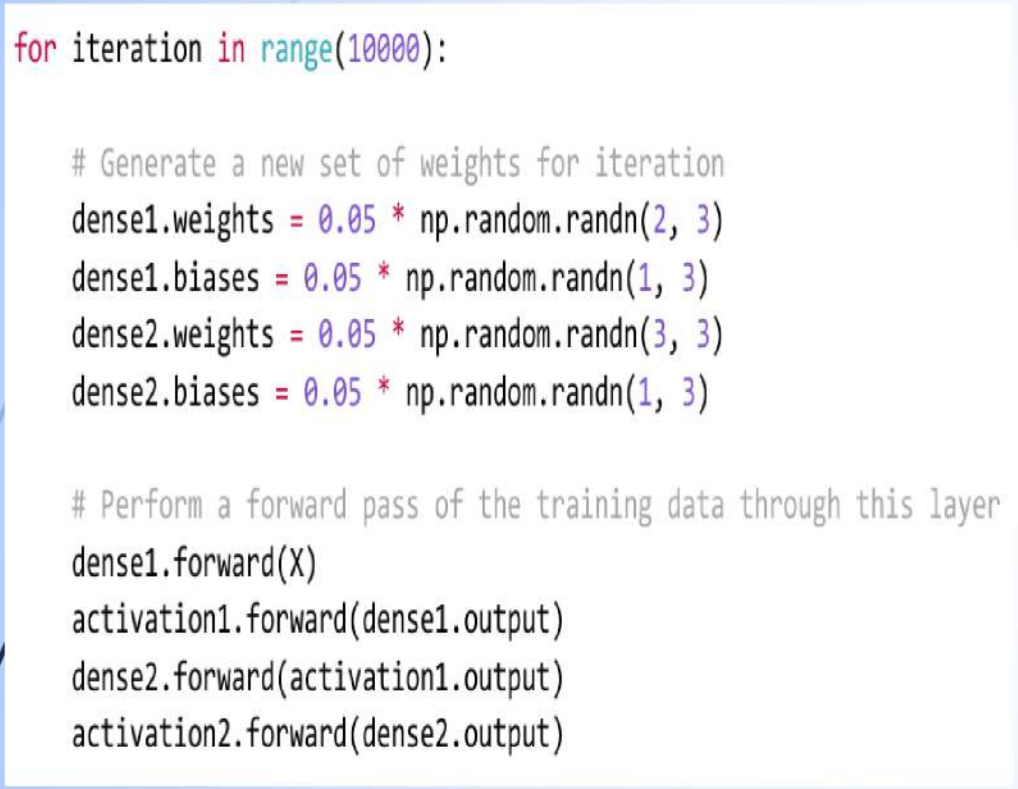
Experimentation

- Let's 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).
- After that, 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:

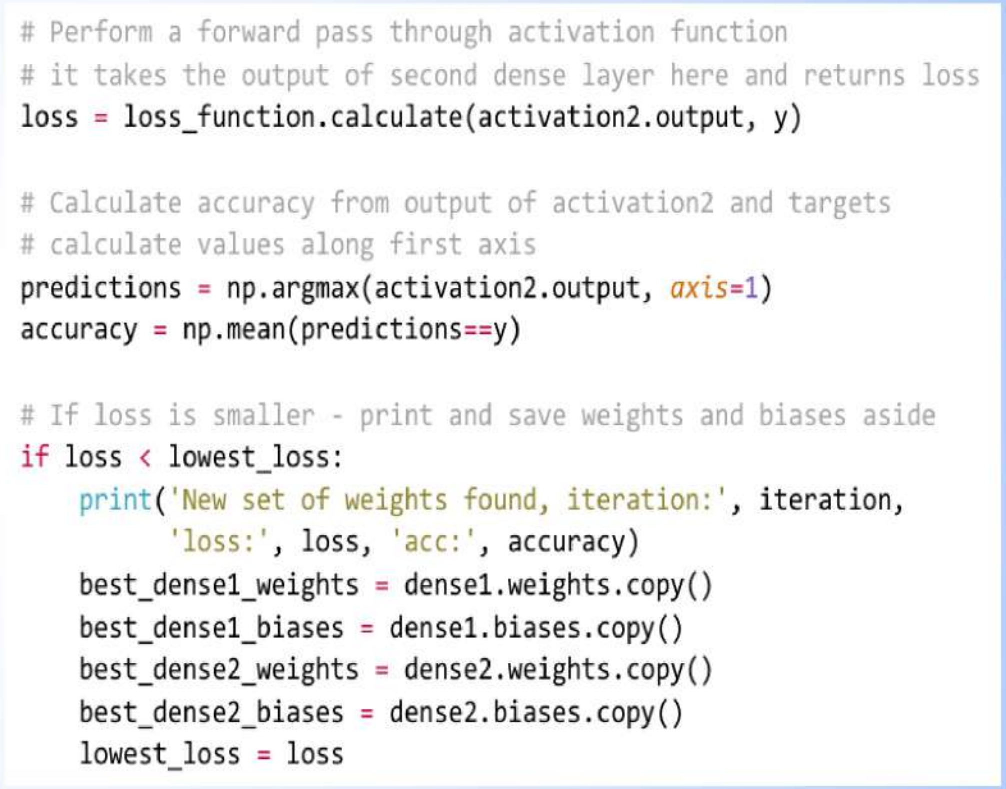
Experimentation



```python
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)
```



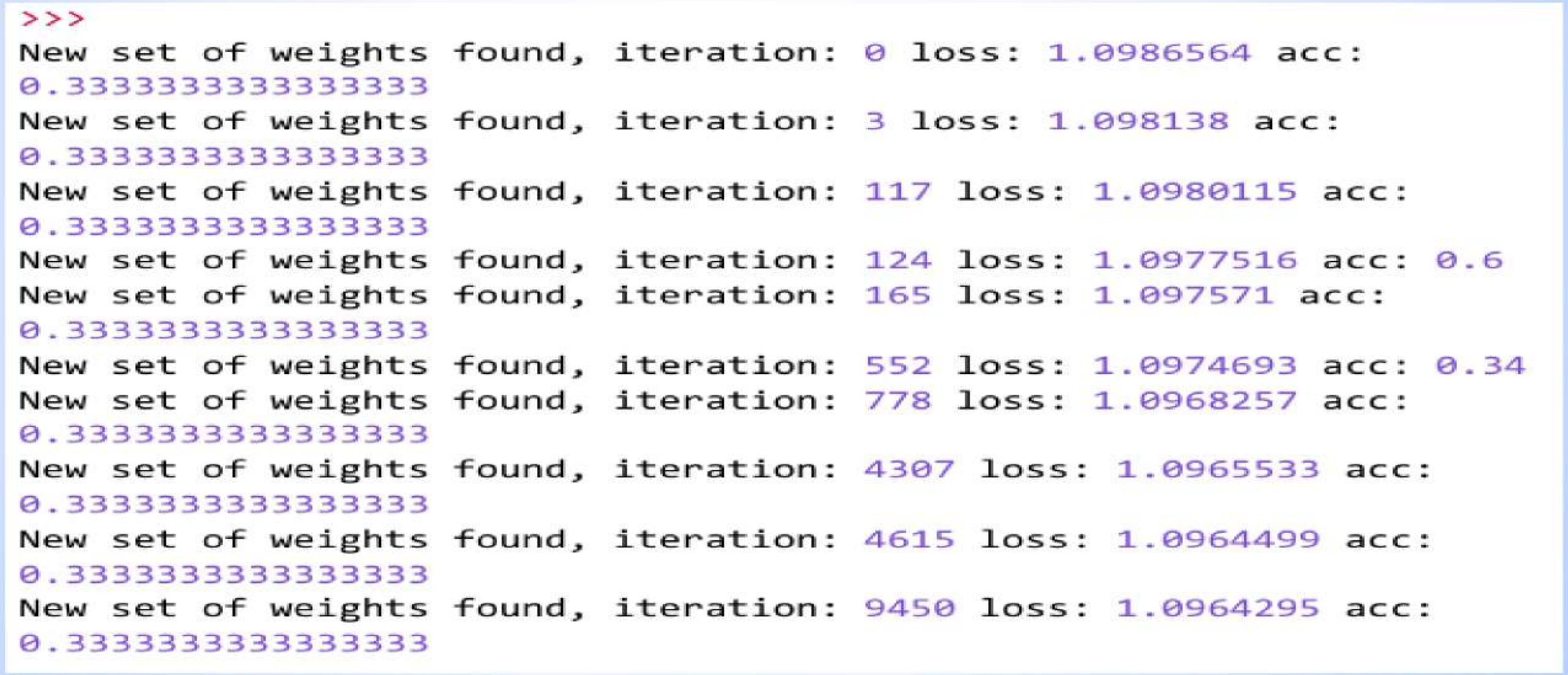
```python
# Perform a forward pass through activation function
# it takes the output of second dense layer here and returns loss
loss = loss_function.calculate(activation2.output, y)

# 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
```

Experimentation

- 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.

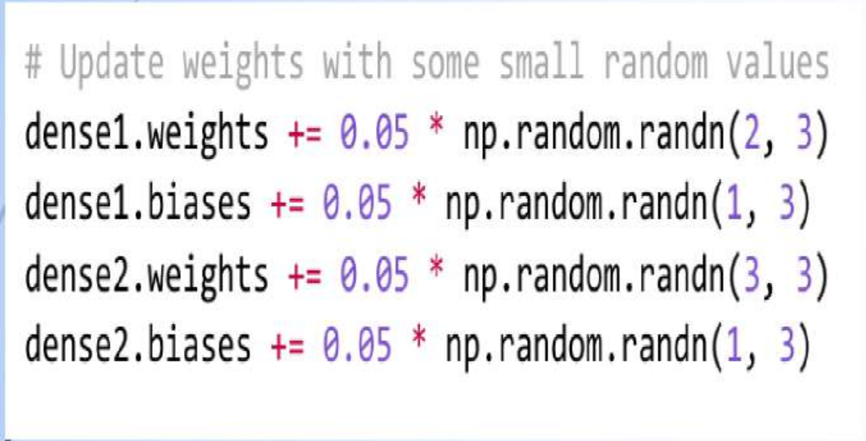


```
>>>
New set of weights found, iteration: 0 loss: 1.0986564 acc:
0.3333333333333333
New set of weights found, iteration: 3 loss: 1.098138 acc:
0.3333333333333333
New set of weights found, iteration: 117 loss: 1.0980115 acc:
0.3333333333333333
New set of weights found, iteration: 124 loss: 1.0977516 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
```

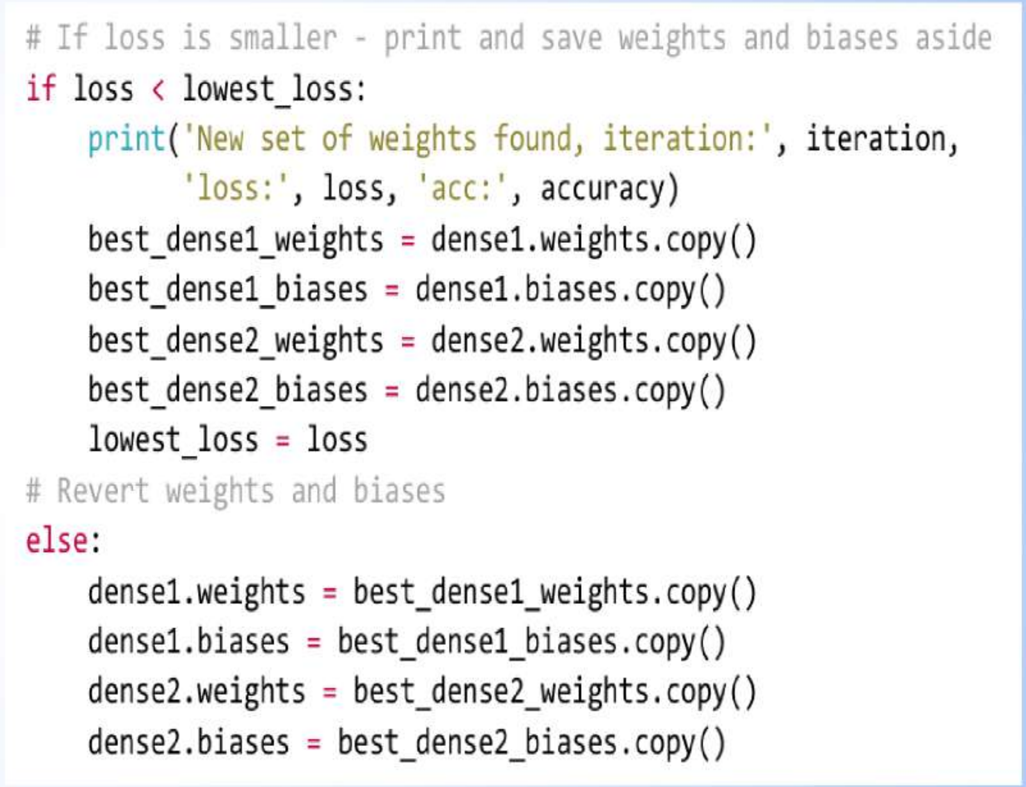
Experimentation

- 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:

Experimentation



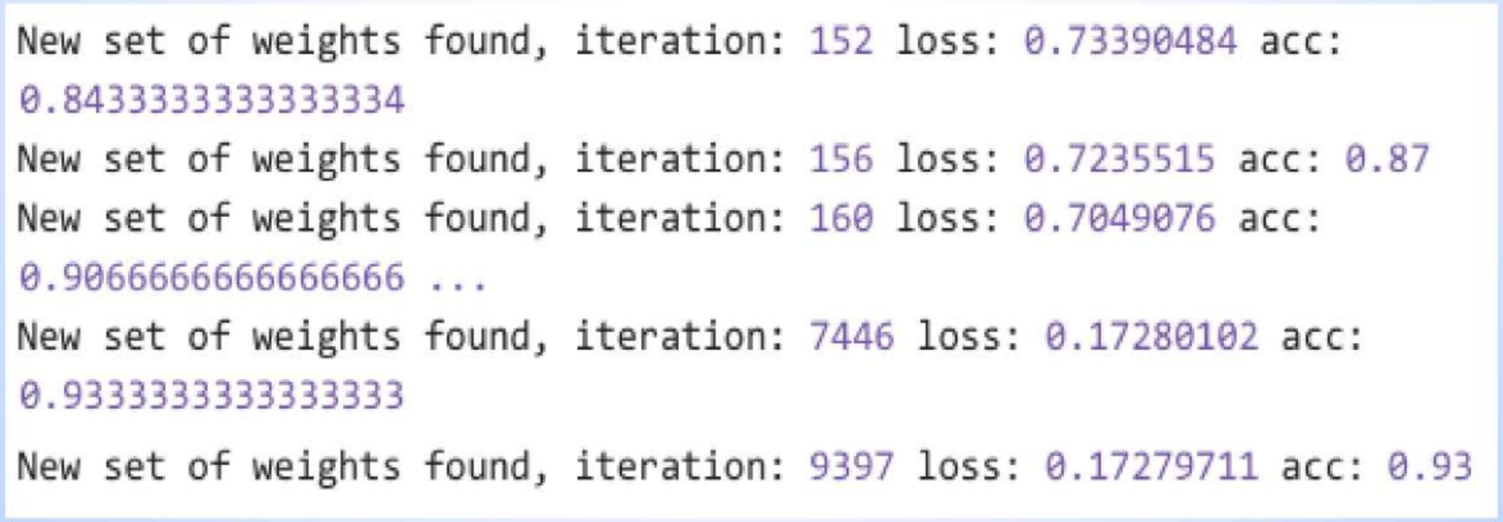
```python
# Update weights with some small random values
dense1.weights += 0.05 * np.random.randn(2, 3)
dense1.biases += 0.05 * np.random.randn(1, 3)
dense2.weights += 0.05 * np.random.randn(3, 3)
dense2.biases += 0.05 * np.random.randn(1, 3)
```



```python
# 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()
```

Experimentation

- 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: 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
```

Conclusion

- In this chapter, experimentations have shown that optimization must not be done blindly.
- You can't just randomly set weights, biases and run for a number of loops to come up with decent loss and accuracy values.
- Neural Networks Optimization is not simple task. For this, an intelligent way of tuning parameters (weights, biases) is needed.
- For optimization to actually work, we need to calculate the impact of each weight and bias used against input and neurons.
- Only after calculating that impact, we can intelligently tune weights and biases to reduce their impact on poor loss and accuracy values.

Thank You