To calculate the partial derivatives with respect to inputs, we need the weights — the partial derivative with respect to the input equals the related weight. This means that the array of partial derivatives with respect to all of the inputs equals the array of weights. Since this array is transposed, we'll need to sum its rows instead of columns. To apply the chain rule, we need to multiply them by the gradient from the subsequent function.

In the code to show this, we take the transposed weights, which are the transposed array of the derivatives with respect to inputs, and multiply them by their respective gradients (related to given neurons) to apply the chain rule. Then we sum along with the inputs. Then we calculate the gradient for the next layer in backpropagation. The "next" layer in backpropagation is the previous layer in the order of creation of the model:

```
import numpy as np

# Passed in gradient from the next layer
# for the purpose of this example we're going to use
# a vector of 1s
dvalues = np.array([[1., 1., 1.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T

# sum weights of given input
# and multiply by the passed in gradient for this neuron
dx0 = sum(weights[0])*dvalues[0]
dx1 = sum(weights[1])*dvalues[0]
dx2 = sum(weights[2])*dvalues[0]
dx3 = sum(weights[3])*dvalues[0]

dinputs = np.array([dx0, dx1, dx2, dx3])

print(dinputs)


>>>
[ 0.44 -0.38 -0.07  1.37]
```

`dinputs` is a gradient of the neuron function with respect to inputs.

We defined the gradient of the subsequent function (dvalues) as a row vector, which we'll explain shortly. From NumPy's perspective, and since both weights and dvalues are NumPy arrays, we can simplify the dx0 to dx3 calculation. Since the weights array is formatted so that the rows contain weights related to each input (weights for all neurons for the given input), we can multiply them by the gradient vector directly:

```
import numpy as np

# Passed in gradient from the next layer
# for the purpose of this example we're going to use
# a vector of 1s
dvalues = np.array([[1., 1., 1.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T

# sum weights of given input
# and multiply by the passed in gradient for this neuron
dx0 = sum(weights[0]*dvalues[0])
dx1 = sum(weights[1]*dvalues[0])
dx2 = sum(weights[2]*dvalues[0])
dx3 = sum(weights[3]*dvalues[0])

dinputs = np.array([dx0, dx1, dx2, dx3])

print(dinputs)


>>>
[ 0.44 -0.38 -0.07  1.37]
```

You might already see where we are going with this — the sum of the multiplication of the elements is the dot product. We can achieve the same result by using the `np.dot` function. For this to be possible, we need to match the "inner" shapes and decide the first dimension of the result, which is the first dimension of the first parameter. We want the output of this calculation to be of the shape of the gradient from the subsequent function — recall that we have one partial derivative for each neuron and multiply it by the neuron's partial derivative with respect to its input. We then want to multiply each of these gradients with each of the partial derivatives that are related to this neuron's inputs, and we already noticed that they are rows. The dot product takes rows from the first argument and columns from the second to perform multiplication and sum; thus, we need to transpose the weights for this calculation:

```
import numpy as np

# Passed in gradient from the next layer
# for the purpose of this example we're going to use
# a vector of 1s
dvalues = np.array([[1., 1., 1.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T

# sum weights of given input
# and multiply by the passed in gradient for this neuron
dinputs = np.dot(dvalues[0], weights.T)

print(dinputs)


>>>
[ 0.44 -0.38 -0.07  1.37]
```

We have to account for one more thing — a batch of samples. So far, we have been using a single sample responsible for a single gradient vector that is backpropagated between layers. The row vector that we created for `dvalues` is in preparation for a batch of data. With more samples, the layer will return a list of gradients, which we *almost* handle correctly for. Let's replace the singular gradient `dvalues[0]` with a full list of gradients, `dvalues`, and add more example gradients to this list:

```
import numpy as np

# Passed in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T
```

```
# sum weights of given input
# and multiply by the passed in gradient for this neuron
dinputs = np.dot(dvalues, weights.T)

print(dinputs)


>>>
[[ 0.44 -0.38 -0.07  1.37]
 [ 0.88 -0.76 -0.14  2.74]
 [ 1.32 -1.14 -0.21  4.11]]
```

Calculating the gradients with respect to weights is very similar, but, in this case, we're going to be using gradients to update the weights, so we need to match the shape of weights, not inputs. Since the derivative with respect to the weights equals inputs, weights are transposed, so we need to transpose inputs to receive the derivative of the neuron with respect to weights. Then we use these transposed inputs as the first parameter to the dot product — the dot product is going to multiply rows by inputs, where each row, as it is transposed, contains data for a given input for all of the samples, by the columns of `dvalues`. These columns are related to the outputs of singular neurons for all of the samples, so the result will contain an array with the shape of the weights, containing the gradients with respect to the inputs, multiplied with the incoming gradient for all of the samples in the batch:

```
import numpy as np

# Passed in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# We have 3 sets of inputs - samples
inputs = np.array([[1, 2, 3, 2.5],
                   [2., 5., -1., 2],
                   [-1.5, 2.7, 3.3, -0.8]])

# sum weights of given input
# and multiply by the passed in gradient for this neuron
dweights = np.dot(inputs.T, dvalues)

print(dweights)


>>>
[[ 0.5  0.5  0.5]
 [20.1 20.1 20.1]
 [10.9 10.9 10.9]
 [ 4.1  4.1  4.1]]
```

This output's shape matches the shape of weights because we summed the inputs for each weight and then multiplied them by the input gradient. `dweights` is a gradient of the neuron function with respect to the weights.

For the biases and derivatives with respect to them, the derivatives come from the sum operation and always equal 1, multiplied by the incoming gradients to apply the chain rule. Since gradients are a list of gradients (a vector of gradients for each neuron for all samples), we just have to sum them with the neurons, column-wise, along axis 0.

```
import numpy as np

# Passed in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# One bias for each neuron
# biases are the row vector with a shape (1, neurons)
biases = np.array([[2, 3, 0.5]])

# dbiases - sum values, do this over samples (first axis), keepdims
# since this by default will produce a plain list -
# we explained this in the chapter 4
dbiases = np.sum(dvalues, axis=0, keepdims=True)

print(dbiases)


>>>
[[6. 6. 6.]]
```

keepdims lets us keep the gradient as a row vector — recall the shape of biases array.

The last thing to cover here is the derivative of the ReLU function. It equals *1* if the input is greater than *0* and *0* otherwise. The layer passes its outputs through the *ReLU()* activation during the forward pass. For the backward pass, *ReLU()* receives a gradient of the same shape. The derivative of the ReLU function will form an array of the same shape, filled with 1 when the related input is greater than 0, and 0 otherwise. To apply the chain rule, we need to multiply this array with the gradients of the following function:

```
import numpy as np

# Example layer output
z = np.array([[1, 2, -3, -4],
              [2, -7, -1, 3],
              [-1, 2, 5, -1]])

dvalues = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]])

# ReLU activation's derivative
drelu = np.zeros_like(z)
drelu[z > 0] = 1

print(drelu)

# The chain rule
drelu *= dvalues

print(drelu)


>>>
[[1 1 0 0]
 [1 0 0 1]
 [0 1 1 0]]
[[ 1  2  0  0]
 [ 5  0  0  8]
 [ 0 10 11  0]]
```

To calculate the ReLU derivative, we created an array filled with zeros. `np.zeros_like` is a NumPy function that creates an array filled with zeros, with the shape of the array from its parameter, the `z` array in our case, which is an example output of the neuron. Following the *ReLU()* derivative, we then set the values related to the inputs greater than *0* as 1. We then print this table to see and compare it to the gradients. In the end, we multiply this array with the gradient of the subsequent function and print the result.

We can now simplify this operation. Since the *ReLU()* derivative array is filled with 1s, which do not change the values multiplied by them, and 0s that zero the multiplying value, this means that we can take the gradients of the subsequent function and set to 0 all of the values that correspond to the *ReLU()* input and are equal to or less than 0:

```
import numpy as np

# Example layer output
z = np.array([[1, 2, -3, -4],
              [2, -7, -1, 3],
              [-1, 2, 5, -1]])

dvalues = np.array([[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]])

# ReLU activation's derivative
# with the chain rule applied
drelu = dvalues.copy()
drelu[z <= 0] = 0

print(drelu)


>>>
[[ 1  2  0  0]
 [ 5  0  0  8]
 [ 0 10 11  0]]
```

The copy of `dvalues` ensures that we don't modify it during the ReLU derivative calculation.

Let's combine the forward and backward pass of a single neuron with a full layer and batch-based partial derivatives. We'll minimize ReLU's output, once again, only for this example:

```
import numpy as np

# Passed in gradient from the next layer
# for the purpose of this example we're going to use
# an array of an incremental gradient values
dvalues = np.array([[1., 1., 1.],
                    [2., 2., 2.],
                    [3., 3., 3.]])

# We have 3 sets of inputs - samples
inputs = np.array([[1, 2, 3, 2.5],
                   [2., 5., -1., 2],
                   [-1.5, 2.7, 3.3, -0.8]])
# We have 3 sets of weights - one set for each neuron
# we have 4 inputs, thus 4 weights
# recall that we keep weights transposed
weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]]).T
```

```
# One bias for each neuron
# biases are the row vector with a shape (1, neurons)
biases = np.array([[2, 3, 0.5]])

# Forward pass
layer_outputs = np.dot(inputs, weights) + biases  # Dense layer
relu_outputs = np.maximum(0, layer_outputs)  # ReLU activation

# Let's optimize and test backpropagation here
# ReLU activation - simulates derivative with respect to input values
# from next layer passed to current layer during backpropagation
drelu = relu_outputs.copy()
drelu[layer_outputs <= 0] = 0

# Dense layer
# dinputs - multiply by weights
dinputs = np.dot(drelu, weights.T)
# dweights - multiply by inputs
dweights = np.dot(inputs.T, drelu)
# dbiases - sum values, do this over samples (first axis), keepdims
# since this by default will produce a plain list -
# we explained this in the chapter 4
dbiases = np.sum(drelu, axis=0, keepdims=True)

# Update parameters
weights += -0.001 * dweights
biases += -0.001 * dbiases

print(weights)
print(biases)

>>>
[[ 0.179515   0.5003665 -0.262746 ]
 [ 0.742093  -0.9152577 -0.2758402]
 [-0.510153   0.2529017  0.1629592]
 [ 0.971328  -0.5021842  0.8636583]]
[[1.98489  2.997739 0.497389]]
```

In this code, we replaced the plain Python functions with NumPy variants, created example data, calculated the forward and backward passes, and updated the parameters. Now we will update the dense layer and ReLU activation code with a `backward` method (for backpropagation), which we'll call during the backpropagation phase of our model.

```
# Dense layer
class Layer_Dense:

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

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

# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        self.output = np.maximum(0, inputs)
```

During the `forward` method for our `Layer_Dense` class, we will want to remember what the inputs were (recall that we'll need them when calculating the partial derivative with respect to weights during backpropagation), which can be easily implemented using an object property (`self.inputs`):

```
# Dense layer
class Layer_Dense:
    ...
    # Forward pass
    def forward(self, inputs):
        ...
        self.inputs = inputs
```

Next, we will add our backward pass (backpropagation) code that we developed previously into a new method in the layer class, which we'll call `backward`:

```
class Layer_Dense:
    ...
    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

We then do the same for our ReLU class:

```
# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify the original variable,
        # let's make a copy of the values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0
```

By this point, we've covered everything we need to perform backpropagation, except for the derivative of the Softmax activation function and the derivative of the cross-entropy loss function.

Categorical Cross-Entropy loss derivative

If you are not interested in the mathematical derivation of the Categorical Cross-Entropy loss, feel free to skip to the code implementation, as derivatives are known for common loss functions, and you won't necessarily need to know how to solve them. It is a good exercise if you plan to create custom loss functions, though.

As we learned in chapter 5, the Categorical Cross-Entropy loss function's formula is:

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

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

This formula is convenient when calculating the loss value itself, as all we need is the output of the Softmax activation function at the index of the correct class. For the purpose of the derivative calculation, we'll use the full equation mentioned back in chapter 5:

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

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

We'll use this full function because our current goal is to calculate the gradient, which is composed of the partial derivatives of the loss function with respect to each of its inputs (being the outputs of the Softmax activation function). This means that we cannot use the equation, which takes just the value at the index of the correct class (the first equation above). To calculate partial derivatives with respect to each of the inputs, we need an equation that takes all of them as parameters, thus the choice to use the full equation.

First, let's define the gradient equation:

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

We defined the equation here as the partial derivative of the loss function with respect to each of its inputs. We already learned that the derivative of the sum equals the sum of the derivatives. We also learned that we can move constants. An example is $y_{i,j}$, as it is not what we are calculating the derivative with respect to. Let's apply these transforms:

$$= -\sum_{j} y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(\hat{y}_{i,j}) =$$

Now we have to solve the derivative of the logarithmic function, which is the reciprocal of its parameter, multiplied (using the chain rule) by the partial derivative of this parameter — using prime (also called Lagrange's) notation:

$$f(x) = log(h(x)) \quad \rightarrow \quad f'(x) = \frac{1}{h(x)} \cdot h'(x)$$

We can solve it further (using Leibniz's notation in this case):

$$f(x) = log(x) \quad \rightarrow \quad \frac{d}{dx} f(x) = \frac{d}{dx} log(x) = \frac{1}{x} \cdot \frac{d}{dx} x = \frac{1}{x} \cdot 1 = \frac{1}{x}$$

Let's apply this derivative:

$$= -\sum_{j} y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} =$$

The partial derivative of a value with respect to this value equals 1:

$$= -\sum_{j} y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 = -\sum_{j} \frac{y_{i,j}}{\hat{y}_{i,j}} =$$

Since we are calculating the partial derivative with respect to the *y* given *j*, the sum is being performed over a single element and can be omitted:

$$= -\frac{y_{i,j}}{\hat{y}_{i,j}}$$

Full solution:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial}{\partial \hat{y}_{i,j}}[-\sum_j y_{i,j} log(\hat{y}_{i,j})] = -\sum_j y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} log(\hat{y}_{i,j}) =$$

$$= -\sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} = -\sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 = -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} = -\frac{y_{i,j}}{\hat{y}_{i,j}}$$

The derivative of this loss function with respect to its inputs (predicted values at the i-th sample, since we are interested in a gradient with respect to the predicted values) equals the negative ground-truth vector, divided by the vector of the predicted values (which is also the output vector of the softmax function).

Categorical Cross-Entropy loss derivative code implementation

Since we derived this equation and have found that it solves to a simple division operation of 2 values, we know that, with NumPy, we can extend this operation to the sample-wise vectors of ground truth and predicted values, and further to the batch-wise arrays of them. From the coding perspective, we need to add a backward method to the Loss_CategoricalCrossentropy class. We need to pass the array of predictions and the array of true values into it and calculate the negated division of them:

```
# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):
    ...
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate gradient
        self.dinputs = -y_true / dvalues
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Along with the partial derivative calculation, we are performing two additional operations. First, we're turning numerical labels into one-hot encoded vectors — prior to this, we need to check how many dimensions y_true consists of. If the shape of the labels returns a single dimension

(which means that they are shaped like a list and not like an array), they consist of discrete numbers and need to be converted to a list of one-hot encoded vectors — a two-dimensional array. If that's the case, we need to turn them into one-hot encoded vectors. We'll use the `np.eye` method which, given a number, *n*, returns an *n*x*n* array filled with ones on the diagonal and zeros everywhere else. For example:

```
import numpy as np
np.eye(5)
```

```
>>>
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

We can then index this table with the numerical label to get the one-hot encoded vector that represents it:

```
np.eye(5)[1]
```

```
>>>
array([0., 1., 0., 0., 0.])
```

```
np.eye(5)[4]
```

```
>>>
array([0., 0., 0., 0., 1.])
```

If *y_true* is already one-hot encoded, we do not perform this step.

The second operation is the gradient normalization. As we'll learn in the next chapter, optimizers sum all of the gradients related to each weight and bias before multiplying them by the learning rate (or some other factor). What this means, in our case, is that the more samples we have in a dataset, the more gradient sets we'll receive at this step, and the bigger this sum will become. As a consequence, we'll have to adjust the learning rate according to each set of samples. To solve this problem, we can divide all of the gradients by the number of samples. A sum of elements divided by a count of them is their mean value (and, as we mentioned, the optimizer will perform the sum) — this way, we'll effectively normalize the gradients and make their sum's magnitude invariant to the number of samples.

Softmax activation derivative

The next calculation that we need to perform is the partial derivative of the Softmax function, which is a bit more complicated task than the derivative of the Categorical Cross-Entropy loss. Let's remind ourselves of the equation of the Softmax activation function and define the derivative:

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \quad \rightarrow \quad \frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}}$$

Where $S_{i,j}$ denotes j-th Softmax's output of i-th sample, z — input array which is a list of input vectors (output vectors from the previous layer), $z_{i,j}$ — j-th Softmax's input of i-th sample, L — number of inputs, $z_{i,k}$ — k-th Softmax's input of i-th sample.

As we described in chapter 4, the Softmax function equals the exponentiated input divided by the sum of all exponentiated inputs. In other words, we need to exponentiate all of the values first, then divide each of them by the sum of all of them to perform the normalization. Each input to the Softmax impacts each of the outputs, and we need to calculate the partial derivative of each output with respect to each input. From the programming side of things, if we calculate the impact of one list on the other list, we'll receive a matrix of values as a result. That's exactly what we'll calculate here — we'll calculate the **Jacobian matrix** (which we'll explain later) of the vectors, which we'll dive deeper into soon.

To calculate this derivative, we need to first define the derivative of the division operation:

$$f(x) = \frac{g(x)}{h(x)} \quad \rightarrow \quad f'(x) = \frac{g'(x) \cdot h(x) - g(x) \cdot h'(x)}{[h(x)]^2}$$

In order to calculate the derivative of the division operation, we need to take the derivative of the numerator multiplied by the denominator, subtract the numerator multiplied by the derivative of the denominator from it, and then divide the result by the squared denominator.

We can now start solving the derivative:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}} =$$

Let's apply the derivative of the division operation:

$$= \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

At this step, we have two partial derivatives present in the equation. For the one on the right side of the numerator (right side of the subtraction operator):

$$\frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}$$

We need to calculate the derivative of the sum of the constant,*e* (Euler's number), raised to power $z_{i,l}$ (where *l* denotes consecutive indices from *1* to the number of the Softmax outputs — *L*) with respect to the $z_{i,k}$. The derivative of the sum operation is the sum of derivatives, and the derivative of the constant *e* raised to power *n* (e^n) with respect to *n* equals e^n:

$$\frac{d}{dn} e^n = e^n \cdot \frac{d}{dn} n = e^n \cdot 1 = e^n$$

It is a special case when the derivative of an exponential function equals this exponential function itself, as its exponent is exactly what we are deriving with respect to, thus its derivative equals *1*. We also know that the range *1...L* contains *k* (*k* is one of the indices from this range) exactly once and then, in this case, the derivative is going to equal *e* to the power of the $z_{i,k}$ (as *j* equals *k*) and *0* otherwise (when *j* does not equal *k* as $z_{i,l}$ won't contain $z_{i,k}$ and will be treated as a constant — The derivative of the constant equals *0*):

$$\frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}} = \frac{\partial}{\partial z_{i,k}} e^{z_{i,1}} + \frac{\partial}{\partial z_{i,k}} e^{z_{i,2}} + ... + \frac{\partial}{\partial z_{i,k}} e^{z_{i,k}} + ... + \frac{\partial}{\partial z_{i,k}} e^{z_{i,L-1}} + \frac{\partial}{\partial z_{i,k}} e^{z_{i,L}}$$

$$= 0 + 0 + ... + e^{z_{i,k}} + ... + 0 + 0 = e^{z_{i,k}}$$

The derivative on the left side of the subtraction operator in the denominator is a slightly different case:

$$\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}}$$

It does not contain the sum over all of the elements like the derivative we solved moments ago, so it can become either *0* if $j \neq k$ or *e* to the power of the $z_{i,j}$ if $j=k$. That means, starting from this step, we need to calculate the derivatives separately for both cases. Let's start with $j=k$.

In the case of $j=k$, the derivative on the left side is going to equal *e* to the power of the $z_{i,j}$ and the derivative on the right solves to the same value in both cases. Let's substitute them:

$$= \frac{e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

The numerator contains the constant *e* to the power of $z_{i,j}$ in both the minuend (the value we are subtracting from) and subtrahend (the value we are subtracting from the minuend) of the subtraction operation. Because of this, we can regroup the numerator to contain this value multiplied by the subtraction of their current multipliers. We can also write the denominator as a multiplication of the value instead of using the power of *2*:

$$= \frac{e^{z_{i,j}} \cdot (\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}})}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

Then let's split the whole equation into 2 parts:

$$= \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} =$$

We moved *e* from the numerator and the sum from the denominator to its own fraction, and the content of the parentheses in the numerator, and the other sum from the denominator as another fraction, both joined by the multiplication operation. Now we can further split the "right" fraction into two separate fractions:

$$= \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot (\frac{\sum_{l=1}^{L} e^{z_{i,l}}}{\sum_{l=1}^{L} e^{z_{i,l}}} - \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}}) =$$

In this case, as it's a subtraction operation, we separated both values from the numerator, dividing them both by the denominator and applying the subtraction operation between new fractions. If

we look closely, the "left" fraction turns into the Softmax function's equation, as well as the "right" one, with the middle fraction solving to *1* as the numerator and the denominator are the same values:

$$= S_{i,j} \cdot (1 - S_{i,k})$$

Note that the "left" Softmax function carries the $_j$ parameter, and the "right" one $_k$ — both came from their numerators, respectively.

Full solution:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}} = \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

$$\frac{e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} = \frac{e^{z_{i,j}} \cdot (\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}})}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

$$\frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{\sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot (\frac{\sum_{l=1}^{L} e^{z_{i,l}}}{\sum_{l=1}^{L} e^{z_{i,l}}} - \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}}) =$$

$$S_{i,j} \cdot (1 - S_{i,k})$$

Now we have to go back and solve the derivative in the case of *j≠k*. In this case, the "left" derivative of the original equation solves to *0* as the whole expression is treated as a constant:

$$= \frac{0 \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

The difference is that now the whole subtrahend solves to *0*, leaving us with just the minuend in the numerator:

$$= \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

Now, exactly like before, we can write the denominator as the multiplication of the values instead of using the power of *2*:

$$= \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

That lets us to split this fraction into 2 fractions, using the multiplication operation:

$$= -\frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} =$$

Now both fractions represent the Softmax function:

$$= -S_{i,j} \cdot S_{i,k}$$

Note that the left Softmax function carries the $_j$ parameter, and the "right" one has $_k$ — both came from their numerators, respectively.

Full solution:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}}}{\partial z_{i,k}} = \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^{L} e^{z_{i,l}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} =$$

$$\frac{0 \cdot \sum_{l=1}^{L} e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} = \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^{L} e^{z_{i,l}}]^2} = \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}} \cdot \sum_{l=1}^{L} e^{z_{i,l}}} =$$

$$-\frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \cdot \frac{e^{z_{i,k}}}{\sum_{l=1}^{L} e^{z_{i,l}}} = -S_{i,j} \cdot S_{i,k}$$

As a summary, the solution of the derivative of the Softmax function with respect to its inputs is:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ -S_{i,j} \cdot S_{i,k} & j \neq k \end{cases}$$

That's not the end of the calculation that we can perform here. When left in this form, we'll have 2 separate equations to code and use in different cases, which isn't very convenient for the speed of calculations. We can, however, further morph the result of the second case of the derivative:

$$-S_{i,j} \cdot S_{i,k} = S_{i,j} \cdot (-S_{i,k}) = S_{i,j} \cdot (0 - S_{i,k})$$

In the first step, we moved the second Softmax along the minus sign into the brackets so we can add a zero inside of them and right before this value. That does not change the solution, but now:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ S_{i,j} \cdot (0 - S_{i,k}) & j \neq k \end{cases}$$

Both solutions look very similar, they differ only in a single value. Conveniently, there exists **Kronecker delta** function (which we'll explain soon) whose equation is:

$$\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

We can apply it here, simplifying our equation further to:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = S_{i,j} \cdot (\delta_{j,k} - S_{i,k})$$

That's the final math solution to the derivative of the Softmax function's outputs with respect to each of its inputs. To make it a little bit easier to implement in Python using NumPy, let's transform the equation for the last time:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = S_{i,j} \cdot (\delta_{j,k} - S_{i,k}) = S_{i,j}\delta_{j,k} - S_{i,j}S_{i,k}$$

We basically multiplied $S_{i,j}$ by both sides of the subtraction operation from the parentheses.

Softmax activation derivative code implementation

This lets us code the solution using just two NumPy functions, which we'll explain now step by step:

Let's make up a single sample:

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

And shape it as a list of samples:

```
import numpy as np

softmax_output = np.array(softmax_output).reshape(-1, 1)
print(softmax_output)


>>>
array([[0.7],
       [0.1],
       [0.2]])
```

The left side of the equation is Softmax's output multiplied by the Kronecker delta. The Kronecker delta equals *1* when both inputs are equal, and *0* otherwise. If we visualize this as an array, we'll have an array of zeros with ones on the diagonal — you might remember that we already have implemented such a solution using the `np.eye` method:

```
print(np.eye(softmax_output.shape[0]))
```

```
>>>
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Now we'll do the multiplication of both of the values from the equation part:

```
print(softmax_output * np.eye(softmax_output.shape[0]))
```

```
>>>
array([[0.7, 0. , 0. ],
       [0. , 0.1, 0. ],
       [0. , 0. , 0.2]])
```

It turns out that we can gain some speed by replacing this by the `np.diagflat` method call, which computes the same solution — the diagflat method creates an array using an input vector as the diagonal:

```
print(np.diagflat(softmax_output))
```

```
>>>
array([[0.7, 0. , 0. ],
       [0. , 0.1, 0. ],
       [0. , 0. , 0.2]])
```

The other part of the equation is $S_{i,j}S_{i,k}$ — the multiplication of the Softmax outputs, iterating over the $_j$ and $_k$ indices respectively. Since, for each sample (the $_i$ index), we'll have to multiply the values from the Softmax function's output (in all of the combinations), we can use the dot product operation. For this, we'll just have to transpose the second argument to get its row vector form (as described in chapter 2):

```
print(np.dot(softmax_output, softmax_output.T))
```

```
>>>
array([[0.49, 0.07, 0.14],
       [0.07, 0.01, 0.02],
       [0.14, 0.02, 0.04]])
```

Finally, we can perform the subtraction of both arrays (following the equation):

```
print(np.diagflat(softmax_output) -
      np.dot(softmax_output, softmax_output.T))


>>>
array([[ 0.21, -0.07, -0.14],
       [-0.07,  0.09, -0.02],
       [-0.14, -0.02,  0.16]])
```

The matrix result of the equation and the array solution provided by the code is called the **Jacobian matrix**. In our case, the Jacobian matrix is an array of partial derivatives in all of the combinations of both input vectors. Remember, we are calculating the partial derivatives of every output of the Softmax function with respect to each input separately. We do this because each input influences each output due to the normalization process, which takes the sum of all the exponentiated inputs. The result of this operation, performed on a batch of samples, is a list of the Jacobian matrices, which effectively forms a 3D matrix — you can visualize it as a column whose levels are Jacobian matrices being the sample-wise gradient of the Softmax function.

This raises a question — if sample-wise gradients are the Jacobian matrices, how do we perform the chain rule with the gradient back-propagated from the loss function, since it's a vector for each sample? Also, what do we do with the fact that the previous layer, which is the Dense layer, will expect the gradients to be a 2D array? Currently, we have a 3D array of the partial derivatives — a list of the Jacobian matrices. The derivative of the Softmax function with respect to any of its inputs returns a vector of partial derivatives (a row from the Jacobian matrix), as this input influences all the outputs, thus also influencing the partial derivative for each of them. We need to sum the values from these vectors so that each of the inputs for each of the samples will return a single partial derivative value instead. Because each input influences all of the outputs, the returned vector of the partial derivatives has to be summed up for the final partial derivative with respect to this input. We can perform this operation on each of the Jacobian matrices directly, applying the chain rule at the same time (applying the gradient from the loss function) using `np.dot()` — For each sample, it'll take the row from the Jacobian matrix and multiply it by the corresponding value from the loss function's gradient. As a result, the dot product of each of these vectors and values will return a singular value, forming a vector of the partial derivatives sample-wise and a 2D array (a list of the resulting vectors) batch-wise.

Let's code the solution:

```
# Softmax activation
class Activation_Softmax:
    ...
    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output and
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)
```

First, we created an empty array (which will become the resulting gradient array) with the same shape as the gradients that we're receiving to apply the chain rule. The `np.empty_like` method creates an empty and uninitialized array. Uninitialized means that we can expect it to contain meaningless values, but we'll set all of them shortly anyway, so there's no need for initialization (for example, with zeros using `np.zeros()` instead). In the next step, we're going to iterate sample-wise over pairs of the outputs and gradients, calculating the partial derivatives as described earlier and calculating the final product (applying the chain rule) of the Jacobian matrix and gradient vector (from the passed-in gradient array), storing the resulting vector as a row in the dinput array. We're going to store each vector in each row while iterating, forming the output array.

Common Categorical Cross-Entropy loss and Softmax activation derivative

At the moment, we have calculated the partial derivatives of the Categorical Cross-Entropy loss and Softmax activation functions, and we can finally use them, but there is still one more step that we can perform to speed the calculations up. Different books and tutorials usually mention the derivative of the loss function with respect to the Softmax inputs, or even weight and biases of the output layer directly and don't go into the details of the partial derivatives of these functions separately. This is partially because the derivatives of both functions combine to solve a simple equation — the whole code implementation is simpler and faster to execute. When we look at our current code, we perform multiple operations to calculate the gradients and even include a loop in the backward step of the activation function.

Let's apply the chain rule to calculate the partial derivative of the Categorical Cross-Entropy loss function with respect to the Softmax function inputs. First, let's define this derivative by applying the chain rule:

$$\frac{\partial L_i}{\partial z_{i,k}} = \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial S_{i,j}}{\partial z_{i,k}} =$$

This partial derivative equals the partial derivative of the loss function with respect to its inputs, multiplied (using the chain rule) by the partial derivative of the activation function with respect to its inputs. Now we need to systematize semantics — we know that the inputs to the loss function, $\hat{y}_{i,j}$, are the outputs of the activation function, $S_{i,j}$:

$$\hat{y}_{i,j} = S_{i,j}$$

That means that we can update the equation to the form of:

$$= \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

Now we can substitute the equation for the partial derivative of the Categorical Cross-Entropy function, but, since we are calculating the partial derivative with respect to the Softmax inputs, we'll use the one containing the sum operator over all of the outputs — it will soon become clear why. The derivative:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}}$$

After substitution to the combined derivative's equation:

$$= -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

Now, as we calculated before, the partial derivative of the Softmax activation, before applying Kronecker delta to it:

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ -S_{i,j} \cdot S_{i,k} & j \neq k \end{cases}$$

Let's actually do the substitution of the $S_{i,j}$ with *y-hat*$_{i,j}$ here as well:

$$\frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = \begin{cases} \hat{y}_{i,j} \cdot (1 - \hat{y}_{i,k}) & j = k \\ -\hat{y}_{i,j} \cdot \hat{y}_{i,k} & j \neq k \end{cases}$$

The solution is different depending on if $j=k$ or $j\neq k$. To handle for this situation, we have to split the current partial derivative following these cases — when they both match and when they do not:

$$-\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}} - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}}$$

For the $j\neq k$ case, we just updated the sum operator to exclude k and that's the only change:

$$-\sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}}$$

For the $j=k$ case, we do not need the sum operator as it will sum only one element, of index k. For

the same reason, we also replace *j*indices with *k*:

$$-\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}}$$

Back to the main equation:

$$= -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}} - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

Now we can substitute the partial derivatives of the activation function for both cases with the newly-defined solutions:

$$= -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \hat{y}_{i,k} \cdot (1 - \hat{y}_{i,k}) - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} (-\hat{y}_{i,j} \hat{y}_{i,k}) =$$

We can cancel out the *y-hat*$_{i,k}$ from both sides of the subtraction in the equation — both contain it as part of the multiplication operations and in their denominators. Then on the "right" side of the equation, we can replace 2 minus signs with the plus one and remove the parentheses:

$$= -y_{i,k} \cdot (1 - \hat{y}_{i,k}) + \sum_{j \neq k} y_{i,j} \hat{y}_{i,k} =$$

Now let's multiply the -$y_{i,k}$ with the content of the parentheses on the "left" side of the equation:

$$= -y_{i,k} + y_{i,k} \hat{y}_{i,k} + \sum_{j \neq k} y_{i,j} \hat{y}_{i,k} =$$

Now let's look at the sum operation — it adds up $y_{i,j}$*y-hat*$_{i,k}$ over all possible values of index *j* except for when it equals *k*. Then, on the left of this part of the equation, we have $y_{i,k}$*y-hat*$_{i,k}$, which contains $y_{i,k}$ — the exact element that is excluded from the sum. We can then join both expressions:

$$= -y_{i,k} + \sum_{j} y_{i,j} \hat{y}_{i,k} =$$

Now the sum operator iterates over all of the possible values of *j* and, since we know that $y_{i,j}$ for each *i* is the one-hot encoded vector of ground-truth values, the sum of all of its elements equals *1*. In other words, following the earlier explanation in this chapter — this sum will multiply *0* by the *y-hat*$_{i,k}$ except for a single situation, the true label, where it'll multiply *1* by this value. We can then simplify it further to:

Full solution:

$$\frac{\partial L_i}{\partial z_{i,k}} = \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\partial L_i}{\partial \hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = -\sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} =$$

$$= -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \frac{\partial \hat{y}_{i,k}}{\partial z_{i,k}} - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} \cdot \frac{\partial \hat{y}_{i,j}}{\partial z_{i,k}} = -\frac{y_{i,k}}{\hat{y}_{i,k}} \cdot \hat{y}_{i,k} \cdot (1 - \hat{y}_{i,k}) - \sum_{j \neq k} \frac{y_{i,j}}{\hat{y}_{i,j}} (-\hat{y}_{i,j} \hat{y}_{i,k}) =$$

$$= -y_{i,k} \cdot (1 - \hat{y}_{i,k}) + \sum_{j \neq k} y_{i,j} \hat{y}_{i,k} = -y_{i,k} + y_{i,k} \hat{y}_{i,k} + \sum_{j \neq k} y_{i,j} \hat{y}_{i,k} =$$

$$= -y_{i,k} + \sum_j y_{i,j} \hat{y}_{i,k} = -y_{i,k} + \hat{y}_{i,k} = \hat{y}_{i,k} - y_{i,k}$$

As we can see, when we apply the chain rule to both partial derivatives, the whole equation simplifies significantly to the subtraction of the predicted and ground truth values. It is also multiple times faster to compute.

Common Categorical Cross-Entropy loss and Softmax activation derivative - code implementation

To code this solution, nothing in the forward pass changes — we still need to perform it on the activation function to receive the outputs and then on the loss function to calculate the loss value. For backpropagation, we'll create the backward step containing the implementation of the new equation, which calculates the combined gradient of the loss and activation functions. We'll code the solution as a separate class, which initializes both the Softmax activation and the Categorical Cross-Entropy objects, calling their forward methods respectively during the forward pass. Then the new backward pass is going to contain the new code:

```
# Softmax classifier - combined Softmax activation
# and cross-entropy loss for faster backward step
class Activation_Softmax_Loss_CategoricalCrossentropy():

    # Creates activation and loss function objects
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)
```

```
        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

To implement the solution $y\text{-}hat_{i,k}\text{-}y_{i,k}$, instead of performing the subtraction of the full arrays, we're taking advantage of the fact that the *y* being `y_true` in the code consists of one-hot encoded vectors, which means that, for each sample, there is only a singular value of *1* in these vectors and the remaining positions are filled with zeros.

This means that we can use NumPy to index the prediction array with the sample number and its true value index, subtracting *1* from these values. This operation requires discrete true labels instead of one-hot encoded ones, thus the additional code that performs the transformation if needed — If the number of dimensions in the ground-truth array equals *2*, it means that it's a matrix consisting of one-hot encoded vectors. We can use `np.argmax()`, which returns the index of the maximum value (index for *1* in this case), but we need to perform this operation sample-wise to get a vector of indices:

```
import numpy as np
y_true = np.array([[1,0,0],[0,0,1],[0,1,0]])
np.argmax(y_true)


>>>
0


print(np.argmax(y_true, axis=1))


>>>
[0, 2, 1]
```

For the last step, we normalize the gradient in exactly the same way and for the same reason as described along with the Categorical Cross-Entropy gradient normalization.

Let's summarize the code for each of the classes that we have updated:

```
# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = 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

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output and
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# 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

        # Backward pass
        def backward(self, dvalues, y_true):

            # Number of samples
            samples = len(dvalues)
            # Number of labels in every sample
            # We'll use the first sample to count them
            labels = len(dvalues[0])

            # If labels are sparse, turn them into one-hot vector
            if len(y_true.shape) == 1:
                y_true = np.eye(labels)[y_true]

            # Calculate gradient
            self.dinputs = -y_true / dvalues
            # Normalize gradient
            self.dinputs = self.dinputs / samples


    # Softmax classifier - combined Softmax activation
    # and cross-entropy loss for faster backward step
    class Activation_Softmax_Loss_CategoricalCrossentropy():

        # Creates activation and loss function objects
        def __init__(self):
            self.activation = Activation_Softmax()
            self.loss = Loss_CategoricalCrossentropy()
```

```
    # Forward pass
    def forward(self, inputs, y_true):
        # Output layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return loss value
        return self.loss.calculate(self.output, y_true)

    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

We can now test if the combined backward step returns the same values compared to when we backpropagate gradients through both of the functions separately. For this example, let's make up an output of the Softmax function and some target values. Next, let's backpropagate them using both solutions:

```
import numpy as np
import nnfs

nnfs.init()

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([0, 1, 1])

softmax_loss = Activation_Softmax_Loss_CategoricalCrossentropy()
softmax_loss.backward(softmax_outputs, class_targets)
dvalues1 = softmax_loss.dinputs

activation = Activation_Softmax()
activation.output = softmax_outputs
loss = Loss_CategoricalCrossentropy()
loss.backward(softmax_outputs, class_targets)
activation.backward(loss.dinputs)
dvalues2 = activation.dinputs
```

```
print('Gradients: combined loss and activation:')
print(dvalues1)
print('Gradients: separate loss and activation:')
print(dvalues2)
```

```
>>>
Gradients: combined loss and activation:
[[-0.1         0.03333333  0.06666667]
 [ 0.03333333 -0.16666667  0.13333333]
 [ 0.00666667 -0.03333333  0.02666667]]
Gradients: separate loss and activation:
[[-0.09999999  0.03333334  0.06666667]
 [ 0.03333334 -0.16666667  0.13333334]
 [ 0.00666667 -0.03333333  0.02666667]]
```

The results are the same. The small difference between values in both arrays results from the precision of floating-point values in raw Python and NumPy. To answer the question of how many times faster this solution is, we can take advantage of Python's timeit module, running both solutions multiple times and combining the execution times. A full description of the timeit module and the code used here is outside of the scope of this book, but we include this code purely to show the speed deltas:

```
import numpy as np
from timeit import timeit
import nnfs

nnfs.init()

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([0, 1, 1])


def f1():
    softmax_loss = Activation_Softmax_Loss_CategoricalCrossentropy()
    softmax_loss.backward(softmax_outputs, class_targets)
    dvalues1 = softmax_loss.dinputs


def f2():
    activation = Activation_Softmax()
    activation.output = softmax_outputs
    loss = Loss_CategoricalCrossentropy()
    loss.backward(softmax_outputs, class_targets)
    activation.backward(loss.dinputs)
    dvalues2 = activation.dinputs
```

```
t1 = timeit(lambda: f1(), number=10000)
t2 = timeit(lambda: f2(), number=10000)
print(t2/t1)


>>>
6.922146504409747
```

Calculating the gradients separately is about 7 times slower. This factor can differ from a machine to a machine, but it clearly shows that it was worth putting in additional effort to calculate and code the optimized solution of the combined loss and activation function derivative.

Let's take the code of the model and initialize the new class of combined accuracy and loss class' object:

```
# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()
```

Instead of the previous:

```
# Create Softmax activation (to be used with Dense layer):
activation2 = Activation_Softmax()

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

Then replace the forward pass calls over these objects:

```
# Perform a forward pass through activation function
# takes the output of second dense layer here
activation2.forward(dense2.output)

...

# Calculate sample losses from output of activation2 (softmax activation)
loss = loss_function.forward(activation2.output, y)
```

With the forward pass call on the new object:

```
# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)
```

And finally add the backward step and printing gradients:

```
# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

# Print gradients
print(dense1.dweights)
print(dense1.dbiases)
print(dense2.dweights)
print(dense2.dbiases)
```

Full model code:

```
# 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 (output values)
dense2 = Layer_Dense(3, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

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

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)
```

```
# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)

# Let's see output of the first few samples:
print(loss_activation.output[:5])

# Print loss value
print('loss:', loss)

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

# Print accuracy
print('acc:', accuracy)

# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

# Print gradients
print(dense1.dweights)
print(dense1.dbiases)
print(dense2.dweights)
print(dense2.dbiases)


>>>
[[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
acc: 0.34
[[ 1.5766358e-04  7.8368575e-05  4.7324404e-05]
 [ 1.8161036e-04  1.1045571e-05 -3.3096316e-05]]
[[-3.6055347e-04  9.6611722e-05 -1.0367142e-04]]
[[ 5.4410957e-05  1.0741142e-04 -1.6182236e-04]
 [-4.0791339e-05 -7.1678100e-05  1.1246944e-04]
 [-5.3011299e-05  8.5817286e-05 -3.2805994e-05]]
[[-1.0732794e-05 -9.4590941e-06  2.0027626e-05]]
```

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):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases

    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)


# ReLU activation
class Activation_ReLU:

    # Forward pass
    def forward(self, inputs):
```

```
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs
        self.output = np.maximum(0, inputs)

    # Backward pass
    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0


# Softmax activation
class Activation_Softmax:

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = 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

    # Backward pass
    def backward(self, dvalues):

        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in \
                enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output and
            jacobian_matrix = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
```

```
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix,
                                         single_dvalues)


# 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

        # Backward pass
        def backward(self, dvalues, y_true):

            # Number of samples
            samples = len(dvalues)
            # Number of labels in every sample
            # We'll use the first sample to count them
            labels = len(dvalues[0])

            # If labels are sparse, turn them into one-hot vector
            if len(y_true.shape) == 1:
                y_true = np.eye(labels)[y_true]

            # Calculate gradient
            self.dinputs = -y_true / dvalues
            # Normalize gradient
            self.dinputs = self.dinputs / samples


    # Softmax classifier - combined Softmax activation
    # and cross-entropy loss for faster backward step
    class Activation_Softmax_Loss_CategoricalCrossentropy():

        # Creates activation and loss function objects
        def __init__(self):
            self.activation = Activation_Softmax()
            self.loss = Loss_CategoricalCrossentropy()

        # Forward pass
        def forward(self, inputs, y_true):
            # Output layer's activation function
            self.activation.forward(inputs)
            # Set the output
            self.output = self.activation.output
            # Calculate and return loss value
            return self.loss.calculate(self.output, y_true)
```

```
    # Backward pass
    def backward(self, dvalues, y_true):

        # Number of samples
        samples = len(dvalues)

        # If labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)

        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# 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 (output values)
dense2 = Layer_Dense(3, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

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

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)
```

```
# Let's see output of the first few samples:
print(loss_activation.output[:5])

# Print loss value
print('loss:', loss)

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

# Print accuracy
print('acc:', accuracy)

# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

# Print gradients
print(dense1.dweights)
print(dense1.dbiases)
print(dense2.dweights)
print(dense2.dbiases)
```

At this point, thanks to gradients and backpropagation using the chain rule, we're able to adjust the weights and biases with the goal of lowering loss, but we'd be doing it in a very rudimentary way. This process of adjusting weights and biases using gradients to decrease loss is the job of the optimizer, which is the subject of the next chapter.

Supplementary Material: https://nnfs.io/ch9
Chapter code, further resources, and errata for this chapter.

Chapter 10

Optimizers

Once we have calculated the gradient, we can use this information to adjust weights and biases to decrease the measure of loss. In a previous toy example, we showed how we could successfully decrease a neuron's activation function's (ReLU) output in this manner. Recall that we subtracted a fraction of the gradient for each weight and bias parameter. While very rudimentary, this is still a commonly used optimizer called **Stochastic Gradient Descent (SGD)**. As you will soon discover, most optimizers are just variants of SGD.

Stochastic Gradient Descent (SGD)

There are some naming conventions with this optimizer that can be confusing, so let's walk through those first. You might hear the following names:

- Stochastic Gradient Descent, SGD
- Vanilla Gradient Descent, Gradient Descent, GD, or Batch Gradient Descent, BGD
- Mini-batch Gradient Descent, MBGD

The first name, **Stochastic Gradient Descent**, historically refers to an optimizer that fits a single sample at a time. The second optimizer, **Batch Gradient Descent**, is an optimizer used to fit a whole dataset at once. The last optimizer, **Mini-batch Gradient Descent**, is used to fit slices of a dataset, which we'd call batches in our context. The naming convention can be confusing here for multiple reasons.

First, in the context of deep learning and this book, we call slices of data **batches**, where, historically, the term to refer to slices of data in the context of Stochastic Gradient Descent was **mini-batches**. In our context, it does not matter if the batch contains a single sample, a slice of the dataset, or the full dataset — as a batch of the data. Additionally, with the current code, we are fitting the full dataset; following this naming convention, we would use **Batch Gradient Descent**. In a future chapter, we'll introduce data slices, or **batches**, so we should start by using the **Mini-batch Gradient Descent** optimizer. That said, current naming trends and conventions with Stochastic Gradient Descent in use with deep learning today have merged and normalized all of these variants, to the point where we think of the **Stochastic Gradient Descent** optimizer as one that assumes a batch of data, whether that batch happens to be a single sample, every sample in a dataset, or some subset of the full dataset at a time.

In the case of Stochastic Gradient Descent, we choose a learning rate, such as *1.0*. We then subtract the *learning_rate · parameter_gradients* from the actual parameter values. If our learning rate is 1, then we're subtracting the exact amount of gradient from our parameters. We're going to start with 1 to see the results, but we'll be diving more into the learning rate shortly. Let's create the SGD optimizer class code. The initialization method will take hyper-parameters, starting with the learning rate, for now, storing them in the class' properties. The `update_params` method, given a layer object, performs the most basic optimization, the same way that we performed it in the previous chapter — it multiplies the gradients stored in the layers by the negated learning rate

and adds the result to the layer's parameters. It seems that, in the previous chapter, we performed SGD optimization without knowing it. The full class so far:

```
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1.0):
        self.learning_rate = learning_rate

    # Update parameters
    def update_params(self, layer):
        layer.weights += -self.learning_rate * layer.dweights
        layer.biases += -self.learning_rate * layer.dbiases
```

To use this, we need to create an optimizer object:

```
optimizer = Optimizer_SGD()
```

Then update our network layer's parameters after calculating the gradient using:

```
optimizer.update_params(dense1)
optimizer.update_params(dense2)
```

Recall that the layer object contains its parameters (weights and biases) and also, at this stage, the gradient that is calculated during backpropagation. We store these in the layer's properties so that the optimizer can make use of them. In our main neural network code, we'd bring the optimization in after backpropagation. Let's make a 1x64 densely-connected neural network (1 hidden layer with 64 neurons) and use the same dataset as before:

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()
```

The next step is to create the optimizer's object:

```
# Create optimizer
optimizer = Optimizer_SGD()
```

Then perform a **forward pass** of our sample data:

```
# Perform a forward pass of our training data through this layer
dense1.forward(X)

# Perform a forward pass through activation function
# takes the output of first dense layer here
activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function
# takes the output of second dense layer here and returns loss
loss = loss_activation.forward(dense2.output, y)

# Let's print loss value
print('loss:', loss)

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

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

Next, we do our **backward pass**, which is also called **backpropagation**:

```
# Backward pass
loss_activation.backward(loss_activation.output, y)
dense2.backward(loss_activation.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)
```

Then we finally use our optimizer to update weights and biases:

```
# Update weights and biases
optimizer.update_params(dense1)
optimizer.update_params(dense2)
```

This is everything we need to train our model! But why would we only perform this optimization once, when we can perform it lots of times by leveraging Python's looping capabilities? We will repeatedly perform a forward pass, backward pass, and optimization until we reach some stopping point. Each full pass through all of the training data is called an **epoch**. In most deep learning tasks, a neural network will be trained for multiple epochs, though the ideal scenario would be to have a perfect model with ideal weights and biases after only one epoch. To add multiple epochs of training into our code, we will initialize our model and run a loop around all the code performing the forward pass, backward pass, and optimization calculations:

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD()

# Train in loop
for epoch in range(10001):

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)
```

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

        if not epoch % 100:
            print(f'epoch: {epoch}, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f}')

        # Backward pass
        loss_activation.backward(loss_activation.output, y)
        dense2.backward(loss_activation.dinputs)
        activation1.backward(dense2.dinputs)
        dense1.backward(activation1.dinputs)

        # Update weights and biases
        optimizer.update_params(dense1)
        optimizer.update_params(dense2)
```

This gives us an update of where we are (epochs), the model's accuracy, and loss every 100 epochs. Initially, we can see consistent improvement:

```
epoch: 0, acc: 0.360, loss: 1.099
epoch: 100, acc: 0.400, loss: 1.087
epoch: 200, acc: 0.417, loss: 1.077
...
epoch: 1000, acc: 0.407, loss: 1.058
...
epoch: 2000, acc: 0.403, loss: 1.038
epoch: 2100, acc: 0.447, loss: 1.022
epoch: 2200, acc: 0.467, loss: 1.023
epoch: 2300, acc: 0.437, loss: 1.005
epoch: 2400, acc: 0.497, loss: 0.993
epoch: 2500, acc: 0.513, loss: 0.981
...
epoch: 9500, acc: 0.590, loss: 0.865
epoch: 9600, acc: 0.627, loss: 0.863
epoch: 9700, acc: 0.630, loss: 0.830
epoch: 9800, acc: 0.663, loss: 0.844
epoch: 9900, acc: 0.627, loss: 0.820
epoch: 10000, acc: 0.633, loss: 0.848
```

Additionally, we've prepared animations to help visualize the training process and to convey the impact of various optimizers and their hyperparameters. The left part of the animation canvas

contains dots, where color represents each of the 3 classes of the data, the coordinates are features, and the background colors show the model prediction areas. Ideally, the points' colors and the background should match if the model classifies correctly. The surrounding area should also follow the data's "trend" — which is what we'd call generalization — the ability of the model to correctly predict unseen data. The colorful squares on the right show weights and biases — red for positive and blue for negative values. The matching areas right below the Dense 1 bar and next to the Dense 2 bar show the updates that the optimizer performs to the layers. The updates might look overly strong compared to the weights and biases, but that's because we've visually normalized them to the maximum value, or else they would be almost invisible since the updates are quite small at a time. The other 3 graphs show the loss, accuracy, and current learning rate values in conjunction with the training time, epochs in this case.

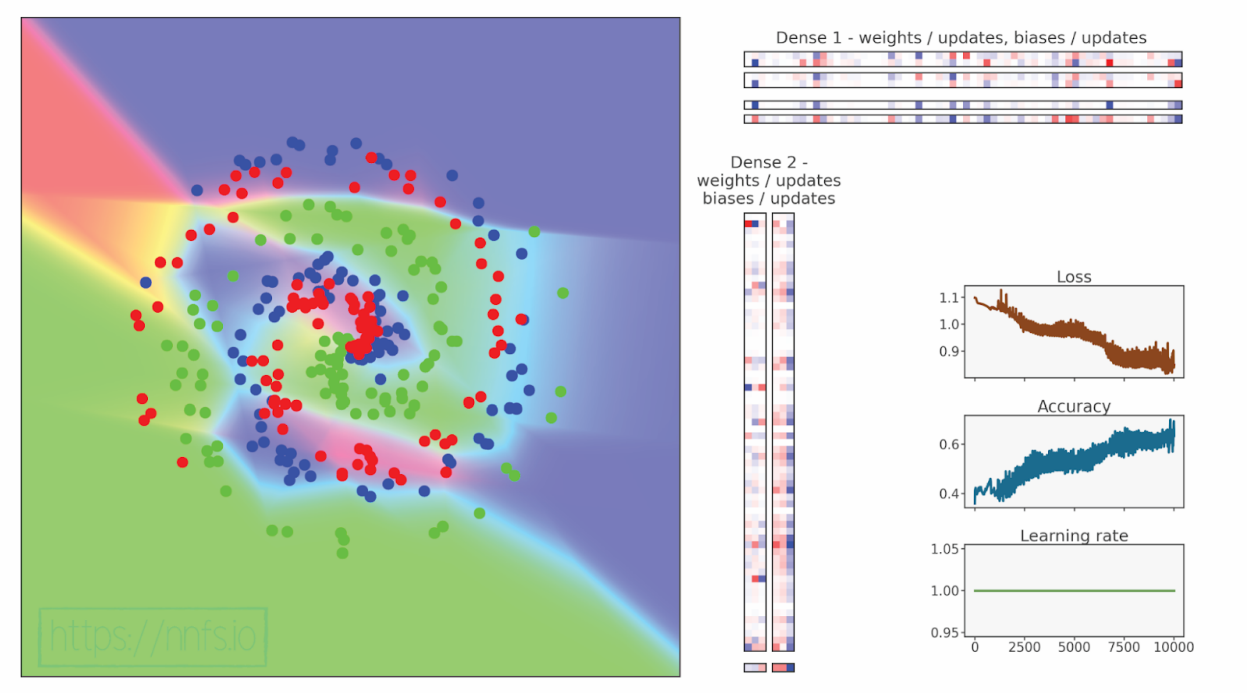


Fig 10.01: Model training with Stochastic Gradient Descent optimizer.

Epilepsy Warning, there are quick flashing colors in the animation:

Anim 10.01: https://nnfs.io/pup

Our neural network mostly stays stuck at around a loss of 1 and later 0.85-0.9, and an accuracy around 0.60. The animation also has a “flashy wiggle” effect, which most likely means we chose too high of a learning rate. Given that loss didn’t decrease much, we can assume that this learning rate, being too high, also caused the model to get stuck in a **local minimum**, which we’ll learn more about soon. Iterating over more epochs doesn’t seem helpful at this point, which tells us that we’re likely stuck with our optimization. Does this mean that this is the most we can get from our optimizer on this dataset?

Recall that we’re adjusting our weights and biases by applying some fraction, in this case, *1.0*, to the gradient and subtracting this from the weights and biases. This fraction is called the **learning rate** (LR) and is the primary adjustable parameter for the optimizer as it decreases loss. To gain an intuition for adjusting, planning, or initially setting the learning rate, we should first understand how the learning rate affects the optimizer and output of the loss function.

Learning Rate

So far, we have a gradient of a model and the loss function with respect to all of the parameters, and we want to apply a fraction of this gradient to the parameters in order to descend the loss value.

In most cases, we won't apply the negative gradient as is, as the direction of the function's steepest descent will be continuously changing, and these values will usually be too big for meaningful model improvements to occur. Instead, we want to perform small steps — calculating the gradient, updating parameters by a negative fraction of this gradient, and repeating this in a loop. Small steps ensure that we are following the direction of the steepest descent, but these steps can also be too small, causing learning stagnation — we'll explain this shortly.

Let's forget, for a while, that we are performing gradient descent of an n-dimensional function (our loss function), where n is the number parameters (weights and biases) that the model contains, and assume that we have just one dimension to the loss function (a singular input). Our goal for the following images and animations is to visualize some concepts and gain an intuition; thus, we will not use or present certain optimizer settings, and instead will be considering things in more general terms. That said, we've used a real SGD optimizer on a real function to prepare all of the following examples. Here's the function where we want to determine what input to it will result in the lowest possible output:

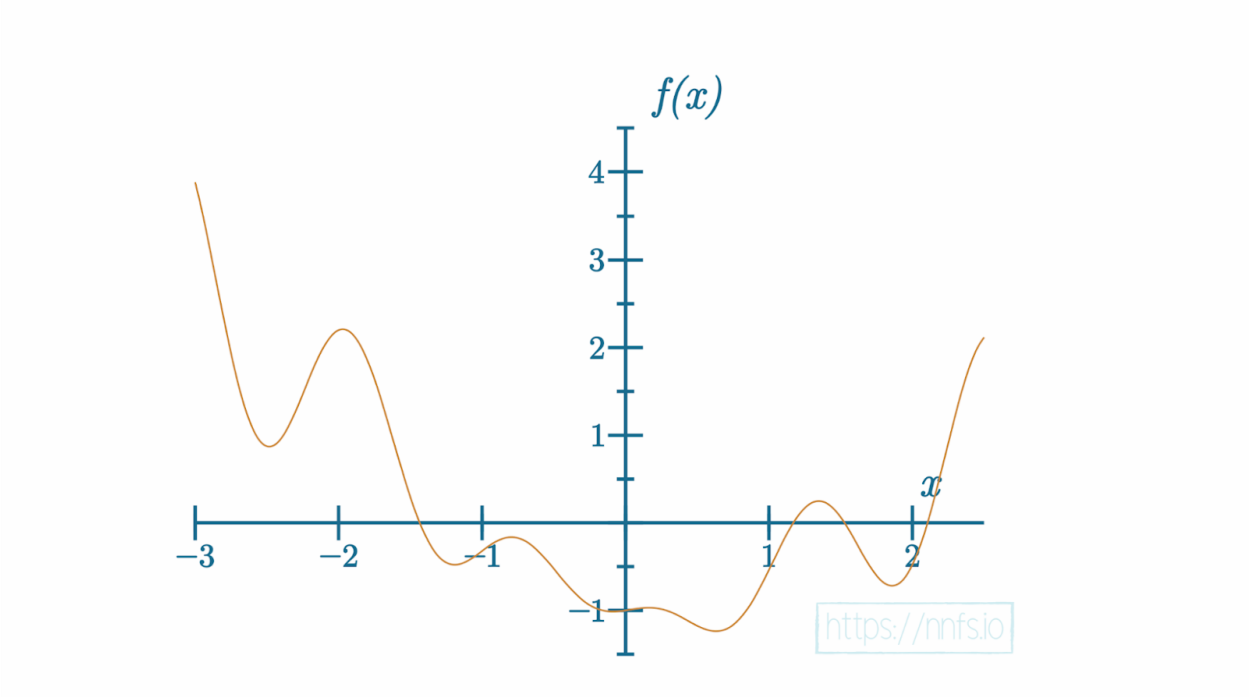


Fig 10.02: Example function to minimize the output.

We can see the **global minimum** of this function, which is the lowest possible *y* value that this function can output. This is the goal — to minimize the function's output to find the global minimum. The values of the axes are not important in this case. The goal is only to show the function and the learning rate concept. Also, remember that this one-dimensional function example is being used merely to aid in visualization. It would be easy to solve this function with simpler math than what is required to solve the much larger n-dimensional loss function for neural networks, where n (which is the number of weights and biases) can be in the millions or even billions (or more). When we have millions of, or more, dimensions, gradient descent is the best-known way to search for a global minimum.

We'll start descending from the left side of this graph. With an example learning rate:

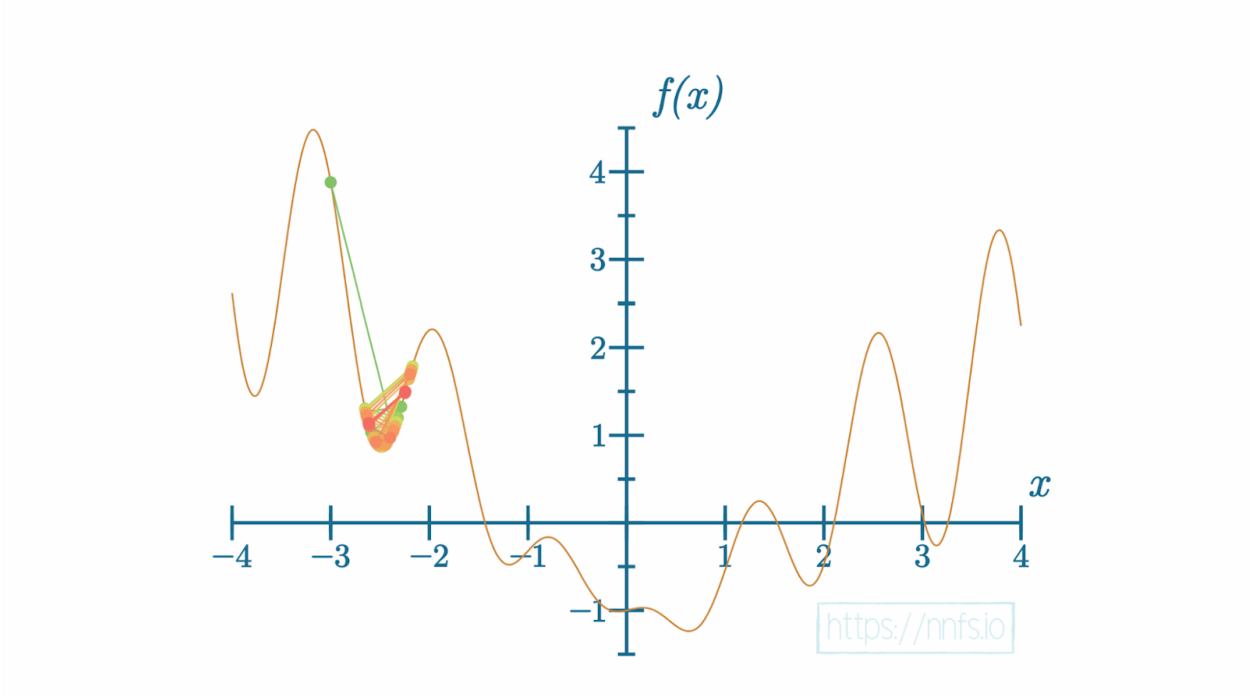


Fig 10.03: Stuck in the first local minimum.

Anim 10.03: https://nnfs.io/and

The learning rate turned out to be too small. Small updates to the parameters caused stagnation in the model's learning — the model got stuck in a local minimum. The **local minimum** is a minimum that is near where we look but isn't necessarily the global minimum, which is the absolute lowest point for a function. With our example here, as well as with optimizing full neural networks, we do not know where the global minimum is. How do we know if we've reached the global minimum or at least gotten close? The loss function measures how far the model is with its predictions to the real target values, so, as long as the loss value is not *0* or very close to *0*, and the model stopped learning, we're at some local minimum. In reality, we almost never approach a loss of *0* for various reasons. One reason for this may be imperfect neural network hyperparameters. Another reason for this may be insufficient data. If you did reach a loss of 0 with a neural network, you should find it suspicious, for reasons we'll get into later in this book.

We can try to modify the learning rate:

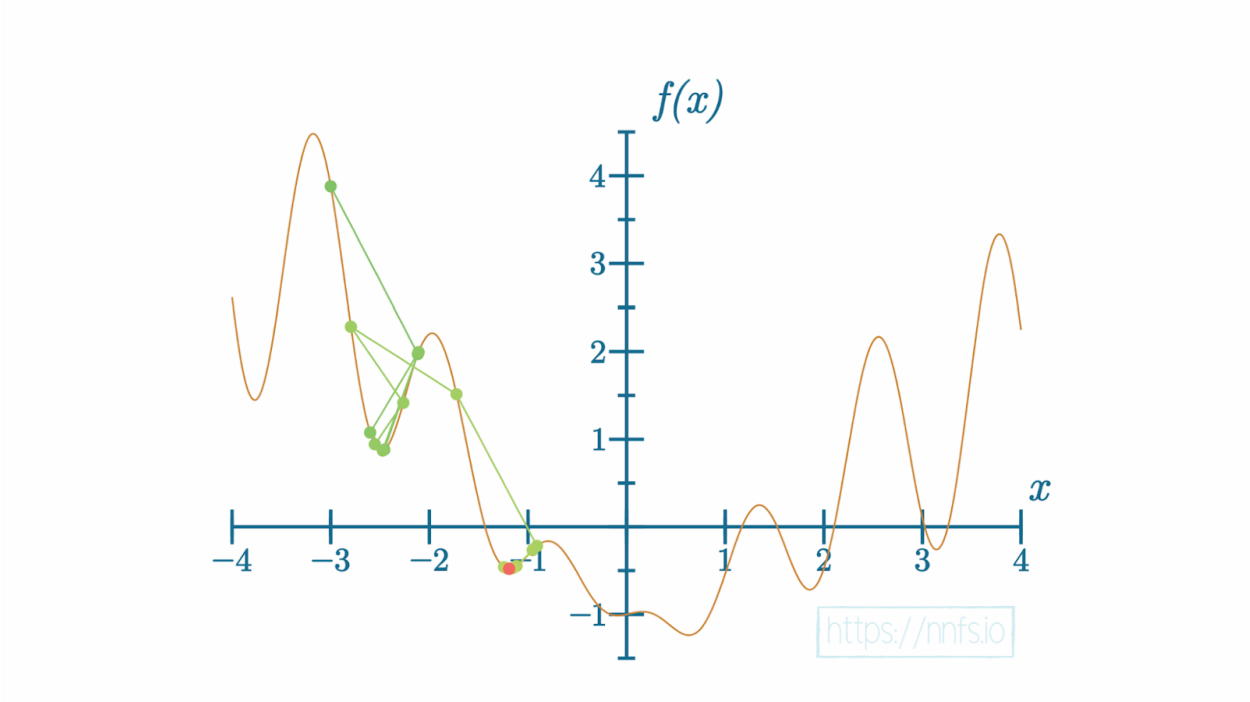


Fig 10.04: Stuck in the second local minimum.

Anim 10.04: https://nnfs.io/xor

This time, the model escaped this local minimum but got stuck at another one. Let's see one more example after another learning rate change:

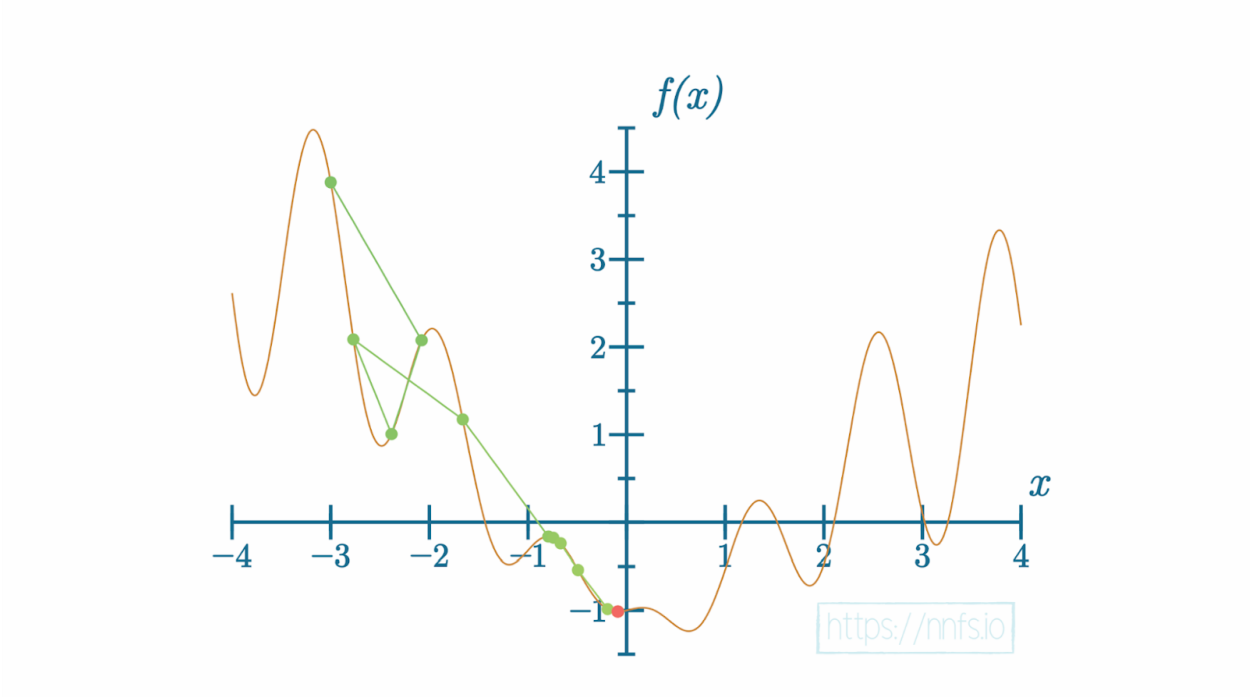


Fig 10.05: Stuck in the third local minimum, near the global minimum.

Anim 10.05: https://nnfs.io/tho

This time the model got stuck at a local minimum near the global minimum. The model was able to escape the “deeper” local minimums, so it might be counter-intuitive why it is stuck here. Remember, the model follows the direction of steepest descent of the loss function, no matter how large or slight the descent is. For this reason, we’ll introduce momentum and the other techniques to prevent such situations.

Momentum, in an optimizer, adds to the gradient what, in the physical world, we could call inertia — for example, we can throw a ball uphill and, with a small enough hill or big enough applied force, the ball can roll-over to the other side of the hill. Let's see how this might look with the model in training:

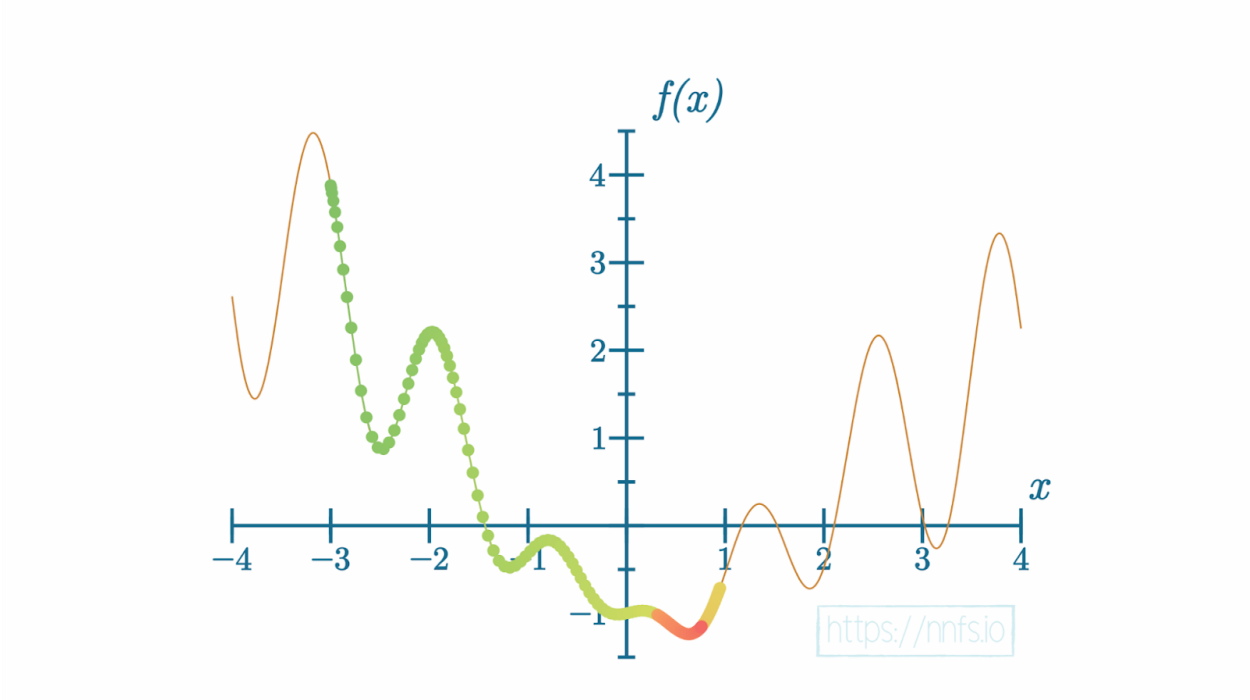


Fig 10.06: Reached the global minimum, too low learning rate.

Anim 10.06: https://nnfs.io/pog

We used a very small learning rate here with a large momentum. The color change from green, through orange to red presents the advancement of the gradient descent process, the steps. We can see that the model achieved the goal and found the global minimum, but this took many steps. Can this be done better?

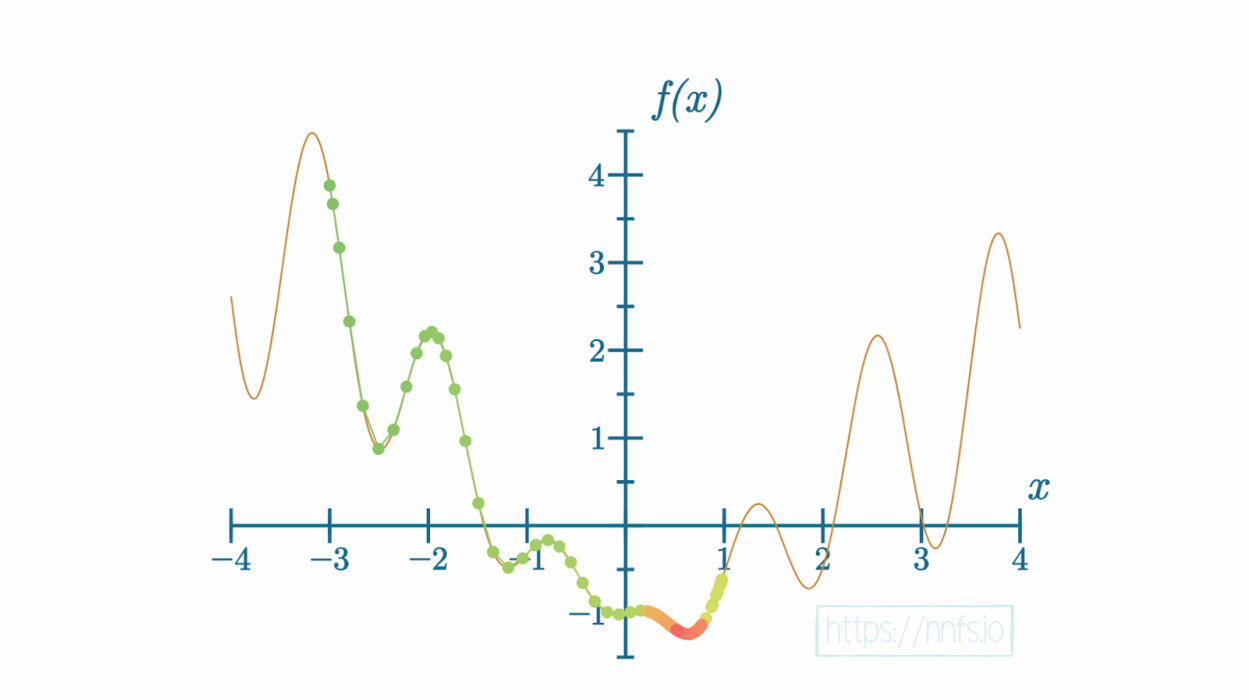


Fig 10.07: Reached the global minimum, better learning rate.

Anim 10.07: https://nnfs.io/jog

And even further:

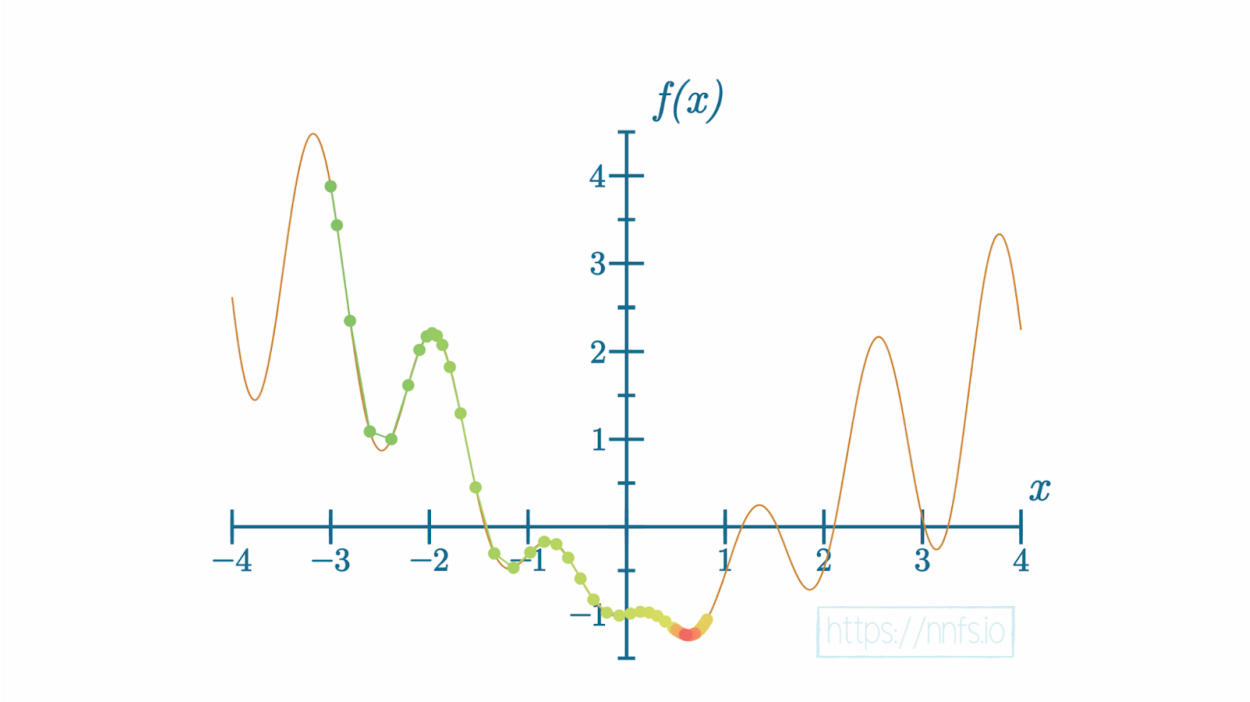


Fig 10.08: Reached the global minimum, significantly better learning rate.

Anim 10.08: https://nnfs.io/mog

With these examples, we were able to find the global minimum in about 200, 100, and 50 steps, respectively, by modifying the learning rate and the momentum. It's possible to significantly shorten the training time by adjusting the parameters of the optimizer. However, we have to be careful with these hyper-parameter adjustments, as this won't necessarily always help the model:

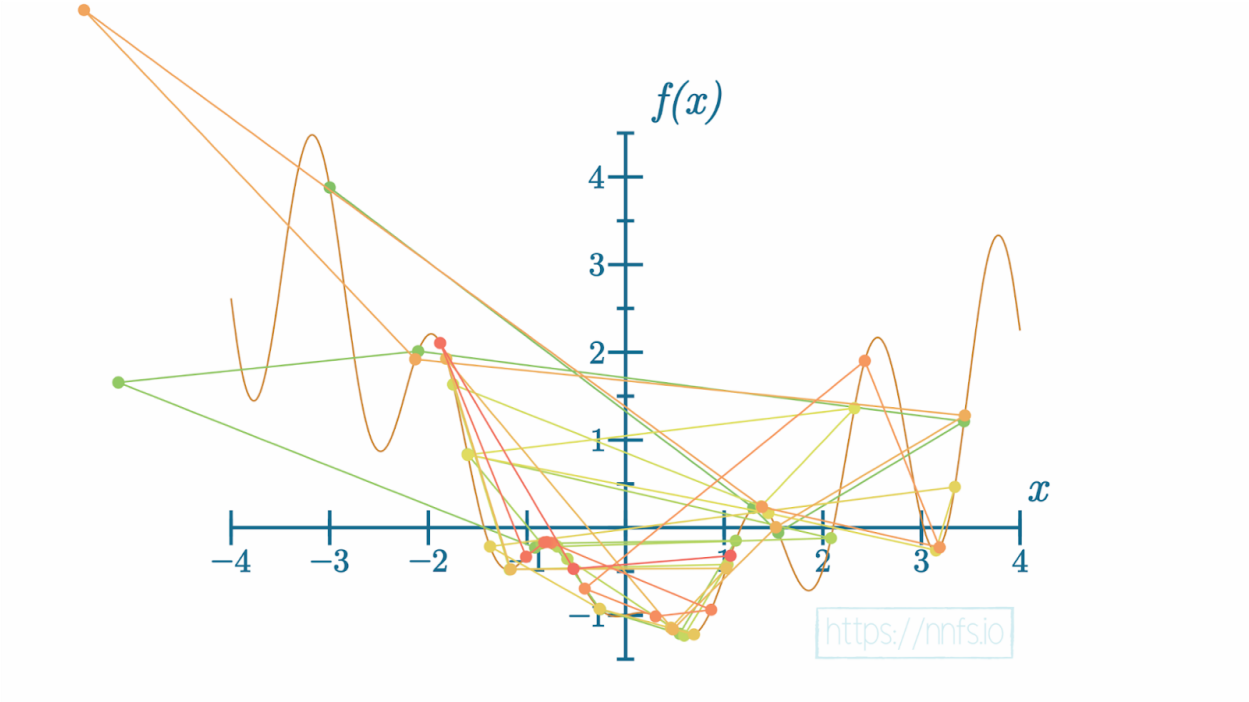


Fig 10.09: Unstable model, learning rate too big.

Anim 10.09: https://nnfs.io/log

With the learning rate set too high, the model might not be able to find the global minimum. Even, at some point, if it does, further adjustments could cause it to jump out of this minimum. We'll see this behavior later in this chapter — try to take a close look at results and see if you can find it, as well as the other issues we've described, from the different optimizers as we work through them.

In this case, the model was “jumping” around some minimum and what this might mean is that we should try to lower the learning rate, raise the momentum, or possibly apply a learning rate decay (lowering the learning rate during training), which we’ll describe in this chapter. If we set the learning rate far too high:

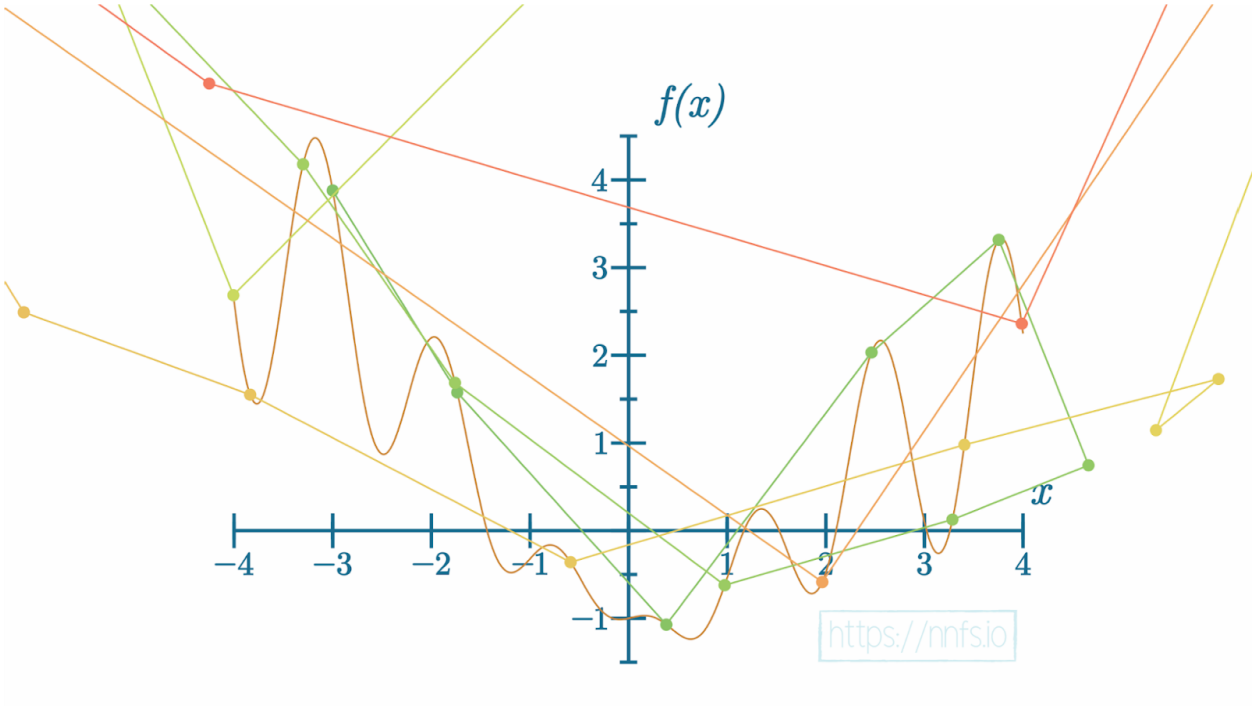


Fig 10.10: Unstable model, learning rate significantly too big.

Anim 10.10: https://nnfs.io/sog

In this situation, the model starts "jumping" around, and moves in what we might observe as random directions. This is an example of "overshooting," with every step — the direction of a change is correct, but the amount of the gradient applied is too large. In an extreme situation, we could cause a **gradient explosion**:

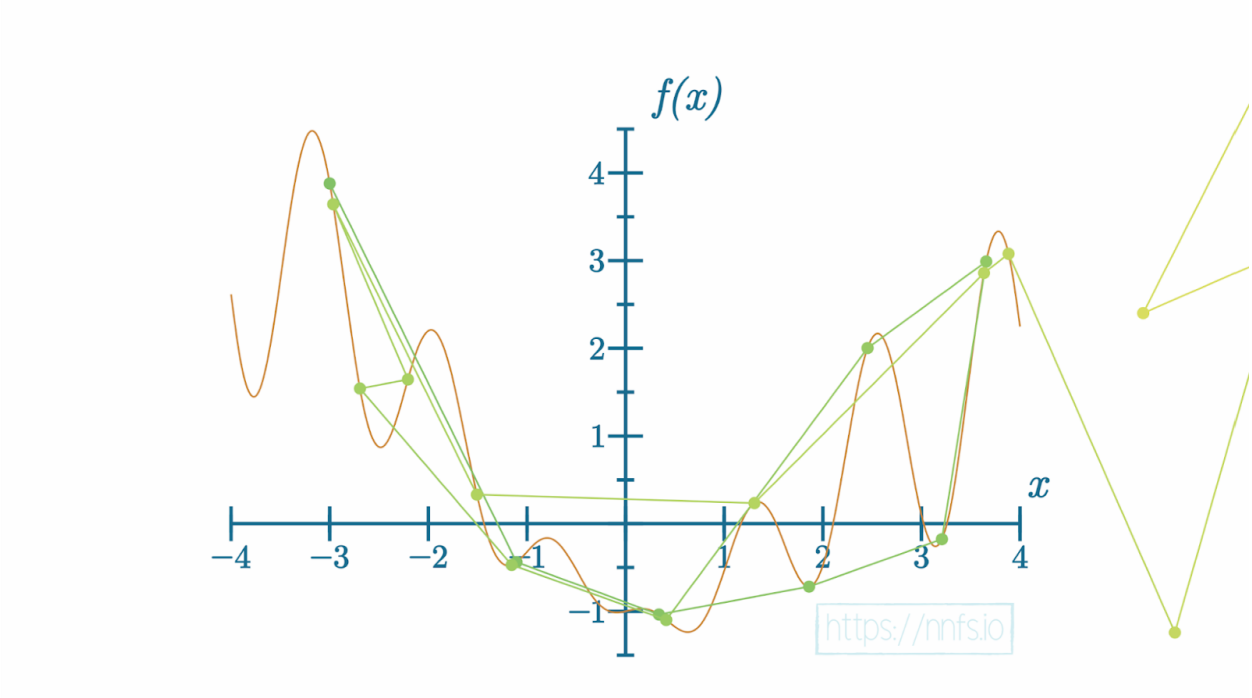


Fig 10.11: Broken model, learning rate critically too big.

Anim 10.11: https://nnfs.io/bog

A gradient explosion is a situation where the parameter updates cause the function's output to rise instead of fall, and, with each step, the loss value and gradient become larger. At some point, the floating-point variable limitation causes an overflow as it cannot hold values of this size anymore, and the model is no longer able to train. It's crucial to recognize this situation forming during training, especially for large models, where the training can take days, weeks, or more. It is possible to tune the model's hyper-parameters in time to save the model and to continue training.

When we choose the learning rate and the other hyper-parameters correctly, the learning process can be relatively quick:

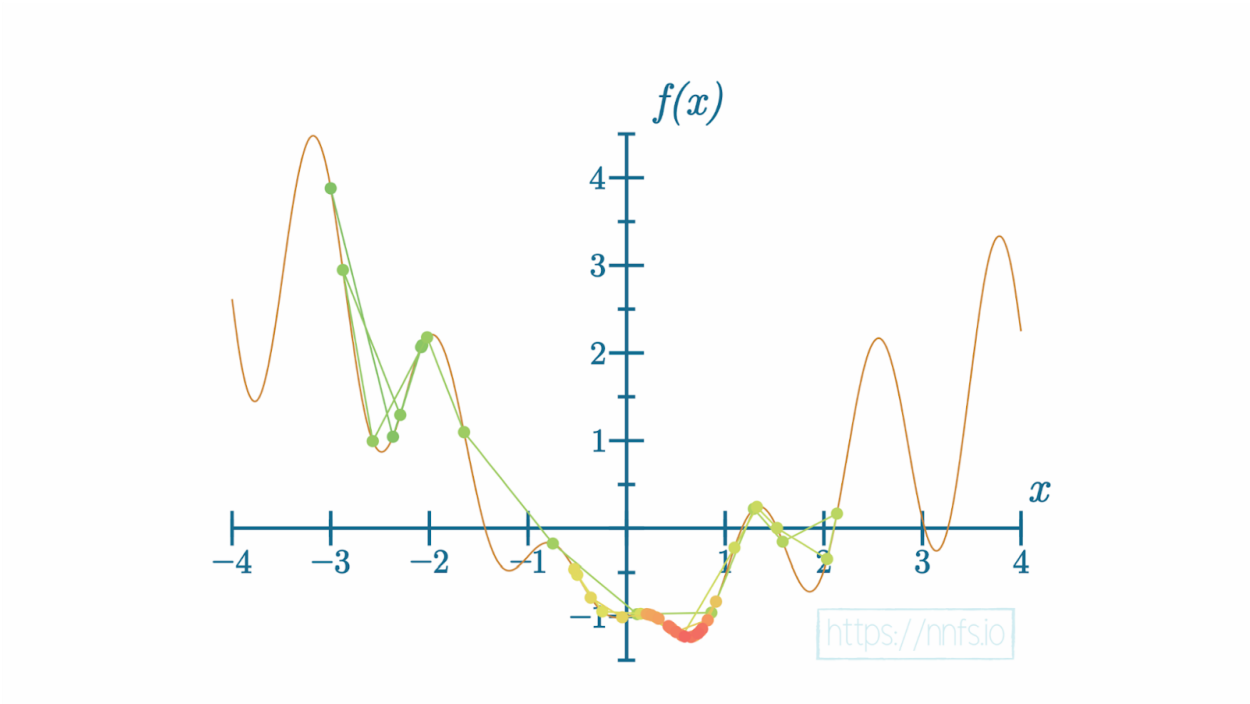


Fig 10.12: Model learned, good learning rate, can be better.

Anim 10.12: https://nnfs.io/cog

This time it took significantly less time for the model to find the global minimum, but it can always be better:

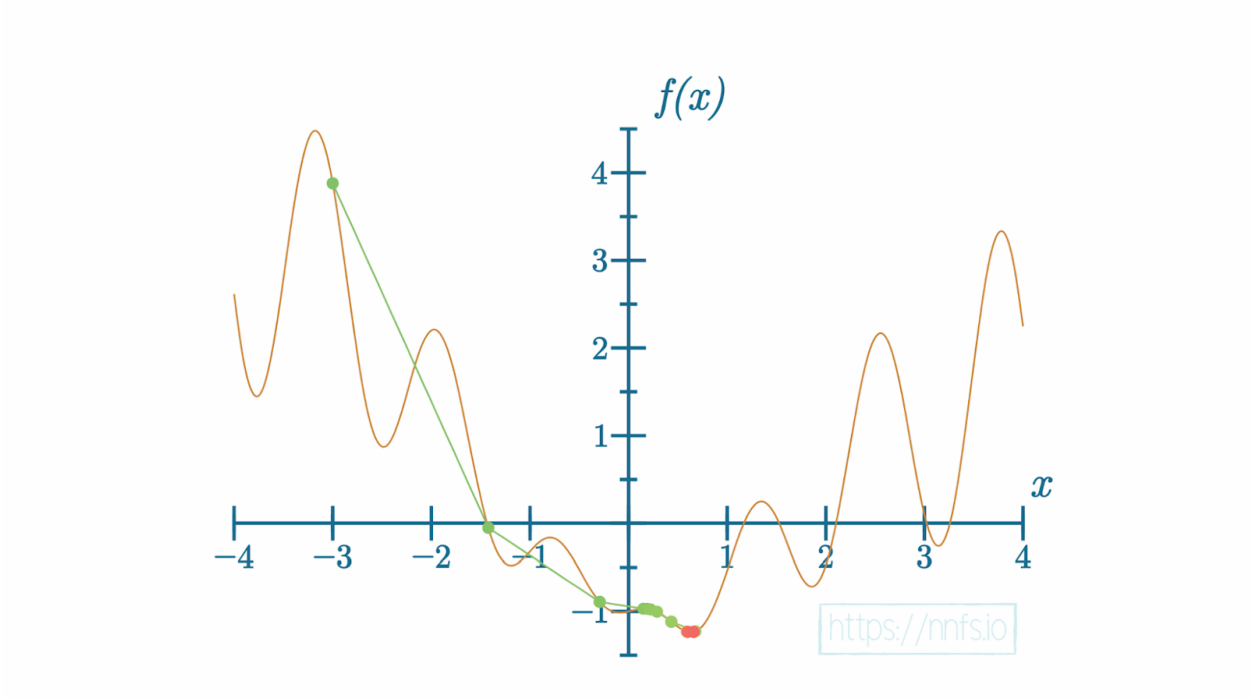


Fig 10.13: An efficient learning example.

Anim 10.13: https://nnfs.io/rog

This time the model needed just a few steps to find the global minimum. The challenge is to choose the hyper-parameters correctly, and it is not always an easy task. It is usually best to start with the optimizer defaults, perform a few steps, and observe the training process when tuning different settings. It is not always possible to see meaningful results in a short-enough period of time, and, in this case, it's good to have the ability to update the optimizer's settings during training. How you choose the learning rate, and other hyper-parameters, depends on the model, data, including the amount of data, the parameter initialization method, etc. There is no single, best way to set hyper-parameters, but experience usually helps. As we mentioned, one

of the challenges during the training of a neural network model is to choose the right settings. The

difference can be anything from a model not learning at all to learning very well.

For a summary of learning rates — if we plot the loss along an axis of steps:

Fig 10.14: Graphs of the loss in a function of steps, different rates

We can see various examples of relative learning rates and what loss will ideally look like as a graph over time (steps) of training.

Knowing what the learning rate should be to get the most out of your training process isn't possible, but a good rule is that your initial training will benefit from a larger learning rate to take initial steps faster. If you start with steps that are too small, you might get stuck in a local minimum and be unable to leave it due to not making large enough updates to the parameters. For example, what if we make the learning rate 0.85 rather than 1.0 with the SGD optimizer?

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()
```

```
# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(learning_rate=.85)

# Train in loop
for epoch in range(10001):

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

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

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099
epoch: 100, acc: 0.403, loss: 1.091
...
epoch: 2000, acc: 0.437, loss: 1.053
epoch: 2100, acc: 0.443, loss: 1.026
epoch: 2200, acc: 0.377, loss: 1.050
epoch: 2300, acc: 0.433, loss: 1.016
epoch: 2400, acc: 0.460, loss: 1.000
epoch: 2500, acc: 0.493, loss: 1.010
epoch: 2600, acc: 0.527, loss: 0.998
epoch: 2700, acc: 0.523, loss: 0.977
...
epoch: 7100, acc: 0.577, loss: 0.941
epoch: 7200, acc: 0.550, loss: 0.921
epoch: 7300, acc: 0.593, loss: 0.943
epoch: 7400, acc: 0.593, loss: 0.940
epoch: 7500, acc: 0.557, loss: 0.907
epoch: 7600, acc: 0.590, loss: 0.949
epoch: 7700, acc: 0.590, loss: 0.935
...
epoch: 9100, acc: 0.597, loss: 0.860
epoch: 9200, acc: 0.630, loss: 0.842
...
epoch: 10000, acc: 0.657, loss: 0.816
```

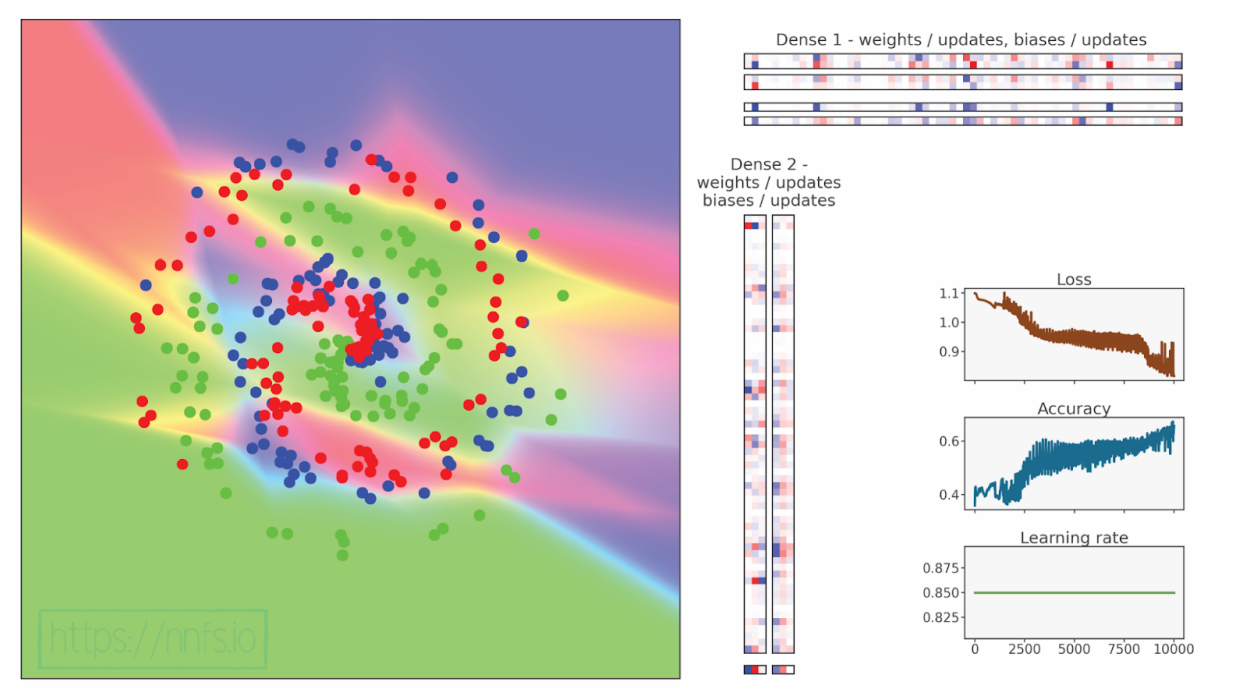


Fig 10.15: Model training with SGD optimizer and lowered learning rate.

Epilepsy Warning (quick flashing colors).

Anim 10.15: https://nnfs.io/cup

As you can see, the neural network did slightly better in terms of accuracy, and it achieved a lower loss; lower loss is not always associated with higher accuracy. Remember, even if we desire the best accuracy out of our model, the optimizer's task is to decrease loss, not raise accuracy directly. Loss is the mean value of all of the sample losses, and some of them could drop significantly, while others might rise just slightly, changing the prediction for them from a correct to an incorrect class at the same time. This would cause a lower mean loss in general, but also more incorrectly predicted samples, which will, at the same time, lower the accuracy. A likely reason for this model's lower accuracy is that it found another local minimum by chance — the descent path has changed, due to smaller steps. In a direct comparison of these two models in training, different learning rates did not show that the lower this learning rate value is, the better. In most cases, we want to start with a larger learning rate and decrease the learning rate over time/steps.

A commonly-used solution to keep initial updates large and explore various learning rates during training is to implement a **learning rate decay**.

Learning Rate Decay

The idea of a **learning rate decay** is to start with a large learning rate, say 1.0 in our case, and then decrease it during training. There are a few methods for doing this. One is to decrease the learning rate in response to the loss across epochs — for example, if the loss begins to level out/ plateau or starts "jumping" over large deltas. You can either program this behavior-monitoring logically or simply track your loss over time and manually decrease the learning rate when you deem it appropriate. Another option, which we will implement, is to program a **Decay Rate**, which steadily decays the learning rate per batch or epoch.

Let's plan to decay per step. This can also be referred to as **1/t decaying** or **exponential decaying**. Basically, we're going to update the learning rate each step by the reciprocal of the step count fraction. This fraction is a new hyper-parameter that we'll add to the optimizer, called the **learning rate decay**. How this decaying works is it takes the step and the decaying ratio and multiplies them. The further in training, the bigger the step is, and the bigger result of this multiplication is. We then take its reciprocal (the further in training, the lower the value) and multiply the initial learning rate by it. The added *1* makes sure that the resulting algorithm never raises the learning rate. For example, for the first step, we might divide 1 by the learning rate, *0.001* for example, which will result in a current learning rate of *1000*. That's definitely not what we wanted. 1 divided by the 1+fraction ensures that the result, a fraction of the starting learning rate, will always be less than or equal to 1, decreasing over time. That's the desired result — start with the current learning rate and make it smaller with time. The code for determining the current decay rate:

```
starting_learning_rate = 1.
learning_rate_decay = 0.1
step = 1

learning_rate = starting_learning_rate * \
                (1. / (1 + learning_rate_decay * step))
print(learning_rate)


>>>
0.9090909090909091
```

In practice, 0.1 would be considered a fairly aggressive decay rate, but this should give you a sense of the concept. If we are on step 20:

```
starting_learning_rate = 1.
learning_rate_decay = 0.1
step = 20

learning_rate = starting_learning_rate * \
                (1. / (1 + learning_rate_decay * step))
print(learning_rate)


>>>
0.3333333333333333
```

We can also simulate this in a loop, which is more comparable to how we will be applying learning rate decay:

```
starting_learning_rate = 1.
learning_rate_decay = 0.1

for step in range(20):
    learning_rate = starting_learning_rate * \
                    (1. / (1 + learning_rate_decay * step))
    print(learning_rate)


>>>
1.0
0.9090909090909091
0.8333333333333334
0.7692307692307692
0.7142857142857143
0.6666666666666666
0.625
0.588235294117647
0.5555555555555556
0.5263157894736842
0.5
0.47619047619047616
0.45454545454545453
0.4347826086956522
0.41666666666666663
0.4
0.3846153846153846
0.37037037037037035
0.35714285714285715
0.3448275862068965
```

This learning rate decay scheme lowers the learning rate each step using the mentioned formula. Initially, the learning rate drops fast, but the change in the learning rate lowers each step, letting the model sit as close as possible to the minimum. The model needs small updates near the end of training to be able to get as close to this point as possible. We can now update our SGD optimizer class to allow for the learning rate decay:

```
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):
        layer.weights += -self.current_learning_rate * layer.dweights
        layer.biases += -self.current_learning_rate * layer.dbiases

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

We've updated a few things in the SGD class. First, in the `__init__` method, we added handling for the current learning rate, and `self.learning_rate` is now the initial learning rate. We also added attributes to track the decay rate and the number of iterations that the optimizer has gone through. Next, we added a new method called `pre_update_params`. This method, if we have a decay rate other than 0, will update our `self.current_learning_rate` using the prior formula. The `update_params` method remains unchanged, but we do have a new `post_update_params` method that will add to our `self.iterations` tracking. With our updated SGD optimizer class, we've added printing the current learning rate, and added pre and post optimizer method calls. Let's use a decay rate of 1e-2 (0.01) and train our model again:

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-2)

# Train in loop
for epoch in range(10001):

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

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

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')
```

```
    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.403, loss: 1.095, lr: 0.5025125628140703
epoch: 200, acc: 0.397, loss: 1.084, lr: 0.33444816053511706
epoch: 300, acc: 0.400, loss: 1.080, lr: 0.2506265664160401
epoch: 400, acc: 0.407, loss: 1.078, lr: 0.2004008016032064
epoch: 500, acc: 0.420, loss: 1.078, lr: 0.1669449081803005
epoch: 600, acc: 0.420, loss: 1.077, lr: 0.14306151645207438
epoch: 700, acc: 0.417, loss: 1.077, lr: 0.1251564455569462
epoch: 800, acc: 0.413, loss: 1.077, lr: 0.11123470522803114
epoch: 900, acc: 0.410, loss: 1.077, lr: 0.10010010010010009
epoch: 1000, acc: 0.417, loss: 1.077, lr: 0.09099181073703366
...
epoch: 2000, acc: 0.420, loss: 1.076, lr: 0.047641734159123386
...
epoch: 3000, acc: 0.413, loss: 1.075, lr: 0.03226847370119393
...
epoch: 4000, acc: 0.407, loss: 1.075, lr: 0.02439619419370578
...
epoch: 5000, acc: 0.403, loss: 1.074, lr: 0.019611688566385566
...
epoch: 7000, acc: 0.400, loss: 1.073, lr: 0.014086491055078181
...
epoch: 10000, acc: 0.397, loss: 1.072, lr: 0.009901970492127933
```

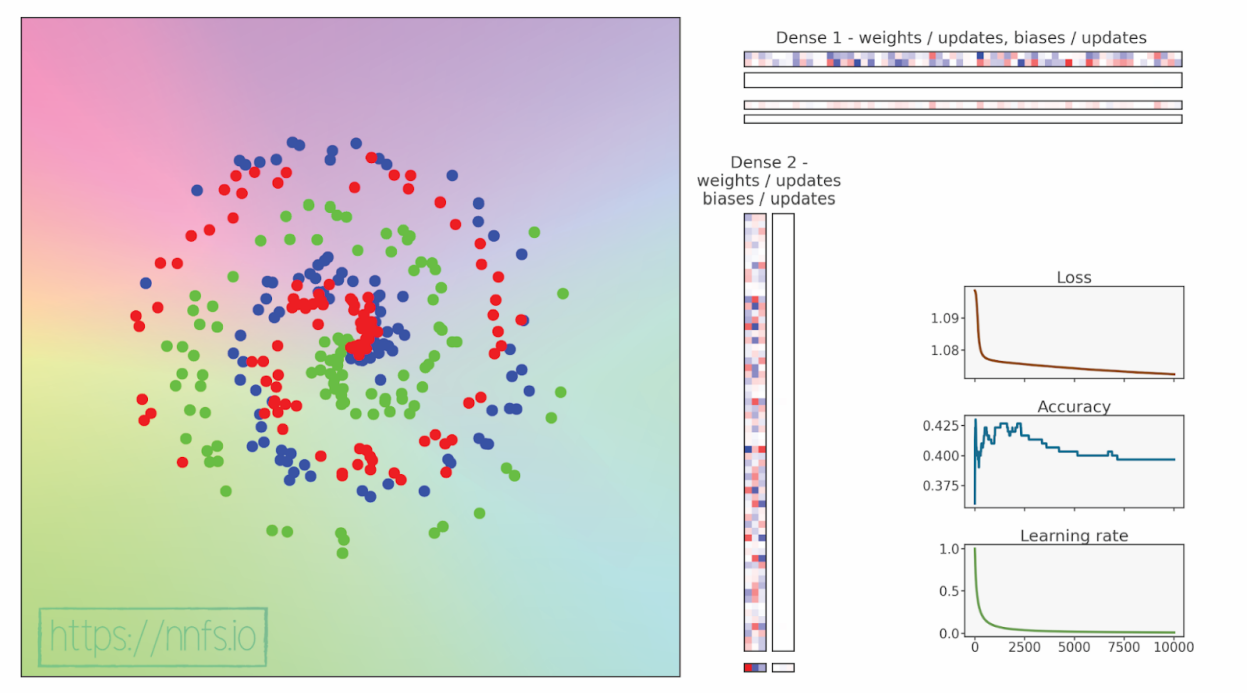


Fig 10.16: Model training with SGD optimizer and and learning rate decay set too high.

Epilepsy Warning (quick flashing colors)

Anim 10.16: https://nnfs.io/zuk

This model definitely got stuck, and the reason is almost certainly because the learning rate decayed far too quickly and became too small, trapping the model in some local minimum. This is most likely why, rather than wiggling, our accuracy and loss stopped changing *at all*.

We can, instead, try to decay a bit slower by making our decay a smaller number. For example, let's go with 1e-3 (0.001):

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-3)

# Train in loop
for epoch in range(10001):

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

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

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')
```

```
    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()


>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.400, loss: 1.088, lr: 0.9099181073703367
epoch: 200, acc: 0.423, loss: 1.078, lr: 0.8340283569641367
...
epoch: 1700, acc: 0.450, loss: 1.025, lr: 0.3705075954057058
epoch: 1800, acc: 0.470, loss: 1.017, lr: 0.35727045373347627
epoch: 1900, acc: 0.460, loss: 1.008, lr: 0.3449465332873405
epoch: 2000, acc: 0.463, loss: 1.000, lr: 0.33344448149383127
epoch: 2100, acc: 0.490, loss: 1.005, lr: 0.32268473701193934
...
epoch: 3200, acc: 0.493, loss: 0.983, lr: 0.23815194093831865
...
epoch: 5000, acc: 0.577, loss: 0.900, lr: 0.16669444907484582
...
epoch: 6000, acc: 0.633, loss: 0.860, lr: 0.1428775539362766
...
epoch: 8000, acc: 0.647, loss: 0.799, lr: 0.11112345816201799
...
epoch: 9800, acc: 0.663, loss: 0.773, lr: 0.09260116677470137
epoch: 9900, acc: 0.663, loss: 0.772, lr: 0.09175153683824203
epoch: 10000, acc: 0.667, loss: 0.771, lr: 0.09091735612328393
```

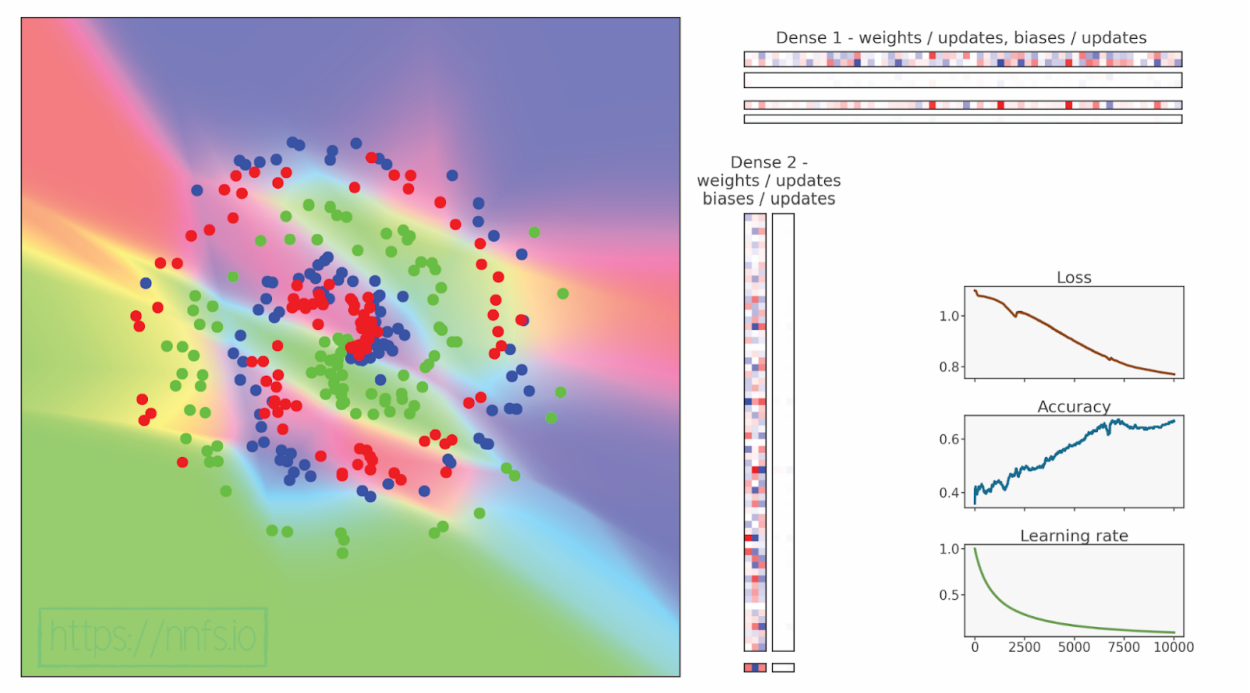


Fig 10.17: Model training with SGD optimizer and more proper learning rate decay.

Epilepsy Warning (quick flashing colors)

Anim 10.17: https://nnfs.io/muk

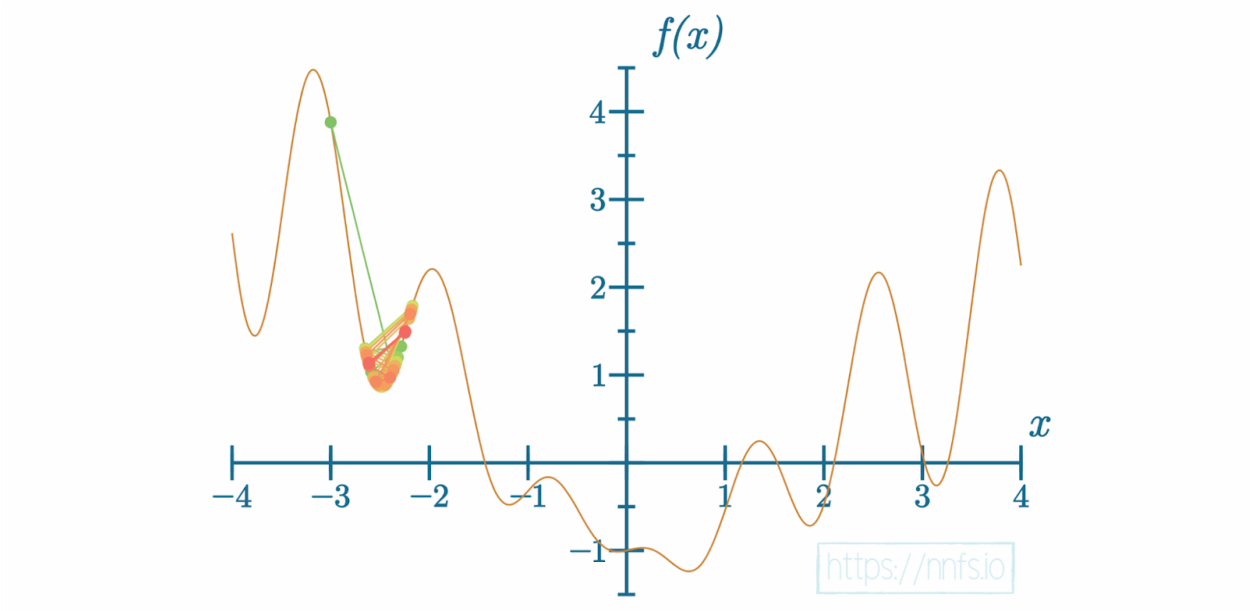
In this case, we've achieved our lowest loss and highest accuracy thus far, but it still should be possible to find parameters that will give us even better results. For example, you may suspect that the initial learning rate is too high. It can make for a great exercise to attempt to find better settings. Feel free to try!

Stochastic Gradient Descent with learning rate decay can do fairly well but is still a fairly basic optimization method that only follows a gradient without any additional logic that could potentially help the model find the **global minimum** to the loss function. One option for improving the SGD optimizer is to introduce **momentum**.

Stochastic Gradient Descent with Momentum

Momentum creates a rolling average of gradients over some number of updates and uses this average with the unique gradient at each step. Another way of understanding this is to imagine a ball going down a hill — even if it finds a small hole or hill, momentum will let it go straight through it towards a lower minimum — the bottom of this hill. This can help in cases where you're stuck in some local minimum (a hole), bouncing back and forth. With momentum, a model is more likely to pass through local minimums, further decreasing loss. Simply put, momentum may still point towards the global gradient descent direction.

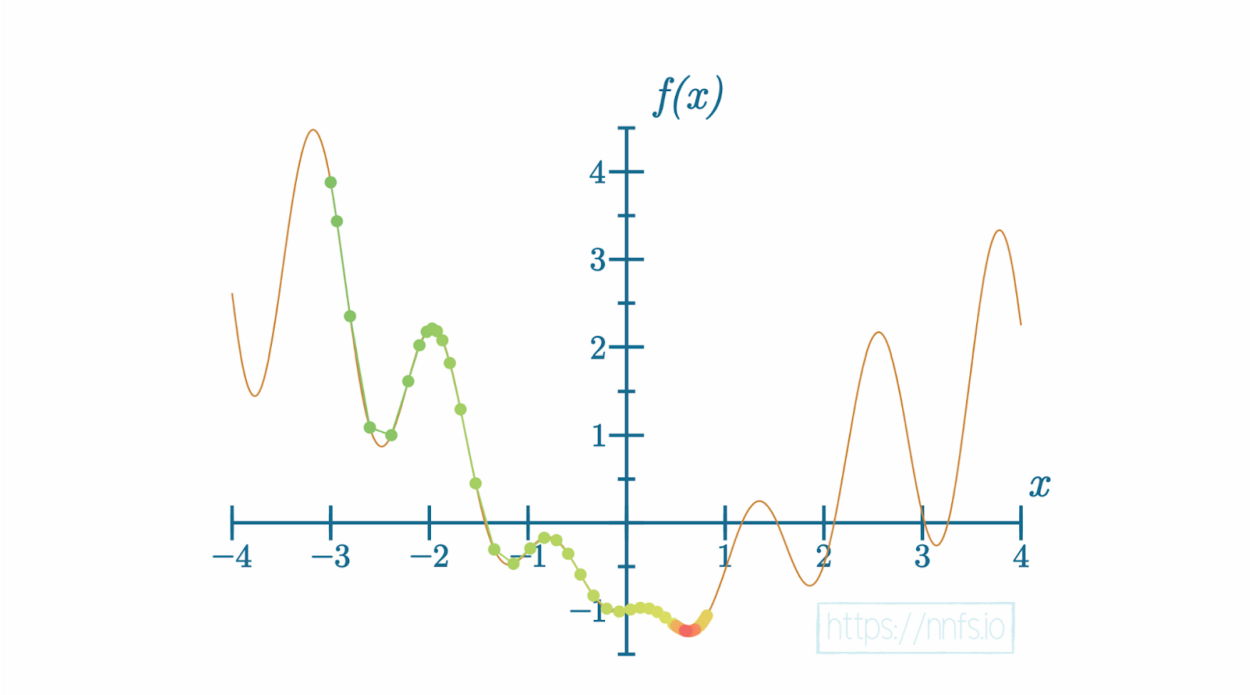
Recall this situation from the beginning of this chapter:



With regular updates, the SGD optimizer might determine that the next best step is one that keeps the model in a local minimum. Remember that the gradient points toward the current steepest loss ascent for that step — taking the negative of the gradient vector flips it toward the current steepest descent, which may not necessarily follow descent towards the global minimum — the current steepest descent may point towards a local minimum. So this step may decrease loss for that update but might not get us out of the local minimum. We might wind up with a gradient

that points in one direction and then the opposite direction in the next update; the gradient could continue to bounce back and forth around a local minimum like this, keeping the optimization of the loss stuck. Instead, momentum uses the previous update's direction to influence the next update's direction, minimizing the chances of bouncing around and getting stuck.

Recall another example shown in this chapter:



We utilize momentum by setting a parameter between 0 and 1, representing the fraction of the previous parameter update to retain, and subtracting (adding the negative) our actual gradient, multiplied by the learning rate (like before), from it. The update contains a portion of the gradient from preceding steps as our momentum (direction of previous changes) and only a portion of the current gradient; together, these portions form the actual change to our parameters and the bigger the role that momentum takes in the update, the slower the update can change the direction. When we set the momentum fraction too high, the model might stop learning at all since the direction of the updates won't be able to follow the global gradient descent. The code for this is as follows:

```
weight_updates = self.momentum * layer.weight_momentums - \
                 self.current_learning_rate * layer.dweights
```

The hyperparameter, `self.momentum`, is chosen at the start and the `layer.weight_momentums` start as all zeros but are altered during training as:

```
layer.weight_momentums = weight_updates
```

This means that the momentum is always the previous update to the parameters. We will perform the same operations as the above with the biases. We can then update our SGD optimizer class' `update_params` method with the momentum calculation, applying with the parameters, and retaining them for the next steps as an alternative chain of operations to the current code. The difference is that we only calculate the updates and we add these updates with the common code:

```
    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)

            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates

        # Vanilla SGD updates (as before momentum update)
        else:
            weight_updates = -self.current_learning_rate * \
                             layer.dweights
            bias_updates = -self.current_learning_rate * \
                           layer.dbiases

        # Update weights and biases using either
        # vanilla or momentum updates
        layer.weights += weight_updates
        layer.biases += bias_updates
```

Making our full SGD optimizer class:

```
# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

            # If layer does not contain momentum arrays, create them
            # filled with zeros
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                # If there is no momentum array for weights
                # The array doesn't exist for biases yet either.
                layer.bias_momentums = np.zeros_like(layer.biases)

            # Build weight updates with momentum - take previous
            # updates multiplied by retain factor and update with
            # current gradients
            weight_updates = \
                self.momentum * layer.weight_momentums - \
                self.current_learning_rate * layer.dweights
            layer.weight_momentums = weight_updates

            # Build bias updates
            bias_updates = \
                self.momentum * layer.bias_momentums - \
                self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates
```

```
            # Vanilla SGD updates (as before momentum update)
            else:
                weight_updates = -self.current_learning_rate * \
                                 layer.dweights
                bias_updates = -self.current_learning_rate * \
                               layer.dbiases

            # Update weights and biases using either
            # vanilla or momentum updates
            layer.weights += weight_updates
            layer.biases += bias_updates

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

Let's show an example illustrating how adding momentum changes the learning process. Keeping the same starting **learning rate** (1) and **decay** (1e-3) from the previous training attempt and using a momentum of 0.5:

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-3, momentum=0.5)

# Train in loop
for epoch in range(10001):

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)
```

```
    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

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

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()


>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.427, loss: 1.078, lr: 0.9099181073703367
epoch: 200, acc: 0.423, loss: 1.075, lr: 0.8340283569641367
...
epoch: 1800, acc: 0.483, loss: 0.978, lr: 0.35727045373347627
epoch: 1900, acc: 0.547, loss: 0.984, lr: 0.3449465332873405
...
epoch: 3100, acc: 0.593, loss: 0.883, lr: 0.2439619419370578
epoch: 3200, acc: 0.570, loss: 0.878, lr: 0.23815194093831865
epoch: 3300, acc: 0.563, loss: 0.863, lr: 0.23261223540358225
epoch: 3400, acc: 0.607, loss: 0.860, lr: 0.22732439190725165
...
epoch: 4600, acc: 0.670, loss: 0.761, lr: 0.1786033220217896
epoch: 4700, acc: 0.690, loss: 0.749, lr: 0.1754693805930865
```

```
...
epoch: 6000, acc: 0.743, loss: 0.661, lr: 0.1428775539362766
...
epoch: 8000, acc: 0.763, loss: 0.586, lr: 0.11112345816201799
...
epoch: 10000, acc: 0.800, loss: 0.539, lr: 0.09091735612328393
```

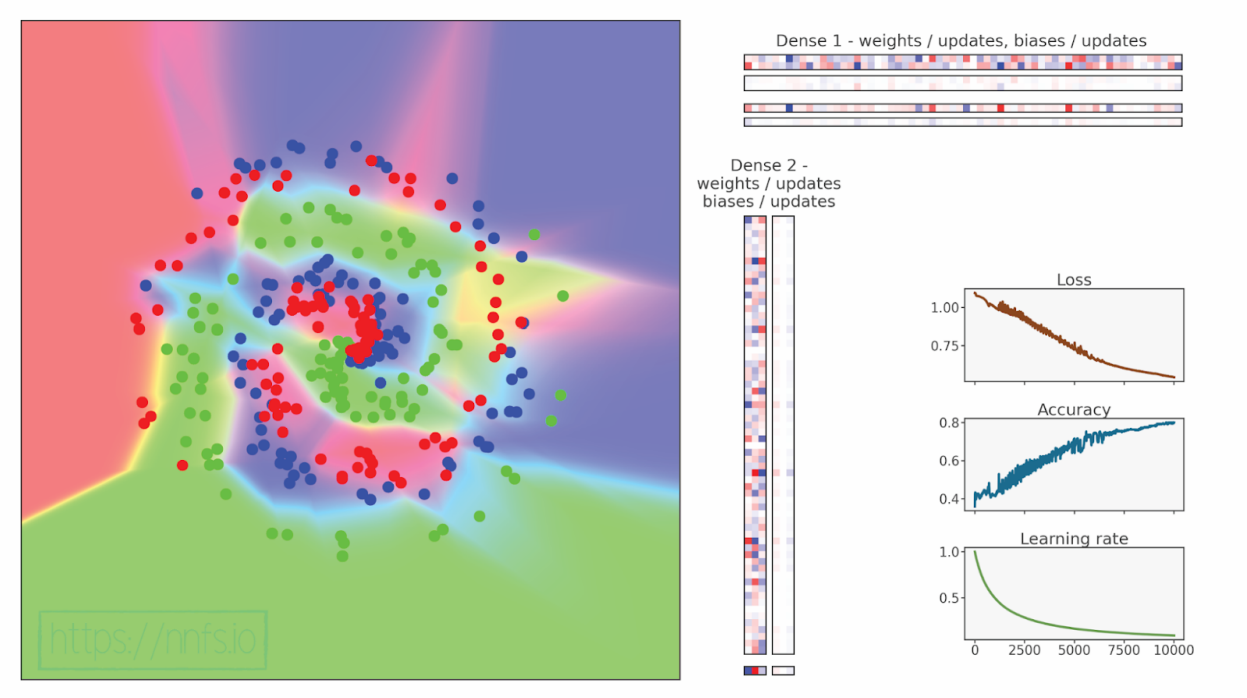


Fig 10.18: Model training with SGD optimizer, learning rate decay and Momentum.

Epilepsy Warning (quick flashing colors)

Anim 10.18: https://nnfs.io/ram

The model achieved the lowest loss and highest accuracy that we've seen so far, but can we do even better? Sure we can! Let's try to set the momentum to 0.9:

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD(decay=1e-3, momentum=0.9)

# Train in loop
for epoch in range(10001):

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

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

```
    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()
```

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.443, loss: 1.053, lr: 0.9099181073703367
epoch: 200, acc: 0.497, loss: 0.999, lr: 0.8340283569641367
epoch: 300, acc: 0.603, loss: 0.810, lr: 0.7698229407236336
epoch: 400, acc: 0.700, loss: 0.700, lr: 0.7147962830593281
epoch: 500, acc: 0.750, loss: 0.595, lr: 0.66711140760507
epoch: 600, acc: 0.810, loss: 0.496, lr: 0.6253908692933083
epoch: 700, acc: 0.810, loss: 0.466, lr: 0.5885815185403178
epoch: 800, acc: 0.847, loss: 0.384, lr: 0.5558643690939411
epoch: 900, acc: 0.850, loss: 0.364, lr: 0.526592943654555
epoch: 1000, acc: 0.877, loss: 0.344, lr: 0.5002501250625312
...
epoch: 2200, acc: 0.900, loss: 0.242, lr: 0.31259768677711786
...
epoch: 2900, acc: 0.910, loss: 0.216, lr: 0.25647601949217746
...
epoch: 3800, acc: 0.920, loss: 0.202, lr: 0.20837674515524068
...
epoch: 7100, acc: 0.930, loss: 0.181, lr: 0.12347203358439313
...
epoch: 10000, acc: 0.933, loss: 0.173, lr: 0.09091735612328393
```

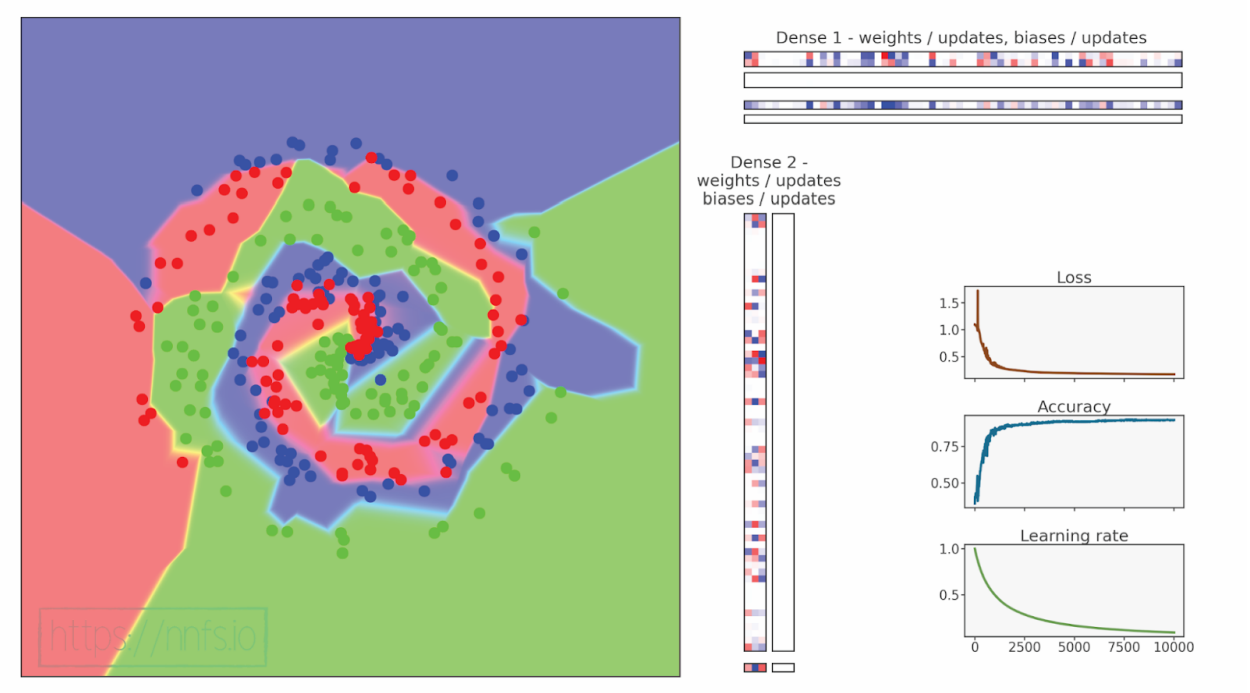


Fig 10.19: Model training with SGD optimizer, learning rate decay and Momentum (tuned).

Epilepsy Warning (quick flashing colors)

Anim 10.19: https://nnfs.io/map

This is a decent enough example of how momentum can prove useful. The model achieved an accuracy of almost 88% in the first 1000 epochs and improved further, ending with an accuracy of 93.3% and a loss of 0.173. These results are a great improvement. The SGD optimizer with momentum is usually one of 2 main choices for an optimizer in practice next to the Adam optimizer, which we'll talk about shortly. First, we have 2 other optimizers to talk about. The next modification to Stochastic Gradient Descent is **AdaGrad**.

AdaGrad

AdaGrad, short for **adaptive gradient**, institutes a per-parameter learning rate rather than a globally-shared rate. The idea here is to normalize updates made to the features. During the training process, some weights can rise significantly, while others tend to not change by much. It is usually better for weights to not rise too high compared to the other weights, and we'll talk about this with regularization techniques. AdaGrad provides a way to normalize parameter updates by keeping a history of previous updates — the bigger the sum of the updates is, in either direction (positive or negative), the smaller updates are made further in training. This lets less-frequently updated parameters to keep-up with changes, effectively utilizing more neurons for training. The concept of AdaGrad can be contained in the following two lines of code:

```
cache += parm_gradient ** 2
parm_updates = learning_rate * parm_gradient / (sqrt(cache) + eps)
```

The `cache` holds a history of squared gradients, and the `parm_updates` is a function of the learning rate multiplied by the gradient (basic SGD so far) and then is divided by the square root of the cache plus some **epsilon** value. The division operation performed with a constantly rising cache might also cause the learning to stall as updates become smaller with time, due to the monotonic nature of updates. That's why this optimizer is not widely used, except for some specific applications. The **epsilon** is a **hyperparameter** (pre-training control knob setting) preventing division by 0. The epsilon value is usually a small value, such as *1e-7*, which we'll be defaulting to. You might also notice that we are summing the squared value, only to calculate the square root later, which might look counter-intuitive as to why we do this. We are adding squared values and taking the square root, which is not the same as just adding the value, for example:

$$\sqrt{1^2 + 3^2} = \sqrt{1 + 9} = \sqrt{10} \approx 2.16$$

$$1 + 3 = 4$$

The resulting cache value grows slower, and in a different way, taking care of the negative numbers (we would not want to divide the update by the negative number and flip its sign). Overall, the impact is the learning rates for parameters with smaller gradients are decreased slowly, while the parameters with larger gradients have their learning rates decreased faster.

To implement AdaGrad, we start by copying and pasting our SGD optimizer class, changing the name, adding a property for **epsilon** with a default of 1e-7 to the `__init__` method, and removing the momentum. Next, inside the `update_params` method, we'll replace the momentum code with:

```
    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)
```

We added the cache and its updates, then added dividing the updates by the square root of the cache. Full code for the AdaGrad optimizer:

```
# Adagrad optimizer
class Optimizer_Adagrad:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=1., decay=0., epsilon=1e-7):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))
```

```
    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        # Vanilla SGD parameter update + normalization
        # with square rooted cache
        layer.weights += -self.current_learning_rate * \
                         layer.dweights / \
                         (np.sqrt(layer.weight_cache) + self.epsilon)
        layer.biases += -self.current_learning_rate * \
                        layer.dbiases / \
                        (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

Testing this optimizer now with decaying set to *1e-4* as well as *1e-5* works better than *1e-3*, which we have used previously. This optimizer with our dataset works better with lesser decaying:

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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)

# Create ReLU activation (to be used with Dense layer):
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
#optimizer = Optimizer_SGD(decay=8e-8, momentum=0.9)
```

```
optimizer = Optimizer_Adagrad(decay=1e-4)
# Train in loop
for epoch in range(10001):

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

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

    if not epoch % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}, ' +
              f'lr: {optimizer.current_learning_rate}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.post_update_params()


>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.457, loss: 1.012, lr: 0.9901970492127933
epoch: 200, acc: 0.527, loss: 0.936, lr: 0.9804882831650161
```

```
epoch: 300, acc: 0.600, loss: 0.874, lr: 0.9709680551509855
...
epoch: 1200, acc: 0.700, loss: 0.640, lr: 0.892936869363336
...
epoch: 1700, acc: 0.750, loss: 0.579, lr: 0.8547739123001966
...
epoch: 4700, acc: 0.800, loss: 0.464, lr: 0.6803183890060548
...
epoch: 5100, acc: 0.810, loss: 0.454, lr: 0.6622955162593549
...
epoch: 6700, acc: 0.820, loss: 0.426, lr: 0.5988382537876519
...
epoch: 7500, acc: 0.830, loss: 0.412, lr: 0.5714612263557918
...
epoch: 9900, acc: 0.847, loss: 0.381, lr: 0.5025378159706518
epoch: 10000, acc: 0.847, loss: 0.379, lr: 0.5000250012500626
```

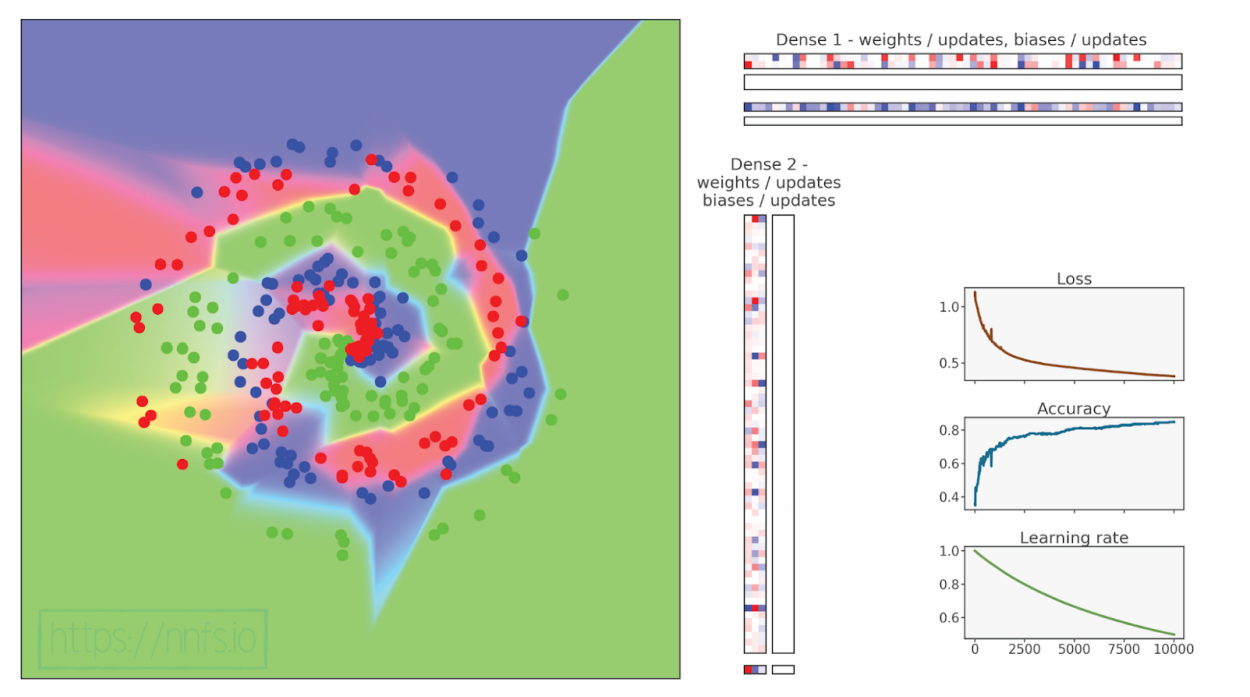


Fig 10.20: Model training with AdaGrad optimizer.

Epilepsy Warning (quick flashing colors)

Anim 10.20: https://nnfs.io/bop

AdaGrad worked quite well here, but not as good as SGD with momentum, and we can see that loss consistently fell throughout the entire training process. It is interesting to note that AdaGrad initially took a few more epochs to reach similar results to Stochastic Gradient Descent with momentum.

RMSProp

Continuing with Stochastic Gradient Descent adaptations, we reach **RMSProp**, short for **Root Mean Square Propagation**. Similar to AdaGrad, RMSProp calculates an adaptive learning rate per parameter; it's just calculated in a different way than AdaGrad.

Where AdaGrad calculates the cache as:

```
cache += gradient ** 2
```

RMSProp calculates the cache as:

```
cache = rho * cache + (1 - rho) * gradient ** 2
```

Note that this is similar to both momentum with the SGD optimizer and cache with the AdaGrad. RMSProp adds a mechanism similar to momentum but also adds a per-parameter adaptive learning rate, so the learning rate changes are smoother. This helps to retain the global direction of changes and slows changes in direction. Instead of continually adding squared gradients to a cache (like in Adagrad), it uses a moving average of the cache. Each update to the cache retains a part of the cache and updates it with a fraction of the new, squared, gradients. In this way, cache contents "move" with data in time, and learning does not stall. In the case of this optimizer, the per-parameter learning rate can either fall or rise, depending on the last updates and current gradient. RMSProp applies the cache in the same way as AdaGrad does.

The new hyperparameter here is *rho*. *Rho* is the cache memory decay rate. Because this optimizer, with default values, carries over so much momentum of gradient and the adaptive learning rate updates, even small gradient updates are enough to keep it going; therefore, a default learning rate of *1* is far too large and causes instant model instability. A learning rate that becomes stable again and gives fast enough updates is around *0.001* (that's also the default value for this optimizer used in well-known machine learning frameworks). That's what we'll use as default from now on too. The following is the full code for RMSProp optimizer class:

```
# RMSprop optimizer
class Optimizer_RMSprop:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 rho=0.9):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If layer does not contain cache arrays,
        # create them filled with zeros
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update cache with squared current gradients
        layer.weight_cache = self.rho * layer.weight_cache + \
            (1 - self.rho) * layer.dweights**2
        layer.bias_cache = self.rho * layer.bias_cache + \
            (1 - self.rho) * layer.dbiases**2
```

```
            # Vanilla SGD parameter update + normalization
            # with square rooted cache
            layer.weights += -self.current_learning_rate * \
                             layer.dweights / \
                             (np.sqrt(layer.weight_cache) + self.epsilon)
            layer.biases += -self.current_learning_rate * \
                            layer.dbiases / \
                            (np.sqrt(layer.bias_cache) + self.epsilon)

    # Call once after any parameter updates
    def post_update_params(self):
        self.iterations += 1
```

Changing the optimizer used in our main neural network testing code:

```
optimizer = Optimizer_RMSprop(decay=1e-4)
```

And running this code gives us:

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.001
epoch: 100, acc: 0.417, loss: 1.077, lr: 0.0009901970492127933
epoch: 200, acc: 0.457, loss: 1.072, lr: 0.0009804882831650162
epoch: 300, acc: 0.480, loss: 1.062, lr: 0.0009709680551509856
...
epoch: 1000, acc: 0.597, loss: 0.961, lr: 0.0009091735612328393
...
epoch: 4800, acc: 0.703, loss: 0.767, lr: 0.0006757213325224677
...
epoch: 5800, acc: 0.713, loss: 0.744, lr: 0.0006329514526235838
...
epoch: 7100, acc: 0.720, loss: 0.718, lr: 0.0005848295221942804
...
epoch: 10000, acc: 0.730, loss: 0.668, lr: 0.0005000250012500625
```

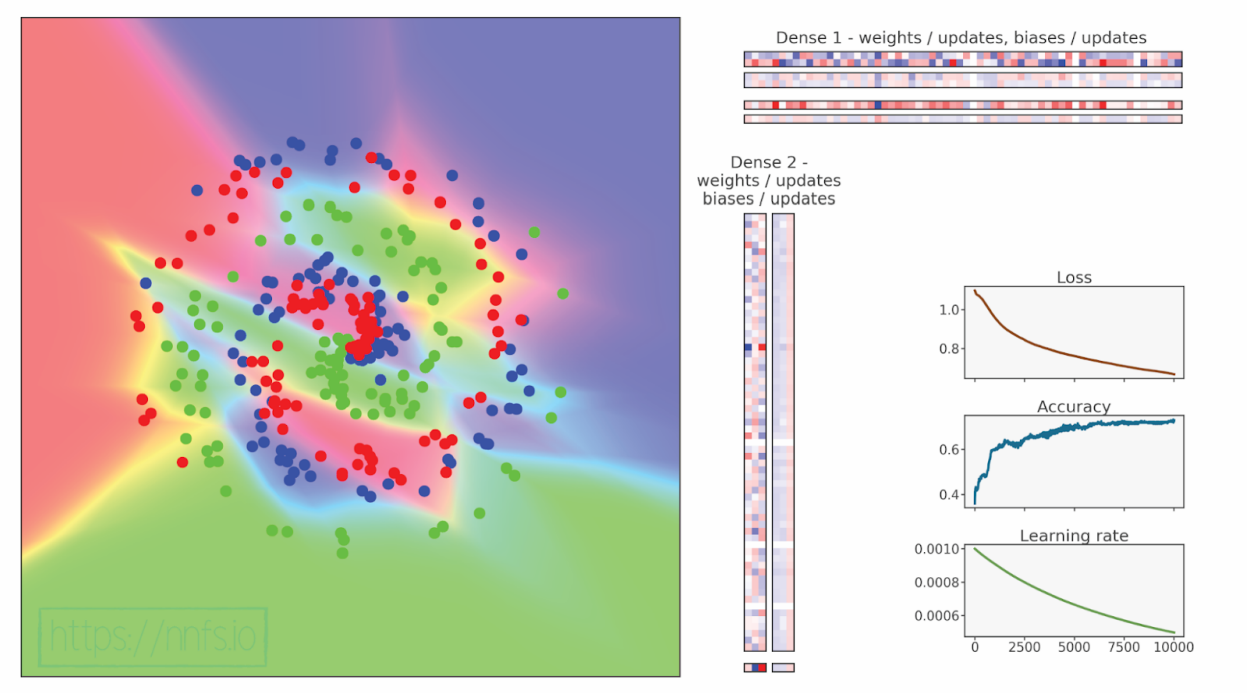


Fig 10.21: Model training with RMSProp optimizer.

Epilepsy Warning (quick flashing colors)

Anim 10.21: https://nnfs.io/pun

The results are not the greatest, but we can slightly tweak the hyperparameters:

```
optimizer = Optimizer_RMSprop(learning_rate=0.02, decay=1e-5,
                              rho=0.999)


>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.02
epoch: 100, acc: 0.467, loss: 1.014, lr: 0.01998021958261321
epoch: 200, acc: 0.530, loss: 0.959, lr: 0.019960279044701046
...
epoch: 600, acc: 0.623, loss: 0.762, lr: 0.019880913329158343
...
epoch: 1000, acc: 0.710, loss: 0.634, lr: 0.019802176259170884
...
epoch: 1800, acc: 0.810, loss: 0.475, lr: 0.01964655841412981
...
epoch: 3800, acc: 0.850, loss: 0.351, lr: 0.01926800836231563
...
epoch: 6200, acc: 0.870, loss: 0.286, lr: 0.018832569044906263
...
epoch: 6600, acc: 0.903, loss: 0.262, lr: 0.018761902081633034
...
epoch: 7100, acc: 0.900, loss: 0.274, lr: 0.018674310684506857
...
epoch: 9500, acc: 0.890, loss: 0.244, lr: 0.018265006986365174
epoch: 9600, acc: 0.893, loss: 0.241, lr: 0.018248341681949654
epoch: 9700, acc: 0.743, loss: 0.794, lr: 0.018231706761228456
epoch: 9800, acc: 0.917, loss: 0.213, lr: 0.018215102141185255
epoch: 9900, acc: 0.907, loss: 0.225, lr: 0.018198527739105907
epoch: 10000, acc: 0.910, loss: 0.221, lr: 0.018181983472577025
```