NEURAL NETWORKS FROM SCRATCH IN PYTHON

By Harrison Kinsley & Daniel Kukiela

Feel free to post comments/questions by highlighting the text and clicking the comment button to the right. Looks like this:

Do not resolve comments that are not yours.

Links to chapters:

Neural Networks from Scratch in Python

Harrison Kinsley & Daniel Kukieła

Copyright

License for Code

Readme

The objective of this book is to break down an extremely complex topic, neural networks, into small pieces, consumable by anyone wishing to embark on this journey. Beyond breaking down this topic, the hope is to dramatically demystify neural networks. As you will soon see, this subject, when explored from scratch, can be an educational and engaging experience. This book is for anyone willing to put in the time to sit down and work through it. In return, you will gain a far deeper understanding than most when it comes to neural networks and deep learning.

This book will be easier to understand if you already have an understanding of Python or another programming language. Python is one of the most clear and understandable programming languages; we have no real interest in padding page counts and exhausting an entire first chapter with a basics of Python tutorial. If you need one, we suggest you start here: *https://pythonprogrammingnet/python-fundamental-tutorials/* To cite this material:

Harrison Kinsley & Daniel Kukieła Neural Networks from Scratch (NNFS) https://nnfs.io

Chapter 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:

```
# SGD optimizer
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.

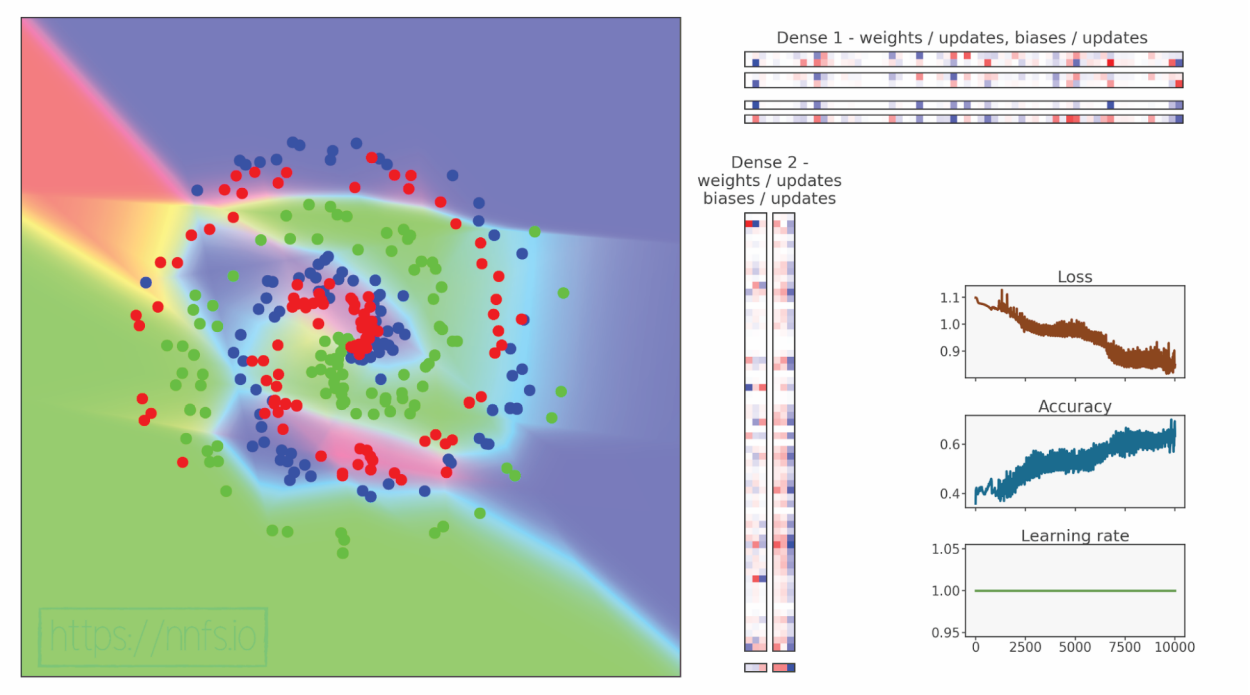


Fig 10.01: Model training with Stochastic Gradient Descent optimizer.

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:

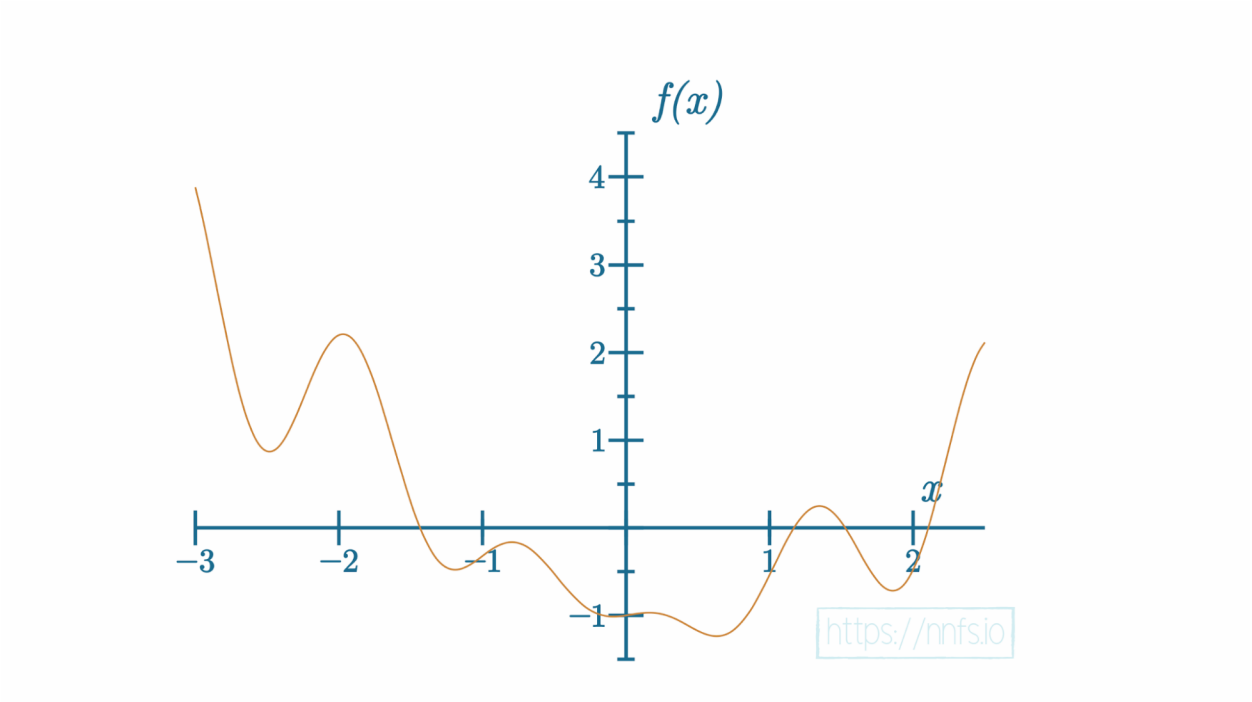


Fig 10.02: Example function to minimize the output.

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:

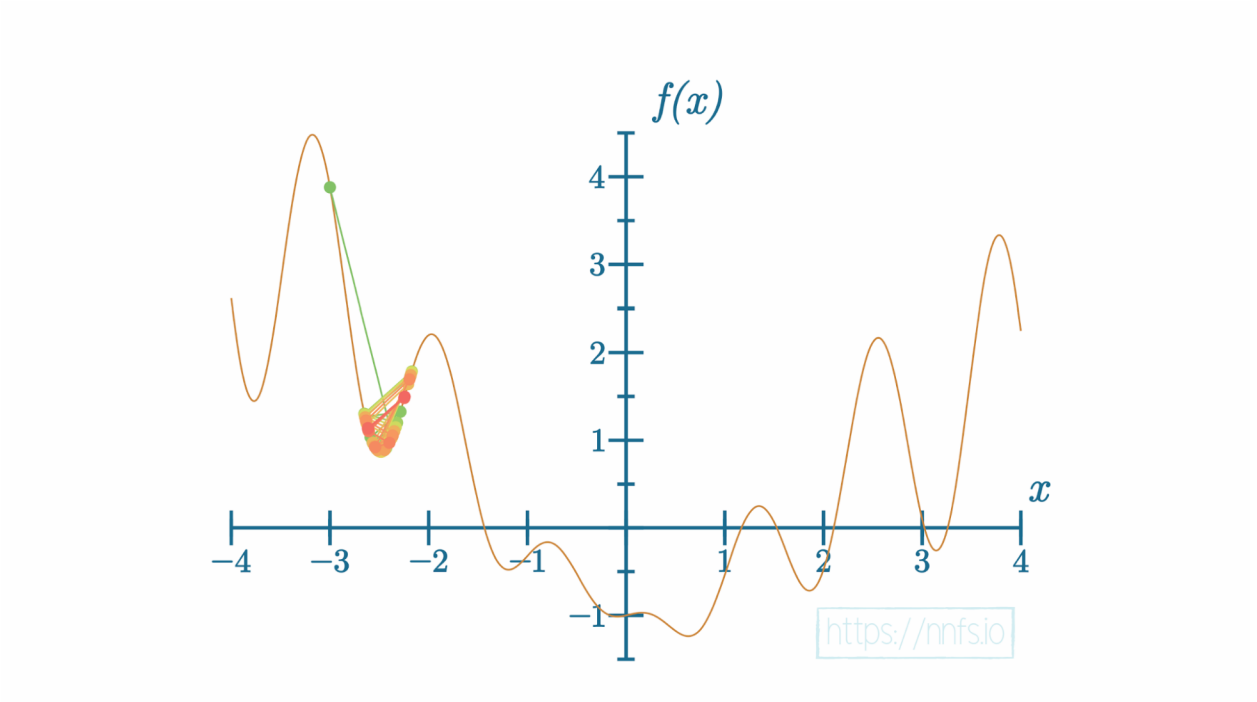


Fig 10.03: Stuck in the first local minimum.

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:

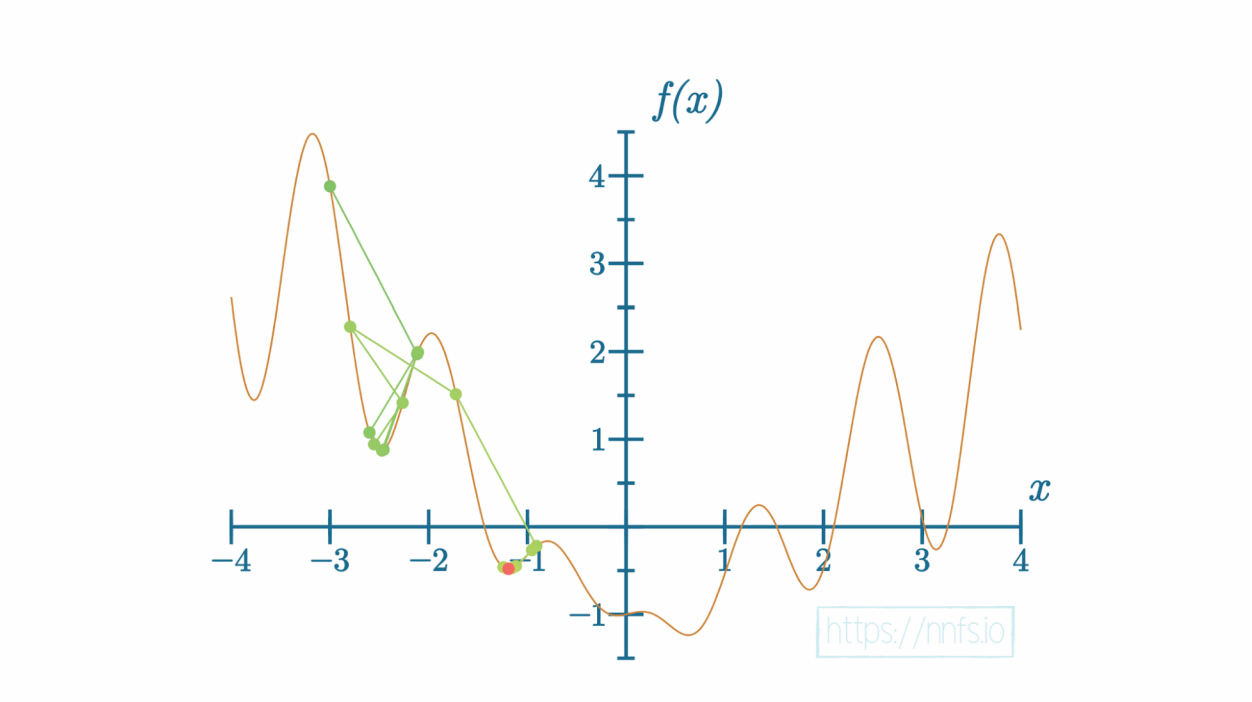


Fig 10.04: Stuck in the second local minimum.

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:

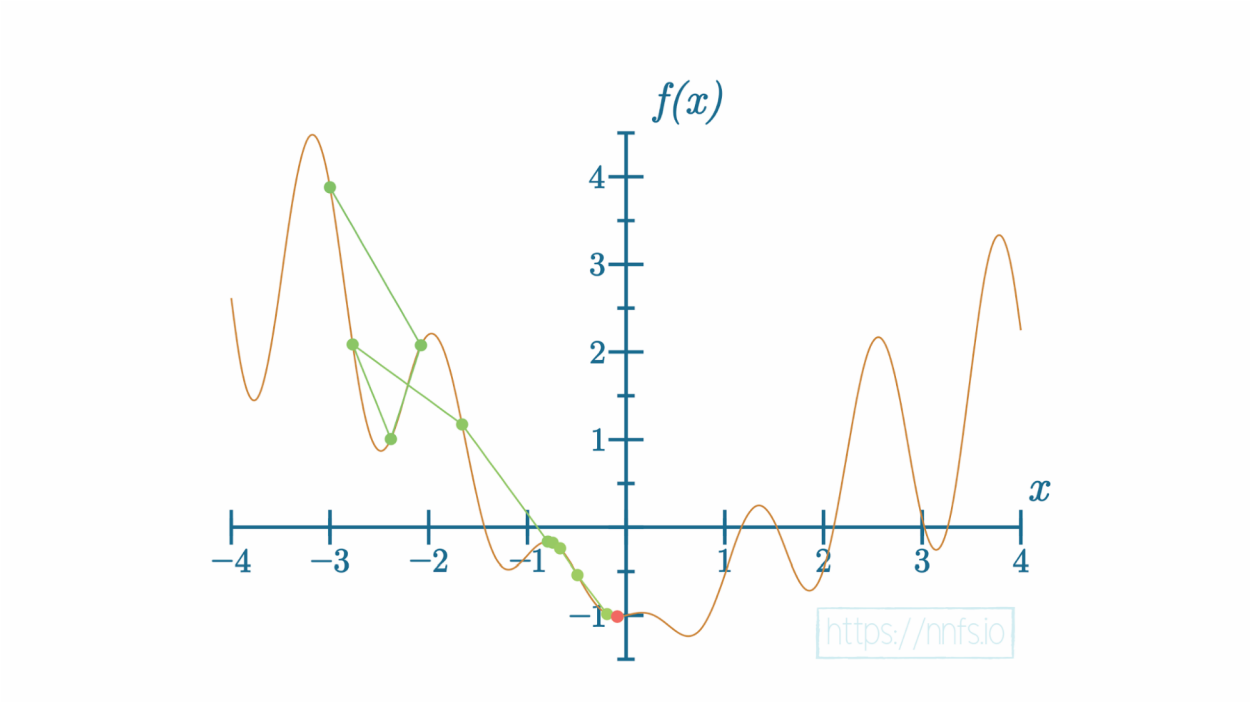


Fig 10.05: Stuck in the third local minimum, near the global minimum.

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:

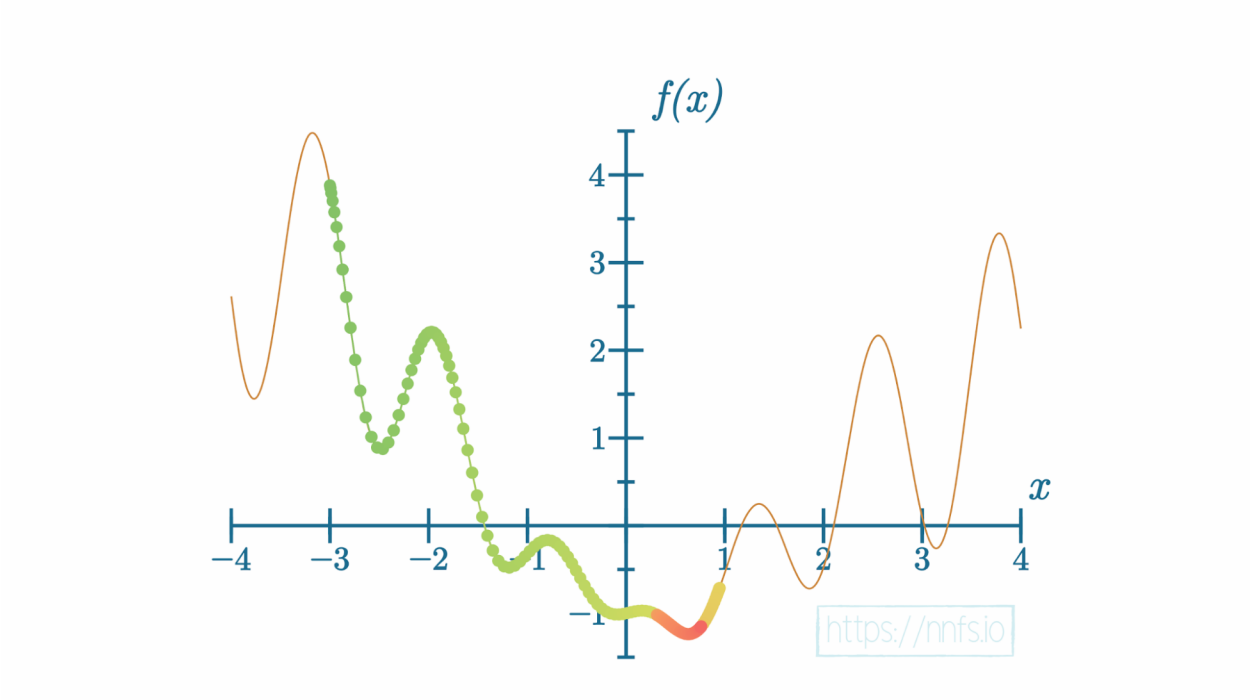


Fig 10.06: Reached the global minimum, too low learning rate.

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?

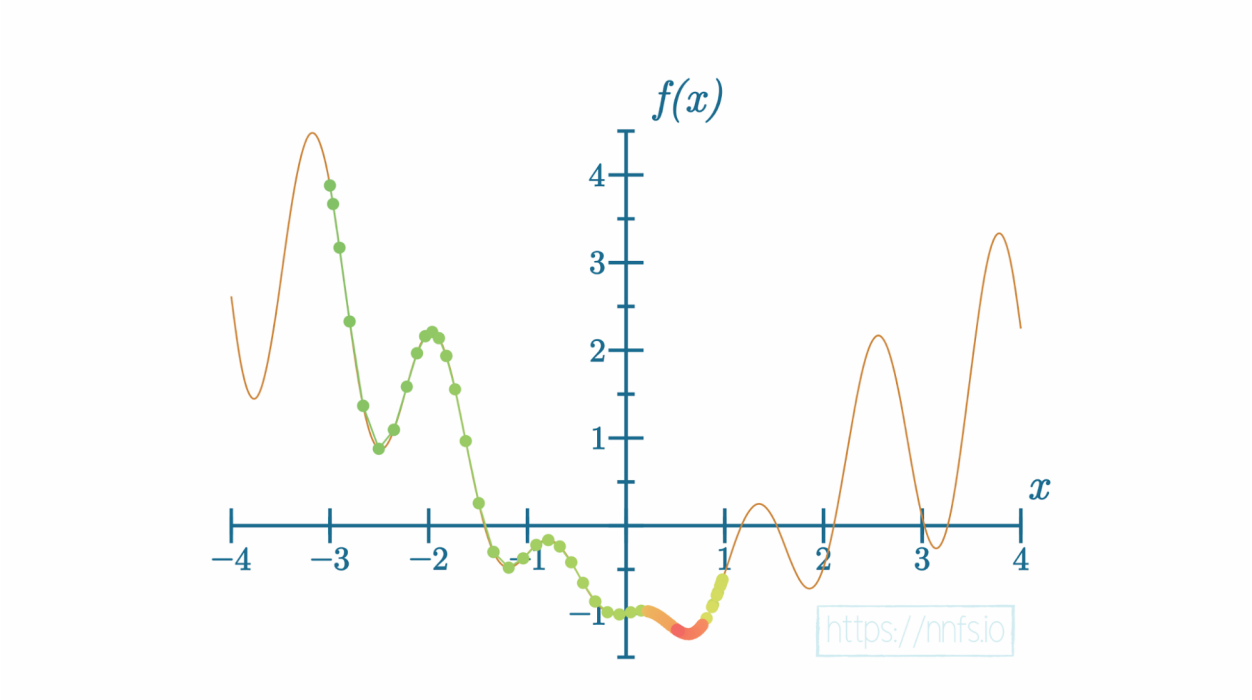


Fig 10.07: Reached the global minimum, better learning rate.

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

And even further:

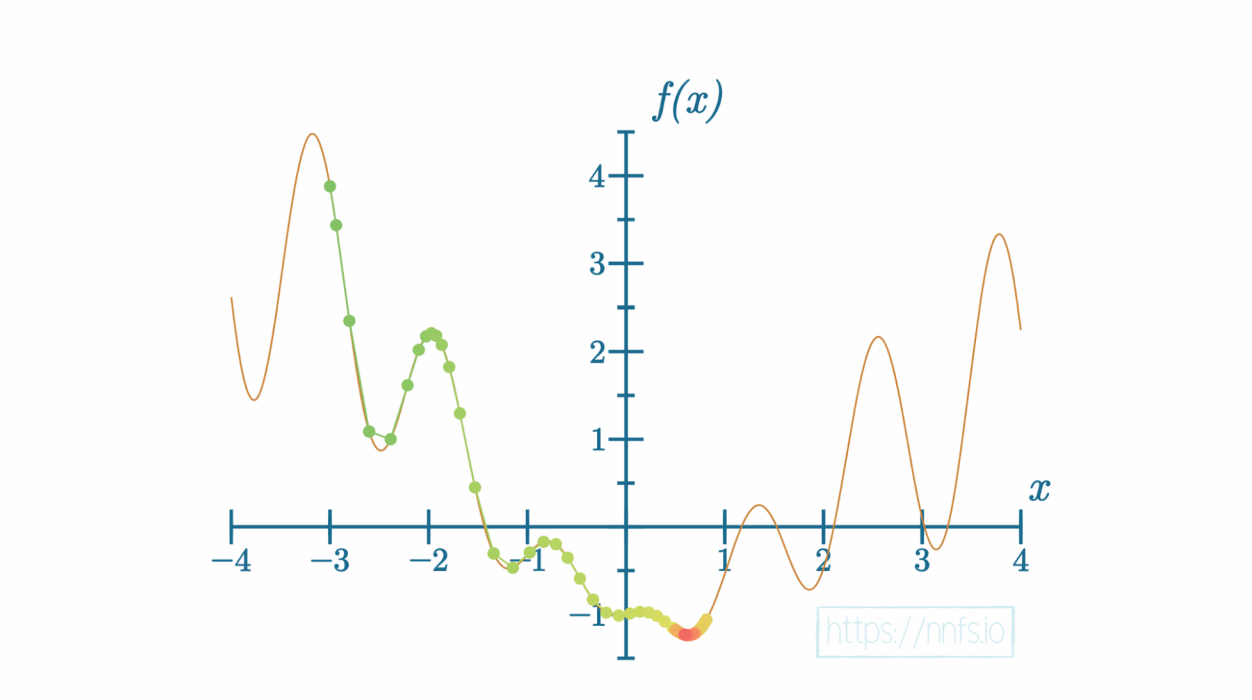


Fig 10.08: Reached the global minimum, significantly better learning rate.

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:

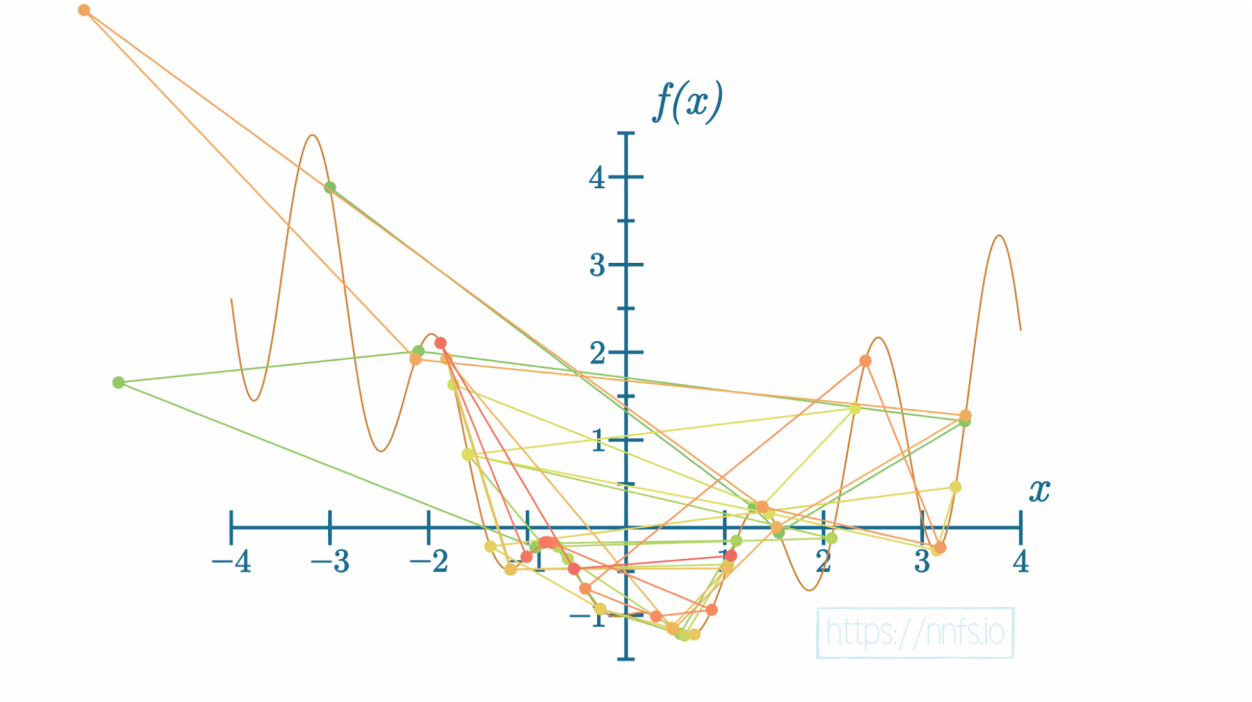


Fig 10.09: Unstable model, learning rate too big.

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:

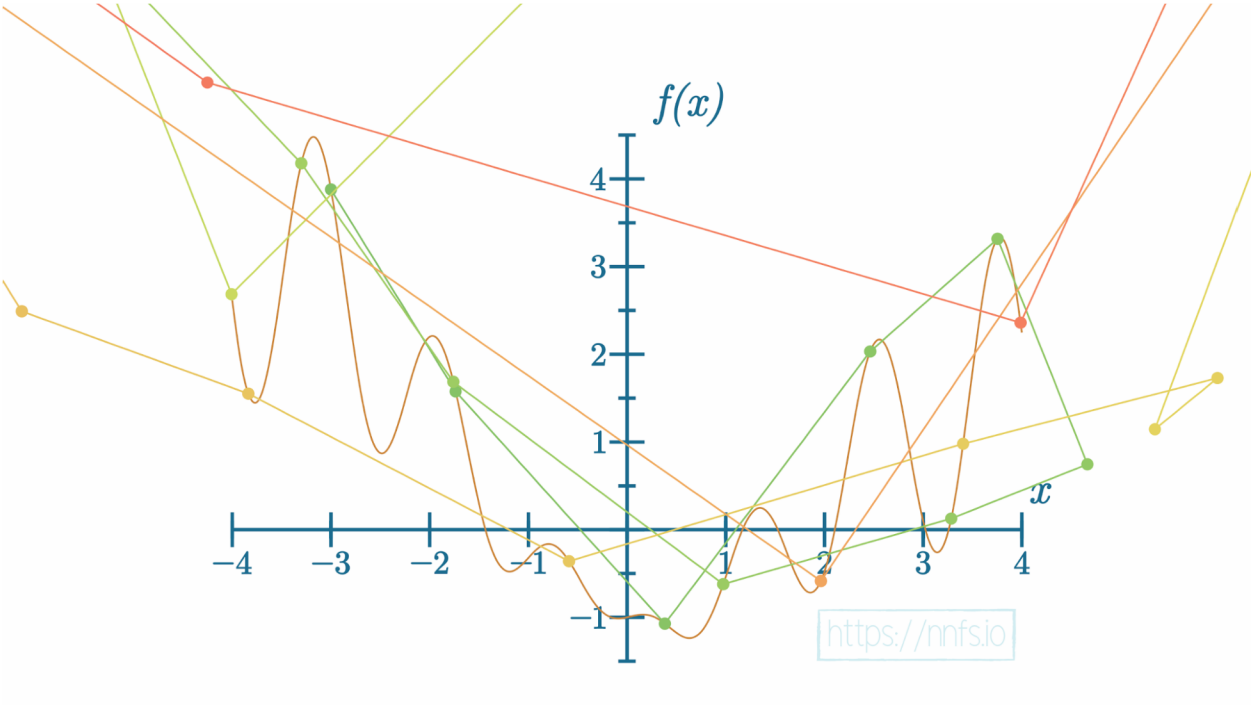


Fig 10.10: Unstable model, learning rate significantly too big.

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**:

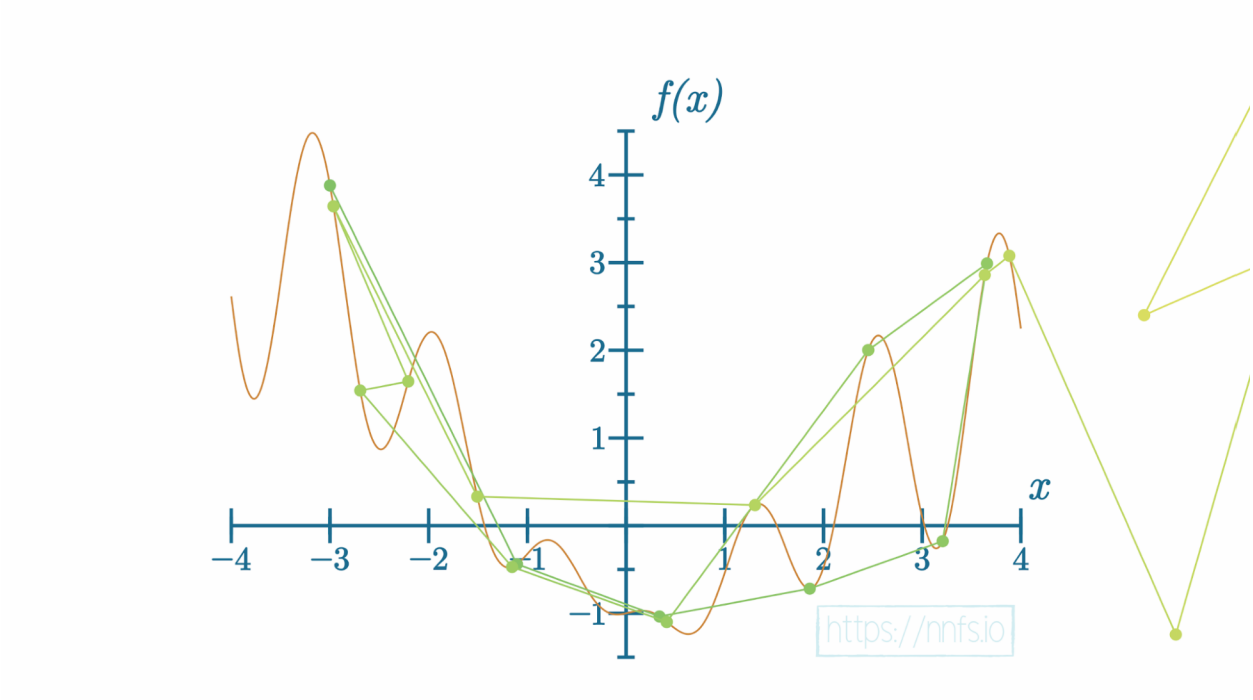


Fig 10.11: Broken model, learning rate critically too big.

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:

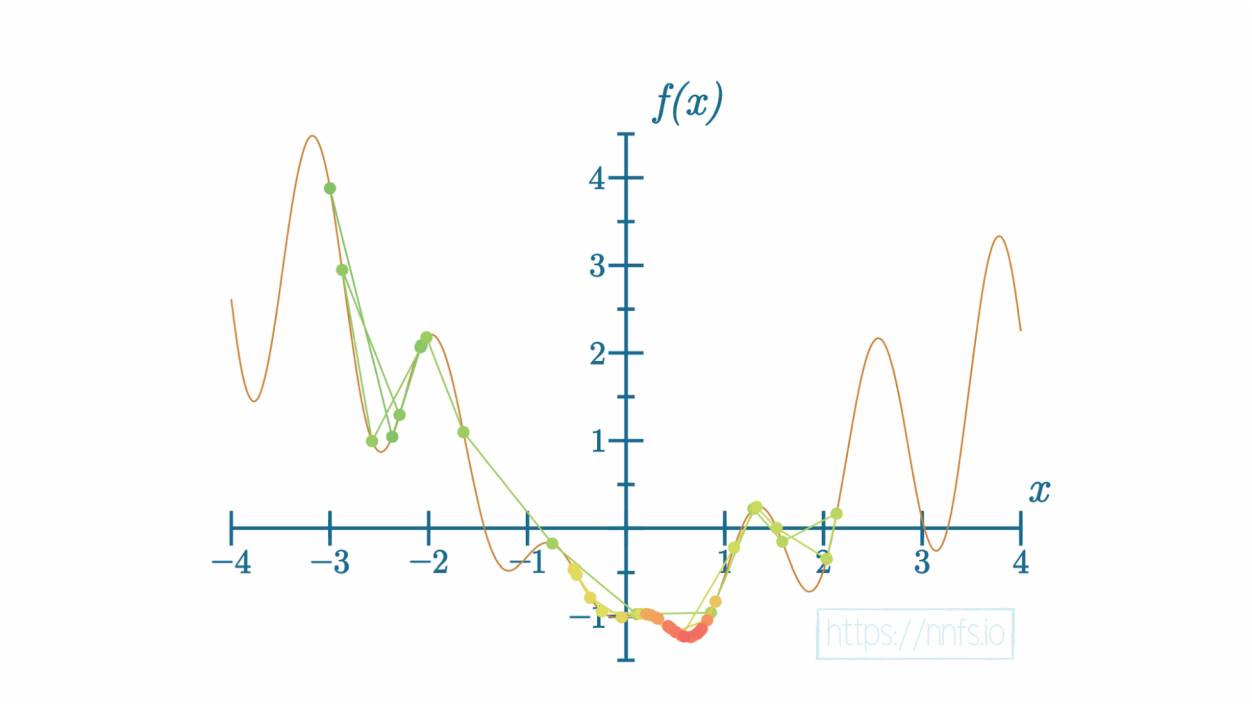


Fig 10.12: Model learned, good learning rate, can be better.

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:

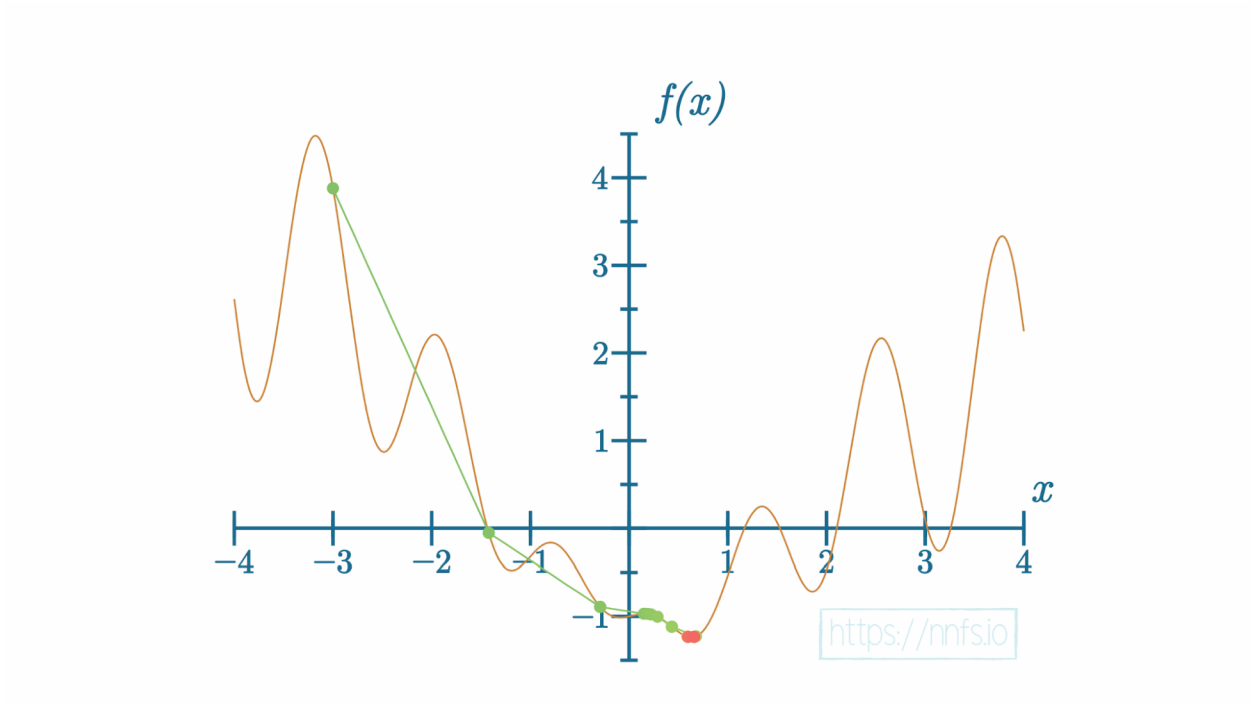


Fig 10.13: An efficient learning example.

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
```

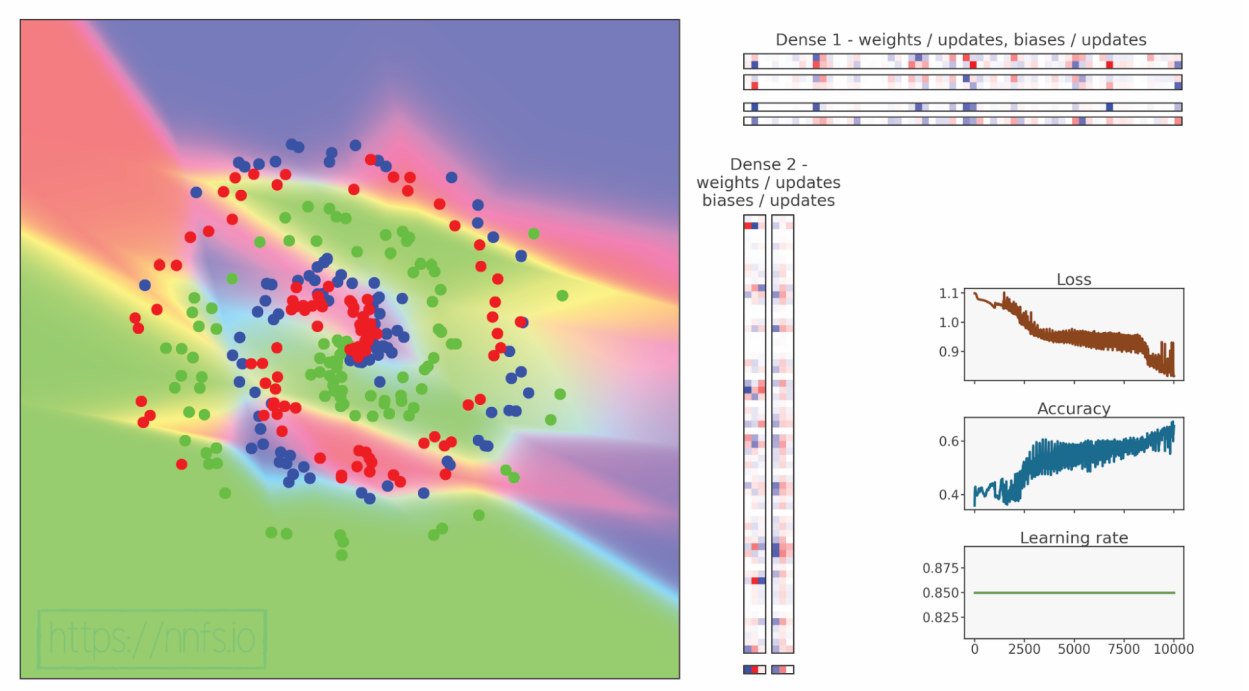


Fig 10.15: Model training with SGD optimizer and lowered learning rate.

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 now updates `self.current_learning_rate`, and we 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
```

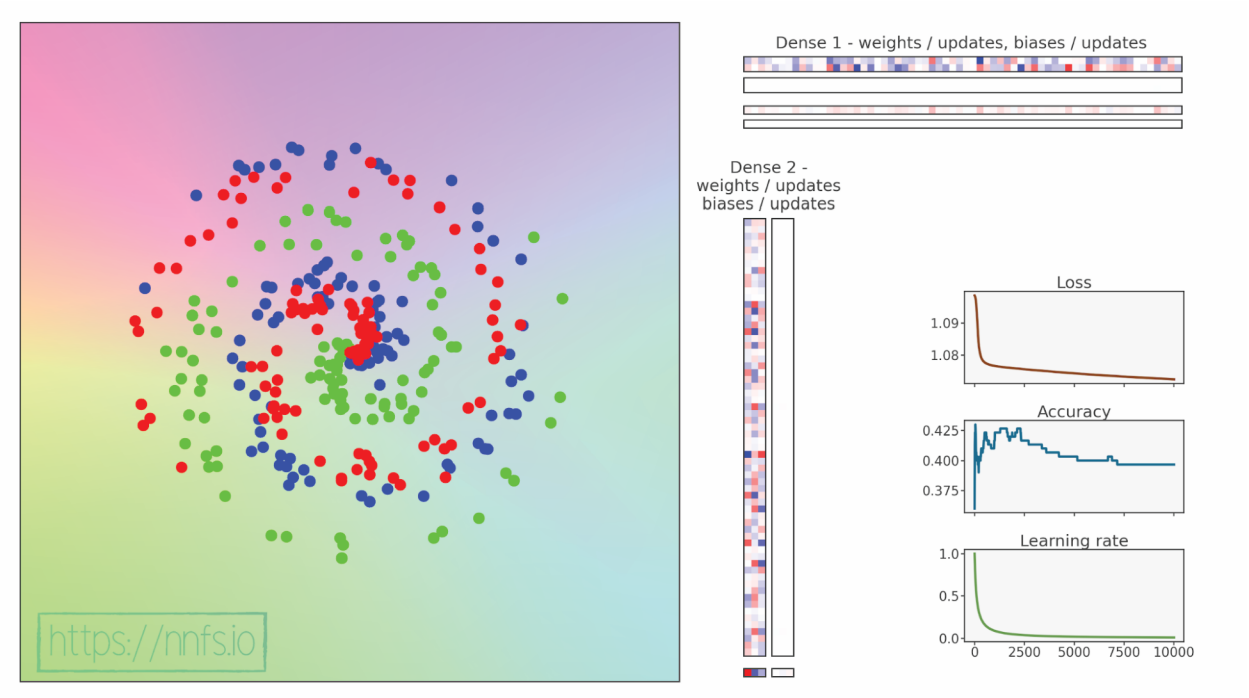


Fig 10.16: Model training with SGD optimizer and learning rate decay set too high.

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
```

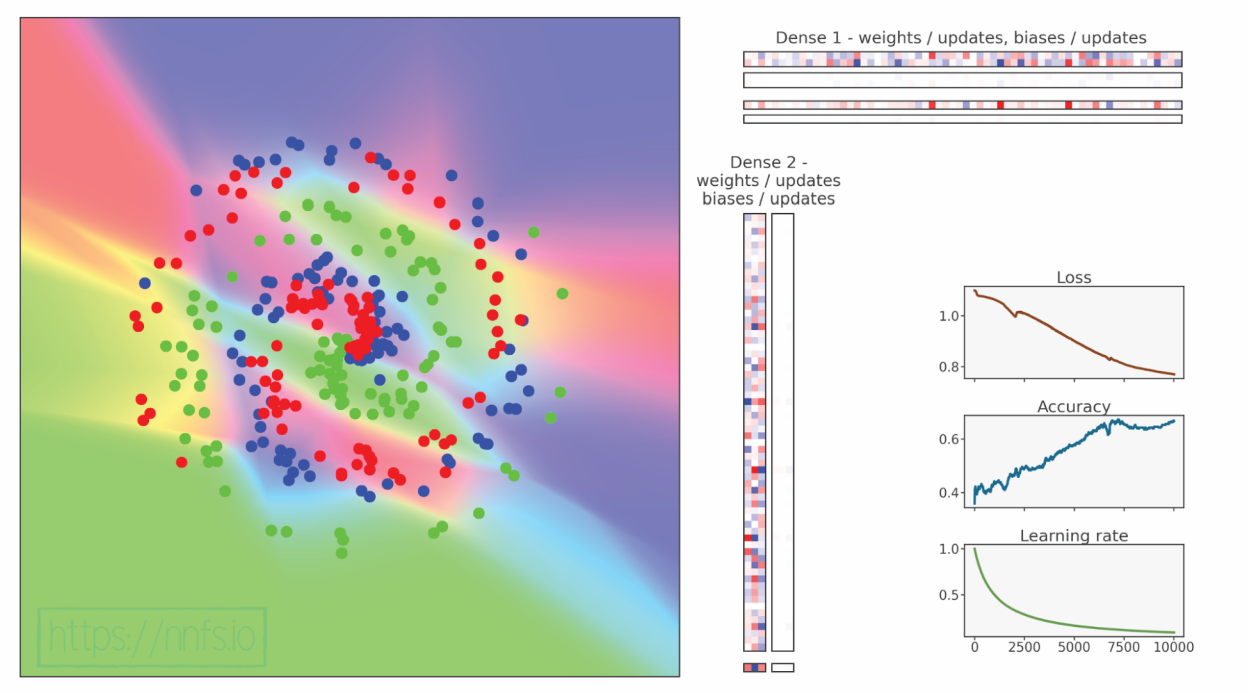


Fig 10.17: Model training with SGD optimizer and more proper learning rate decay.

Epilepsy Warning (quick flashing colors)

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

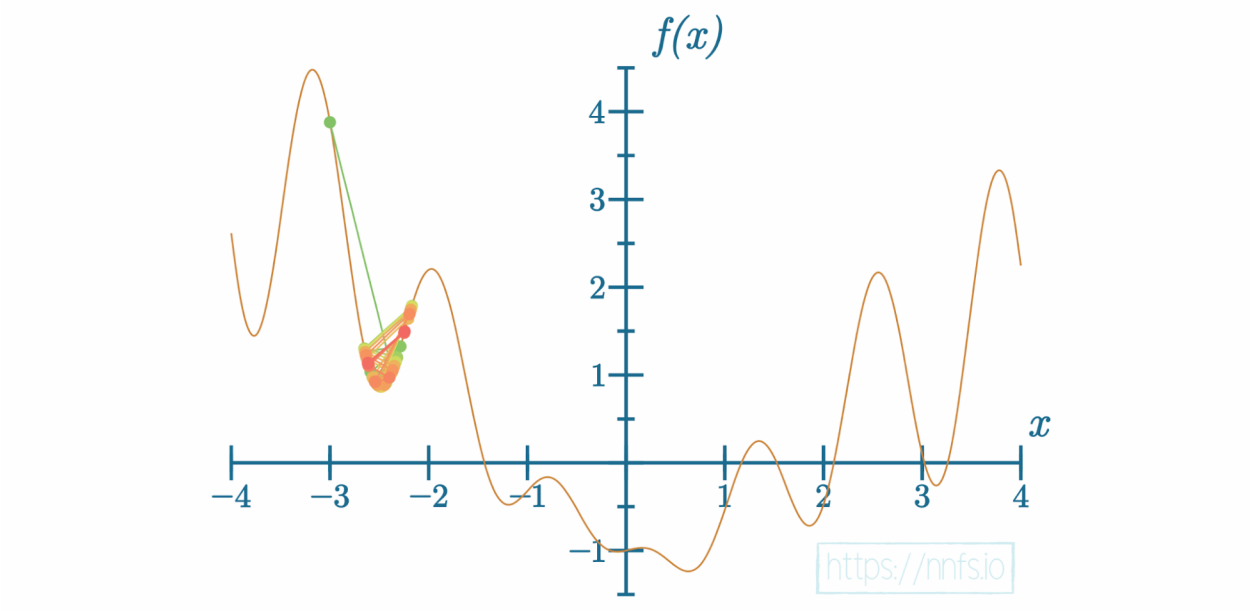
In this case, we've achieved our lowest loss and highest accuracy thus far, but it still should be possible to find parameters that will give us even better results. For example, you may suspect that the initial learning rate is too high. It can make for a great exercise to attempt to find better settings.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

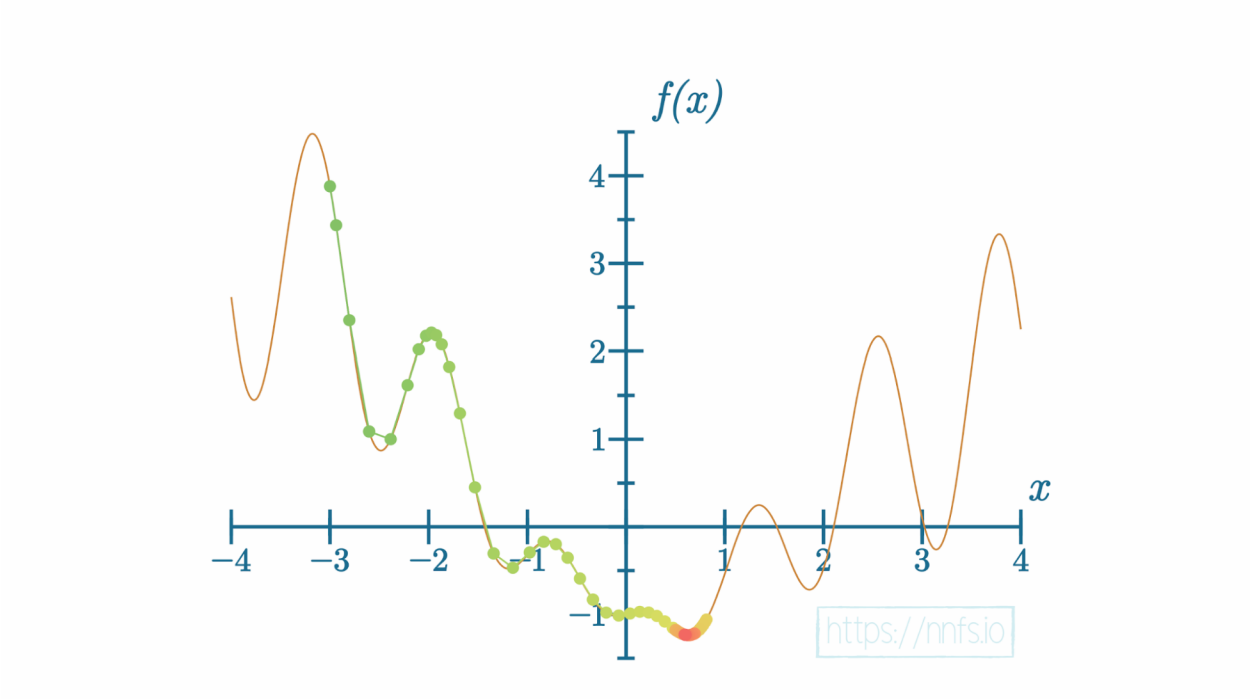
Recall this situation from the beginning of this chapter:



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
```

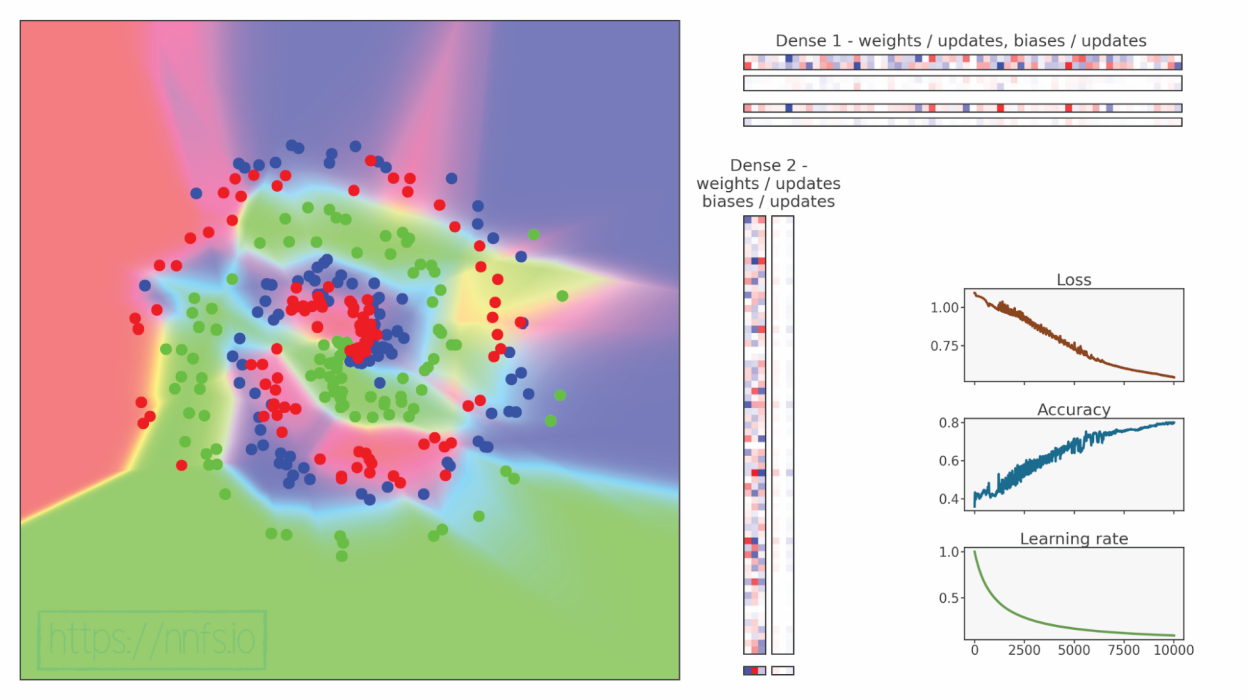


Fig 10.18: Model training with SGD optimizer, learning rate decay and Momentum.

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
```

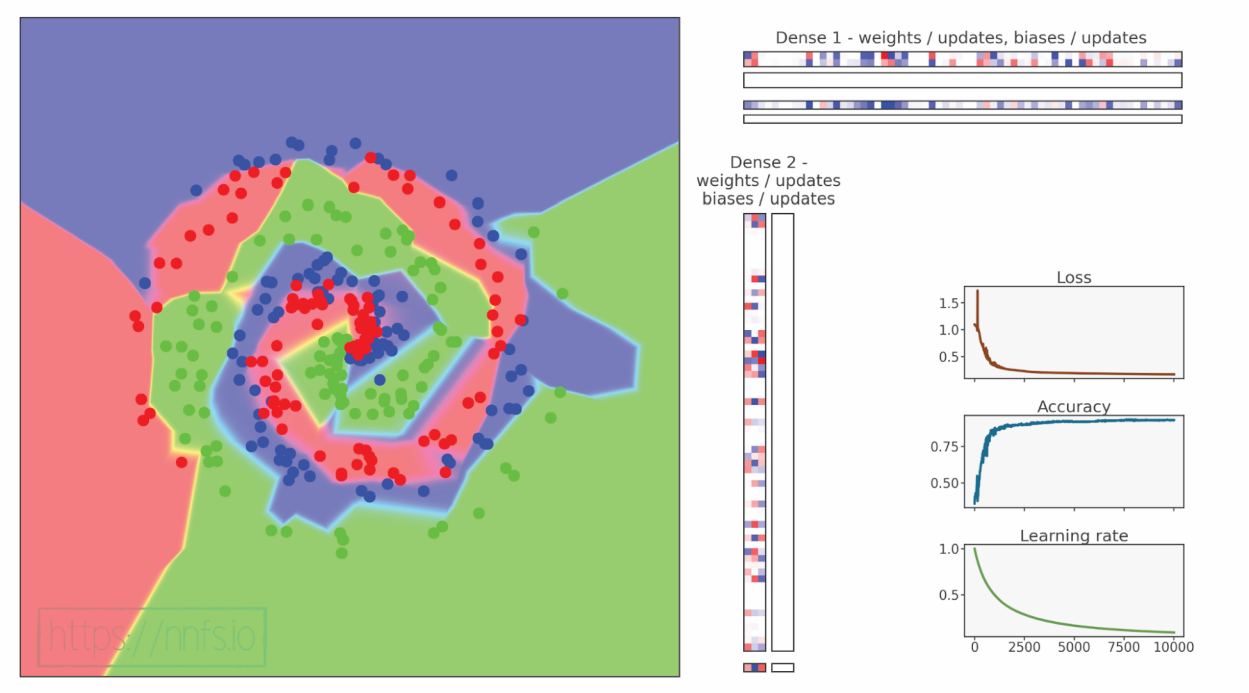


Fig 10.19: Model training with SGD optimizer, learning rate decay and Momentum (tuned).

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:

$$\sqrt{1^2 + 3^2} = \sqrt{1 + 9} = \sqrt{10} \approx 3.16$$

$$1 + 3 = 4$$

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
```

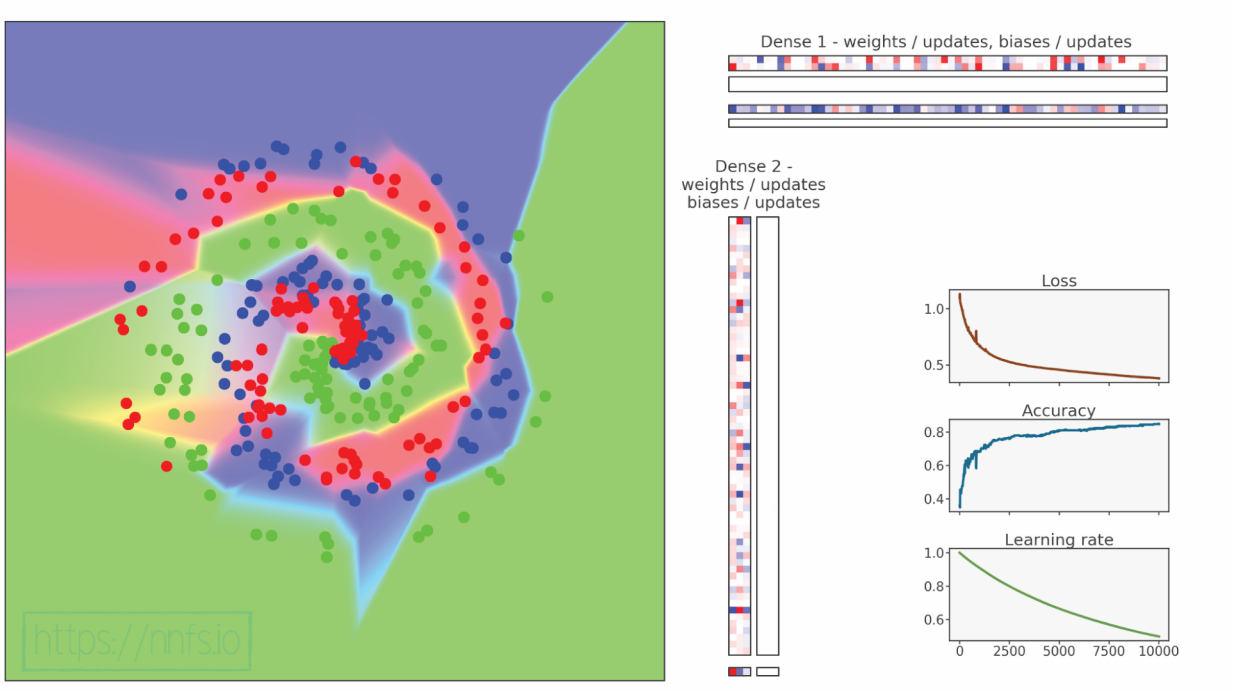


Fig 10.20: Model training with AdaGrad optimizer.

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
```

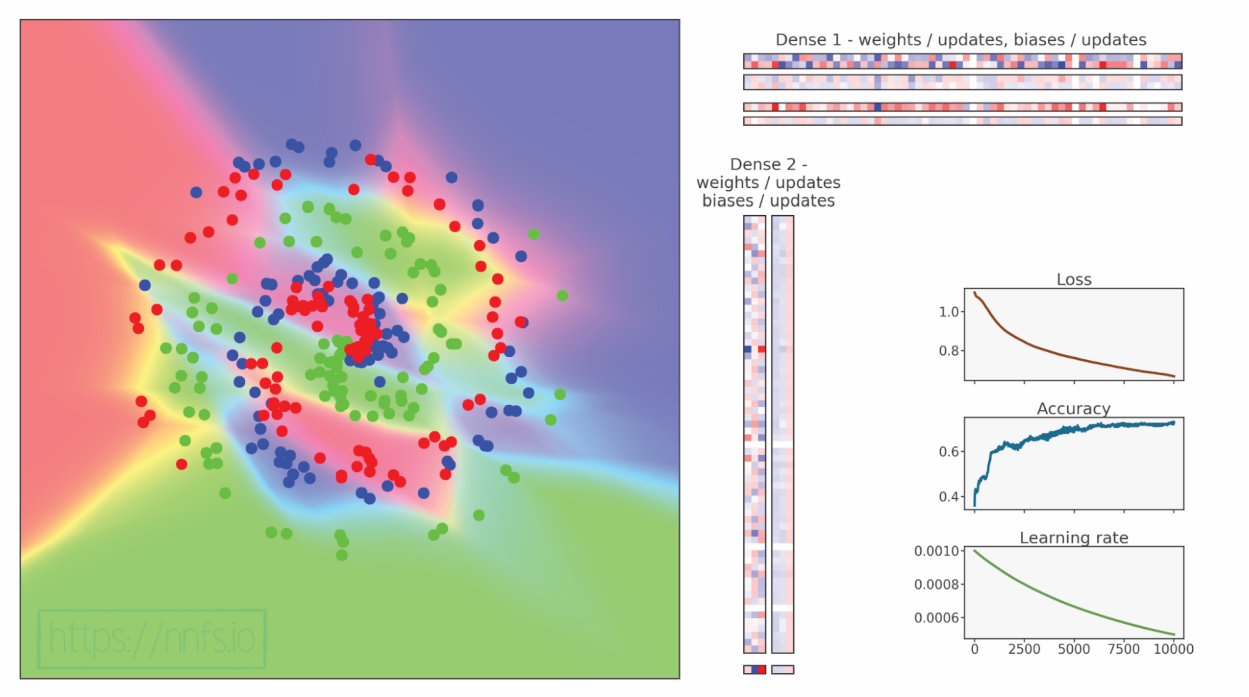


Fig 10.21: Model training with RMSProp optimizer.

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
```

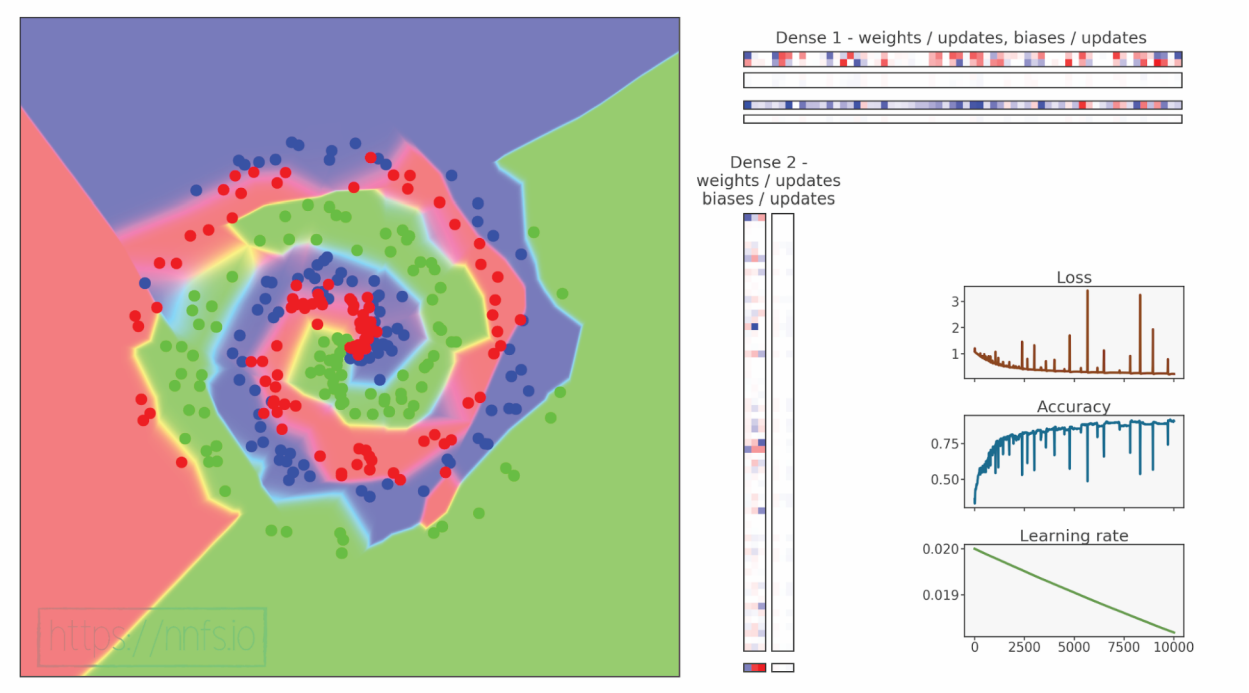


Fig 10.22: Model training with RMSProp optimizer (tuned).

Epilepsy Warning (quick flashing colors)

Anim 10.22: https://nnfs.io/not

Pretty good result, close to SGD with momentum but not as good. We still have one final adaptation to stochastic gradient descent to cover.

Adam

Adam, short for **Adaptive Momentum**, is currently the most widely-used optimizer and is built atop RMSProp, with the momentum concept from SGD added back in. This means that, instead of applying current gradients, we're going to apply momentums like in the SGD optimizer with momentum, then apply a per-weight adaptive learning rate with the cache as done in RMSProp.

The Adam optimizer additionally adds a bias correction mechanism. Do not confuse this with the layer's bias. The bias correction mechanism is applied to the cache and momentum, compensating for the initial zeroed values before they warm up with initial steps. To achieve this correction, both momentum and caches are divided by $1\text{-}beta^{step}$. As step raises, $beta^{step}$ approaches *0* (a fraction to the power of a rising value decreases), solving this whole expression to a fraction during the first steps and approaching *1* as training progresses. For example, *beta 1*, a fraction of momentum to apply, defaults to *0.9*. This means that, during the first step, the correction value equals:

$$1 - 0.9^1 = 1 - 0.9 = 0.1$$

With training progression, as step count rises:

$$1 - \lim_{step \to \infty} 0.9^{step} = 1 - 0 = 1$$

The same applies to the cache and the *beta 2* — in this case, the starting value is *0.001* and also approaches *1*. These values divide the momentums and the cache, respectively. Division by a fraction causes them to be multiple times bigger, significantly speeding up training in the initial stages before both tables warm up during multiple initial steps. We also previously mentioned that both of these bias-correcting coefficients go towards a value of *1* as training progresses and return parameter updates to their typical values for the later training steps. To get parameter updates, we divide the scaled momentum by the scaled square-rooted cache.

The code for the Adam Optimizer is based on the RMSProp optimizer. It adds the momentum seen from the SGD along with the *beta 1* hyper-parameter. Next, it introduces the bias correction mechanism for both the momentum and the cache. We've also modified the way the parameter updates are calculated — using corrected momentums and corrected caches, instead of gradients and caches. The full list of changes made from RMSProp are posted after the following code:

```
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # 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_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
```

```
            layer.bias_cache = self.beta_2 * layer.bias_cache + \
                (1 - self.beta_2) * layer.dbiases**2
            # Get corrected cache
            weight_cache_corrected = layer.weight_cache / \
                (1 - self.beta_2 ** (self.iterations + 1))
            bias_cache_corrected = layer.bias_cache / \
                (1 - self.beta_2 ** (self.iterations + 1))

            # Vanilla SGD parameter update + normalization
            # with square rooted cache
            layer.weights += -self.current_learning_rate * \
                             weight_momentums_corrected / \
                             (np.sqrt(weight_cache_corrected) +
                                 self.epsilon)
            layer.biases += -self.current_learning_rate * \
                             bias_momentums_corrected / \
                             (np.sqrt(bias_cache_corrected) +
                                 self.epsilon)

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

The following changes were made from copying the RMSProp class code:

1. renamed class from `Optimizer_RMSprop` to `Optimizer_Adam`
2. renamed the *rho* hyperparameter and property to *beta_2* in `__init__`
3. added *beta_1* hyperparameter and property in `__init__`
4. added *momentum* array creation in `update_params()`
5. added *momentum* calculation
6. renamed *self*`.rho` to *self*`.beta_2` with cache calculation code in `update_params()`
7. added `*_corrected` variables as corrected momentums and weights
8. replaced `layer.dweights`, `layer.dbiases`, `layer.weight_cache`, and `layer.bias_cache` with corrected arrays of values in parameter updates with momentum arrays

Back to our main neural network code. We can now set our optimizer to Adam, run the code, and see what impact these changes had:

```
optimizer = Optimizer_Adam(learning_rate=0.02, decay=1e-5)
```

With our default settings, we end with:

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.02
epoch: 100, acc: 0.683, loss: 0.772, lr: 0.01998021958261321
epoch: 200, acc: 0.793, loss: 0.560, lr: 0.019960279044701046
epoch: 300, acc: 0.850, loss: 0.458, lr: 0.019940378268975763
```

```
epoch: 400, acc: 0.873, loss: 0.374, lr: 0.01992051713662487
epoch: 500, acc: 0.897, loss: 0.321, lr: 0.01990069552930875
epoch: 600, acc: 0.893, loss: 0.286, lr: 0.019880913329158343
epoch: 700, acc: 0.900, loss: 0.260, lr: 0.019861170418772778
...
epoch: 1700, acc: 0.930, loss: 0.164, lr: 0.019665876753950384
...
epoch: 2600, acc: 0.950, loss: 0.132, lr: 0.019493367381748363
...
epoch: 9900, acc: 0.967, loss: 0.078, lr: 0.018198527739105907
epoch: 10000, acc: 0.963, loss: 0.079, lr: 0.018181983472577025
```

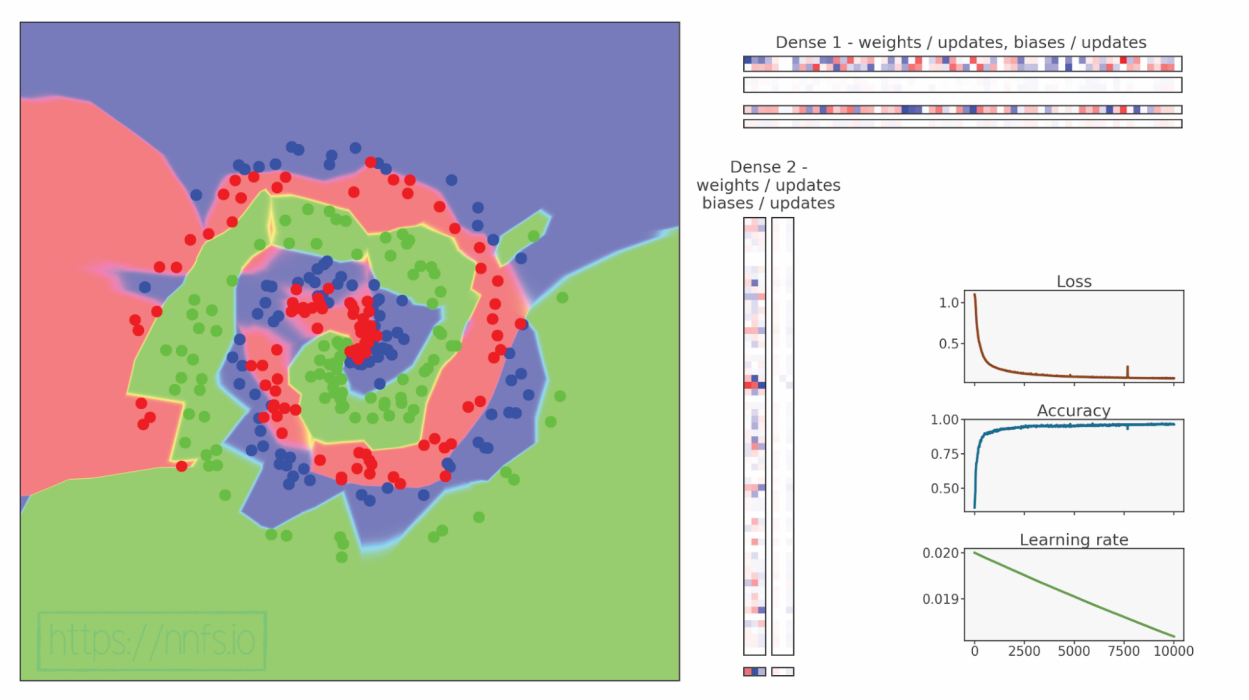


Fig 10.23: Model training with Adam optimizer.

Epilepsy Warning (quick flashing colors)

Anim 10.23: https://nnfs.io/you

This is the best result so far, but let's adjust the learning rate to be a bit higher, to *0.05* and change decay to *5e-7*:

```
optimizer = Optimizer_Adam(learning_rate=0.05, decay=5e-7)
```

In this case, loss and accuracy slightly improved, ending on:

```
>>>
epoch: 0, acc: 0.360, loss: 1.099, lr: 0.05
epoch: 100, acc: 0.713, loss: 0.684, lr: 0.04999752512250644
epoch: 200, acc: 0.827, loss: 0.511, lr: 0.04999502549496326
...
epoch: 700, acc: 0.907, loss: 0.264, lr: 0.049982531105378675
epoch: 800, acc: 0.897, loss: 0.278, lr: 0.04998003297682575
epoch: 900, acc: 0.923, loss: 0.230, lr: 0.049977535097973466
...
epoch: 2000, acc: 0.930, loss: 0.170, lr: 0.04995007490013731
...
epoch: 3300, acc: 0.950, loss: 0.136, lr: 0.04991766081847992
...
epoch: 7800, acc: 0.973, loss: 0.089, lr: 0.04980578235171948
epoch: 7900, acc: 0.970, loss: 0.089, lr: 0.04980330185930667
epoch: 8000, acc: 0.980, loss: 0.088, lr: 0.04980082161395499
...
epoch: 9900, acc: 0.983, loss: 0.074, lr: 0.049753743844839965
epoch: 10000, acc: 0.983, loss: 0.074, lr: 0.04975126853296942
```

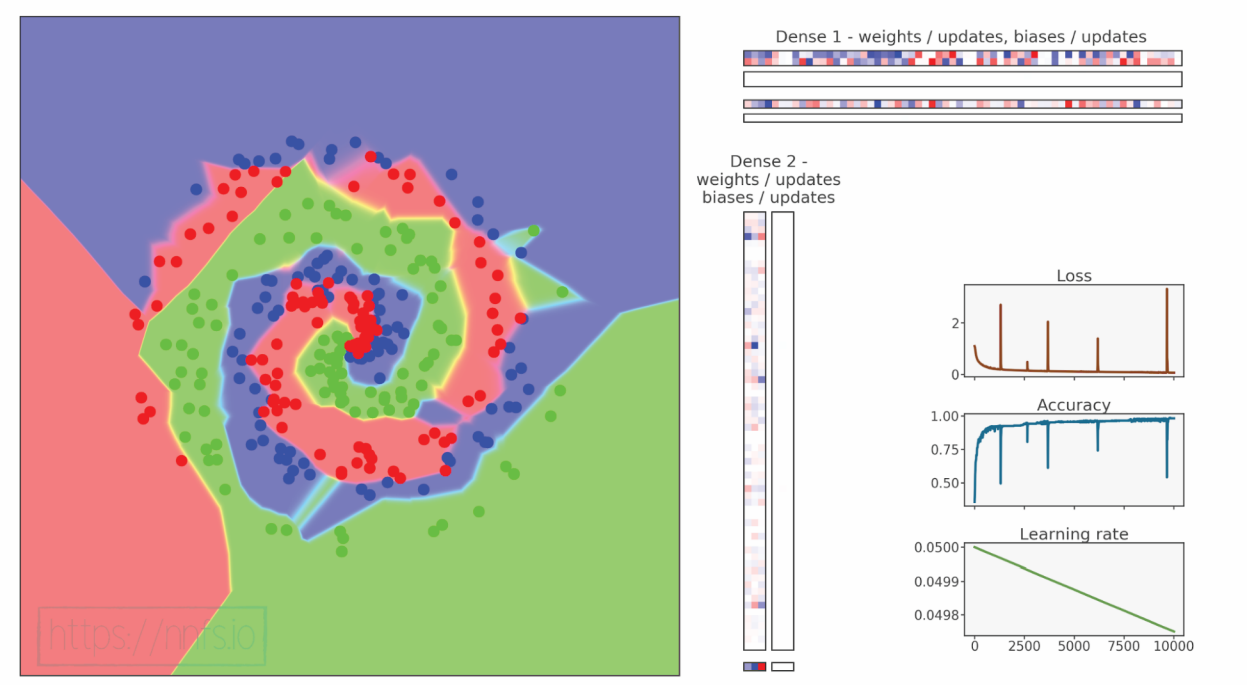


Fig 10.24: Model training with Adam optimizer (tuned).

Epilepsy Warning (quick flashing colors)

Anim 10.24 : https://nnfs.io/car

It doesn't get much better, both for accuracy and loss. While Adam has performed the best here and is usually the best optimizer of those shown, that's not always the case. It's usually a good idea to try the Adam optimizer first but to also try the others, especially if you're not getting the results you hoped for. Sometimes simple SGD or SGD + momentum performs better than Adam. Reasons why will vary, but keep this in mind.

We will cover choosing various hyperparameters (such as the learning rate) when training, but a general starting learning rate for SGD is *1.0*, with a decay down to *0.1*. For Adam, a good starting LR is *0.001* (*1e-3*), decaying down to *0.0001* (*1e-4*). Different problems may require different values here, but these are decent to start.

We achieved 98.3% accuracy on the generated dataset in this section, and a loss approaching perfection (0). Rather than being excited, you will soon learn to fear results this good, or at least approach them cautiously. There are cases where you can truly achieve valid results as good as these, but, in this case, we've been ignoring a major concept in machine learning: out-of-sample testing data (which can shed light on over-fitting), which is the subject of the next section.

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
            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)


# 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


# 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
```

```
# 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
```

```
# Adam optimizer
class Optimizer_Adam:

    # Initialize optimizer - set settings
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
                 beta_1=0.9, beta_2=0.999):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.beta_1 = beta_1
        self.beta_2 = beta_2

    # 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_momentums = np.zeros_like(layer.weights)
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)
            layer.bias_cache = np.zeros_like(layer.biases)

        # Update momentum  with current gradients
        layer.weight_momentums = self.beta_1 * \
                                 layer.weight_momentums + \
                                 (1 - self.beta_1) * layer.dweights
        layer.bias_momentums = self.beta_1 * \
                               layer.bias_momentums + \
                               (1 - self.beta_1) * layer.dbiases
        # Get corrected momentum
        # self.iteration is 0 at first pass
        # and we need to start with 1 here
        weight_momentums_corrected = layer.weight_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        bias_momentums_corrected = layer.bias_momentums / \
            (1 - self.beta_1 ** (self.iterations + 1))
        # Update cache with squared current gradients
        layer.weight_cache = self.beta_2 * layer.weight_cache + \
            (1 - self.beta_2) * layer.dweights**2
```

```
            layer.bias_cache = self.beta_2 * layer.bias_cache + \
                (1 - self.beta_2) * layer.dbiases**2
            # Get corrected cache
            weight_cache_corrected = layer.weight_cache / \
                (1 - self.beta_2 ** (self.iterations + 1))
            bias_cache_corrected = layer.bias_cache / \
                (1 - self.beta_2 ** (self.iterations + 1))

            # Vanilla SGD parameter update + normalization
            # with square rooted cache
            layer.weights += -self.current_learning_rate * \
                             weight_momentums_corrected / \
                             (np.sqrt(weight_cache_corrected) +
                                 self.epsilon)
            layer.biases += -self.current_learning_rate * \
                             bias_momentums_corrected / \
                             (np.sqrt(bias_cache_corrected) +
                                 self.epsilon)

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


# 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, y_pred, y_true):

        # Number of samples
        samples = len(y_pred)
        # Number of labels in every sample
        # We'll use the first sample to count them
        labels = len(y_pred[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 / y_pred
        # 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, y_pred, y_true):

        # Number of samples
        samples = len(y_pred)

        # 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 = y_pred.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 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_Adam(learning_rate=0.05, decay=5e-7)

# 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()
```

Supplementary Material: https://nnfs.io/ch10 Chapter code, further resources, and errata for this chapter.