Network Error With Loss

1. The Categorical Cross-Entropy Loss Class:

In the later chapters, we'll be adding more loss functions and some of the operations that we'll be performing are common for all of them. One of these operations is how we calculate the overall loss — no matter which loss function we'll use, the overall loss is always a mean value of all sample losses. Let's create the Loss class containing the calculate method that will call our loss object's forward method and calculate the mean value of the returned sample losses:

```
# Common loss class
class Loss:
    # Calculates the data and regularization losses
        # given model output and ground truth values
def calculate(self, output, y):
    # Calculate sample losses
    sample_losses = self.forward(output, y)
    # Calculate mean loss
    data_loss = np.mean(sample_losses)
    # Return loss
    return data_loss
```

In later chapters, we'll add more code to this class, and the reason for it to exist will become more clear. For now, we'll use it for this single purpose.

Let's convert our loss code into a class for convenience down the line:

```
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):
    # Forward pass
    def forward(self, y_pred, y_true):
    # Number of samples in a batch
    samples = len(y_pred)
```

```
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
        # Probabilities for target values -
        # only if categorical labels
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
        )
        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods
```

This class inherits the Loss class and performs all the error calculations that we derived throughout this chapter and can be used as an object. For example, using the manually-created output and targets:

```
loss_function = Loss_CategoricalCrossentropy()
loss = loss_function.calculate(softmax_outputs, class_targets)
print(loss)
>>>
0.38506088005216804
```

Combining everything 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
# ReLU activation
class Activation_ReLU:
    # Forward pass
    def forward(self, inputs):
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)
# Softmax activation
class Activation_Softmax:
    # Forward pass
    def forward(self, inputs):
```

```
                # Get unnormalized probabilities
                exp_values = np.exp(inputs - np.max(inputs, axis=1,
                    keepdims=True))
                # Normalize them for each sample
                probabilities = exp_values / np.sum(exp_values, axis=1,
                    keepdims=True)
                self.output = probabilities
# Common loss class
class Loss:
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):
        # Calculate sample losses
        sample_losses = self.forward(output, y)
        # Calculate mean loss
        data_loss = np.mean(sample_losses)
        # Return loss
        return data_loss
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):
    # Forward pass
    def forward(self, y_pred, y_true):
        # Number of samples in a batch
        samples = len(y_pred)
        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
        # Probabilities for target values -
        # only if categorical labels
```

```
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[
                range(samples),
                y_true
            ]
        # Mask values - only for one-hot encoded labels
        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(
                y_pred_clipped * y_true,
                axis=1
        )
        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods
# 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)
# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()
# Create second Dense layer with 3 input features (as we take output
# of previous layer here) and 3 output values
dense2 = Layer_Dense(3, 3)
# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()
# Create loss function
loss_function = Loss_CategoricalCrossentropy()
# Perform a forward pass of our training data through this layer
dense1.forward(X)
```

```
        # Perform a forward pass through activation function
        # it takes the output of first dense layer here
        activation1.forward(dense1.output)
        # Perform a forward pass through second Dense layer
        # it takes outputs of activation function of first layer as inputs
        dense2.forward(activation1.output)
        # Perform a forward pass through activation function
        # it takes the output of second dense layer here
        activation2.forward(dense2.output)
        # Let's see output of the first few samples:
        print(activation2.output[:5])
        # Perform a forward pass through loss function
        # it takes the output of second dense layer here and returns loss
        loss = loss_function.calculate(activation2.output, y)
        # Print loss value
        print('loss:', loss)
>>>
        [[0.33333334 0.33333334 0.33333334]
        [0.33333316 0.3333332 0.33333364]
        [0.33333287 0.3333329 0.33333418]
        [0.3333326 0.33333263 0.33333477]
        [0.33333233 0.3333324 0.33333528]]
loss: 1.0986104
```

Again, we get ~0.33 values since the model is random, and its average loss is also not great for these data, as we've not yet trained our model on how to correct its errors.

2. Accuracy Calculation:

While loss is a useful metric for optimizing a model, the metric commonly used in practice along with loss is the accuracy, which describes how often the largest confidence is the correct class in

terms of a fraction. Conveniently, we can reuse existing variable definitions to calculate the accuracy metric. We will use the argmax values from the softmax outputs and then compare these to the targets. This is as simple as doing (note that we slightly modified the softmax_outputs for the purpose of this example):

```
import numpy as np
# Probabilities of 3 samples
softmax_outputs = np.array([[0.7, 0.2, 0.1],
                            [0.5, 0.1, 0.4],
                            [0.02, 0.9, 0.08]])
# Target (ground-truth) labels for 3 samples
class_targets = np.array([0, 1, 1])
# Calculate values along second axis (axis of index 1)
predictions = np.argmax(softmax_outputs, axis=1)
# If targets are one-hot encoded - convert them
if len(class_targets.shape) == 2:
    class_targets = np.argmax(class_targets, axis=1)
# True evaluates to 1; False to 0
accuracy = np.mean(predictions==class_targets)
print('acc:', accuracy)
```

>>>

acc: 0.6666666666666666

We are also handling one-hot encoded targets by converting them to sparse values using np.argmax().

We can add the following to the end of our full script above to calculate its accuracy:

```
# Calculate accuracy from output of activation2 and targets
# calculate values along first axis
predictions = np.argmax(activation2.output, axis=1)
if len(y.shape) == 2:
    y = np.argmax(y, axis=1)
```

```
    accuracy = np.mean(predictions==y)

    # Print accuracy

    print('acc:', accuracy)
```

>>>

acc: 0.34

Now that you've learned how to perform a forward pass through our network and calculate the metrics to signal if the model is performing poorly, we will embark on optimization in the next chapter!