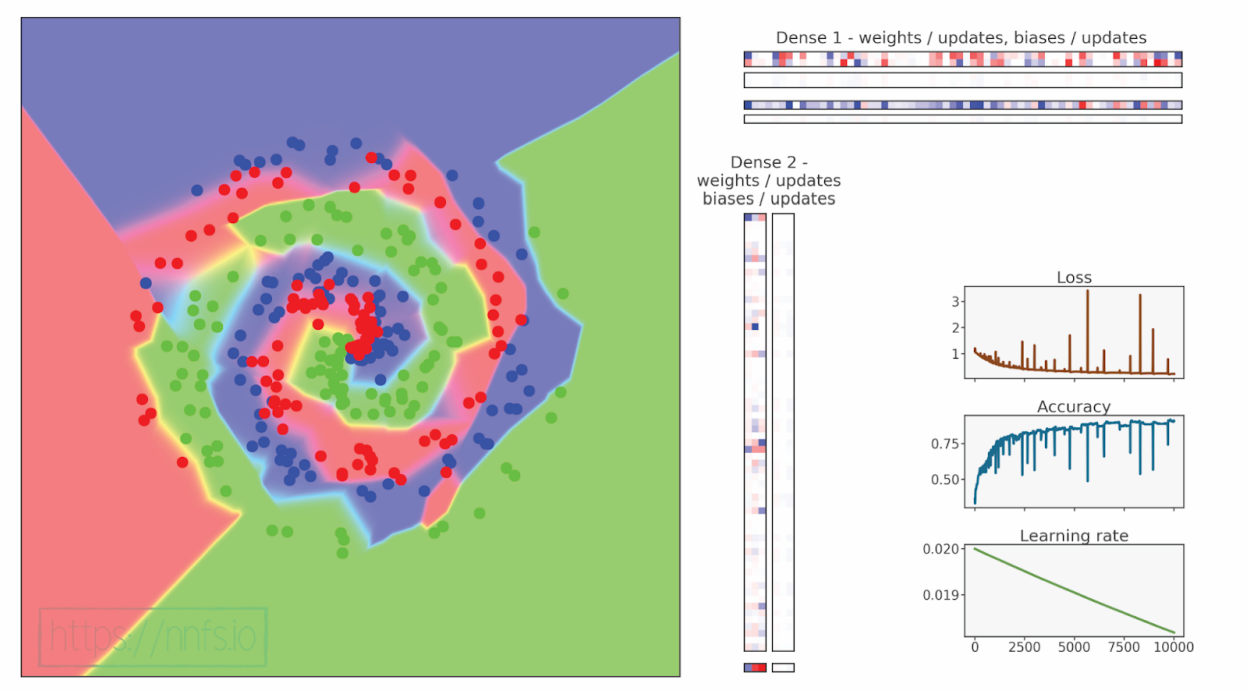

Fig 10.22: Model training with RMSProp optimizer (tuned).

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-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 cache 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
```

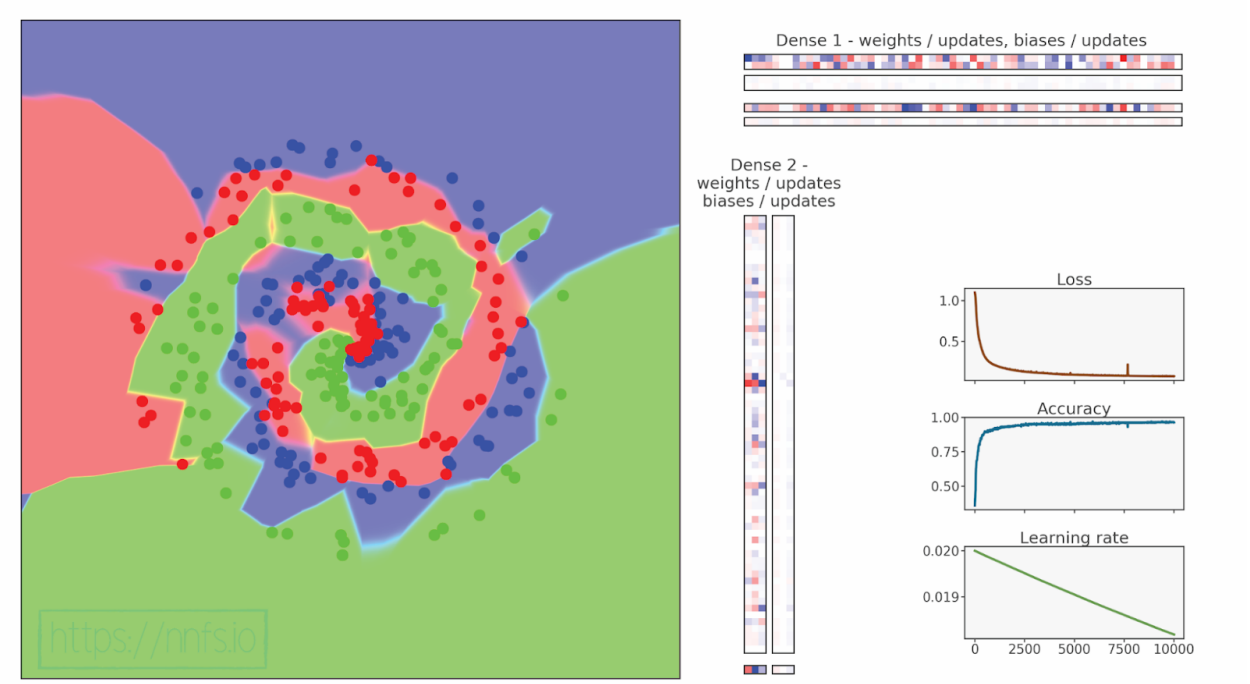

Fig 10.23: Model training with Adam optimizer.

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
```

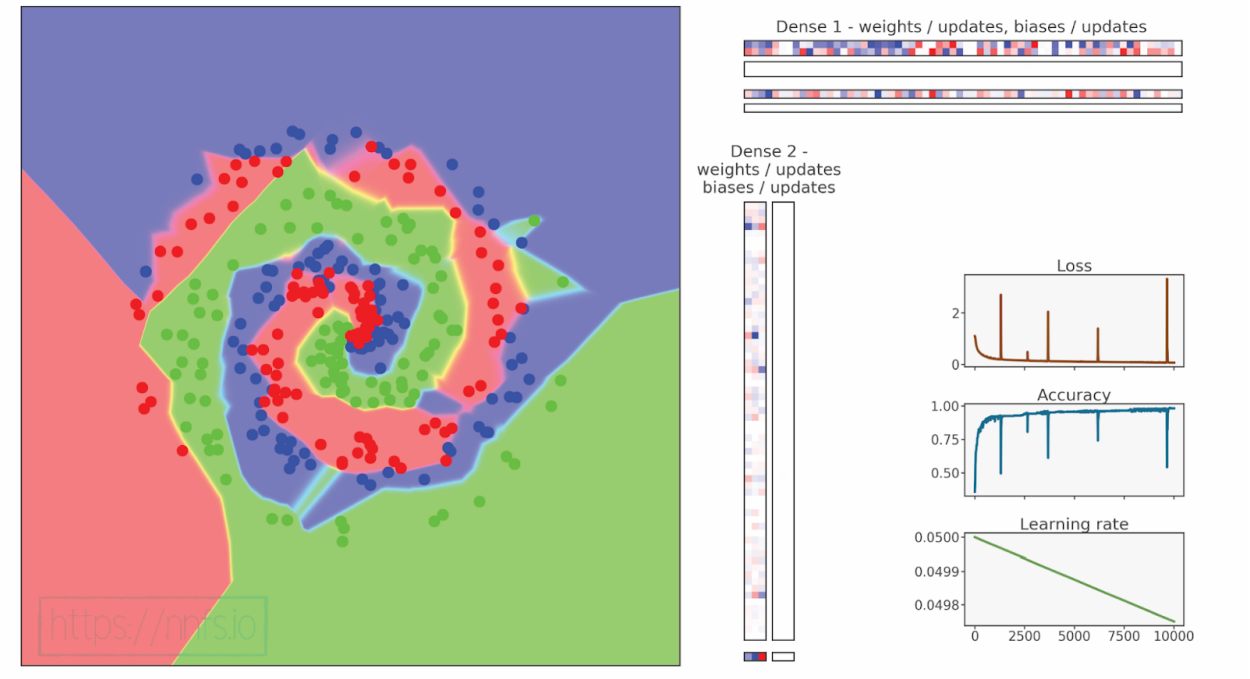

Fig 10.24: Model training with Adam optimizer (tuned).
Epilepsy Warning (quick flashing colors)

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


# 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, dvalues, y_true):

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

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

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

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

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

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

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

        # Number of samples
        samples = len(dvalues)

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

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


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

# Create Dense layer with 2 input features and 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.

Chapter 11

Testing with Out-of-Sample Data

Up to this point, we've created a model that is seemingly 98% accurate at predicting the testing dataset that we've generated. These generated data are created based on a very clear set of rules outlined in the *spiral_data* function. The expectation is that a well-trained neural network can learn a representation of these rules and use this representation to predict classes of additional generated data.

Imagine that you've trained a neural network model to read license plates on vehicles. The expectation for a well-trained model, in this case, would be that it could see future examples of license plates and still accurately predict them (a prediction, in this case, would be correctly identifying the characters on the license plate).

The complexity of neural networks is their biggest issue and strength. By having a massive amount of tunable parameters, they are exceptional at "fitting" to data. This is a gift, a curse,

and something that we must constantly try to balance. With enough neurons, a model can easily memorize a dataset; however, it can not generalize the data with too few. This is one reason why we do not simply solve problems with neural networks by using the most neurons or biggest models possible.

At the moment, we're uncertain whether our latest neural network's 98% accuracy is due to learning to meaningfully represent the underlying data-generation function or instead **overfitting** the data. So far, we have only tuned hyper-parameters to achieve the highest possible accuracy on the training data, and have never tried to challenge the model with the previously unseen data.

Overfitting is effectively just memorizing the data without any understanding of it. An overfit model will do very well predicting the data that it has already seen, but often significantly worse on unseen data.

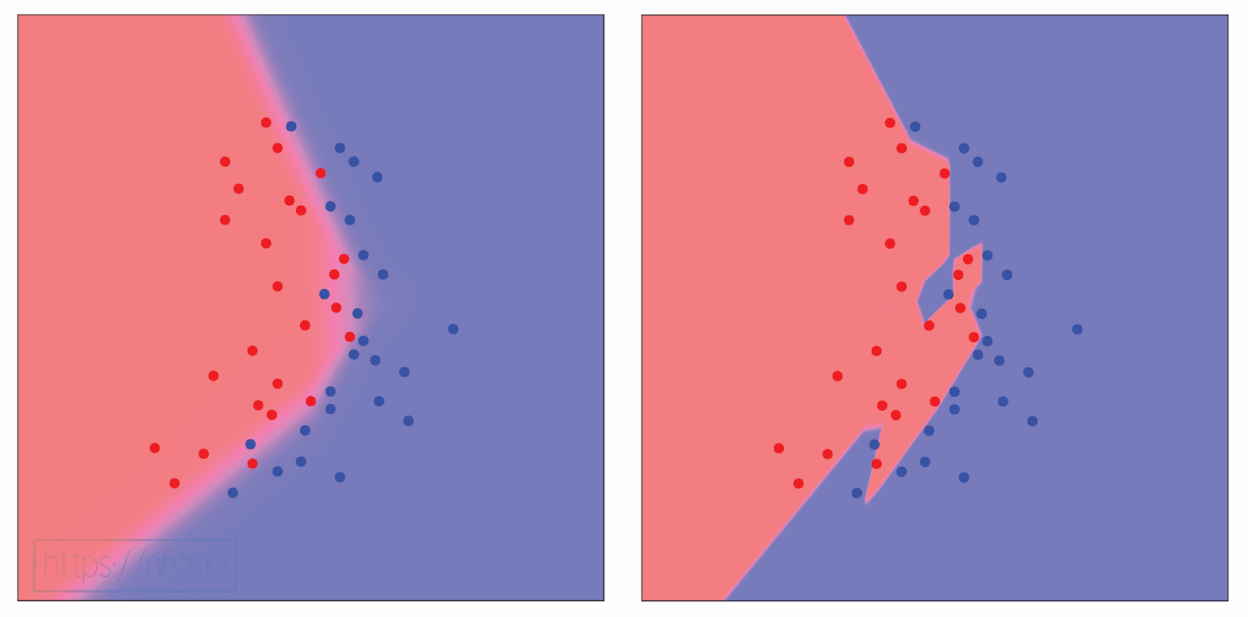

Fig 11.01: Good generalization (left) and overfitting (right) on the data

The left image shows an example of generalization. In this example, the model learned to separate red and blue data points, even if some of them will be predicted incorrectly. One reason for this might be the data that contains some "confusing" samples. When you look at the image, you can see that, for example, some of these blue dots might not be there, which would raise the data quality and make it easier to fit. A good dataset is one of the biggest challenges with neural networks. The image on the right shows the model that memorized the data, fitting them perfectly and ruining generalization.

Without knowing if a model overfits the training data, we cannot trust the model's results. For this reason, it's essential to have both **training** and **testing data** as separate sets for different purposes.

Training data should only be used to train a model. The **testing,** or **out-of-sample** data, should only be used to validate a model's performance after training (we are using the testing data during training later in this chapter for demonstration purposes only). The idea is that some data are reserved and withheld from the training data for testing the model's performance.

In many cases, one can take a random sampling of available data to train with and make the remaining data the testing dataset. You still need to be very careful about information leaking through. One common area where this can be problematic is in time-series data. Consider a scenario where you have data from sensors collected every second. You might have millions of observations collected, and randomly selecting your data for the **testing** data might result in samples in your **testing** dataset that are only a second in time apart from your **training** data, thus are very similar. This means overfitting can spill into your testing data, and the model can achieve good results on both the training and the testing data, which won't mean it generalized well. Randomly allocating time-series data as testing data may be very similar to training data. Both datasets must differ enough to prove the model's ability to generalize. In time-series data, a better approach is to take multiple slices of your data, entire blocks of time, and reserve those for testing.

Other biases like these can sneak into your testing dataset, and this is something you must be vigilant about, carefully considering if data leakage has occurred and how to truly isolate **out-of-sample** data.

In our case, we can use our data-generating function to create new data that will serve as out-of-sample/testing data:

```
# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)
```

Given what was just said about overfitting, it may look wrong to only generate more data, as the testing data could look similar to the training data. Intuition and experience are both important to spot potential issues with out-of-sample data. By looking at the image representation of the data, we can see that another set of data generated by the same function will be adequate. This is just about as safe as it gets for out-of-sample data as the classes are partially mixing at the edges (also, we're quite literally using the "underlying function" to make more data).

With these data, we evaluate the model's performance by doing a forward pass and calculating loss and accuracy the same as before:

```
# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)

# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# 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_test)

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

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')


>>>
...
epoch: 9800, acc: 0.983, loss: 0.075, lr: 0.04975621940303483
epoch: 9900, acc: 0.983, loss: 0.074, lr: 0.049753743844839965
epoch: 10000, acc: 0.983, loss: 0.074, lr: 0.04975126853296942
validation, acc: 0.803, loss: 0.858
```

While 80.3% accuracy and a loss of *0.858* is not terrible, this contrasts with our training data that achieved 98% accuracy and a loss of 0.074. This is evidence of over-fitting. In the following image, the training data is dimmed, and validation data points are shown on top of it at the same positions for both the well-generalized (on the left) and overfitted (on the right) models.

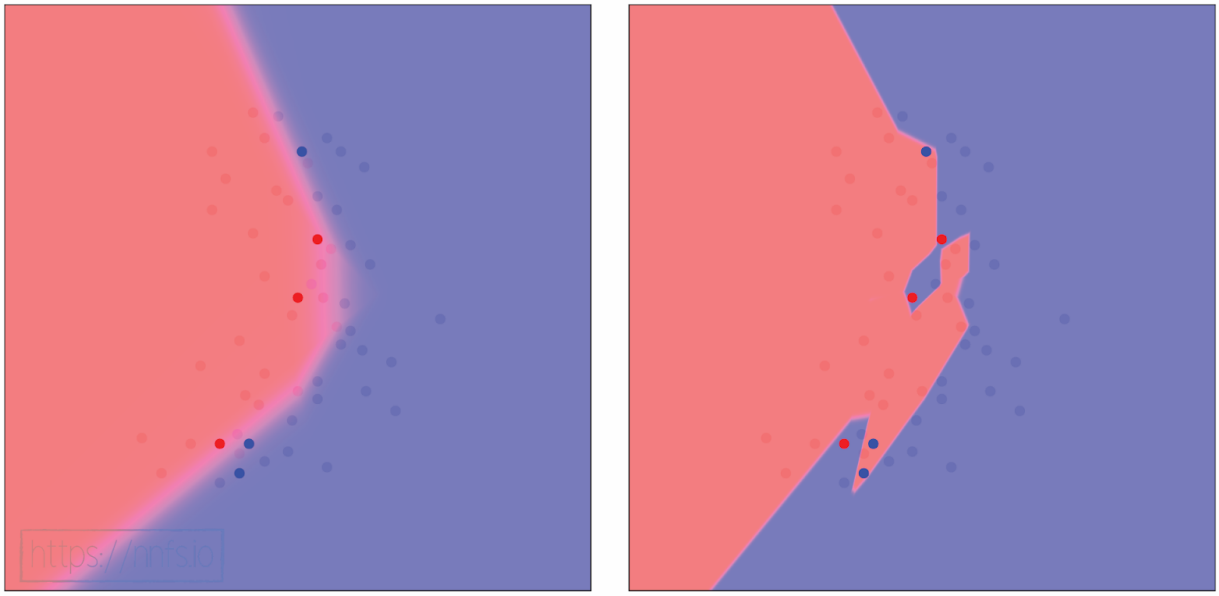

Fig 11.02: Left - prediction with well-generalized model; right - prediction mistakes with an overfit model.

We can recognize overfitting when testing data results begin to diverge in trend from training data. It will usually be the case that performance against your training data is better, but having training loss differ from test performance by over 10% approximately is a common sign of serious overfitting from our anecdotal experience. Ideally, both datasets would have identical performance. Even a small difference means that the model did not correctly predict some testing samples, implying slight overfitting of training data. In most cases, modest overfitting is not a serious problem, but something we hope to minimize.

Let's see the training process of this model once again, but with the training data, training accuracy, and loss plots dimmed. We add the test data and its loss and accuracy plotted on top of the training counterparts to show this model overfitting:

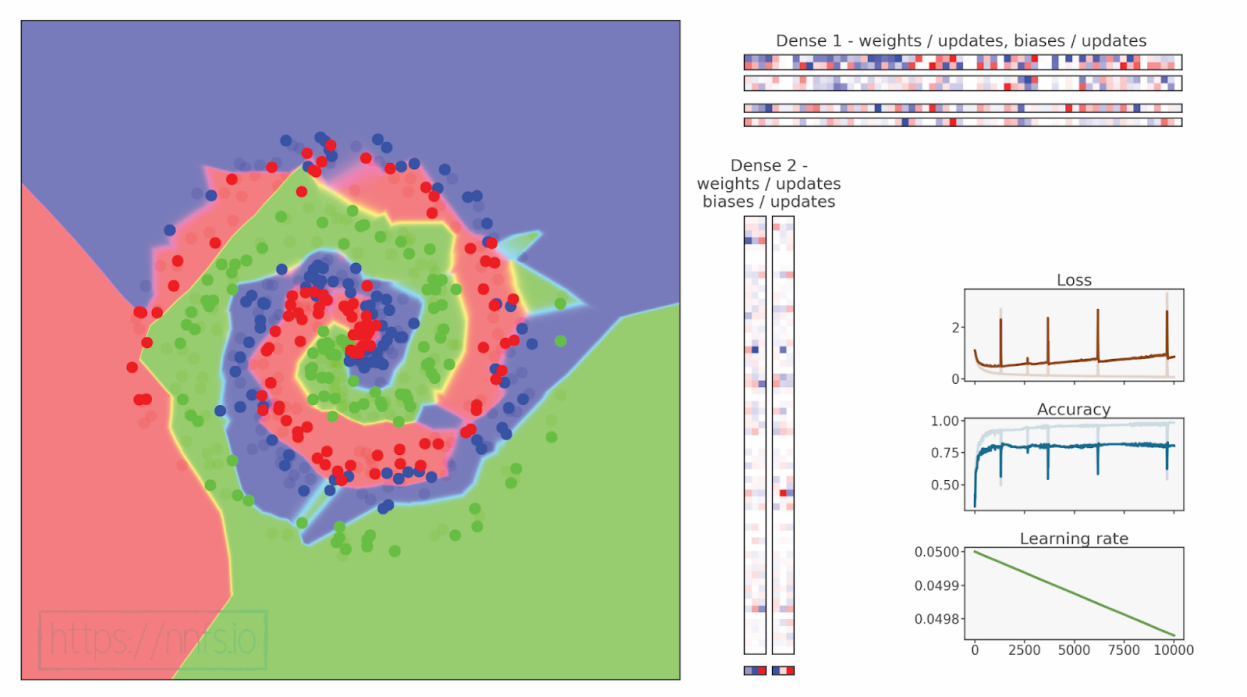

Fig 11.03: Prediction issues on the testing data — overfitted model.

Anim 11.03: https://nnfs.io/zog

This is a classic example of overfitting — the validation loss falls down, then starts rising once the model starts overfitting. The dots representing classes in the validation data can be spotted over areas of effect of other classes. Previously, we weren't aware that this was happening; we were just seeing very good training results. That's why we usually should use the testing data to test the model after training. The model is currently tuned to achieve the best possible score on the training data, and most likely the learning rate is too high, there are too many training epochs, or the model is too big. There are other possible causes and ways to fix this, but this is the topic of the following chapters. In general, the goal is to have the testing loss identical to the training loss, even if that means higher loss and lower accuracy on the training data. Similar performance on both datasets means that model generalized instead of overfitting on the training data.

As mentioned, one option to prevent overfitting is to change the model's size. If a model is not learning at all, one solution might be to try a larger model. If your model is learning, but there's a divergence between the training and testing data, it could mean that you should try a smaller model. One general rule to follow when selecting initial model hyperparameters is to find the smallest model possible that still learns. Other possible ways to avoid overfitting are regularization techniques we'll discuss in chapter 14, and the *Dropout* layer explained in chapter 15. Often the divergence of the training and testing data can take a long time to occur. The process of trying different model settings is called hyperparameter searching. Initially, you can very quickly (usually within minutes) try different settings (e.g., layer sizes) to see if the models are learning *something*. If they are, train the models fully — or at least significantly longer — and compare results to pick the best set of hyperparameters. Another possibility is to create a list of different hyperparameter sets and train the model in a loop using each of those sets at a time to pick the best set at the end. The reasoning here is that the fewer neurons you have, the less chance you have that the model is memorizing the data. Fewer neurons can mean it's easier for a neural network to generalize (actually learn the meaning of the data) compared to memorizing the data. With enough neurons, it's easier for a neural network to memorize the data. Remember that the neural network wants to decrease training loss and follows the path of least resistance to meet that objective. Our job as the programmer is to make the path to generalization the easiest path. This can often mean our job is actually to make the path to lowering loss for the model more challenging!

Supplementary Material: https://nnfs.io/ch11
Chapter code, further resources, and errata for this chapter.

Chapter 12

Validation Data

In the chapter on optimization, we used hyperparameter tuning to select hyperparameters that lead to better results, but one more thing requires clarification. We *should not* check different hyperparameters using the test dataset; if we do that, we're going to be manually optimizing the model to the test dataset, biasing it towards overfitting these data, and these data are supposed to be used only to perform the last check if the model trains and generalizes well. In other words, if we're tuning our network's parameters to fit the testing data, then we're essentially optimizing our network on the testing data, which is another way for overfitting on these data.

Thus, hyperparameter tuning using the test dataset is a mistake. The test dataset should only be used as unseen data, not informing the model in any way, which hyperparameter tuning is, other than to test performance.

Hyperparameter tuning can be performed using yet another dataset called **validation data**. The test dataset needs to contain real out-of-sample data, but with a validation dataset, we have more freedom with choosing data. If we have a lot of training data and can afford to use some for validation purposes, we can take it as an out-of-sample dataset, similar to a test dataset. We can now search for parameters that work best using this new validation dataset and test our model

at the end using the test dataset to see if we really tuned the model or just overfitted it to the validation data.

There are situations when we'll be short on data and cannot afford to create yet another dataset from the training data. In those situations, we have two options:

The first is to temporarily split the training data into a smaller training dataset and validation dataset for hyperparameter tuning. Afterward, with the final hyperparameter set, train the model on all the training data. We allow ourselves to do that as we tune the model to the part of training data that we put aside as validation data. Keep in mind that we still have a test dataset to check the model's performance after training.

The second possibility in situations where we are short on data is a process called **cross-validation**. Cross-validation is primarily used when we have a small training dataset and cannot afford any data for validation purposes. How it works is we split the training dataset into a given number of parts, let's say 5. We now train the model on the first 4 chunks and validate it on the last. So far, this is similar to the case described previously — we are also only using the training dataset and can validate on data that was not used for training. What makes cross-validation different is that we then swap samples. For example, if we have 5 chunks, we can call them chunks A, B, C, D, and E. We may first train on A, B, C, and D, then validate on E. We'll then train on A, B, C, E, and validate on D, doing this until we've validated on each of the 5 sample groups. This way, we do not lose any training data. We validate using the data that was not used for training during any given iteration and validate on more data than if we just temporarily split the training dataset and train on all of the samples. This validation method is often called k-fold cross-validation; here, our k is 5. Here's an example of 2 steps of cross-validation:

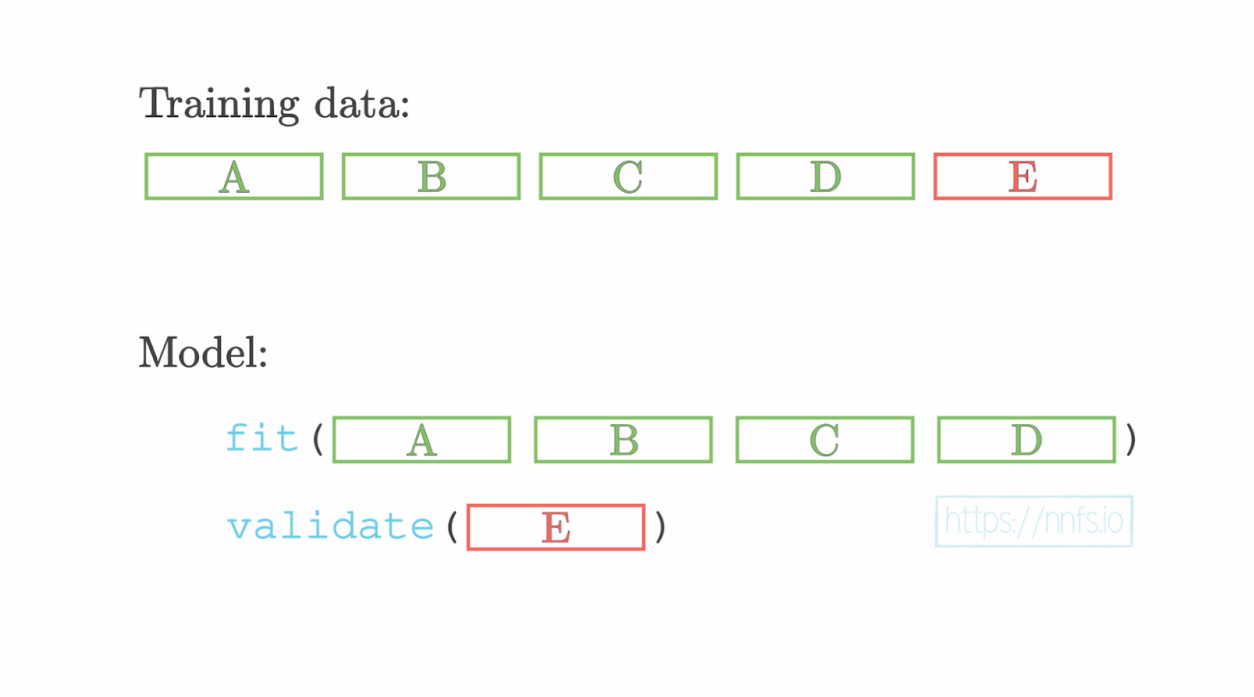

Fig 12.01: Cross-validation, first step.

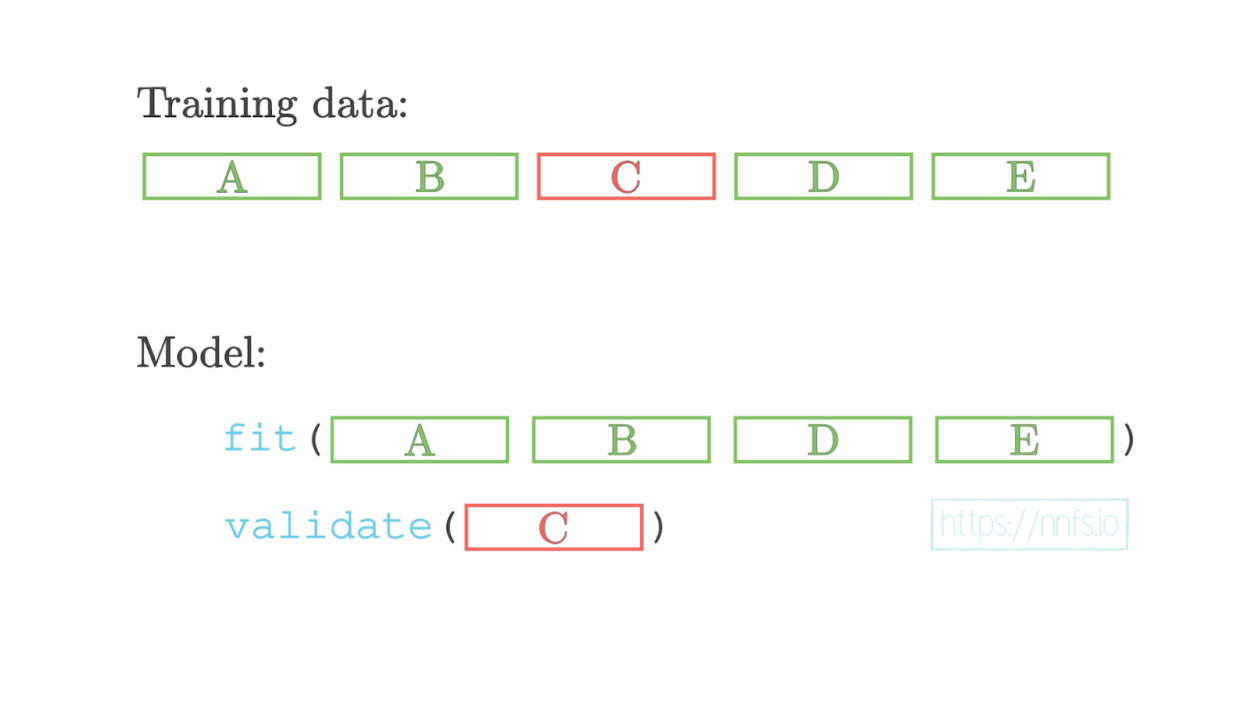

Fig 12.02: Cross-validation, third step.

Anim 12.01-12.02: https://nnfs.io/lho

When using a validation dataset and cross-validation, it is common to loop over different hyperparameter sets, leaving the code to run training multiple times, applying different settings each run, and reviewing the results to choose the best set of hyperparameters. In general, we should not loop over *all* possible setting combinations that we would like to check unless training is exceptionally fast. It's usually better to check some settings that we suspect will work well, pick the best combination of those settings, tweak them to create the next list of setting sets, and train the model on new sets. We can repeat this process as many times as we'd like.

Supplementary Material: https://nnfs.io/ch12 Chapter code, further resources, and errata for this chapter.

Chapter 13

Training Dataset

Since we are talking about datasets and testing, it's worth mentioning a few things about the training dataset and operations that we can perform on it; this technique is referred to as **preprocessing**. However, it's important to remember that any preprocessing we do to our training data also needs to be done to our validation and testing data and later done to the prediction data.

Neural networks usually perform best on data consisting of numbers in a range of 0 to 1 or -1 to 1, with the latter being preferable. Centering data on the value of 0 can help with model training as it attenuates weight biasing in some direction. Models can work fine with data in the range of 0 to 1 in most cases, but sometimes we're going to need to rescale them to a range of -1 to 1 to get training to behave or achieve better results.

Speaking of the data range, the values do not have to strictly be in the range of -1 and 1 — the model will perform well with data slightly outside of this range or with just some values being many times bigger. The case here is that when we multiply data by a weight and sum the results with a bias, we're usually passing the resulting output to an activation function. Many activation functions behave properly within this described range. For example, *softmax* outputs a vector of probabilities containing numbers in the range of 0 to 1; *sigmoid* also has an output range of 0 to 1,

but *tanh* outputs a range from -1 to 1.

Another reason why this scaling is ideal is a neural network's reliance on many multiplication operations. If we multiply by numbers above 1 or below -1, the resulting value is larger in scale than the original one. Within the -1 to 1 range, the result becomes a fraction, a smaller value. Multiplying big numbers from our training data with weights might cause floating-point overflow or instability — weights growing too fast. It's easier to control the training process with smaller numbers.

There are many terms related to data **preprocessing**: standardization, scaling, variance scaling, mean removal (as mentioned above), non-linear transformations, scaling to outliers, etc., but they are out of the scope of this book. We're only going to scale data to a range by simply dividing all of the numbers by the maximum of their absolute values. For the example of an image that consists of numbers in the range between *0* and *255*, we divide the whole dataset by *255* and return data in the range from *0* to *1*. We can also subtract *127.5* (to get a range from *-127.5* to *127.5*) and divide by 127.5, returning data in the range from -1 to 1.

We need to ensure identical scaling for all the datasets (same scale parameters). For example, we can find the maximum for training data and divide training, validation and testing data by this number. In general, we should prepare a scaler of our choice and use its instance on every dataset. It is important to remember that once we train our model and want to predict using new samples, we need to scale those new samples by using the same scaler instance we used on the training, validation, and testing data. In most cases, when we are working with data (e.g., sensor data), we will need to save the scaler object along with the model and use it during prediction as well; otherwise, results are likely to vary as the model might not effectively recognize these data without being scaled. It is usually fine to scale datasets that consist of larger numbers than the training data using a scaler prepared on the training data. If the resulting numbers are slightly outside of the *-1* to *1* range, it does not affect validation or testing negatively, since we do not train on these data. Additionally, for linear scaling, we can use different datasets to find the maximum as well, but be aware that non-linear scaling can leak the information from other datasets to the training dataset and, in this case, the scaler should be prepared on the training data only.

In cases where we do not have many training samples, we could use **data augmentation**. One easy way to understand augmentation is in the case of images. Let's imagine that our model's goal is to detect rotten fruits — apples, for example. We will take a photo of an apple from different angles and predict whether it's rotten. We should get more pictures in this case, but let's assume that we cannot. What we could do is to take photos that we have, rotate, crop, and save those as worthy data too. This way, we have added more samples to the dataset, which can help with model generalization. In general, if we use augmentation, then it's only useful if the augmentations that we make are similar to variations that we could see in reality. For example, we may refrain from using a rotation when creating a model to detect road signs as they are not being rotated in real-life scenarios (in most cases, anyway). The case of a rotated road sign, however, is one you better

consider if you're making a self-driving car. Just because a bolt came loose on a stop sign, flipping it over, doesn't mean you no longer need to stop there!

How many samples do we need to train the model? There is no single answer to this question — one model might require just a few per class, and another may require a few million or billion. Usually, a few thousand per class will be necessary, and a few tens of thousands should be preferable to start. The difference depends on the data complexity and model size. If the model has to predict sensor data with 2 simple classes, for example, if an image contains a dark area or does not, hundreds of samples per class might be enough. To train on data with many features and several classes, tens of thousands of samples are what you should start with. If you're attempting to train a chatbot the intricacies of written language, then you're going to likely want at least millions of samples.

Supplementary Material: https://nnfs.io/ch13
Chapter code, further resources, and errata for this chapter.

Chapter 14

L1 and L2 Regularization

Regularization methods are those which reduce generalization error. The first forms of regularization that we'll address are **L1** and **L2 regularization**. L1 and L2 regularization are used to calculate a number (called a **penalty**) added to the loss value to penalize the model for large weights and biases. Large weights might indicate that a neuron is attempting to memorize a data element; generally, it is believed that it would be better to have many neurons contributing to a model's output, rather than a select few.

Forward Pass

L1 regularization's penalty is the sum of all the absolute values for the weights and biases. This is a linear penalty as regularization loss returned by this function is directly proportional to parameter values. L2 regularization's penalty is the sum of the squared weights and biases. This non-linear approach penalizes larger weights and biases more than smaller ones because of the square function used to calculate the result. In other words, L2 regularization is commonly used as it does not affect small parameter values substantially and does not allow the model to grow weights too large by heavily penalizing relatively big values. L1 regularization, because of its linear nature, penalizes small weights more than L2 regularization, causing the model to start being invariant to small inputs and variant only to the bigger ones. That's why L1 regularization is rarely used alone and usually combined with L2 regularization if it's even used at all. Regularization functions of this type drive the sum of weights and the sum of parameters towards *0*, which can also help in cases of exploding gradients (model instability, which might cause weights to become very large values). Beyond this, we also want to dictate how much of an impact we want this regularization penalty to carry. We use a value referred to as **lambda** in this equation — where a higher value means a more significant penalty.

L1 weight regularization:

$$L_{1w} = \lambda \sum_{m} |w_m|$$

L1 bias regularization:

$$L_{1b} = \lambda \sum_{n} |b_n|$$

L2 weight regularization:

$$L_{2w} = \lambda \sum_{m} w_m^2$$

L2 bias regularization:

$$L_{2b} = \lambda \sum_{n} b_n^2$$

Overall loss:

$$Loss = DataLoss + L_{1w} + L_{1b} + L_{2w} + L_{2b}$$

Using code notation:

```
l1w = lambda_l1w * sum(abs(weights))
l1b = lambda_l1b * sum(abs(biases))
l2w = lambda_l2w * sum(weights**2)
l2b = lambda_l2b * sum(biases**2)
loss = data_loss + l1w + l1b + l2w + l2b
```

Regularization losses are calculated separately, then summed with the data loss, to form the overall loss. Parameter *m* is an arbitrary iterator over all of the weights in a model, parameter *n* is the bias equivalent of this iterator, w_m is the given weight, and b_n is the given bias.

To implement regularization in our neural network code, we'll start with the `__init__` method of the `Dense` layer's class, which will house the **lambda** regularization strength hyperparameters, since these can be set separately for every layer:

```
# Layer initialization
def __init__(self, n_inputs, n_neurons,
             weight_regularizer_l1=0, weight_regularizer_l2=0,
             bias_regularizer_l1=0, bias_regularizer_l2=0):
    # Initialize weights and biases
    self.weights = 0.01 * np.random.randn(inputs, neurons)
    self.biases = np.zeros((1, neurons))
    # Set regularization strength
    self.weight_regularizer_l1 = weight_regularizer_l1
    self.weight_regularizer_l2 = weight_regularizer_l2
    self.bias_regularizer_l1 = bias_regularizer_l1
    self.bias_regularizer_l2 = bias_regularizer_l2
```

This method sets the lambda hyperparameters. Now we update our loss class to include the additional penalty if we choose to set the lambda hyperparameter for any of the regularizers in the layer's initialization. We will implement this code into the *Loss* class as it is common for the hidden layers. What's more, the regularization calculation is the same, regardless of

the type of loss used. It's only a penalty that is summed with the data loss value resulting in a final, overall loss value. For this reason, we're going to add a new method to a general loss class, which is inherited by all of our specific loss functions (such as our existing `Loss_CategoricalCrossentropy`). For the code of this method, we'll create the layer's regularization loss variable. We'll add to it each of the atomic regularization losses if its corresponding lambda value is greater than *0*. To perform these calculations, we read the lambda hyperparameters, weights, and biases from the passed-in layer object. For our general loss class:

```
    # Regularization loss calculation
    def regularization_loss(self, layer):

        # 0 by default
        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * \
                                   np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * \
                                   np.sum(layer.weights * \
                                          layer.weights)

        # L1 regularization - biases
        # calculate only when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * \
                                   np.sum(np.abs(layer.biases))

        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * \
                                   np.sum(layer.biases * \
                                          layer.biases)

        return regularization_loss
```

Then we'll calculate the regularization loss and add it to our calculated loss in the training loop:

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

# Calculate regularization penalty
regularization_loss = loss_function.regularization_loss(dense1) + \
                      loss_function.regularization_loss(dense2)

# Calculate overall loss
loss = data_loss + regularization_loss
```

We created a new `regularization_loss` variable and added all layer's regularization losses to it. This completes the forward pass for regularization, but this also means our overall loss has changed since part of the calculation can include regularization, which must be accounted for in the backpropagation of the gradients. Thus, we will now cover the partial derivatives for both L1 and L2 regularization.

Backward pass

The derivative of L2 regularization is relatively simple:

$$L_{2w} = \lambda \sum_m w_m^2 \quad \rightarrow \quad \frac{\partial L_{2w}}{\partial w_m} = \frac{\partial}{\partial w_m}[\lambda \sum_m w_m^2] =$$

$$= \lambda \frac{\partial}{\partial w_m} w_m^2 = \lambda \cdot 2w_m^{2-1} = 2\lambda w_m$$

This might look complicated, but is one of the simpler derivative calculations that we have to derive in this book. Lambda is a constant, so we can move it outside of the derivative term. We can remove the sum operator since we calculate the partial derivative with respect to the given parameter only, and the sum of one element equals this element. So, we only need to calculate the derivative of w^2, which we know is $2w$. From the coding perspective, we will multiply all of the weights by 2λ. We'll implement this with NumPy directly as it's just a simple multiplication operation.

L1 regularization's derivative, on the other hand, requires more explanation. In the case of L1 regularization, we must calculate the derivative of the absolute value piecewise function, which effectively multiplies a value by -1 if it is less than 0; otherwise, it's multiplied by 1. This is because the absolute value function is linear for positive values, and we know that a linear function's derivative is:

$$f(x) = x \quad \rightarrow \quad f'(x) = 1$$

For negative values, it negates the sign of the value to make it positive. In other words, it multiplies values by -1:

$$f(x) = -x \quad \rightarrow \quad f'(x) = -1$$

When we combine that:

$$abs(x) = \begin{cases} x & x > 0 \\ -x & x < 0 \end{cases} \quad \rightarrow \quad abs'(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \end{cases}$$

And the complete partial derivative of L1 regularization with respect to given weight:

$$L_{1w} = \lambda \sum_m |w_m| \quad \rightarrow \quad L'_{1w} = \frac{\partial}{\partial w_m} \lambda \sum_m |w_m| = \lambda \frac{\partial}{\partial w_m} |w_m| = \lambda \begin{cases} 1 & w_m > 0 \\ -1 & w_m < 0 \end{cases}$$

Like L2 regularization, lambda is a constant, and we calculate the partial derivative of this regularization with respect to the specific input. The partial derivative, in this case, equals 1 or -1 depending on the w_m (weight) value.
We are calculating this derivative with respect to weights, and the resulting gradient, which has the same shape as the weights, is what we'll use to update the weights. To put this into pure Python code:

```
weights = [0.2, 0.8, -0.5]  # weights of one neuron
dL1 = []  # array of partial derivatives of L1 regularization
for weight in weights:
    if weight >= 0:
        dL1.append(1)
    else:
        dL1.append(-1)
print(dL1)


>>>
[1, 1, -1]
```

You may have noticed that we're using `>= 0` in the code where the equation above clearly depicts *> 0*. If we picture the `np.abs` function, it's a line going down and "bouncing" at the value *0*, like a saw tooth. At the pointed end (i.e., the value of *0*), the derivative of the `np.abs` function is undefined, but we cannot code it this way, so we need to handle this situation and break this rule a bit.

Now let's try to modify this L1 derivative to work with multiple neurons in a layer:

```
weights = [[0.2, 0.8, -0.5, 1],  # now we have 3 sets of weights
           [0.5, -0.91, 0.26, -0.5],
           [-0.26, -0.27, 0.17, 0.87]]
dL1 = []  # array of partial derivatives of L1 regularization
for neuron in weights:
    neuron_dL1 = []  # derivatives related to one neuron
    for weight in neuron:
        if weight >= 0:
            neuron_dL1.append(1)
        else:
            neuron_dL1.append(-1)
    dL1.append(neuron_dL1)
print(dL1)


>>>
[[1, 1, -1, 1], [1, -1, 1, -1], [-1, -1, 1, 1]]
```

That's the vanilla Python version, now for the NumPy version. With NumPy, we're going to use conditions and binary masks. We'll create the gradient as an array filled with values of *1* and shaped like weights, using `np.ones_like(weights)`. Next, the condition `weights < 0` returns an array of the same shape as `dL1`, containing `0` where the condition is false and `1` where it's true. We're using this as a binary mask to `dL1` to set values to *-1* only where the condition is true (where weight values are less than 0):

```
import numpy as np

weights = np.array([[0.2, 0.8, -0.5, 1],
                    [0.5, -0.91, 0.26, -0.5],
                    [-0.26, -0.27, 0.17, 0.87]])

dL1 = np.ones_like(weights)

dL1[weights < 0] = -1

print(dL1)


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

This returned an array of the same shape containing values of 1 and -1 — the partial gradient of the `np.abs` function (we still have to multiply it by the lambda hyperparameter). We can now

take these and update the backward pass method for the dense layer object. For L1 regularization, we'll take the code above and multiply it by λ for weights and perform the same operation for biases. For L2 regularization, as discussed at the beginning of this chapter, all we need to do is take the weights/biases, multiply them by 2λ, and add that product to the gradients:

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

        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

With this, we can update our print to include new information — regularization loss and overall loss:

```
print(f'epoch: {epoch}, ' +
      f'acc: {accuracy:.3f}, ' +
      f'loss: {loss:.3f} (' +
      f'data_loss: {data_loss:.3f}, ' +
      f'reg_loss: {regularization_loss:.3f}), ' +
      f'lr: {optimizer.current_learning_rate}')
```

Then we can add weight and bias regularizer parameters when defining a layer:

```
# Create Dense layer with 2 input features and 3 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)
```

We usually add regularization terms to the hidden layers only. Even if we are calling the regularization method on the output layer as well, it won't modify gradients if we do not set the lambda hyperparameters to values other than *0*.

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,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # 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)

            # Gradients on regularization
            # L1 on weights
            if self.weight_regularizer_l1 > 0:
                dL1 = np.ones_like(self.weights)
                dL1[self.weights < 0] = -1
                self.dweights += self.weight_regularizer_l1 * dL1
            # L2 on weights
            if self.weight_regularizer_l2 > 0:
                self.dweights += 2 * self.weight_regularizer_l2 * \
                                 self.weights
            # L1 on biases
            if self.bias_regularizer_l1 > 0:
                dL1 = np.ones_like(self.biases)
                dL1[self.biases < 0] = -1
                self.dbiases += self.bias_regularizer_l1 * dL1
            # L2 on biases
            if self.bias_regularizer_l2 > 0:
                self.dbiases += 2 * self.bias_regularizer_l2 * \
                                self.biases

            # Gradient on values
            self.dinputs = np.dot(dvalues, self.weights.T)


    # ReLU activation
    class Activation_ReLU:

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

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

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

```
# Softmax activation
class Activation_Softmax:

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

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

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

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


# 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:

    # Regularization loss calculation
    def regularization_loss(self, layer):

        # 0 by default
        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * \
                                   np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * \
                                   np.sum(layer.weights * \
                                          layer.weights)
```

```
            # L1 regularization - biases
            # calculate only when factor greater than 0
            if layer.bias_regularizer_l1 > 0:
                regularization_loss += layer.bias_regularizer_l1 * \
                                       np.sum(np.abs(layer.biases))

            # L2 regularization - biases
            if layer.bias_regularizer_l2 > 0:
                regularization_loss += layer.bias_regularizer_l2 * \
                                       np.sum(layer.biases * \
                                              layer.biases)

            return regularization_loss

        # Calculates the data and regularization losses
        # given model output and ground truth values
        def calculate(self, output, y):

            # Calculate sample losses
            sample_losses = self.forward(output, y)

            # Calculate mean loss
            data_loss = np.mean(sample_losses)

            # Return loss
            return data_loss


    # Cross-entropy loss
    class Loss_CategoricalCrossentropy(Loss):

        # Forward pass
        def forward(self, y_pred, y_true):

            # Number of samples in a batch
            samples = len(y_pred)

            # Clip data to prevent division by 0
            # Clip both sides to not drag mean towards any value
            y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

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

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

            # Losses
            negative_log_likelihoods = -np.log(correct_confidences)
            return negative_log_likelihoods

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

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

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

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


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

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

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

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

        # Number of samples
        samples = len(dvalues)

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

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


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

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

# 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=0.02, 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
        data_loss = loss_activation.forward(dense2.output, y)

        # Calculate regularization penalty
        regularization_loss = \
            loss_activation.loss.regularization_loss(dense1) + \
            loss_activation.loss.regularization_loss(dense2)

        # Calculate overall loss
        loss = data_loss + regularization_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)

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

```
# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)

# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# 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_test)

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

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')


>>>
...
epoch: 10000, acc: 0.947, loss: 0.217 (data_loss: 0.157, reg_loss: 0.060),
lr: 0.019900507413187767
validation, acc: 0.830, loss: 0.435
```

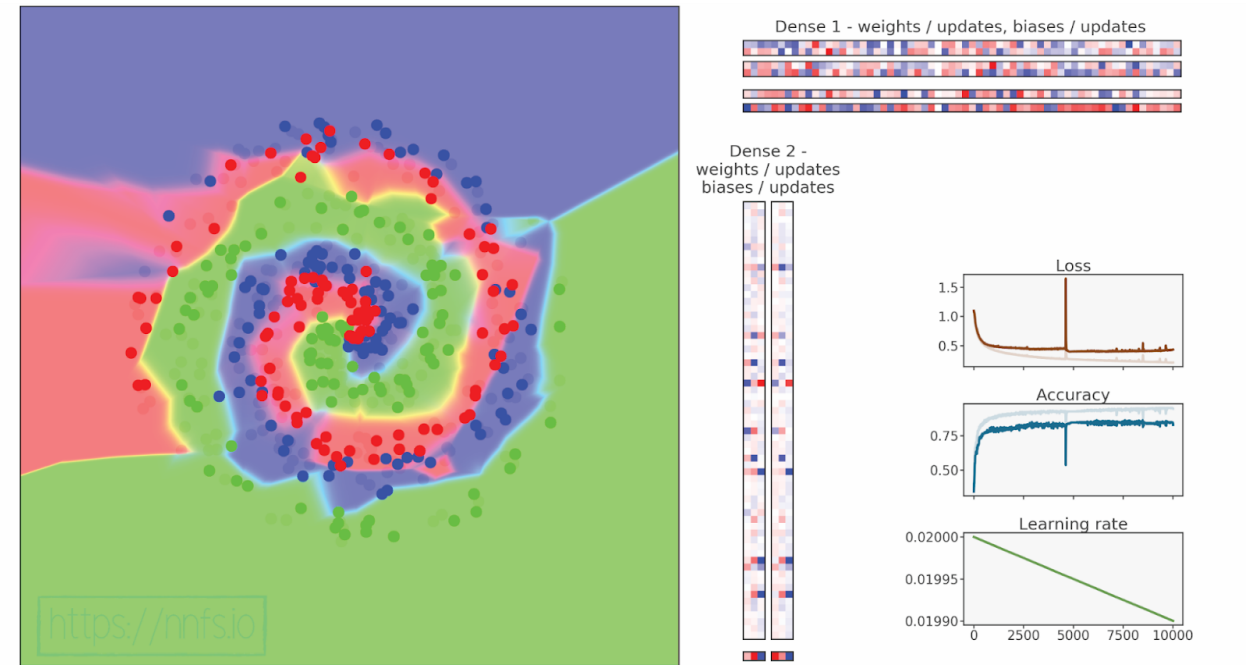

Fig 14.01: Training with regularization

Anim 14.01: https://nnfs.io/abc

This animation shows the training data in the background (dimmed dots) and the validation data in the foreground. After adding the L2 regularization term to the hidden layer, we achieved a lower validation loss (0.858 before adding regularization in, 0.435 now) and higher accuracy (0.803 before, 0.830 now). We can also take a moment to exemplify how a simple increase in data for training can make a large difference. If we grow from 100 samples to 1,000 samples:

```
# Create dataset
X, y = spiral_data(samples=1000, classes=3)
```

And run the code again:

```
>>>
epoch: 10000, acc: 0.895, loss: 0.357 (data_loss: 0.293, reg_loss: 0.063),
lr: 0.019900507413187767
validation, acc: 0.873, loss: 0.332
```

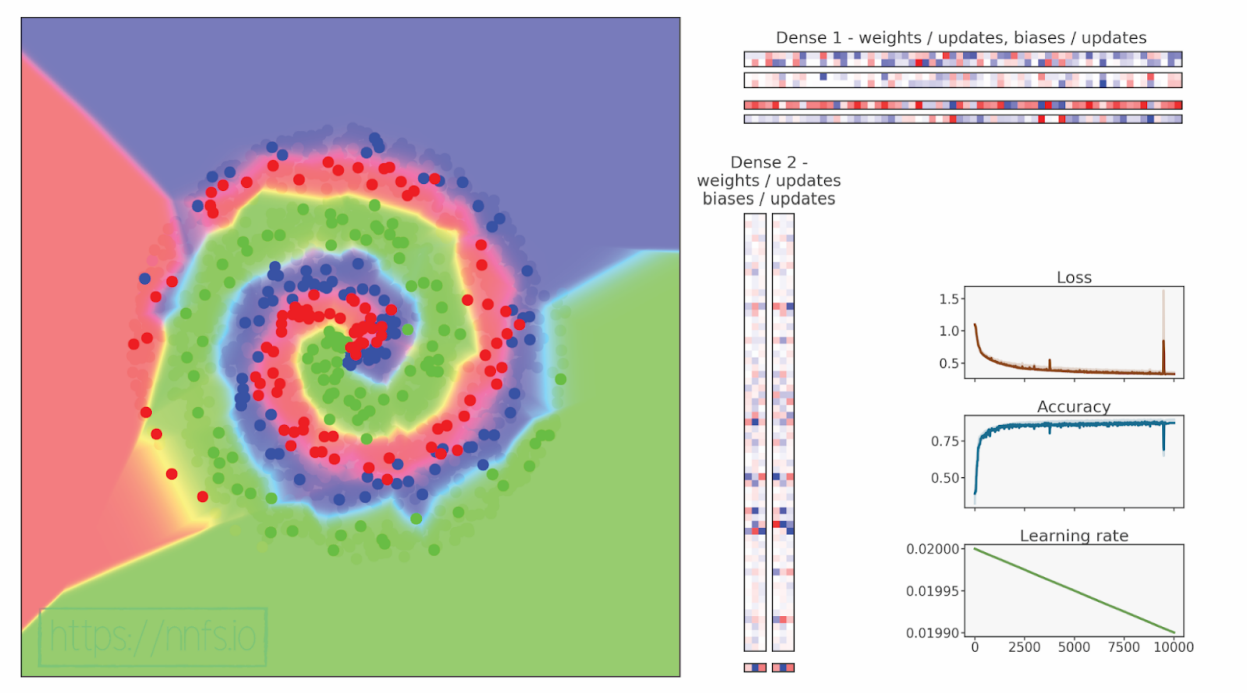

Fig 14.02: Training with regularization and more training data.

Anim 14.02: https://nnfs.io/bcd

We can see that this change alone also had a considerable impact on both validation accuracy in general, as well as the delta between the validation and training accuracies — lower accuracy and higher training loss suggest that the capacity of the model might be too low. A large delta earlier and a small one now suggests that the model was most likely overfitting previously. In theory, this regularization allows us to create much larger models without fear of overfitting (or memorization). We can test this by increasing the number of neurons per layer. Going with 128 or 256 neurons per layer helps with the training accuracy but not that much with the validation accuracy:

```
# Create Dense layer with 2 input features and 256 output values
dense1 = Layer_Dense(2, 256, weight_regularizer_l2=5e-4,
                             bias_regularizer_l2=5e-4)
```

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

# Create second Dense layer with 256 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(256, 3)


>>>
epoch: 10000, acc: 0.920, loss: 0.261 (data_loss: 0.214, reg_loss: 0.047),
lr: 0.019900507413187767
validation, acc: 0.893, loss: 0.332
```

This didn't produce much of a change in results, but raising this number again to 512 did improve validation accuracy and loss as well:

```
# Create Dense layer with 2 input features and 512 output values
dense1 = Layer_Dense(2, 512, weight_regularizer_l2=5e-4,
                             bias_regularizer_l2=5e-4)

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

# Create second Dense layer with 512 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(512, 3)


>>>
epoch: 10000, acc: 0.918, loss: 0.253 (data_loss: 0.210, reg_loss: 0.043),
lr: 0.019900507413187767
validation, acc: 0.920, loss: 0.256
```

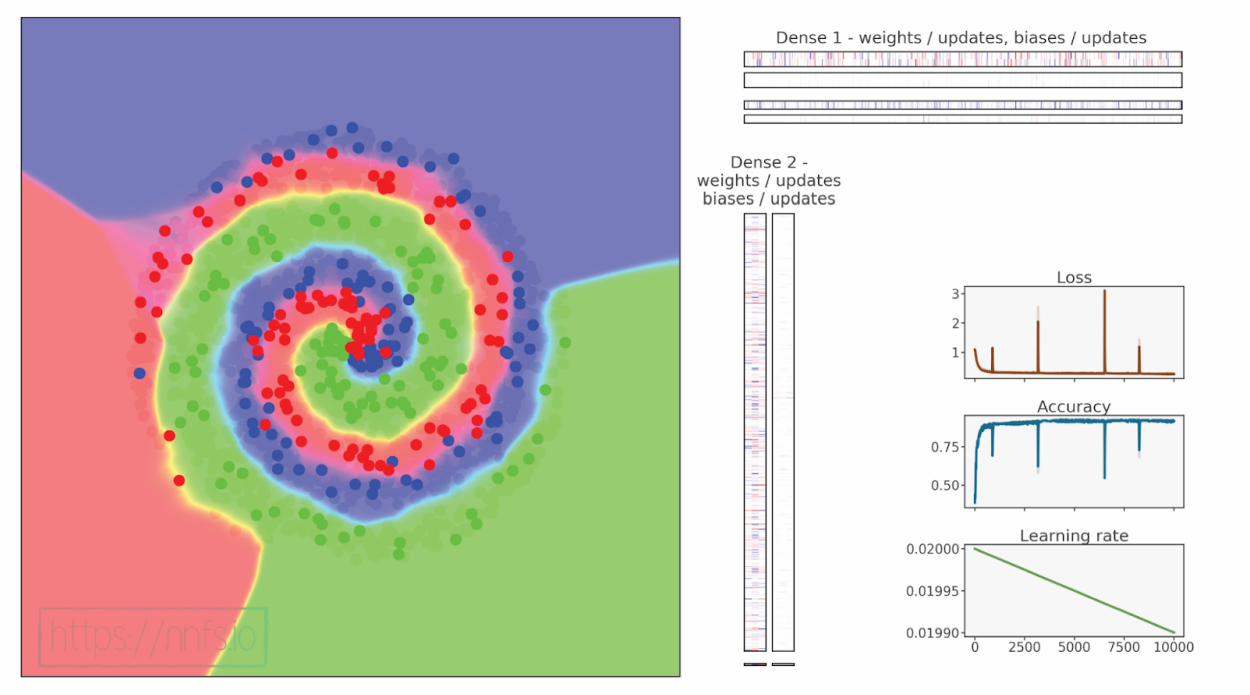

Fig 14.03: Training with regularization and more training data (tuned).

Anim 14.03: https://nnfs.io/cde

In this case, we see that the accuracies and losses for in-sample and out-of-sample data are almost identical. From here, we could add either more layers and neurons or both. Feel free to tinker with this to try to improve it. Next, we're going to cover another regularization method: **dropout**.

Supplementary Material: https://nnfs.io/ch14
Chapter code, further resources, and errata for this chapter.

Chapter 15

Dropout

Another option for neural network regularization is adding a **dropout layer**. This type of layer disables some neurons, while the others pass through unchanged. The idea here, similarly to regularization, is to prevent a neural network from becoming too dependent on any neuron or for any neuron to be relied upon entirely in a specific instance (which can be common if a model overfits the training data). Another problem dropout can help with is **co-adoption**, which happens when neurons depend on the output values of other neurons and do not learn the underlying function on their own. Dropout can also help with **noise** and other perturbations in the training data as more neurons working together mean that the model can learn more complex functions.

The Dropout function works by randomly disabling neurons at a given rate during every forward pass, forcing the network to learn how to make accurate predictions with only a random part of neurons remaining. Dropout forces the model to use more neurons for the same purpose, resulting in a higher chance of learning the underlying function that describes the data. For example, if we disable one half of the neurons during the current step, and the other half during the next step, we are forcing more neurons to learn the data, as only a part of them "sees" the data and gets updates in a given pass. These alternating halves of neurons are an example, and in reality, we'll use a hyperparameter to inform the dropout layer of the number of neurons to disable randomly.

Also, since active neurons are changing, dropout helps prevent overfitting, as the model can't use specific neurons to memorize certain samples. It's also worth mentioning that the dropout layer does not truly disable neurons, but instead zeroes their outputs. In other words, dropout does not decrease the number of neurons used, nor does it make the training process twice as fast when half the neurons are disabled.

Forward Pass

In the code, we will "turn off" neurons with a filter that is an array with the same shape as the layer output but filled with numbers drawn from a Bernoulli distribution. A **Bernoulli distribution** is a binary (or discrete) probability distribution where we can get a value of *1* with a probability of *p* and value of *0* with a probability of *q*. Let's take some random value from this distribution, r_i, then:

$$P(r_i = 1) = p$$

$$P(r_i = 0) = q = 1 - p = 1 - P(r_i = 1)$$

What this means is that the probability of this value being *1* is *p*. The probability of it being *0* is *q = 1 - p*, therefore:

$$r_i \sim Bernoulli(p)$$

This means that the given r_i is an equivalent of a value from the Bernoulli distribution with a probability *p* for this value to be *1*. If r_i is a single value from this distribution, a draw from this distribution, reshaped to match the shape of the layer outputs, can be used as a mask to these outputs.

We are returned an array filled with values of *1* with a probability of *p* and otherwise values of *0*. We then apply this filter to the output of a layer we want to add dropout to.

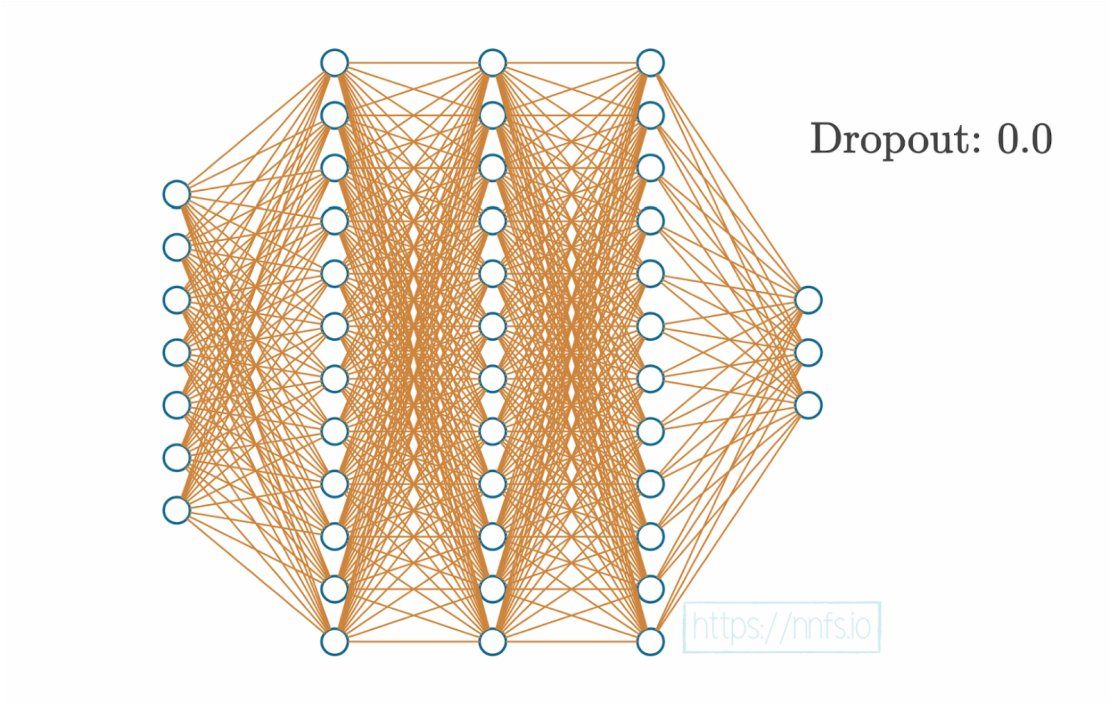

Fig 15.01: Example model with no dropout applied.

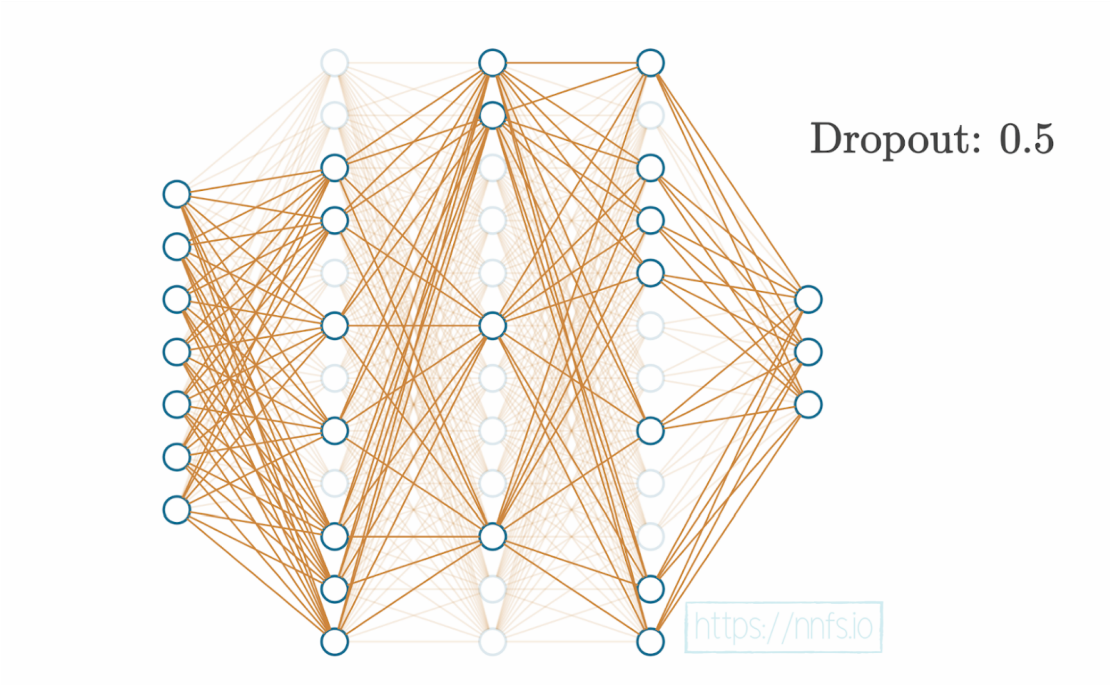

Fig 15.02: Example model with 0.5 dropout.

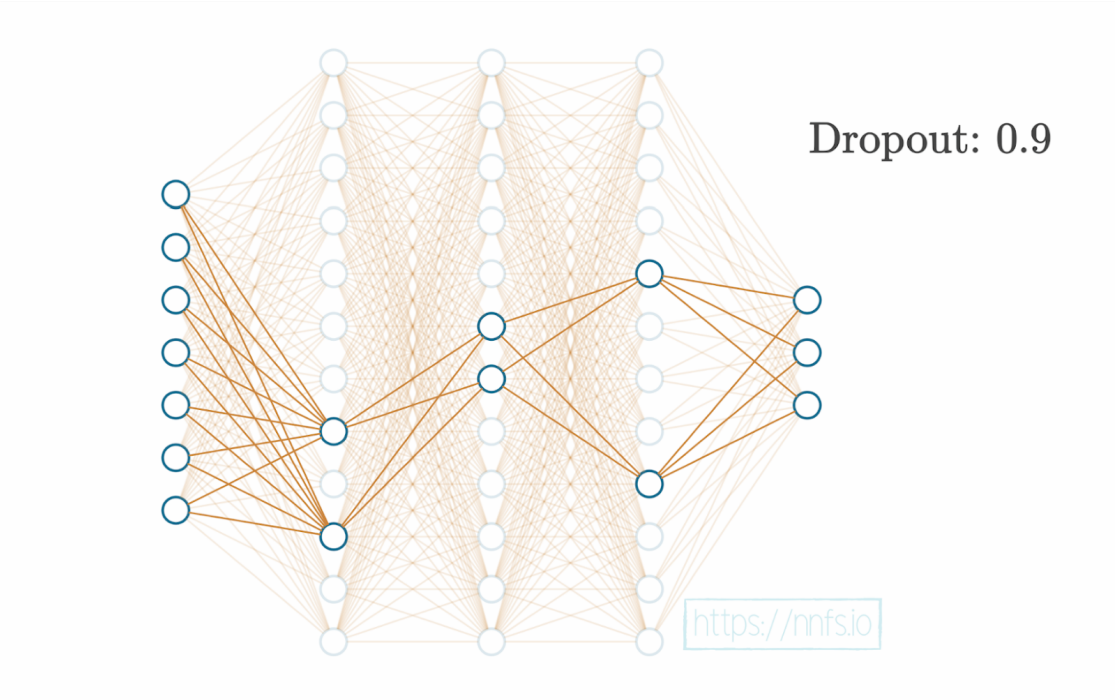

Fig 15.03: Example model with 0.9 dropout.

Anim 15.01-15.03: https://nnfs.io/def

With the code, we have one hyperparameter for a dropout layer. This is a value for the percentage of neurons to disable in that layer. For example, if you chose 0.10 for the dropout parameter, 10% of the neurons will be disabled at random during each forward pass. Before we use NumPy, we'll demonstrate this with an example in pure Python:

```
import random

dropout_rate = 0.5
# Example output containing 10 values
example_output = [0.27, -1.03, 0.67, 0.99, 0.05,
                  -0.37, -2.01, 1.13, -0.07, 0.73]
```

```
# Repeat as long as necessary
while True:

    # Randomly choose index and set value to 0
    index = random.randint(0, len(example_output) - 1)
    example_output[index] = 0

    # We might set an index that already is zeroed
    # There are different ways of overcoming this problem,
    # for simplicity we count values that are exactly 0
    # while it's extremely rare in real model that weights
    # are exactly 0, this is not the best method for sure
    dropped_out = 0
    for value in example_output:
        if value == 0:
            dropped_out += 1

    # If required number of outputs is zeroed - leave the loop
    if dropped_out / len(example_output) >= dropout_rate:
        break

print(example_output)


>>>
[0, -1.03, 0.67, 0.99, 0, -0.37, 0, 0, 0, 0.73]
```

The code is relatively rudimentary, but the idea is to keep zeroing neuron outputs (setting them to 0) randomly until we've disabled whatever target % of neurons we require. If we consider a Bernoulli distribution as a special case of a Binomial distribution with *n=1* and look at a list of available methods in NumPy, it turns out that there's a much cleaner way to do this using *numpy.random.binomial*. A binomial distribution differs from Bernoulli distribution in one way, as it adds a parameter, *n,* which is the number of concurrent experiments (instead of just one) and returns the number of successes from these *n* experiments.

`np.random.binomial()` works by taking the already discussed parameters *n* (number of experiments) and *p* (probability of the true value of the experiment) as well as an additional parameter *size*: `np.random.binomial(n, p, size)`.

The function itself can be thought of like a coin toss, where the result will be 0 or 1. The *n* is how many tosses of the coin do you want to do. The *p* is the probability for the toss result to be a 1. The overall result is a sum of all toss results. The *size* is how many of these "tests" to run, and the return is a list of overall results. For example:

```
np.random.binomial(2, 0.5, size=10)
```

This will produce an array that is of size 10, where each element will be the sum of 2 coin tosses, where the probability of 1 will be 0.5, or 50%. The resulting array:

```
array([0, 0, 1, 2, 0, 2, 0, 1, 0, 2])
```

We can use this to create our dropout layer. Our goal here is to create a filter where the intended dropout % is represented as 0, with everything else as 1. For example, let's say we have a dropout layer that we'll add after a layer that consists of 5 neurons, and we wish to have a 20% dropout. An example of a dropout layer might look like:

```
[1, 0, 1, 1, 1]
```

As you can see, ⅕ of that list is a 0. This is an example of the filter we're going to apply to the output of the dense layer. If we multiplied a neural network's layer output by this, we'd be effectively disabling the neuron at the same index as the 0.
We can mimic that with `np.random.binomial()` by doing:

```
dropout_rate = 0.20
np.random.binomial(1, 1-dropout_rate, size=5)

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

This is based on probabilities, so there will be times when it does not look like the above array. There could be times no neurons zero out, or all neurons zero out. On average, these random draws will tend toward the probability we desire. Also, this was an example using a very small layer (5 neurons). On a realistically sized layer, you should find the probability more consistently matches your intended value.

Assume a neural network layer's output is:

```
example_output = np.array([0.27, -1.03, 0.67, 0.99, 0.05,
                           -0.37, -2.01, 1.13, -0.07, 0.73])
```

Next, let's assume our target dropout rate is 0.3, or 30%. We apply a dropout layer:

```
import numpy as np

dropout_rate = 0.3
example_output = np.array([0.27, -1.03, 0.67, 0.99, 0.05,
                           -0.37, -2.01, 1.13, -0.07, 0.73])

example_output *= np.random.binomial(1, 1-dropout_rate,
                                     example_output.shape)

print(example_output)


>>>
[ 0.27 -1.03  0.00  0.99  0.   -0.37 -2.01  1.13 -0.07  0.  ]
```

Note that our dropout rate is the ratio of neurons we intend to *disable (q)*. Sometimes, the implementation of dropout will include a rate parameter that instead means the fraction of neurons you intend to *keep (p)*. At the time of writing this, the dropout parameter in deep learning frameworks, TensorFlow and Keras, represents the neurons you intend to disable. On the other hand, the dropout parameter in PyTorch and the original paper on dropout (http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf) signal the ratio of neurons you intend to keep.

The way it's implemented is not important. What *is* important is that you know which method you're using!

While dropout helps a neural network generalize and is helpful for training, it's not something we want to utilize when predicting. It's not as simple as only omitting it because the magnitude of inputs to the next neurons can be dramatically different. If you have a dropout of 50%, for example, this would suggest that, on average, your inputs to the next layer neurons will be 50% smaller when summed, assuming they are fully-connected. What that means is that we used dropout during training, and, in this example, a random 50% of neurons output a value of 0 at each of the steps. Neurons in the next layer multiply inputs by weights, sum them, and receive values of 0 for half of their inputs. If we don't use dropout during prediction, all neurons will output their values, and this state won't match the state seen during training, since the sums will be statistically about twice as big. To handle this, during prediction, we might multiply all of the outputs by the dropout fraction, but that'd add another step for the forward pass, and there is a better way to achieve this. Instead, we want to scale the data back up after a dropout, during the training phase, to mimic the mean of the sum when all of the neurons output their values. *Example_output* becomes:

```
example_output *= np.random.binomial(1, 1-dropout_rate,
                                     example_output.shape) / \
                  (1-dropout_rate)
```

Notice that we added the division of the dropout's result by the dropout rate. Since this rate is a fraction, it makes the resulting values larger, accounting for the value lost because a fraction of the neuron outputs being zeroed out. This way, we don't have to worry about the prediction and can simply omit the dropout during prediction. In any specific example, you will find that scaling doesn't equal the same sum as before because we're randomly dropping neurons. That said, after enough samples, the scaling will average out overall. To prove this:

```
import numpy as np

dropout_rate = 0.2
example_output = np.array([0.27, -1.03, 0.67, 0.99, 0.05,
                           -0.37, -2.01, 1.13, -0.07, 0.73])
print(f'sum initial {sum(example_output)}')

sums = []
for i in range(10000):

    example_output2 = example_output * \
        np.random.binomial(1, 1-dropout_rate, example_output.shape) / \
        (1-dropout_rate)
    sums.append(sum(example_output2))

print(f'mean sum: {np.mean(sums)}')


>>>
sum initial 0.36000000000000015
mean sum: 0.36282000000000014
```

It's not exact yet, but you should get the idea.

Backward Pass

The last missing piece to implement dropout as a layer is a backward pass method. As before, we need to calculate the partial derivative of the dropout operation:

When the value of element r_i equals *1*, its function and derivative becomes the neuron's output, z, compensated for the loss value by *1-q*, where q is the dropout rate, as we just described:

$$f(z,q) = \frac{z}{1-q} \quad \rightarrow \quad \frac{\partial}{\partial z}[\frac{z}{1-q}] = \frac{1}{1-q} \cdot \frac{\partial}{\partial z} z = \frac{1}{1-q} \cdot 1 = \frac{1}{1-q}$$

That's because the derivative with respect to z of z is 1, and we treat the rest as a constant. Whenr_i*=0*:

$$f(z,q) = 0 \quad \rightarrow \quad \frac{\partial}{\partial z} 0 = 0$$

And that's because we are zeroing this element of the dropout filter, and the derivative of any constant value (including *0*) is *0*. Let's combine both cases and denote *Dropout* as *Dr*:

$$Dr_i = \begin{cases} \frac{z_i}{1-q} & r_i = 1 \\ 0 & r_i = 0 \end{cases} \quad \rightarrow \quad \frac{\partial}{\partial z_i} Dr_i = \begin{cases} \frac{1}{1-q} & r_i = 1 \\ 0 & r_i = 0 \end{cases} = \frac{r_i}{1-q}$$

i denotes the index of the given input (and the layer output). When we write a derivative of the dropout function this way, we can simplify it to a value from the Bernoulli distribution divided by *1-q*, which is identical to our scaled mask, the function the dropout applies during the forward pass, as it's also either *1* divided by *1-q*, or *0*. Thus, we can save this mask during the forward pass and use it with the chain rule as the gradient of this function.

The Code

We can now implement this concept in a new layer type, the dropout layer:

```
# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs):
        # Save input values
        self.inputs = inputs
        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask
```

Let's take this new dropout layer, and add it between our two dense layers. First defining it:

```
# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

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

# Create dropout layer
dropout1 = Layer_Dropout(0.1)
```

```
# 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)
```

During the forward pass, add in the dropout:

```
# Perform a forward pass through Dropout layer
dropout1.forward(activation1.output)

# Perform a forward pass through second Dense layer
# takes outputs of activation function of first layer as inputs
dense2.forward(dropout1.output)
```

And of course in the backward pass:

```
dropout1.backward(dense2.dinputs)
activation1.backward(dropout1.dinputs)
```

Let's also raise the learning rate a bit, from 0.02 to 0.05 and raise the learning rate decaying from 5e-7 to 5e-5 as these parameters work better with our model and dropout layer.

Full code up to now:

```
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,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        # Initialize weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros((1, n_neurons))
```

```
        # Set regularization strength
        self.weight_regularizer_l1 = weight_regularizer_l1
        self.weight_regularizer_l2 = weight_regularizer_l2
        self.bias_regularizer_l1 = bias_regularizer_l1
        self.bias_regularizer_l2 = bias_regularizer_l2

    # 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)

        # Gradients on regularization
        # L1 on weights
        if self.weight_regularizer_l1 > 0:
            dL1 = np.ones_like(self.weights)
            dL1[self.weights < 0] = -1
            self.dweights += self.weight_regularizer_l1 * dL1
        # L2 on weights
        if self.weight_regularizer_l2 > 0:
            self.dweights += 2 * self.weight_regularizer_l2 * \
                             self.weights
        # L1 on biases
        if self.bias_regularizer_l1 > 0:
            dL1 = np.ones_like(self.biases)
            dL1[self.biases < 0] = -1
            self.dbiases += self.bias_regularizer_l1 * dL1
        # L2 on biases
        if self.bias_regularizer_l2 > 0:
            self.dbiases += 2 * self.bias_regularizer_l2 * \
                            self.biases

        # Gradient on values
        self.dinputs = np.dot(dvalues, self.weights.T)
```

```
# Dropout
class Layer_Dropout:

    # Init
    def __init__(self, rate):
        # Store rate, we invert it as for example for dropout
        # of 0.1 we need success rate of 0.9
        self.rate = 1 - rate

    # Forward pass
    def forward(self, inputs):
        # Save input values
        self.inputs = inputs
        # Generate and save scaled mask
        self.binary_mask = np.random.binomial(1, self.rate,
                           size=inputs.shape) / self.rate
        # Apply mask to output values
        self.output = inputs * self.binary_mask

    # Backward pass
    def backward(self, dvalues):
        # Gradient on values
        self.dinputs = dvalues * self.binary_mask


# ReLU activation
class Activation_ReLU:

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

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

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

```
# Softmax activation
class Activation_Softmax:

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

        # Get unnormalized probabilities
        exp_values = np.exp(inputs - np.max(inputs, axis=1,
                                            keepdims=True))
        # Normalize them for each sample
        probabilities = exp_values / np.sum(exp_values, axis=1,
                                            keepdims=True)

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

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

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


# 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:

    # Regularization loss calculation
    def regularization_loss(self, layer):

        # 0 by default
        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * \
                                   np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * \
                                   np.sum(layer.weights * \
                                          layer.weights)
```

```
        # L1 regularization - biases
        # calculate only when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * \
                                   np.sum(np.abs(layer.biases))

        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * \
                                   np.sum(layer.biases * \
                                          layer.biases)

        return regularization_loss

    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y):

        # Calculate sample losses
        sample_losses = self.forward(output, y)

        # Calculate mean loss
        data_loss = np.mean(sample_losses)

        # Return loss
        return data_loss


# Cross-entropy loss
class Loss_CategoricalCrossentropy(Loss):

    # Forward pass
    def forward(self, y_pred, y_true):

        # Number of samples in a batch
        samples = len(y_pred)

        # Clip data to prevent division by 0
        # Clip both sides to not drag mean towards any value
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)

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

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

        # Losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

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

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

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

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


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

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

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

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

        # Number of samples
        samples = len(dvalues)

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

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


# Create dataset
X, y = spiral_data(samples=1000, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                            bias_regularizer_l2=5e-4)

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

# Create dropout layer
dropout1 = Layer_Dropout(0.1)

# 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-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 Dropout layer
    dropout1.forward(activation1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(dropout1.output)

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

    # Calculate regularization penalty
    regularization_loss = \
        loss_activation.loss.regularization_loss(dense1) + \
        loss_activation.loss.regularization_loss(dense2)

    # Calculate overall loss
    loss = data_loss + regularization_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)

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

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    dropout1.backward(dense2.dinputs)
    activation1.backward(dropout1.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()
```

```
# Validate the model

# Create test dataset
X_test, y_test = spiral_data(samples=100, classes=3)

# Perform a forward pass of our testing data through this layer
dense1.forward(X_test)

# 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_test)

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

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')


>>>
epoch: 9900, acc: 0.668, loss: 0.733 (data_loss: 0.717, reg_loss: 0.016), lr:
0.0334459346466437
epoch: 10000, acc: 0.688, loss: 0.727 (data_loss: 0.711, reg_loss: 0.016), lr:
0.03333444448148271
validation, acc: 0.757, loss: 0.712
```

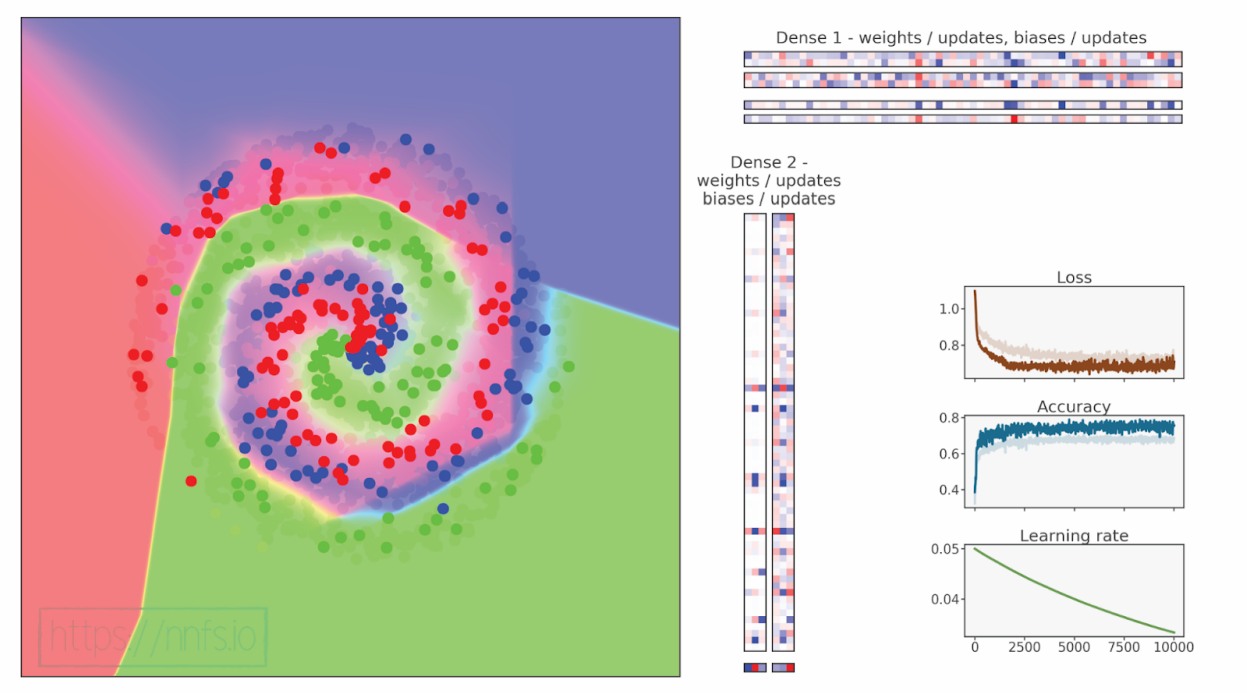

Fig 15.04: Model trained with dropout.

Anim 15.04: https://nnfs.io/efg

While our accuracy and loss have suffered considerably, we've found a scenario where our validation set performs *better* than our in-sample set (because we do not apply dropout when testing so you don't disable some of the connections). Further tweaking would likely fix the accuracy issue; for example, due to our regularization tactics, we can change our layer sizes to 512:

```
# Create Dense layer with 2 input features and 512 output values
dense1 = Layer_Dense(2, 512, weight_regularizer_l2=5e-4,
                             bias_regularizer_l2=5e-4)

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

```
# Create dropout layer
dropout1 = Layer_Dropout(0.1)

# Create second Dense layer with 512 input features
# and 3 output values
dense2 = Layer_Dense(512, 3)
```

Adding more neurons ends with:

```
epoch: 0, acc: 0.373, loss: 1.099 (data_loss: 1.099, reg_loss: 0.000), lr:
0.05
epoch: 100, acc: 0.719, loss: 0.735 (data_loss: 0.672, reg_loss: 0.063), lr:
0.04975371909050202
epoch: 200, acc: 0.782, loss: 0.627 (data_loss: 0.548, reg_loss: 0.079), lr:
0.049507401356502806
epoch: 300, acc: 0.800, loss: 0.603 (data_loss: 0.521, reg_loss: 0.082), lr:
0.0492635105177595
epoch: 400, acc: 0.802, loss: 0.595 (data_loss: 0.513, reg_loss: 0.082), lr:
0.04902201088288642
epoch: 500, acc: 0.809, loss: 0.562 (data_loss: 0.482, reg_loss: 0.079), lr:
0.048782867456949125
epoch: 600, acc: 0.836, loss: 0.521 (data_loss: 0.445, reg_loss: 0.076), lr:
0.04854604592455945
epoch: 700, acc: 0.816, loss: 0.532 (data_loss: 0.457, reg_loss: 0.076), lr:
0.048311512633460556
epoch: 800, acc: 0.839, loss: 0.515 (data_loss: 0.442, reg_loss: 0.073), lr:
0.04807923457858551
epoch: 900, acc: 0.842, loss: 0.499 (data_loss: 0.426, reg_loss: 0.072), lr:
0.04784917938657352
epoch: 1000, acc: 0.837, loss: 0.480 (data_loss: 0.408, reg_loss: 0.071),
lr: 0.04762131530072861
...
epoch: 9800, acc: 0.848, loss: 0.443 (data_loss: 0.391, reg_loss: 0.052),
lr: 0.033558173093056816
epoch: 9900, acc: 0.841, loss: 0.468 (data_loss: 0.416, reg_loss: 0.052),
lr: 0.0334459346466437
epoch: 10000, acc: 0.859, loss: 0.468 (data_loss: 0.417, reg_loss: 0.051),
lr: 0.03333444448148271
validation, acc: 0.857, loss: 0.397
```

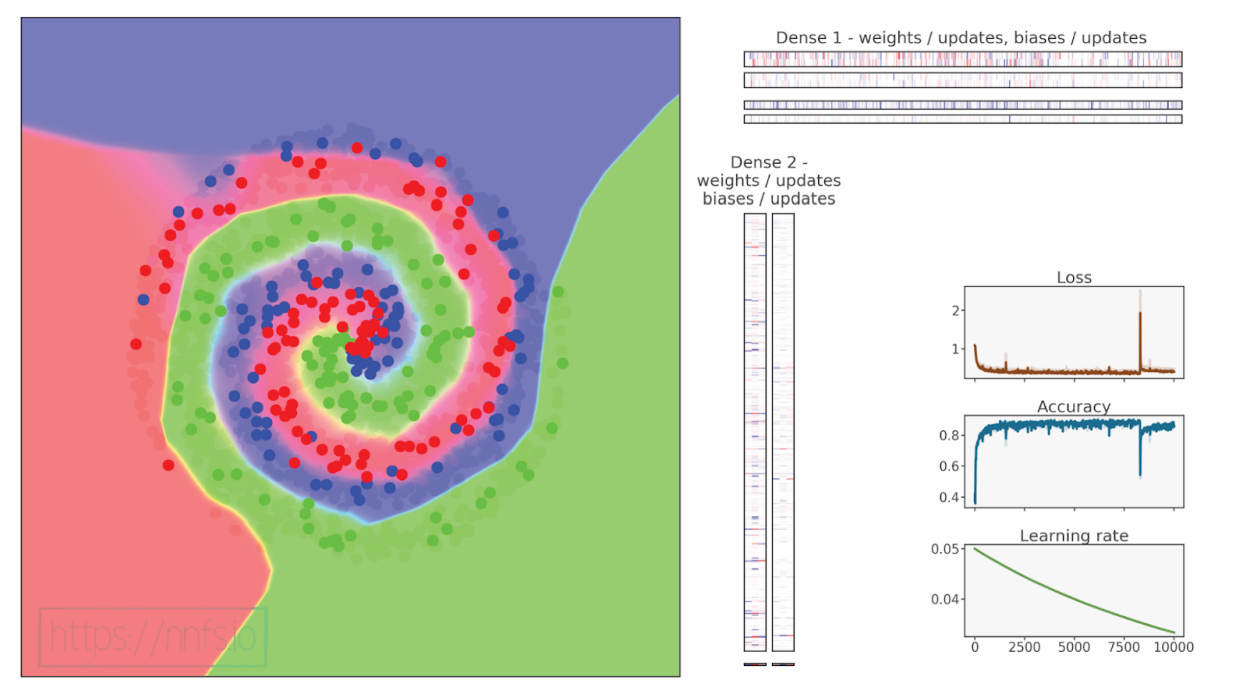

Fig 15.05: Model trained with dropout and bigger hidden layer.

Anim 15.05: https://nnfs.io/fgh

Pretty good result, but worse compared to the "no dropout" model. Interestingly, validation accuracy is close to the training accuracy with dropout — usually validation accuracy will be higher, so we might suspect these as signs of overfitting here (validation loss is lower than expected).

Supplementary Material: https://nnfs.io/ch15
Chapter code, further resources, and errata for this chapter.

Chapter 16

Binary Logistic Regression

Now that we've learned how to create and train a neural network, let's consider an alternative output layer for a neural network. Until now, we've used an output layer that is a probability distribution, where all of the values represent a confidence level of a given class being the correct class, and where these confidences sum to *1*. We're now going to cover an alternate output layer option, where each neuron separately represents two classes — 0 for one of the classes, and a 1 for the other. A model with this type of output layer is called **binary logistic regression**. This single neuron could represent two classes like *cat* vs. *dog*, but it could also represent *cat* vs. *not cat* or any combination of 2 classes, and you could have many of these. For example, a model may have two binary output neurons. One of these neurons could be distinguishing between *person/not person*, and the other neuron could be deciding between *indoors/outdoors*. Binary logistic regression is a regressor type of algorithm, which will differ as we'll use a **sigmoid** activation function for the output layer rather than **softmax**, and **binary cross-entropy** rather than **categorical cross-entropy** for calculating loss.

Sigmoid Activation Function

The sigmoid activation function is used with regressors because it "squishes" a range of outputs from negative infinity to positive infinity to be between 0 and 1. The bounds represent the two possible classes. The sigmoid equation is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

For the purpose of neural networks, we'll use our common notation:

$$\sigma_{i,j} = \frac{1}{1 + e^{-z_{i,j}}}$$

The denominator of the **Sigmoid** function contains *e* raised to the power of $z_{i,j}$, where *z*, given indices, means a singular output value of the layer that this activation function takes as input. The index *i* means the current sample, and the index *j* means the current output in this sample.

If we plot the sigmoid function:

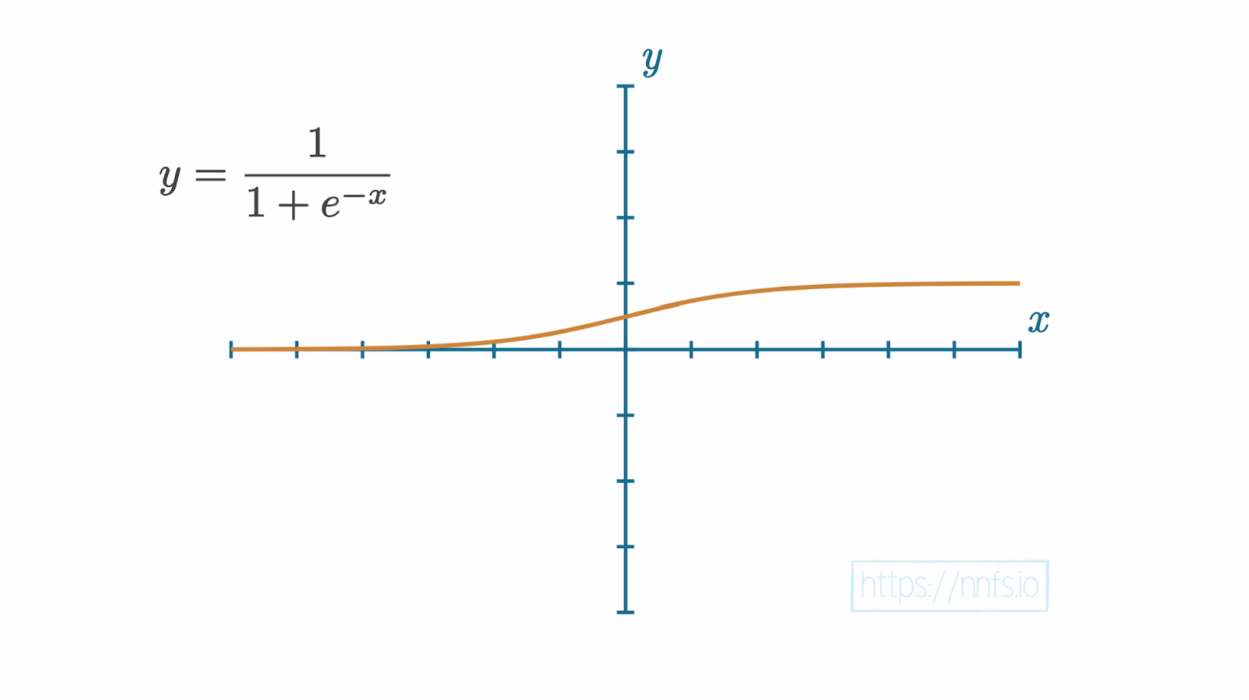

Fig 16.1: The sigmoid function graph.

Note the output from this function averages at *0.5*, and squishes down to a flat line as it approaches *0* or *1*. The sigmoid function approaches both maximum and minimum values exponentially fast. For example, for an input of *2*, the output is *~0.88*, which is already pretty close to *1*. With an input of *3*, the output is *~0.95*, and so on. It's also similar for negative values: $\sigma(-2) \approx 0.12$ and $\sigma(-3) \approx 0.05$. This property makes the sigmoid activation function a good candidate to apply to the final layer's output with a binary logistic regression model.

For commonly-used functions, such as the sigmoid function, the derivatives are almost always public knowledge. Unless you're inventing a function, you won't need to calculate derivatives by hand, but it can still be a good exercise. The sigmoid function's derivative solves to $\sigma_{i,j}(1-\sigma_{i,j})$. If you would like to leverage this fact without diving into the mathematical derivation, feel free to skip to the next section.

Sigmoid Function Derivative

Let's define the derivative of the **Sigmoid** function with respect to its input:

$$\sigma_{i,j} = \frac{1}{1 + e^{-z_{i,j}}} \quad \rightarrow \quad \sigma'_{i,j} = \frac{d}{dz_{i,j}}[\frac{1}{1 + e^{-z_{i,j}}}]$$

At this point, we might start calculating the derivative of the division operation, but, since the numerator contains just the value *1*, the whole fraction is effectively just a reciprocal of its denominator and can be represented as its negative power:

$$\frac{1}{x} = x^{-1}$$

It's easier to calculate the derivative of the power operation than the derivative of the division operation, so let's update our equation to follow this:

$$\frac{d}{dz_{i,j}}(1 + e^{-z_{i,j}})^{-1} =$$

Now, we can calculate the derivative of the expression raised to the power of the *-1*, which equals this exponent multiplied by the expression itself, raised to the power lowered by *1*. Then, following the chain rule, we have to calculate the derivative of the expression itself:

$$= -1 \cdot (1 + e^{-z_{i,j}})^{-1-1} \cdot \frac{d}{dz_{i,j}}(1 + e^{-z_{i,j}}) =$$

As we already learned, the derivative of the sum operation is the sum of derivatives:

$$= -(1 + e^{-z_{i,j}})^{-2} \cdot (\frac{d}{dz_{i,j}}1 + \frac{d}{dz_{i,j}}e^{-z_{i,j}}) =$$

The derivative of *1* with respect to $z_{i,j}$ equals *0*, as the derivative of a constant is always *0*. The

derivative of the constant *e* raised to the power $-z_{i,j}$ equals this value multiplied by the derivative of the exponent:

$$= -(1 + e^{-z_{i,j}})^{-2} \cdot (0 + e^{-z_{i,j}} \cdot \frac{d}{dz_{i,j}}[-z_{i,j}]) =$$

The derivative of the $-z_{i,j}$ with respect to $z_{i,j}$ equals *-1* as *-1* is a constant and can be moved outside of the derivative, leaving us with the derivative of $z_{i,j}$ with respect to $z_{i,j}$ which, as we know, equals *1*:

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1 \cdot \frac{d}{dz_{i,j}} z_{i,j})) = -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1)) =$$

Now we can move the minus sign outside of the parentheses and cancel out the other minus:

$$= -(1 + e^{-z_{i,j}})^{-2} \cdot (-e^{-z_{i,j}}) = (1 + e^{-z_{i,j}})^{-2} \cdot e^{-z_{i,j}} =$$

Let's rewrite the resulting equation — the expression raised to the power of *-2* can be written as its reciprocal raised to the power of *2*, then the multiplier (the value we multiply by) from the equation can become the numerator of the resulting fraction:

$$= \frac{e^{-z_{i,j}}}{(1 + e^{-z_{i,j}})^2} =$$

The denominator of this fraction can be written as the multiplication of the expression by itself instead of raising it to the power of *2*:

$$= \frac{e^{-z_{i,j}}}{(1 + e^{-z_{i,j}})(1 + e^{-z_{i,j}})} =$$

Now we can split this fraction into two separate ones — one containing *1* in the numerator and the other one *e* to the power of $-z_{i,j}$, both having each of the expressions that are separated by the multiplication operator in the denominator in their respective denominators. We can do this as we are performing the multiplication operation between both fractions:

$$= \frac{1}{1 + e^{-z_{i,j}}} \cdot \frac{e^{-z_{i,j}}}{1 + e^{-z_{i,j}}} =$$

If you remember the equation of the sigmoid function, you might already see where we are going with this — the multiplicand (the value that is being multiplied by the multiplier) is the equation of the sigmoid function. Let's work on this equation further — it'd be ideal if the numerator of the

multiplicator could be represented as some sort of equation containing the sigmoid function's equation as well. What we can do is add *1* and remove *1* from it as it won't change its value:

$$= \frac{1}{1 + e^{-z_{i,j}}} \cdot \frac{1 + e^{-z_{i,j}} - 1}{1 + e^{-z_{i,j}}} =$$

What this allows us to do is split the multiplicator into two separate fractions by the minus sign in the multiplicator:

$$= \frac{1}{1 + e^{-z_{i,j}}} \cdot (\frac{1 + e^{-z_{i,j}}}{1 + e^{-z_{i,j}}} - \frac{1}{1 + e^{-z_{i,j}}}) =$$

The minuend (the value we are subtracting from) of the multiplicator equals *1* as the numerator, and the denominator of the fraction are equal, and the subtrahend (the value we are subtracting from the minuend) is actually the equation of the sigmoid function as well:

$$= \frac{1}{1 + e^{-z_{i,j}}} \cdot (1 - \frac{1}{1 + e^{-z_{i,j}}}) = \sigma_{i,j} \cdot (1 - \sigma_{i,j})$$

It turns out that the derivative of the sigmoid function equals this function multiplied by the difference of *1* and this function as well. That allows us to easily write this derivative in the code.

Full solution:

$$\sigma_{i,j} = \frac{1}{1+e^{-z_{i,j}}} \quad \rightarrow \quad \sigma'_{i,j} = \frac{d}{dz_{i,j}}[\frac{1}{1+e^{-z_{i,j}}}] = \frac{d}{dz_{i,j}}(1+e^{-z_{i,j}})^{-1} =$$

$$= -1 \cdot (1+e^{-z_{i,j}})^{-1-1} \cdot \frac{d}{dz_{i,j}}(1+e^{-z_{i,j}}) = -(1+e^{-z_{i,j}})^{-2} \cdot (\frac{d}{dz_{i,j}}1 + \frac{d}{dz_{i,j}}e^{-z_{i,j}}) =$$

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (0 + e^{-z_{i,j}} \cdot \frac{d}{dz_{i,j}}[-z_{i,j}]) =$$

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1 \cdot \frac{d}{dz_{i,j}}z_{i,j})) = -(1+e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1)) =$$

$$= -(1+e^{-z_{i,j}})^{-2} \cdot (-e^{-z_{i,j}}) = (1+e^{-z_{i,j}})^{-2} \cdot e^{-z_{i,j}} =$$

$$= \frac{e^{-z_{i,j}}}{(1+e^{-z_{i,j}})^2} = \frac{e^{-z_{i,j}}}{(1+e^{-z_{i,j}})(1+e^{-z_{i,j}})} = \frac{1}{1+e^{-z_{i,j}}} \cdot \frac{e^{-z_{i,j}}}{1+e^{-z_{i,j}}} =$$

$$= \frac{1}{1+e^{-z_{i,j}}} \cdot \frac{1+e^{-z_{i,j}}-1}{1+e^{-z_{i,j}}} == \frac{1}{1+e^{-z_{i,j}}} \cdot (\frac{1+e^{-z_{i,j}}}{1+e^{-z_{i,j}}} - \frac{1}{1+e^{-z_{i,j}}}) =$$

$$= \frac{1}{1+e^{-z_{i,j}}} \cdot (1 - \frac{1}{1+e^{-z_{i,j}}}) = \sigma_{i,j} \cdot (1-\sigma_{i,j})$$

Sigmoid Function Code

As with other activation functions, we'll write a forward pass method and a backward pass method. For the forward pass, we'll take the inputs and apply the sigmoid function. For the backward pass, we'll leverage the sigmoid function's derivative, which, as we figured out during derivation of the sigmoid function's derivative, equals the sigmoid output from the forward pass multiplied by the difference of 1 and this output.

```
# Sigmoid activation
class Activation_Sigmoid:

    # Forward pass
    def forward(self, inputs):
        # Save input and calculate/save output
        # of the sigmoid function
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))

    # Backward pass
    def backward(self, dvalues):
        # Derivative - calculates from output of the sigmoid function
        self.dinputs = dvalues * (1 - self.output) * self.output
```

Now that we have the new activation function, we need to code our new calculation for the binary cross-entropy loss.

Binary Cross-Entropy Loss

To calculate binary cross-entropy loss, we will continue to use the negative log concept from categorical cross-entropy loss. Rather than only calculating this on the target class, we will sum the log-likelihoods of the correct and incorrect classes for each neuron separately. Because class values are either *0* or *1*, we can simplify the incorrect class to be *1-correct class* as this inverts the value. We can then calculate the negative log-likelihood of the correct and incorrect classes, adding them together. We are presenting two forms of the equation — the first is following the description just given, then the optimized version differentiating only in the minus signs being moved over and redundant parentheses removed:

$$L_{i,j} = (y_{i,j})(-log(\hat{y}_{i,j})) + (1 - y_{i,j})(-log(1 - \hat{y}_{i,j})) =$$

$$= -y_{i,j} \cdot log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})$$

In code, this will start as (but will be modified shortly, so do not commit this to your codebase yet):

```
sample_losses = -(y_true * np.log(y_pred) +
                (1 - y_true) * np.log(1 - y_pred))
```

Since a model can contain multiple binary outputs, and each of them, unlike in the cross-entropy loss, outputs its own prediction, loss calculated on a single output is going to be a vector of losses containing one value for each output. What we need is a sample loss and, to achieve that, we need to calculate a mean of all of these losses from a single sample:

$$L_i = \frac{1}{J} \sum_j L_{i,j}$$

Where index *i* means the current sample, the index *j* means the current output in this sample, and the *J* means the number of outputs. Since we are operating on a set of samples (the output is an array containing the set of loss vectors), we can use NumPy to perform this operation on a single call:

```
sample_losses = np.mean(sample_losses, axis=-1)
```

The last parameter, `axis=-1`, informs NumPy to calculate the mean value along the last dimension. To make it easier to visualize, let's use a simple example. Assume that this is an output of the model containing 3 neurons in the output layer, and it's passed through the binary cross-entropy loss function:

```
outputs = np.array([[1,  2,  3],
                    [2,  4,  6],
                    [0,  5,  10],
                    [11, 12, 13],
                    [5,  10, 15]])
```

These numbers are completely made up for this example. We want to take each of the output vectors, `[1,  2,  3]` for example, and calculate a mean value from the numbers they hold, putting the result on the output vector. We then want to repeat this for the other vectors and return the resulting vector, which will be a one-dimensional array. Using NumPy:

```
np.mean(outputs, axis=-1)
```

```
>>>
array([ 2.,  4.,  5., 12., 10.])
```

If we calculate the mean value of the first output, it's indeed *2*, the mean value of the second output is indeed *4*, and so on.

We are also going to inherit from the **Loss** class, so the overall loss calculation will be handled by the `calculate` method that we already created for the categorical cross-entropy loss class.

Binary Cross-Entropy Loss Derivative

To calculate the gradient from here, we already know that the derivative for the natural logarithm is *1/x* and that the derivative of *1-x* is *-1*. In simplified form, this gives us *-(y_true / y + (1 - y_true) / (1 - y)) · (-1)*.

To calculate the partial derivative of this loss function with respect to the predicted input, we'll use the latter version of the loss equation. It doesn't really matter in this case which one we use:

$$\frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} = \frac{\partial}{\partial \hat{y}_{i,j}}[-y_{i,j} \cdot log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

The expression that we have to calculate the partial derivative of consists of two sub-expressions, which are components of the sum operation. We can write that as the sum of derivatives:

$$= \frac{\partial}{\partial \hat{y}_{i,j}}[-y_{i,j} \cdot log(\hat{y}_{i,j})] + \frac{\partial}{\partial \hat{y}_{i,j}}[-(1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

Both components contain $y_{i,j}$ (the target value) inside of their derivatives, which are the constants that we are deriving with respect to $\hat{y}_{i,j}$ (the predicted value, which is a different variable), so we can move them outside of the derivative along with the other constants and minus sign:

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

Now, like in the **Categorical Cross-Entropy** loss' derivative, we have to calculate the derivative of the logarithmic function, which equals the reciprocal of its parameter multiplied (following the chain rule) by the derivative of this parameter. Let's apply that to both of the partial derivatives:

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [1 - \hat{y}_{i,j}] =$$

Now the first partial derivative equals *1*, since the value we derive, and the value we derive with respect to, are the same values. The second partial derivative can be written as the difference of

the derivatives:

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

From the two new derivatives, the first one equals *0* as the derivative of the constant always equals *0*, then the second derivative equals *1* as the value we derive, and the value we derive with respect to, are the same values:

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot (0 - 1) =$$

We can finally clean up to get the resulting equation:

$$= -\frac{y_{i,j}}{\hat{y}_{i,j}} + \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} = -(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}})$$

The partial derivative of the **Binary Cross-Entropy** loss solves to a pretty simple equation that will be easy to implement in code.

Full solution:

$$\frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} = \frac{\partial}{\partial \hat{y}_{i,j}} [-y_{i,j} \cdot log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

$$= \frac{\partial}{\partial \hat{y}_{i,j}} [-y_{i,j} \cdot log(\hat{y}_{i,j})] + \frac{\partial}{\partial \hat{y}_{i,j}} [-(1 - y_{i,j}) \cdot log(1 - \hat{y}_{i,j})] =$$

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

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [1 - \hat{y}_{i,j}] =$$

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

$$= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot (0 - 1) =$$

$$= -\frac{y_{i,j}}{\hat{y}_{i,j}} + \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} = -(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}})$$

This partial derivative is a derivative of the single output's loss and, with any type of output, we always need to calculate it with respect to a sample loss, not an atomic output loss, since we have to calculate the mean value of all output losses in a sample to form a *sample loss* during the forward pass:

$$L_i = \frac{1}{J}\sum_j L_{i,j}$$

For backpropagation, we have to calculate the partial derivative of the *sample loss* with respect to each input:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial L_i}{\partial L_{i,j}} \cdot \frac{\partial L_{ij}}{\partial \hat{y}_{i,j}}$$

We have just calculated the second derivative, the partial derivative of the single output loss with respect to the related prediction. We have to calculate the partial derivative of the sample loss with respect to the single output loss:

$$\frac{\partial L_i}{\partial L_{i,j}} = \frac{\partial}{\partial L_{i,j}}[\frac{1}{J}\sum_j L_{i,j}] =$$

1 divided by *J* (the number of outputs), is a constant and can be moved outside of the derivative. Since we are calculating the derivative with respect to a given output, *j*, the sum of one element equals this element:

$$= \frac{1}{J} \cdot \frac{\partial}{\partial L_{i,j}} L_{i,j} =$$

The remaining derivative equals *1* as the derivative of a variable with respect to the same variable equals *1*.

$$= \frac{1}{J} \cdot 1 = \frac{1}{J}$$

Full solution:

$$\frac{\partial L_i}{\partial L_{i,j}} = \frac{\partial}{\partial L_{i,j}}[\frac{1}{J}\sum_j L_{i,j}] = \frac{1}{J} \cdot \frac{\partial}{\partial L_{i,j}} L_{i,j} = \frac{1}{J} \cdot 1 = \frac{1}{J}$$

Now we can update the equation of the partial derivative of a sample loss with respect to a single output loss by applying the chain rule:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial L_i}{\partial L_{i,j}} \cdot \frac{\partial L_{ij}}{\partial \hat{y}_{i,j}} = \frac{1}{J} \cdot (-(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}})) = -\frac{1}{J} \cdot (\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}}) =$$

We have to perform this normalization since each output returns its own derivative, and without normalization, each additional input will raise gradients and require changing other hyperparameters, including the learning rate.

Binary Cross-Entropy Code

In our code, this will be:

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

# Calculate gradient
self.dinputs = -(y_true / clipped_dvalues -
                 (1 - y_true) / (1 - clipped_dvalues)) / outputs
```

Similar to what we did in the categorical cross-entropy loss, we need to normalize gradient so it'll become invariant to the number of samples we calculate it for:

```
# Normalize gradient
self.dinputs = self.dinputs / samples
```

Finally, we need to address the numerical instability of the logarithmic function. The sigmoid activation can return a value in the range of *0* to *1* (inclusive), but the *log(0)* presents a slight issue due to how it's calculated and will return *negative infinity*. This alone isn't necessarily a big deal, but any list with *-inf* in it will have a mean of *-inf*, which is the same for any list with positive infinity averaging to infinity.

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


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


print(np.mean([5, 2, 4, np.log(0)]))


>>>
-inf
```

This is a similar issue to the one we discussed earlier regarding categorical cross-entropy loss in chapter 5. To prevent this issue, we'll add clipping on the batch of values:

```
# 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)
```

We now will use these clipped values for the forward pass, rather than the originals:

```
# Calculate sample-wise loss
sample_losses = -(y_true * np.log(y_pred_clipped) +
                  (1 - y_true) * np.log(1 - y_pred_clipped))
```

As we perform the division operation during the derivative calculation, the gradient passed in may contain both values, *0* and *1*. Either of these values will cause a problem in either the `y_true / dvalues` or `(1 - y_true) / (1 - dvalues)` parts respectively (*0* in the first and *1-1=0* in the second case will also cause division by *0*), so we need to clip this gradient as well:

```
# Clip data to prevent division by 0
# Clip both sides to not drag mean towards any value
clipped_dvalues = np.clip(dvalues, 1e-7, 1 - 1e-7)
```

Now, similar to the forward pass, we can use these clipped values:

```
# Calculate gradient
self.dinputs = -(y_true / clipped_dvalues -
                 (1 - y_true) / (1 - clipped_dvalues)) / outputs
```