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 5

Calculating Network Error with Loss

With a randomly-initialized model, or even a model initialized with more sophisticated approaches, our goal is to train, or teach, a model over time. To train a model, we tweak the weights and biases to improve the model's accuracy and confidence. To do this, we calculate how much error the model has. The **loss function**, also referred to as the **cost function**, is the algorithm that quantifies how wrong a model is. **Loss** is the measure of this metric. Since loss is the model's error, we ideally want it to be 0.

You may wonder why we do not calculate the error of a model based on the argmax accuracy. Recall our earlier example of confidence: `[0.22, 0.6, 0.18]` vs `[0.32, 0.36, 0.32]`. If the correct class were indeed the middle one (index 1), the model accuracy would be identical between the two above. But are these two examples *really* as accurate as each other? They are not, because accuracy is simply applying an argmax to the output to find the index of the biggest value. The output of a neural network is actually confidence, and more confidence in the correct answer is better. Because of this, we strive to increase correct confidence and decrease misplaced confidence.

Categorical Cross-Entropy Loss

If you're familiar with linear regression, then you already know one of the loss functions used with neural networks that do regression: **squared error** (or **mean squared error** with neural networks).

We're not performing regression in this example; we're classifying, so we need a different loss function. The model has a softmax activation function for the output layer, which means it's outputting a probability distribution. **Categorical cross-entropy** is explicitly used to compare a "ground-truth" probability (*y* or "*targets*") and some predicted distribution (*y-hat* or "*predictions*"), so it makes sense to use cross-entropy here. It is also one of the most commonly used loss functions with a softmax activation on the output layer.

The formula for calculating the categorical cross-entropy of *y* (actual/desired distribution) and *y-hat* (predicted distribution) is:

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j})$$

Where L_i denotes sample loss value, *i* is the i-th sample in the set, *j* is the label/output index, *y* denotes the target values, and *y-hat* denotes the predicted values.

Once we start coding the solution, we'll simplify it further to *-log(correct_class_confidence)*, the formula for which is:

$$L_i = -\log\left(\hat{y}_{i,k}\right) \quad where\ \mathbf{k}\ is\ an\ index\ of\ "true"\ probability$$

Where L_i denotes sample loss value, *i* is the i-th sample in a set, *k* is the index of the target label (ground-true label) and *y-hat* denotes the predicted values.

You may ask why we call this cross-entropy and not **log loss**, which is also a type of loss. If you do not know what log loss is, you may wonder why there is such a fancy looking formula for what looks to be a fairly basic description.

In general, the log loss error function is what we apply to the output of a binary logistic regression model (which we'll describe in chapter 16) — there are only two classes in the distribution, each of them applying to a single output (neuron) which is targeted as a 0 or 1. In our case, we have a classification model that returns a probability distribution over all of the outputs. Cross-entropy compares two probability distributions. In our case, we have a softmax output, let's say it's:

```
softmax_output = [0.7, 0.1, 0.2]
```

Which probability distribution do we intend to compare this to? We have 3 class confidences in the above output, and let's assume that the desired prediction is the first class (index 0, which is currently 0.7). If that's the intended prediction, then the desired probability distribution is `[1, 0, 0]`. Cross-entropy can also work on probability distributions like `[0.2, 0.5, 0.3]`; they wouldn't have to look like the one above. That said, the desired probabilities will consist of a 1 in the desired class, and a 0 in the remaining undesired classes. Arrays or vectors like this are called **one-hot**, meaning one of the values is "hot" (on), with a value of 1, and the rest are "cold" (off), with values of 0. When comparing the model's results to a one-hot vector using cross-entropy, the other parts of the equation zero out, and the target probability's log loss is multiplied by 1, making the cross-entropy calculation relatively simple. This is also a special case of the cross-entropy calculation, called categorical cross-entropy. To exemplify this — if we take a softmax output of `[0.7, 0.1, 0.2]` and targets of `[1, 0, 0]`, we can apply the calculations as follows:

$$L_i = -\sum_j y_{i,j} log(\hat{y}_{i,j}) = -(1 \cdot log(0.7) + 0 \cdot log(0.1) + 0 \cdot log(0.2)) =$$

$$= -(-0.35667494393873245 + 0 + 0) = 0.35667494393873245$$

Let's see the Python code for this:

```
import math


# An example output from the output layer of the neural network
softmax_output = [0.7, 0.1, 0.2]
# Ground truth
target_output = [1, 0, 0]

loss = -(math.log(softmax_output[0])*target_output[0] +
         math.log(softmax_output[1])*target_output[1] +
         math.log(softmax_output[2])*target_output[2])

print(loss)


>>>
0.35667494393873245
```

That's the full categorical cross-entropy calculation, but we can make a few assumptions given one-hot target vectors. First, what are the values for `target_output[1]` and `target_output[2]` in this case? They're both 0, and anything multiplied by 0 is 0. Thus, we don't need to calculate these indices. Next, what's the value for `target_output[0]` in this case? It's 1. So this can be omitted as any number multiplied by 1 remains the same. The same output then, in this example, can be calculated with:

```
loss = -math.log(softmax_output[0])
```

Which still gives us:

```
>>>
0.35667494393873245
```

As you can see with one-hot vector targets, or scalar values that represent them, we can make some simple assumptions and use a more basic calculation — what was once an involved formula reduces to the negative log of the target class' confidence score — the second formula presented at the beginning of this chapter.

As we've already discussed, the example confidence level might look like `[0.22, 0.6, 0.18]` or `[0.32, 0.36, 0.32]`. In both cases, the *argmax* of these vectors will return the second class as the prediction, but the model's confidence about these predictions is high only for one of them. The **Categorical Cross-Entropy Loss** accounts for that and outputs a larger loss the lower the confidence is:

```python
import math

print(math.log(1.))
print(math.log(0.95))
print(math.log(0.9))
print(math.log(0.8))
print('...')
print(math.log(0.2))
print(math.log(0.1))
print(math.log(0.05))
print(math.log(0.01))
```

```
>>>
0.0
-0.05129329438755058
-0.10536051565782628
-0.2231435513142097
...
-1.6094379124341003
-2.3025850929940455
-2.995732273553991
-4.605170185988091
```

We've printed different log values for a few example confidences. When the confidence level equals *1*, meaning the model is 100% "sure" about its prediction, the loss value for this sample equals *0*. The loss value raises with the confidence level, approaching 0. You might also wonder why we did not print the result of *log(0)* — we'll explain that shortly.

So far, we've applied *log()* to the softmax output, but have neither explained what "log" is nor why we use it. We will save the discussion of "why" until the next chapters, which cover derivatives, gradients, optimizations, and more; suffice it to say that the log function has some desirable properties. **Log** is short for **logarithm** and is defined as the solution for the x-term in an equation of the form $a^x = b$. The logarithm equation is $x = \log_a(b)$. For example, $10^x = 100$ can be solved with a log: $x = \log_{10}(100)$, which evaluates to 2. This property of the log function is *especially* beneficial when *e* (Euler's number or *~2.71828*) is used in the base (where 10 is in the example). The logarithm with *e* as its base is referred to as the **natural logarithm**, **natural log**, or simply **log** — you may also see this written as **ln**: $\ln(x) = \log(x) = \log_e(x)$. The variety of conventions can make this confusing, so to simplify things, **any mention of log will always be a natural logarithm throughout this book**. The natural log represents the solution for the x-term in the equation $e^x = b$; for example, $e^x = 5.2$ is solved by *log(5.2)*.

In Python code:

```
import numpy as np

b = 5.2
print(np.log(b))


>>>
1.6486586255873816
```

We can confirm this by exponentiating our result:

```
import math

print(math.e ** 1.6486586255873816)


>>>
5.199999999999999
```

The small difference is the result of floating-point precision in Python. Getting back to the loss calculation, we need to modify our output in two additional ways. First, we'll update our process to work on batches of softmax output distributions; and second, make the negative log calculation dynamic to the target index (the target index has been hard-coded so far).

Consider a scenario with a neural network that performs classification between three classes, and the neural network classifies in batches of three. After running through the softmax activation function with a batch of 3 samples and 3 classes, the network's output layer yields:

```
# Probabilities for 3 samples
softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
```

We need a way to dynamically calculate the categorical cross-entropy, which we now know is a negative log calculation. To determine which value in the softmax output to calculate the negative log from, we simply need to know our target values. In this example, there are 3 classes; let's say we're trying to classify something as a "dog," "cat," or "human." A dog is class 0 (at index 0), a cat class 1 (index 1), and a human class 2 (index 2). Let's assume the batch of three sample inputs to this neural network is being mapped to the target values of a dog, cat, and cat. So the targets (as a list of target indices) would be *[0, 1, 1]*.

```
softmax_outputs = [[0.7, 0.1, 0.2],
                   [0.1, 0.5, 0.4],
                   [0.02, 0.9, 0.08]]

class_targets = [0, 1, 1]  # dog, cat, cat
```

The first value, 0, in `class_targets` means the first softmax output distribution's intended prediction was the one at the 0th index of `[0.7, 0.1, 0.2]`; the model has a `0.7` confidence score that this observation is a dog. This continues throughout the batch, where the intended target of the 2nd softmax distribution, `[0.1, 0.5, 0.4]`, was at an index of 1; the model only has a `0.5` confidence score that this is a cat — the model is less certain about this observation. In the last sample, it's also the 2nd index from the softmax distribution, a value of `0.9` in this case — a pretty high confidence.

With a collection of softmax outputs and their intended targets, we can map these indices to retrieve the values from the softmax distributions:

```
softmax_outputs = [[0.7, 0.1, 0.2],
                   [0.1, 0.5, 0.4],
                   [0.02, 0.9, 0.08]]

class_targets = [0, 1, 1]

for targ_idx, distribution in zip(class_targets, softmax_outputs):
    print(distribution[targ_idx])
```

```
>>>
0.7
0.5
0.9
```

The `zip` function, again, lets us iterate over multiple iterables at the same time in Python. This can be further simplified using NumPy (we're creating a NumPy array of the Softmax outputs this time):

```
softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
class_targets = [0, 1, 1]
```

```
print(softmax_outputs[[0, 1, 2], class_targets])


>>>
[0.7 0.5 0.9]
```

What are the 0, 1, and 2 values? NumPy lets us index an array in multiple ways. One of them is to use a list filled with indices and that's convenient for us — we could use the `class_targets` for this purpose as it already contains the list of indices that we are interested in. The problem is that this has to filter data rows in the array — the second dimension. To perform that, we also need to explicitly filter this array in its first dimension. This dimension contains the predictions and we, of course, want to retain them all. We can achieve that by using a list containing numbers from 0 through all of the indices. We know we're going to have as many indices as distributions in our entire batch, so we can use a `range()` instead of typing each value ourselves:

```
print(softmax_outputs[
    range(len(softmax_outputs)), class_targets
])


>>>
[0.7 0.5 0.9]
```

This returns a list of the confidences at the target indices for each of the samples. Now we apply the negative log to this list:

```
print(-np.log(softmax_outputs[
    range(len(softmax_outputs)), class_targets
]))


>>>
[0.35667494 0.69314718 0.10536052]
```

Finally, we want an average loss per batch to have an idea about how our model is doing during training. There are many ways to calculate an average in Python; the most basic form of an average is the **arithmetic mean**: *sum(iterable) / len(iterable)*. NumPy has a method that computes this average on arrays, so we will use that instead. We add NumPy's average to the code:

```
neg_log = -np.log(softmax_outputs[
              range(len(softmax_outputs)), class_targets
          ])
average_loss = np.mean(neg_log)
print(average_loss)


>>>
0.38506088005216804
```

We have already learned that targets can be one-hot encoded, where all values, except for one, are zeros, and the correct label's position is filled with 1. They can also be sparse, which means that the numbers they contain are the correct class numbers — we are generating them this way with the `spiral_data` function, and we can allow the loss calculation to accept any of these forms. Since we implemented this to work with sparse labels (as in our training data), we have to add a check if they are one-hot encoded and handle it a bit differently in this new case. The check can be performed by counting the dimensions — if targets are single-dimensional (like a list), they are sparse, but if there are 2 dimensions (like a list of lists), then there is a set of one-hot encoded vectors. In this second case, we'll implement a solution using the first equation from this chapter, instead of filtering out the confidences at the target labels. We have to multiply confidences by the targets, zeroing out all values except the ones at correct labels, performing a sum along the row axis (axis *1*). We have to add a test to the code we just wrote for the number of dimensions, move calculations of the log values outside of this new *if* statement, and implement the solution for the one-hot encoded labels following the first equation:

```
import numpy as np

softmax_outputs = np.array([[0.7, 0.1, 0.2],
                            [0.1, 0.5, 0.4],
                            [0.02, 0.9, 0.08]])
class_targets = np.array([[1, 0, 0],
                          [0, 1, 0],
                          [0, 1, 0]])


# Probabilities for target values -
# only if categorical labels
if len(class_targets.shape) == 1:
    correct_confidences = softmax_outputs[
        range(len(softmax_outputs)),
        class_targets
    ]
```

```
# Mask values - only for one-hot encoded labels
elif len(class_targets.shape) == 2:
    correct_confidences = np.sum(
        softmax_outputs*class_targets,
        axis=1
    )

# Losses
neg_log = -np.log(correct_confidences)

average_loss = np.mean(neg_log)
print(average_loss)

>>>
0.38506088005216804
```

Before we move on, there is one additional problem to solve. The softmax output, which is also an input to this loss function, consists of numbers in the range from 0 to 1 - a list of confidences. It is possible that the model will have full confidence for one label making all the remaining confidences zero. Similarly, it is also possible that the model will assign full confidence to a value that wasn't the target. If we then try to calculate the loss of this confidence of 0:

```
import numpy as np
print(-np.log(0))


>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
inf
```

Before we explain this, we need to talk about *log(0)*. From the mathematical point of view, *log(0)* is undefined. We already know the following dependence: if *y=log(x)*, then $e^y=x$. The question of what the resulting *y* is in *y=log(0)* is the same as the question of what's the *y* in $e^y=0$. In simplified terms, the constant *e* to any power is always a positive number, and there is no *y* resulting in $e^y=0$. This means the *log(0)* is undefined. We need to be aware of what the *log(0)* is, and "undefined" does not mean that we don't know anything about it. Since *log(0)* is undefined, what's the result for a value very close to *0*? We can calculate the limit of a function. How to exactly calculate it exceeds this book, but the solution is:

$$\lim_{x \to 0^+} log(x) = -\infty$$

We read it as the limit of a natural logarithm of *x*, with x approaching *0* from a positive (it is impossible to calculate the natural logarithm of a negative value) equals negative infinity. What this means is that the limit is negative infinity for an infinitely small *x*, where *x* never reaches *0*.

The situation is a bit different in programming languages. We do not have limits here, just a function which, given a parameter, returns some value. The negative natural logarithm of *0*, in Python with NumPy, equals an infinitely big number, rather than undefined, and prints a warning about a division by *0* (which is a result of how this calculation is done). If `-np.log(0)` equals `inf`, is it possible to calculate e to the power of negative infinity with Python?

```
print(np.e**(-np.inf))
```

```
>>>
0.0
```

In programming, the fewer things that are undefined, the better. Later on, we'll see similar simplifications, for example when calculating a derivative of the absolute value function, which does not exist for an input of *0* and we'll have to make some decisions to work around this.

Back to the result of `inf` for `-np.log(0)` — as much as that makes sense, since the model would be fully wrong, this will be a problem for us to do further calculations with. Later, with optimization, we will also have a problem calculating gradients, starting with a mean value of all sample-wise losses since a single infinite value in a list will cause the average of that list to also be infinite:

```
import numpy as np
print(np.mean([1, 2, 3, -np.log(0)]))
```

```
>>>
__main__:1: RuntimeWarning: divide by zero encountered in log
inf
```

We could add a very small value to the confidence to prevent it from being a zero, for example, *1e-7*:

```
print(-np.log(1e-7))
```

```
>>>
16.11809565095832
```

Adding a very small value, one-tenth of a million, to the confidence at its far edge will insignificantly impact the result, but this method yields an additional 2 issues. First, in the case where the confidence value is *1*:

```
print(-np.log(1+1e-7))


>>>
-9.999999505838704e-08
```

When the model is fully correct in a prediction and puts all the confidence in the correct label, loss becomes a negative value instead of being 0. The other problem here is shifting confidence towards *1*, even if by a very small value. To prevent both issues, it's better to clip values from both sides by the same number, *1e-7* in our case. That means that the lowest possible value will become *1e-7* (like in the demonstration we just performed) but the highest possible value, instead of being *1+1e-7*, will become *1-1e-7* (so slightly less than *1*):

```
print(-np.log(1-1e-7))

>>>
1.0000000494736474e-07
```

This will prevent loss from being exactly *0*, making it a very small value instead, but won't make it a negative value and won't bias overall loss towards *1*. Within our code and using numpy, we'll accomplish that using `np.clip` method:

```
y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
```

This method can perform clipping on an array of values, so we can apply it to the predictions directly and save this as a separate array, which we'll use shortly.

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:

```python
# 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 activation function 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.

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:

```python
# 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!

Supplementary Material: https://nnfs.io/ch5
Chapter code, further resources, and errata for this chapter.

Chapter 6 - Optimization