The full code now for the binary cross-entropy:

```
# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

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

        # 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)

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

        # Return losses
        return sample_losses

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

        # 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])

        # 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)

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

Now that we have this new activation function and loss calculation, we'll make edits to our existing softmax classifier to implement the binary logistic regression model.

Implementing Binary Logistic Regression and Binary Cross-Entropy Loss

With these new classes, our code changes will be in the execution of actual code (instead of modifying the classes). The first change is to make the `spiral_data` object output 2 classes, rather than 3, like so:

```
# Create dataset
X, y = spiral_data(samples=100, classes=2)
```

Next, we'll reshape our labels, as they're not sparse anymore. They're binary, *0* or *1*:

```
# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)
```

Consider the difference here. Initially, the *y* output from the `spiral_data` function would look something like:

```
X, y = spiral_data(samples=100, classes=2)
print(y[:5])


>>>
[0 0 0 0 0]
```

Then we reshape it here for binary logistic regression:

```
y = y.reshape(-1, 1)
print(y[:5])


>>>
[[0]
 [0]
 [0]
 [0]
 [0]]
```

Why have we done this? Initially, with the softmax classifier, the values from `spiral_data` could be used directly as the target labels, as they contain the correct class labels in numerical form — an index of the correct class, where each neuron in the output layer is a separate class, for example `[0, 1, 1, 0, 1]`. In this case, however, we're trying to represent some binary outputs, where each neuron represents 2 possible classes on its own. For the example we're currently working on, we have a single output neuron so the output from our neural network should be a tensor (array), containing one value, of a target value of either *0* or *1*, for example, `[[0], [1], [1], [0], [1]]`. The `.reshape(-1, 1)` means to reshape the data into 2 dimensions, where the second dimension contains a single element, and the first dimension contains how many elements the result will contain (`-1`) following other conditions. You are allowed to use `-1` only once in a shape with NumPy, letting you have that dimension be variable. Thanks to this ability, we do not always need the same number of samples every time, and NumPy can handle the calculation for us. In the case above, they're all *0* because the `spiral_data` function makes the dataset one class at a time, starting with *0*. We will also need to reshape the y-testing data in the same way.

Let's create our layers and use the appropriate activation functions:

```
# Create dataset
X, y = spiral_data(100, 2)

# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)

# 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 1 output value
dense2 = Layer_Dense(64, 1)

# Create Sigmoid activation:
activation2 = Activation_Sigmoid()
```

Notice that we're still using the rectified linear activation for the hidden layer. The hidden layer activation functions don't necessarily need to change, even though we're effectively building a different type of classifier. You should also notice that because this is now a binary classifier, the `dense2` object has only 1 output. Its output represents exactly 2 classes (*0* or *1*) being mapped to one neuron. We can now select a loss function and optimizer. For the **Adam** optimizer settings,

we are going to use the default learning rate and the decaying of *5e-7*:

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

# Create optimizer
optimizer = Optimizer_Adam(decay=5e-7)
```

While we require a different calculation for loss (since we use a different activation function for the output layer), we can still use the same optimizer as in the softmax classifier. Another small change is how we measure predictions. With probability distributions, we use *argmax* and determine which index is associated with the largest value, which becomes the classification result. With a binary classifier, we are determining if the output is closer to *0* or to *1*. To do this, we simplify the output to:

```
predictions = (activation2.output > 0.5) * 1
```

This results in *True/False* evaluations to the statement that the output is above *05* for all values. *True* and *False*, when treated as numbers, are *1* and *0*, respectively. For example, if we execute: `int(True)`, the result will be `1` and `int(False)` will be `0`. If we want to convert a list of *True/False* boolean values to numbers, we can't just wrap the list in `int()`. However, we *can* perform math operations directly on an array of boolean values and return the arithmetic answer. For example, we can run:

```
import numpy as np
a = np.array([True, False, True])
print(a)

>>>
[ True False  True]
```

And then:

```
b = a*1
print(b)

>>>
[1 0 1]
```

Thus, to evaluate predictive accuracy, we can do the following in our code:

```
predictions = (activation2.output > 0.5) * 1
accuracy = np.mean(predictions==y_test)
```

The `* 1` multiplication turns an array of boolean True/False values into numerical 1/0 values, respectively. We will need to implement this accuracy calculation for validation data too.

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)


# 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)


# 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


# 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


# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

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

        # 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)

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

        # Return losses
        return sample_losses

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

        # 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])

        # 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)

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

```
# Create dataset
X, y = spiral_data(samples=100, classes=2)

# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)

# 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 1 output value
dense2 = Layer_Dense(64, 1)

# Create Sigmoid activation:
activation2 = Activation_Sigmoid()

# Create loss function
loss_function = Loss_BinaryCrossentropy()

# Create optimizer
optimizer = Optimizer_Adam(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 activation function
    # takes the output of second dense layer here
    activation2.forward(dense2.output)

    # Calculate the data loss
    data_loss = loss_function.calculate(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

        # Calculate accuracy from output of activation2 and targets
        # Part in the brackets returns a binary mask - array consisting
        # of True/False values, multiplying it by 1 changes it into array
        # of 1s and 0s
        predictions = (activation2.output > 0.5) * 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_function.backward(activation2.output, y)
        activation2.backward(loss_function.dinputs)
        dense2.backward(activation2.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=2)

    # Reshape labels to be a list of lists
    # Inner list contains one output (either 0 or 1)
    # per each output neuron, 1 in this case
    y_test = y_test.reshape(-1, 1)
```

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

# Calculate the data loss
loss = loss_function.calculate(activation2.output, y_test)

# Calculate accuracy from output of activation2 and targets
# Part in the brackets returns a binary mask - array consisting of
# True/False values, multiplying it by 1 changes it into array
# of 1s and 0s
predictions = (activation2.output > 0.5) * 1
accuracy = np.mean(predictions==y_test)

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


>>>
epoch: 0, acc: 0.500, loss: 0.693 (data_loss: 0.693, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.630, loss: 0.674 (data_loss: 0.673, reg_loss: 0.001), lr:
0.0009999505024501287
epoch: 200, acc: 0.625, loss: 0.669 (data_loss: 0.668, reg_loss: 0.001), lr:
0.0009999005098992651
epoch: 300, acc: 0.650, loss: 0.664 (data_loss: 0.663, reg_loss: 0.002), lr:
0.000999850522346909
epoch: 400, acc: 0.650, loss: 0.659 (data_loss: 0.657, reg_loss: 0.002), lr:
0.0009998005397923115
epoch: 500, acc: 0.675, loss: 0.647 (data_loss: 0.644, reg_loss: 0.004), lr:
0.0009997505622347225
epoch: 600, acc: 0.720, loss: 0.632 (data_loss: 0.625, reg_loss: 0.006), lr:
0.0009997005896733929
...
epoch: 1500, acc: 0.805, loss: 0.503 (data_loss: 0.464, reg_loss: 0.039),
lr: 0.0009992510613295335
...
epoch: 2500, acc: 0.855, loss: 0.430 (data_loss: 0.379, reg_loss: 0.052),
lr: 0.0009987520593019025
```

```
...
epoch: 4500, acc: 0.910, loss: 0.346 (data_loss: 0.285, reg_loss: 0.061),
lr: 0.0009977555488927658
epoch: 4600, acc: 0.905, loss: 0.340 (data_loss: 0.278, reg_loss: 0.062),
lr: 0.000997705775569079
epoch: 4700, acc: 0.910, loss: 0.330 (data_loss: 0.268, reg_loss: 0.062),
lr: 0.0009976560072110577
epoch: 4800, acc: 0.920, loss: 0.326 (data_loss: 0.263, reg_loss: 0.063),
lr: 0.0009976062438179587
...
epoch: 6100, acc: 0.940, loss: 0.291 (data_loss: 0.223, reg_loss: 0.069),
lr: 0.0009969597711777935
...
epoch: 6600, acc: 0.950, loss: 0.279 (data_loss: 0.211, reg_loss: 0.068),
lr: 0.000996711350897713
epoch: 6700, acc: 0.955, loss: 0.272 (data_loss: 0.203, reg_loss: 0.069),
lr: 0.0009966616816971556
epoch: 6800, acc: 0.955, loss: 0.269 (data_loss: 0.200, reg_loss: 0.069),
lr: 0.00099661201744669
epoch: 6900, acc: 0.960, loss: 0.266 (data_loss: 0.197, reg_loss: 0.069),
lr: 0.0009965623581455767
...
epoch: 9800, acc: 0.965, loss: 0.222 (data_loss: 0.158, reg_loss: 0.063),
lr: 0.0009951243880606966
epoch: 9900, acc: 0.965, loss: 0.221 (data_loss: 0.157, reg_loss: 0.063),
lr: 0.0009950748768967994
epoch: 10000, acc: 0.965, loss: 0.219 (data_loss: 0.156, reg_loss: 0.063),
lr: 0.0009950253706593885
validation, acc: 0.945, loss: 0.207
```

The model performed quite well here! You should have some intuition about tweaking the output layer to better fit the problem you're attempting to solve while keeping your hidden layers mostly the same. In the next chapter, we're going to work on regression, where our intended output is not a classification at all, but rather to predict a scalar value, like the price of a house.

Supplementary Material: https://nnfs.io/ch16
Chapter code, further resources, and errata for this chapter.

Chapter 17

Regression

Up until this point, we've been working with classification models, where we try to determine *what* something is. Now we're curious about determining a *specific* value based on an input. For example, you might want to use a neural network to predict what the temperature will be tomorrow or what the price of a car should be. For a task like this, we need something with a much more granular output. This also means that we require a new way to measure loss, as well as a new output layer activation function! It also means our data are different. We need training data that have target scalar values, not classes.

```
import matplotlib.pyplot as plt
import nnfs
from nnfs.datasets import sine_data

nnfs.init()


X, y = sine_data()
```

```
plt.plot(X, y)
plt.show()
```

The data above will produce a graph like:

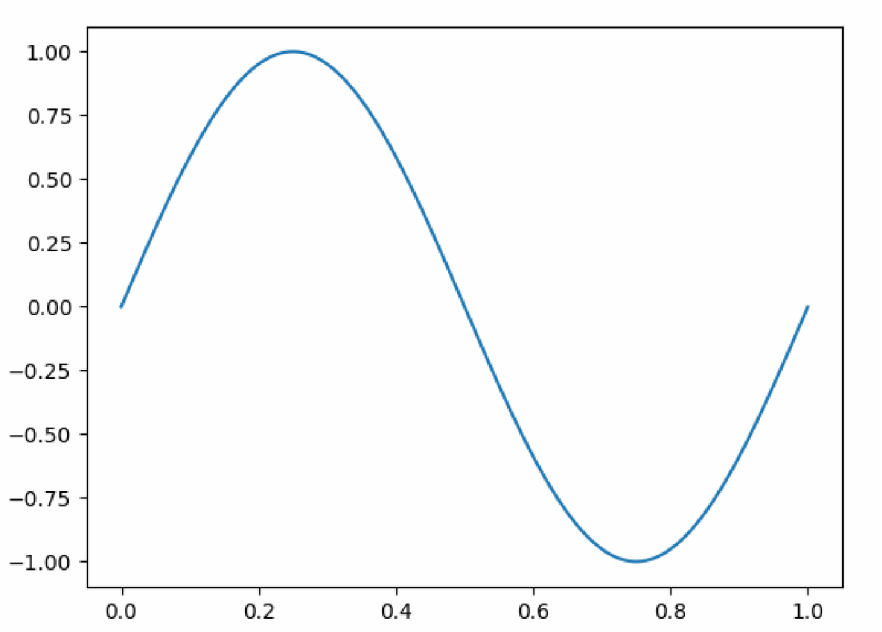

Fig 17.01: The sine data graph.

Linear Activation

Since we're no longer using classification labels and want to predict a scalar value, we're going to use a linear activation function for the output layer. This linear function does not modify its input and passes it to the output: $y=x$. For the backward pass, we already know the derivative of $f(x)=x$ is 1; thus, the full class for our new linear activation function is:

```
# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()
```

This might raise a question — why do we even write some code that does nothing? We just pass inputs to outputs for the forward pass and do the same with gradients during the backward pass since, to apply the chain rule, we multiply incoming gradients by the derivative, which is 1. We do this only for completeness and clarity to see the activation function of the output layer in the model definition code. From a computational time point of view, this adds almost nothing to the processing time, at least not enough to noticeably impact training times.

Now we just need to figure out loss!

Mean Squared Error Loss

Since we aren't working with classification labels anymore, we cannot calculate cross-entropy. Instead, we need some new methods. The two main methods for calculating error in regression are **mean squared error** (MSE) and **mean absolute error** (MAE).

With **mean squared error**, you square the difference between the predicted and true values of single outputs (as the model can have multiple regression outputs) and average those squared values.

$$L_i = \frac{1}{J}\sum_{j}(y_{i,j} - \hat{y}_{i,j})^2$$

Where *y* means the target value, *y-hat* means predicted value, index *i* means the current sample, index *j* means the current output in this sample, and the *J* means the number of outputs.

The idea here is to penalize more harshly the further away we get from the intended target.

Mean Squared Error Loss Derivative

The partial derivative of squared error with respect to the predicted value is:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_{j} (y_{i,j} - \hat{y}_{i,j})^2] =$$

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 the given output, *j*, the sum of one element equals this element:

$$= \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} (y_{i,j} - \hat{y}_{i,j})^2 =$$

To calculate the partial derivative of an expression to the power of some value, we need to multiply this exponent by the expression, subtract 1 from the exponent, and multiply this by the partial derivative of the inner function:

$$= \frac{1}{J} \cdot 2 \cdot (y_{i,j} - \hat{y}_{i,j})^{2-1} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [y_{i,j} - \hat{y}_{i,j}] =$$

The partial derivative of the subtraction equals the subtraction of the partial derivatives:

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})^1 \cdot (\frac{\partial}{\partial \hat{y}_{i,j}} y_{i,j} - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}) =$$

The partial derivative of the ground truth value with respect to the predicted value equals *0* since we treat other variables as constants. The partial derivative of the predicted value with respect to itself equals *1*, which results in *0-1=-1*. This is multiplied by the rest of the equation and forms the solution:

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (0 - 1) = \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (-1) = -\frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})$$

Full solution:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_j (y_{i,j} - \hat{y}_{i,j})^2] = \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} (y_{i,j} - \hat{y}_{i,j})^2 =$$

$$= \frac{1}{J} \cdot 2 \cdot (y_{i,j} - \hat{y}_{i,j})^{2-1} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [y_{i,j} - \hat{y}_{i,j}] =$$

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})^1 \cdot (\frac{\partial}{\partial \hat{y}_{i,j}} y_{i,j} - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}) = \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (0 - 1) =$$

$$= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (-1) = -\frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})$$

The partial derivative equals *-2*, multiplied by the subtraction of the true and predicted values, and then divided by the number of outputs to normalize the gradients, making their magnitude invariant to the number of outputs.

Mean Squared Error (MSE) Loss Code

The code for MSE includes an implementation of the equation to calculate the sample loss from multiple outputs. `axis=-1` with the mean calculation was explained in the previous chapter in detail and, in short words, it informs NumPy to calculate mean across outputs, for each sample separately. For the backward pass, we implemented the derivative equation, which results in *-2* multiplied by the difference of true and predicted values, and normalized by the number of outputs. Similarly to the other loss function implementations, we also normalize gradients by the number of samples to make them invariant to the batch size, or the number of samples in general:

```
# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

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

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)
```

```
        # Return losses
        return sample_losses

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

        # 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])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Mean Absolute Error Loss

With **mean absolute error**, you take the absolute difference between the predicted and true values in a single output and average those absolute values.

$$L_i = \frac{1}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}|$$

Where *y* means the target value, *y-hat* means predicted value, index *i* means the current sample, index *j* means the current output in this sample, and the *J* means the number of outputs.

This function, used as a loss, penalizes the error linearly. It produces sparser results and is robust to outliers, which can be both advantageous and disadvantageous. In reality, L1 (MAE) loss is used less frequently than L2 (MSE) loss.

Mean Absolute Error Loss Derivative

The partial derivative for absolute error with respect to the predicted values is:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}|]$$

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 the given output, *j*, the sum of one element equals this element:

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

We already calculated the partial derivative of an absolute value for the L1 regularization, which is similar to the L1 loss. The derivative of an absolute value equals *1* if this value is greater than *0*, or *-1* if it's less than *0*. The derivative does not exist for a value of *0*:

$$= \frac{1}{J} \cdot \begin{cases} 1 & y_{i,j} - \hat{y}_{i,j} > 0 \\ -1 & y_{i,j} - \hat{y}_{i,j} < 0 \end{cases}$$

Full solution:

$$\frac{\partial}{\partial \hat{y}_{i,j}} L_i = \frac{\partial}{\partial \hat{y}_{i,j}} [\frac{1}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}|] = \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} |y_{i,j} - \hat{y}_{i,j}| = \frac{1}{J} \cdot \begin{cases} 1 & y_{i,j} - \hat{y}_{i,j} > 0 \\ -1 & y_{i,j} - \hat{y}_{i,j} < 0 \end{cases}$$

Mean Absolute Error Loss Code

The code for mean absolute error is very similar to the mean squared error. The forward pass includes NumPy's `np.abs()` to calculate absolute values before calculating the mean. For the backward pass, we'll use `np.sign(),` which returns 1 or -1 given the sign of the input and *0* if the parameter equals *0*, then normalize gradients by the number of samples to make them invariant to the batch size, or number of samples in general:

```
# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses

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

        # 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 = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples
```

Accuracy in Regression

Now that we've got data, an activation function, and a loss calculation for regression, we'd like to measure performance.

With cross-entropy, we were able to count the number of matches (situations where the prediction equals the ground truth target), and then divide it by the number of samples to measure the model's accuracy. With a regression model, we have two problems: the first problem is that each output neuron in the model (there might be many) is a separate output — like in a binary regression model and unlike in a classifier, where all outputs contribute toward a common prediction. The second problem is that the prediction is a float value, and we can't simply check if the output value equals the ground truth one, as it most likely won't — if it differs even slightly, the accuracy will be a 0. For example, if your model predicts home prices and one of the samples has the target price of $192,500, and the predicted value is $192,495, then a pure "is it equal" assessment would return False. We'd likely consider the predicted price to be correct or "close enough" in this scenario, given the magnitude of the numbers in consideration. There's no perfect way to show accuracy with regression. Still, it is preferable to have some accuracy metric.

For example, Keras, a popular deep learning framework, shows both accuracy and loss for regression models, and we'll also make our own accuracy metric.

First, we need some "limit" value, which we'll call "precision." To calculate this precision, we'll calculate the standard deviation from the ground truth target values and then divide it by *250*. This value can certainly vary depending on your goals. The larger the number you divide by, the more "strict" the accuracy metric will be. *250* is our value of choice. Code to represent this:

```
accuracy_precision = np.std(y) / 250
```

Then we could use this precision value as a sort of "cushion allowance" for regression outputs when comparing targets and predicted values for accuracy. We perform the comparison by applying the absolute value on the difference between the ground truth values and the predictions. Then we check if the difference is smaller than our previously calculated precision:

```
predictions = activation2.output
accuracy = np.mean(np.absolute(predictions - y) <
           accuracy_precision)
```

Regression Model Training

With this new activation function, loss, and way of calculating accuracy, we now create our model:

```
# Create dataset
X, y = sine_data()

# Create Dense layer with 1 input feature and 64 output values
dense1 = Layer_Dense(1, 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 1 output value
dense2 = Layer_Dense(64, 1)
# Create Linear activation:
activation2 = Activation_Linear()

# Create loss function
loss_function = Loss_MeanSquaredError()

# Create optimizer
optimizer = Optimizer_Adam()

# Accuracy precision for accuracy calculation
# There are no really accuracy factor for regression problem,
# but we can simulate/approximate it. We'll calculate it by checking
# how many values have a difference to their ground truth equivalent
# less than given precision
# We'll calculate this precision as a fraction of standard deviation
# of al the ground truth values
accuracy_precision = np.std(y) / 250

# 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 activation function
# takes the output of second dense layer here
activation2.forward(dense2.output)

# Calculate the data loss
data_loss = loss_function.calculate(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

# Calculate accuracy from output of activation2 and targets
# To calculate it we're taking absolute difference between
# predictions and ground truth values and compare if differences
# are lower than given precision value
predictions = activation2.output
accuracy = np.mean(np.absolute(predictions - y) <
                   accuracy_precision)

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_function.backward(activation2.output, y)
activation2.backward(loss_function.dinputs)
dense2.backward(activation2.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)
```

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

```
>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.003, loss: 0.346 (data_loss: 0.346, reg_loss: 0.000), lr:
0.001
...
epoch: 9900, acc: 0.003, loss: 0.145 (data_loss: 0.145, reg_loss: 0.000),
lr: 0.001
epoch: 10000, acc: 0.004, loss: 0.145 (data_loss: 0.145, reg_loss: 0.000),
lr: 0.001
```

Training didn't work out here very well! Let's add an ability to draw the testing data and let's also do a forward pass on the testing data, drawing output data on the same plot as well:

```
import matplotlib.pyplot as plt

X_test, y_test = sine_data()

dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)

plt.plot(X_test, y_test)
plt.plot(X_test, activation2.output)
plt.show()
```

First, we are importing matplotlib, then creating a new set of data. Next, we have 4 lines of the code that are the same as the forward pass from our code above. We could call it a prediction or, in the context of what we are going to do, validation. We'll cover both topics and explain what validation and prediction are in the future chapters. For now, it's enough to know that what we are doing here is predicting on the same feature-set that we've used to train the model in order to see what the model learned and returns for our data — seeing how close outputs are to the training ground-true values. We are then plotting the training data, which are obviously a sine, and prediction data, what we'd hope to form a sine as well. Let's run this code again and take a look at the generated image:

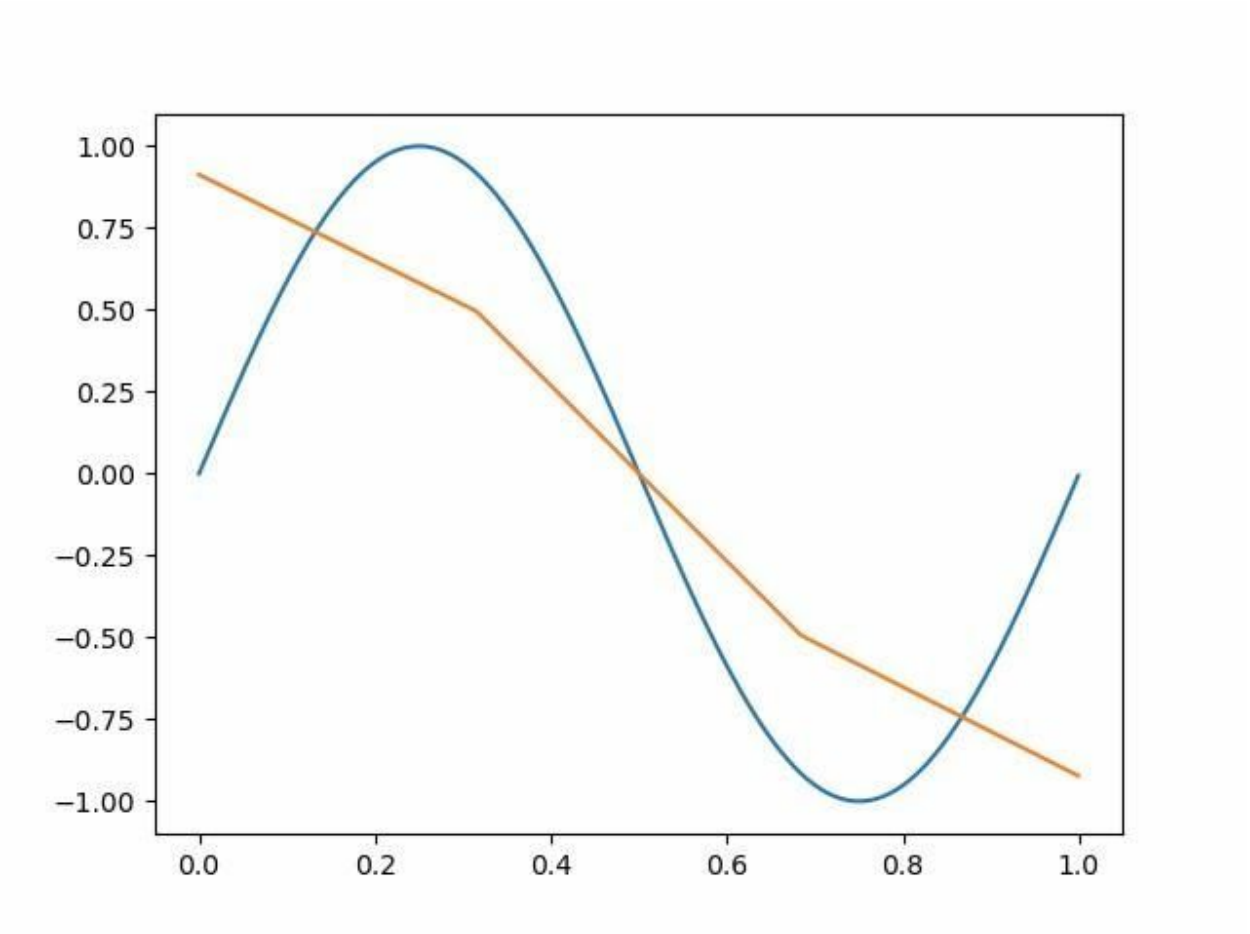

Fig 17.02: Model prediction - could not fit the sine data.

Animation of the training process:

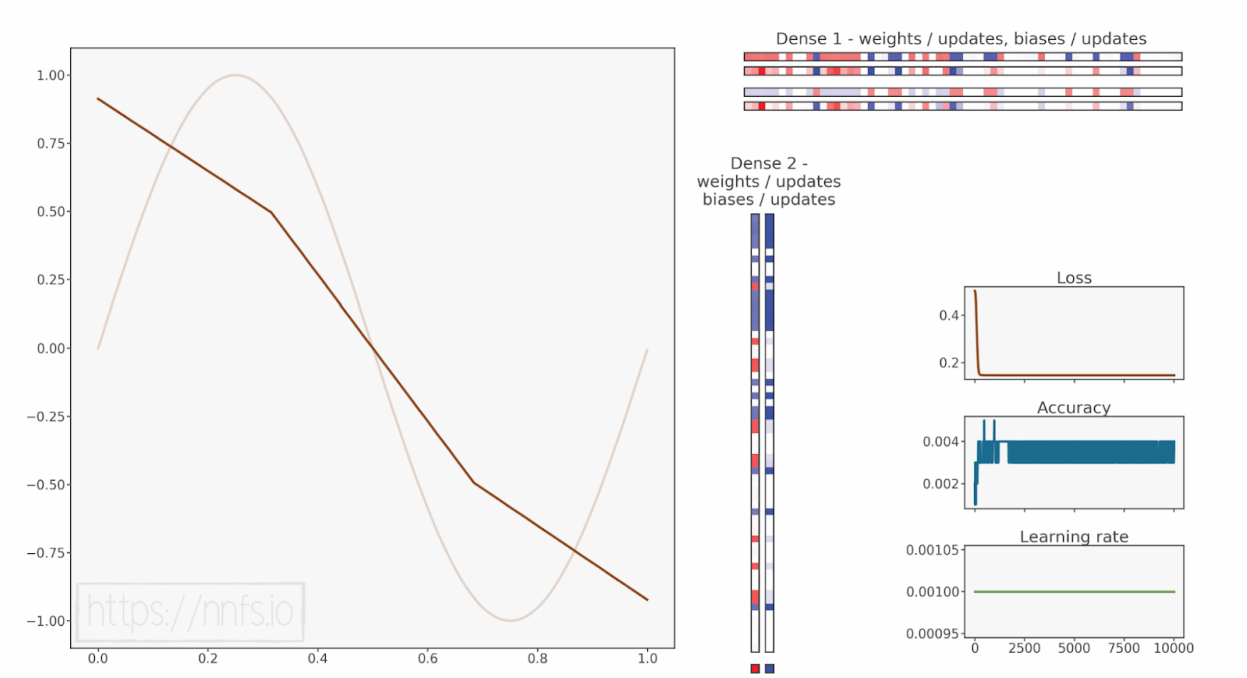

Fig 17.03: Model stoppped training immediately.

Anim 17.03: https://nnfs.io/ghi

Recall the rectified linear activation function and how its nonlinear behavior allowed us to map nonlinear functions, but we also needed two or more hidden layers. In this case, we have only 1 hidden layer followed by the output layer. As we should know by now, this is simply not enough!

If we add just one more layer:

```
# Create dataset
X, y = sine_data()

# Create Dense layer with 1 input feature and 64 output values
dense1 = Layer_Dense(1, 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 64 output values
dense2 = Layer_Dense(64, 64)

# Create ReLU activation (to be used with Dense layer):
activation2 = Activation_ReLU()

# Create third Dense layer with 64 input features (as we take output
# of previous layer here) and 1 output value
dense3 = Layer_Dense(64, 1)

# Create Linear activation:
activation3 = Activation_Linear()

# Create loss function
loss_function = Loss_MeanSquaredError()

# Create optimizer
optimizer = Optimizer_Adam()

# Accuracy precision for accuracy calculation
# There are no really accuracy factor for regression problem,
# but we can simulate/approximate it. We'll calculate it by checking
# how many values have a difference to their ground truth equivalent
# less than given precision
# We'll calculate this precision as a fraction of standard deviation
# of al the ground truth values
accuracy_precision = np.std(y) / 250

# 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 activation function
# takes the output of second dense layer here
activation2.forward(dense2.output)

# Perform a forward pass through third Dense layer
# takes outputs of activation function of second layer as inputs
dense3.forward(activation2.output)

# Perform a forward pass through activation function
# takes the output of third dense layer here
activation3.forward(dense3.output)

# Calculate the data loss
data_loss = loss_function.calculate(activation3.output, y)

# Calculate regularization penalty
regularization_loss = \
    loss_function.regularization_loss(dense1) + \
    loss_function.regularization_loss(dense2) + \
    loss_function.regularization_loss(dense3)

# Calculate overall loss
loss = data_loss + regularization_loss

# Calculate accuracy from output of activation2 and targets
# To calculate it we're taking absolute difference between
# predictions and ground truth values and compare if differences
# are lower than given precision value
predictions = activation3.output
accuracy = np.mean(np.absolute(predictions - y) <
                               accuracy_precision)

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_function.backward(activation3.output, y)
    activation3.backward(loss_function.dinputs)
    dense3.backward(activation3.dinputs)
    activation2.backward(dense3.dinputs)
    dense2.backward(activation2.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.update_params(dense3)
    optimizer.post_update_params()


import matplotlib.pyplot as plt

X_test, y_test = sine_data()

dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)
dense3.forward(activation2.output)
activation3.forward(dense3.output)

plt.plot(X_test, y_test)
plt.plot(X_test, activation3.output)
plt.show()


>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.003, loss: 0.187 (data_loss: 0.187, reg_loss: 0.000), lr:
0.001
...
epoch: 9900, acc: 0.617, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.001
epoch: 10000, acc: 0.620, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.001
```

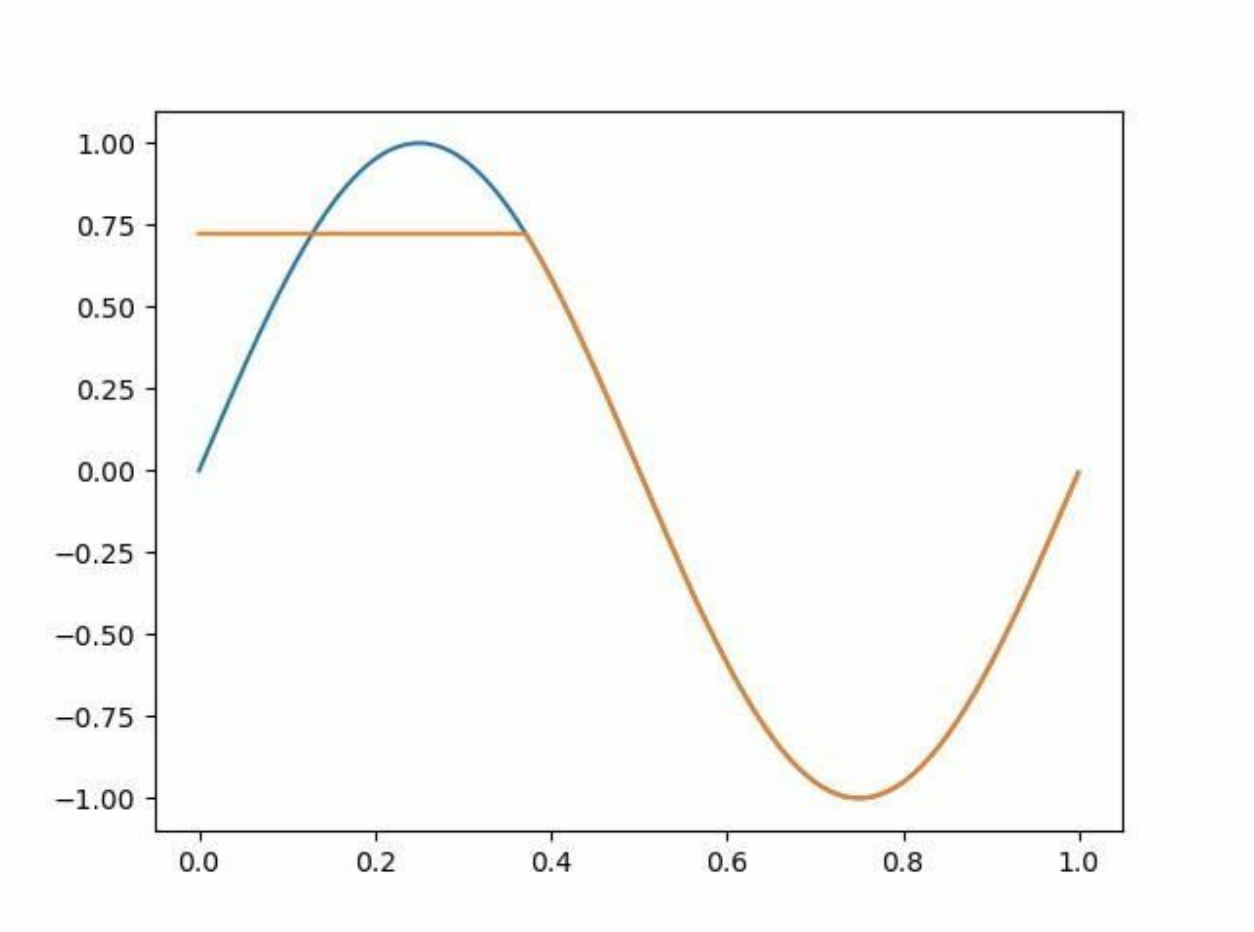

Fig 17.04: Model prediction - better fit to the data.

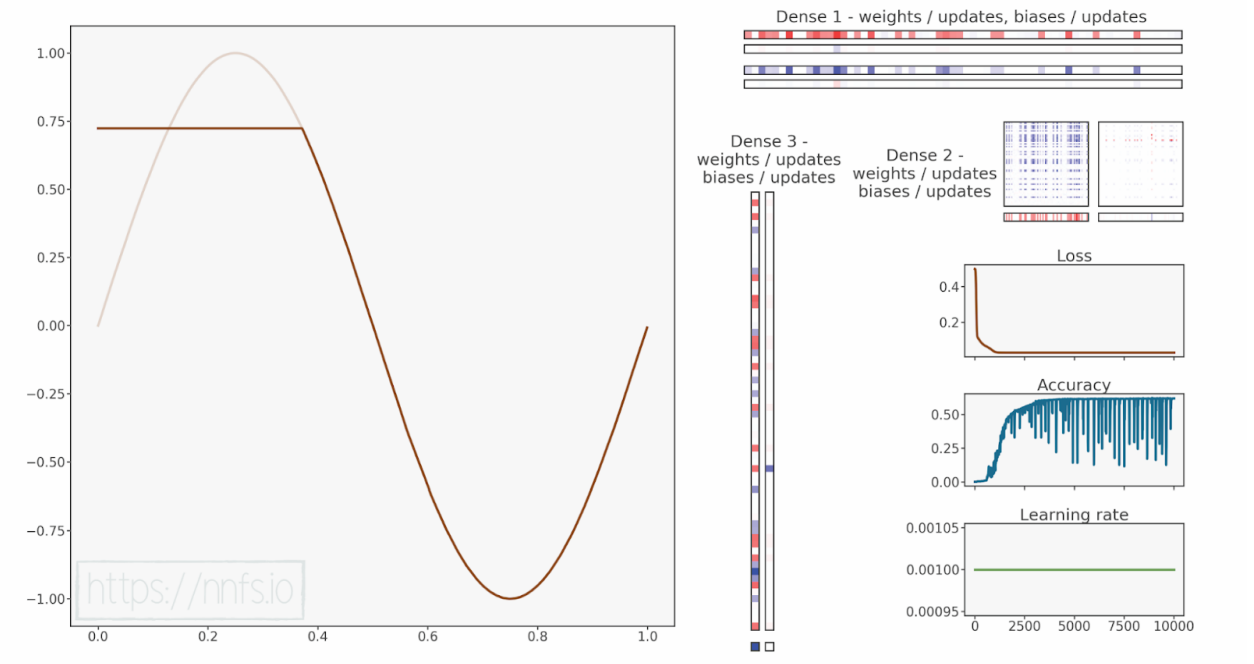

Fig 17.05: Model trained to better fit the sine data.

Anim 17.05: https://nnfs.io/hij

Our model's accuracy is not very good, and loss seems stuck at a pretty high level for this model. The image shows us why this is the case, the model has some trouble fitting our data, and it looks like it might be stuck in a local minimum. As we have already learned, to try to help the model with being stuck at a local minimum, we might use a higher learning rate and add a learning rate decay. In the previous model, we have used the default learning rate, which is *0.001*. Let's set it to *0.01* and add learning rate decaying:

```
optimizer = Optimizer_Adam(learning_rate=0.01, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.01
epoch: 100, acc: 0.027, loss: 0.061 (data_loss: 0.061, reg_loss: 0.000), lr:
0.009099181073703368
...
epoch: 9900, acc: 0.565, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.0009175153683824203
epoch: 10000, acc: 0.564, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.0009091735612328393
```

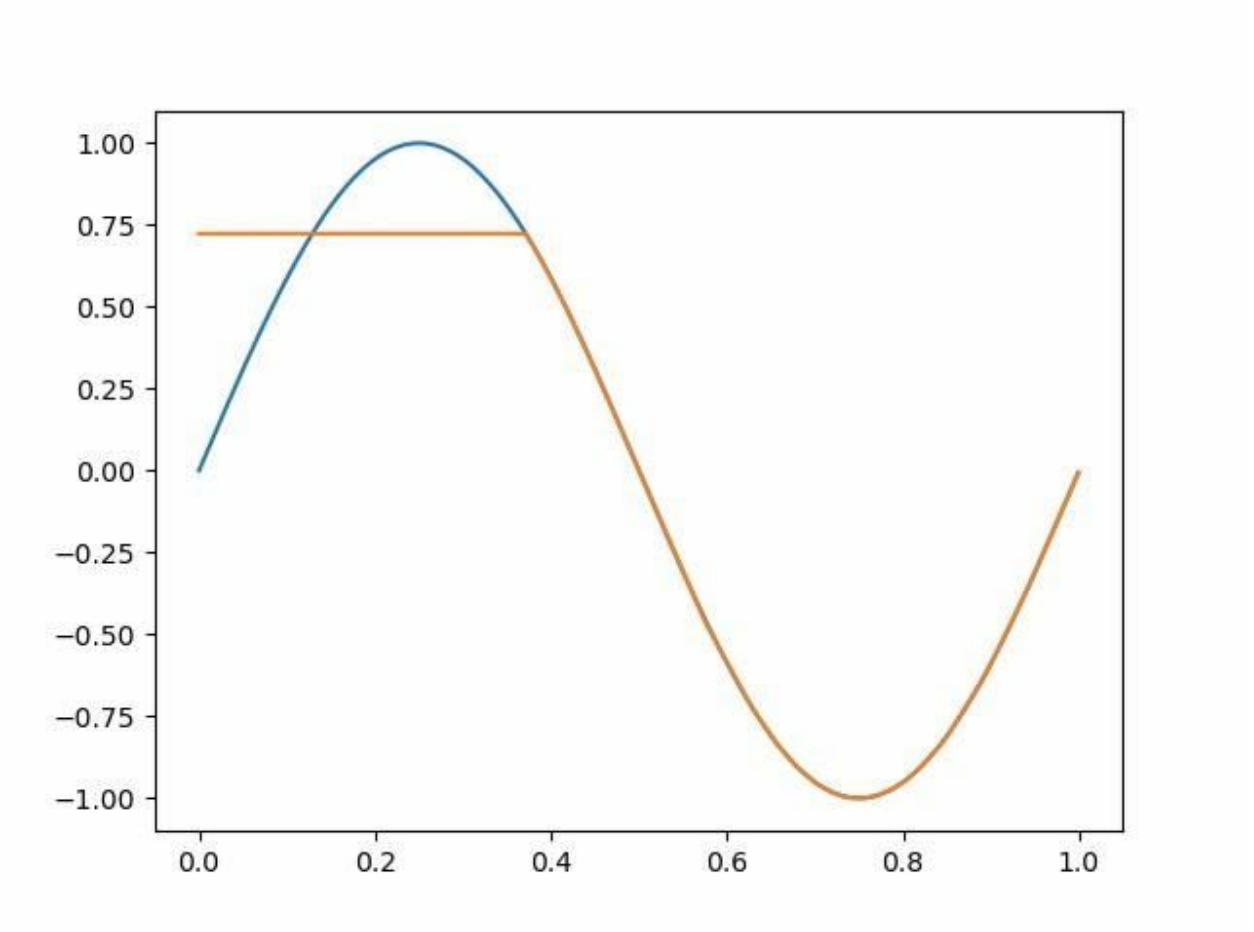

Fig 17.06: Model prediction - similar fit to the data.

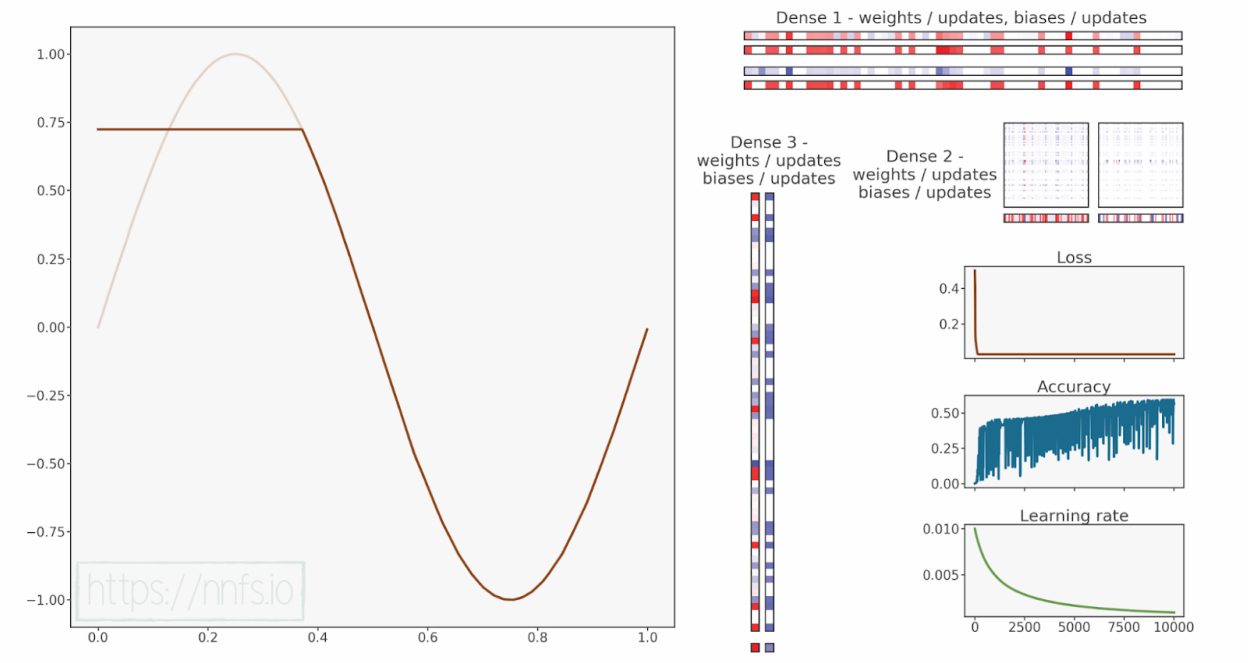

Fig 17.07: Model trained to fit the sine data, similar fit.

Anim 17.07: https://nnfs.io/ijk

Our model seems to still be stuck with even lower accuracy this time. Let's try to use an even bigger learning rate then:

```
optimizer = Optimizer_Adam(learning_rate=0.05, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.002, loss: 0.500 (data_loss: 0.500, reg_loss: 0.000), lr:
0.05
epoch: 100, acc: 0.087, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000), lr:
0.04549590536851684
...
epoch: 9900, acc: 0.275, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.004587576841912101
epoch: 10000, acc: 0.229, loss: 0.031 (data_loss: 0.031, reg_loss: 0.000),
lr: 0.0045458678061641965
```

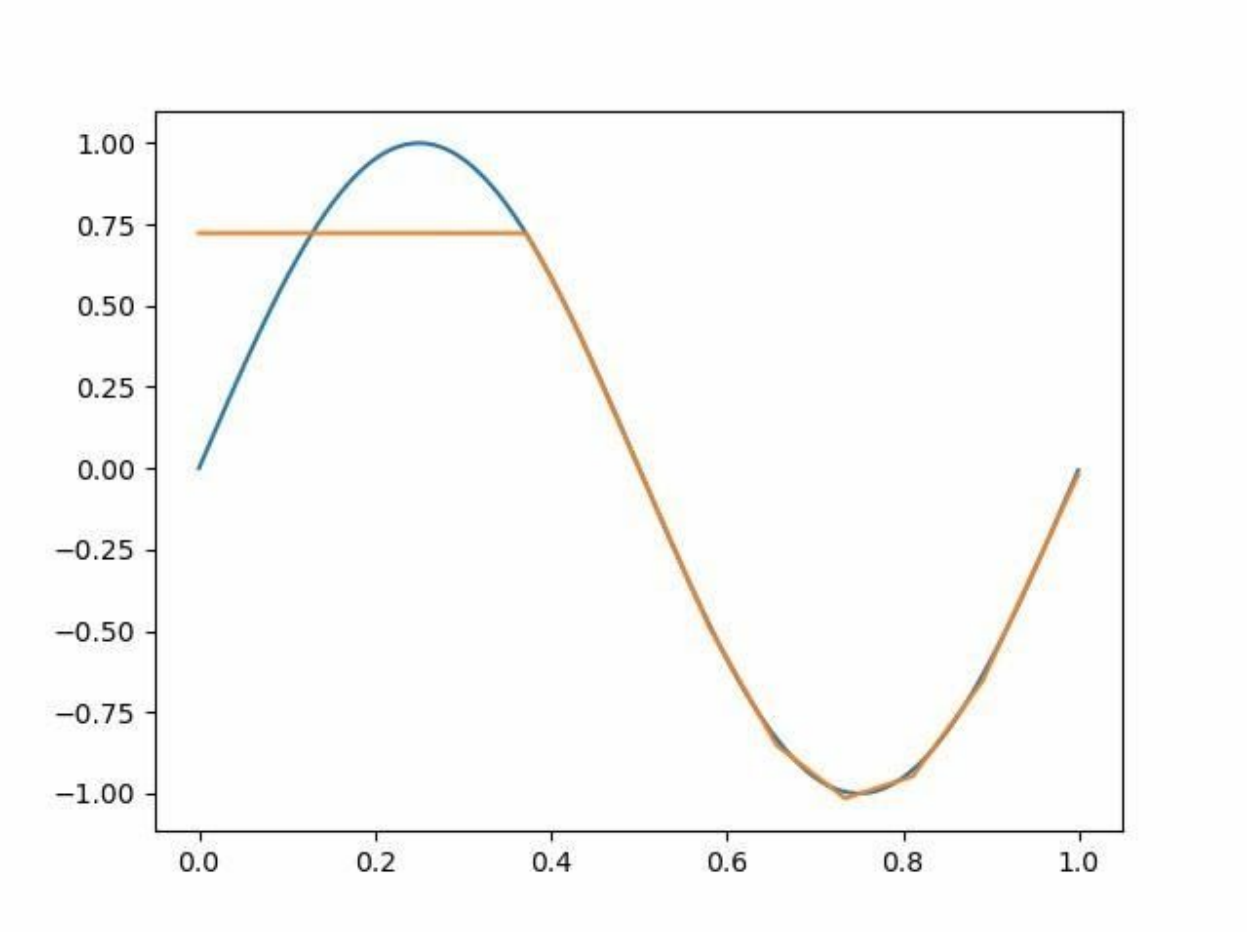

Fig 17.08: Model prediction - similar fit to the data.

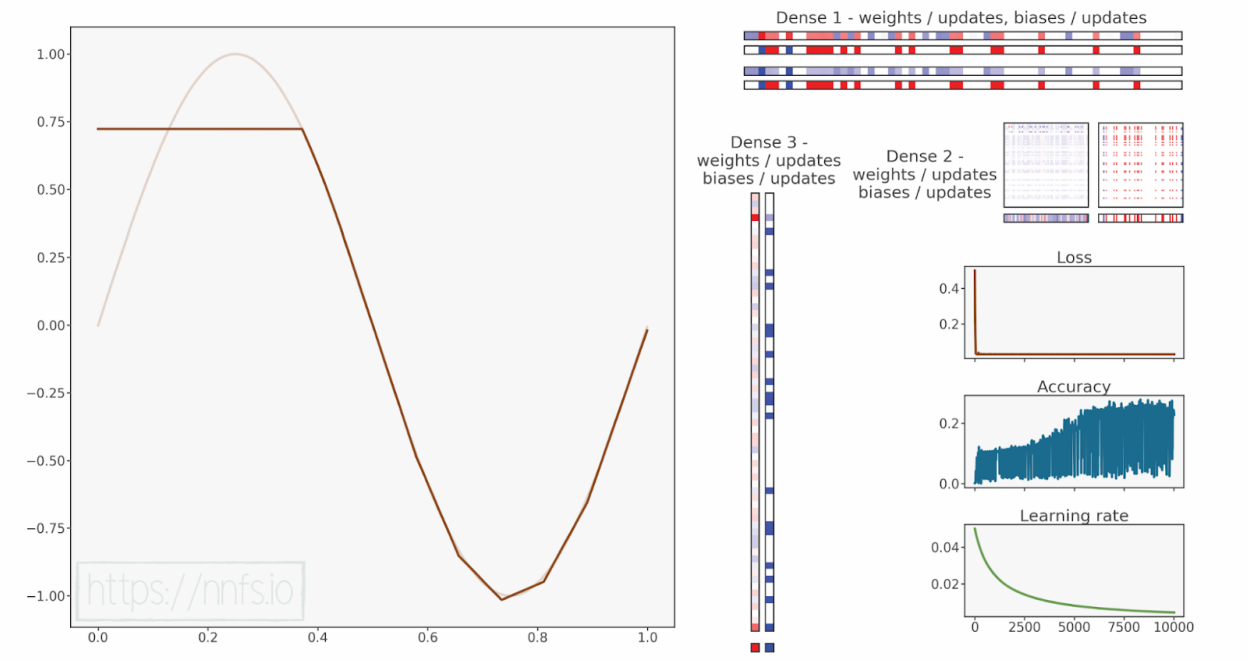

Fig 17.09: Model trained to fit the sine data, similar fit.

Anim 17.09: https://nnfs.io/jkl

It's getting even worse. Accuracy drops significantly, and we can observe the lower part of the sine being of a worse shape as well. It seems like we are not able to make this model learn the data, but after multiple tests and tuning hyperparameters, we could find a learning rate of 0.005:

```
optimizer = Optimizer_Adam(learning_rate=0.005, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.005
epoch: 100, acc: 0.017, loss: 0.048 (data_loss: 0.048, reg_loss: 0.000), lr:
0.004549590536851684
...
epoch: 9900, acc: 0.982, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045875768419121016
epoch: 10000, acc: 0.981, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045458678061641964
```

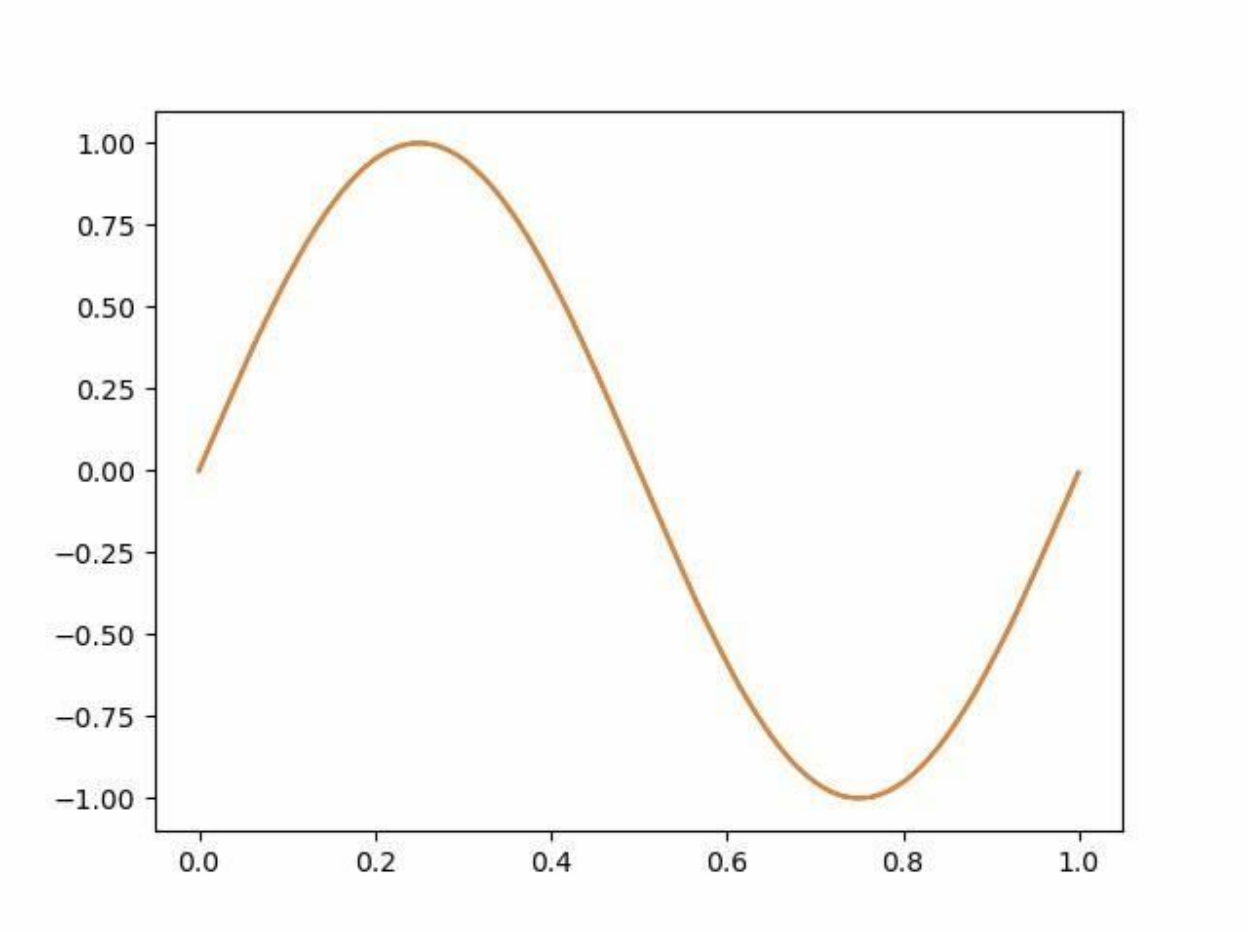

Fig 17.10: Model prediction - good fit to the data.

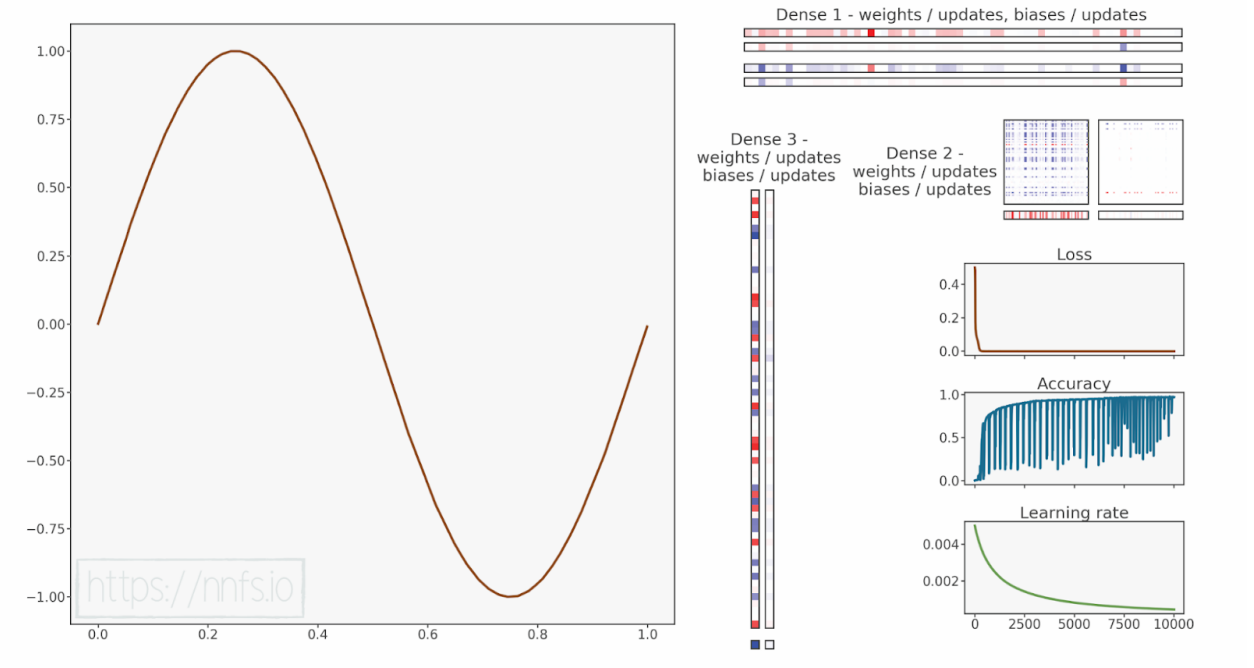

Fig 17.11: Model trained to fit the sine data.

Anim 17.11: https://nnfs.io/klm

This time model has learned pretty well, but the curious part is that both lower and higher learning rates than what we used here initially caused accuracy to be pretty low and loss be stuck at the same value when the learning rate in between them actually worked.

Debugging such a problem is usually a pretty hard task and out of the scope of this book. The accuracy and loss suggest that updates to the parameters are not big enough, but the rising learning rate makes things only worse, and there is just this single spot that we were able to find that lets our model learn. You might recall that, back in chapter 3, we were discussing parameter initialization methods and why it's important to initialize them wisely. It turns out that, in the current case, we can help the model learn by changing the factor of 0.01 to 0.1 in the Dense layer's weight initialization. But then you might ask — since the learning rate is being used to decide how much of a gradient to apply to the parameters, why does changing these initial values help instead? As you may recall, the back-propagated gradient is calculated using weights, and the learning rate does not affect it. That's why it's important to use right weight initialization, and so far, we have been using the same values for each model.

If we, for example, take a look at the source code of Keras, a neural network framework, we can learn that:

```
def glorot_uniform(seed=None):
    """Glorot uniform initializer, also called Xavier uniform initializer.

    It draws samples from a uniform distribution within [-limit, limit]
    where `limit` is `sqrt(6 / (fan_in + fan_out))`
    where `fan_in` is the number of input units in the weight tensor
    and `fan_out` is the number of output units in the weight tensor.

    # Arguments
        seed: A Python integer. Used to seed the random generator.

    # Returns
        An initializer.

    # References
        Glorot & Bengio, AISTATS 2010
        http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf
    """
    return VarianceScaling(scale=1.,
                           mode='fan_avg',
                           distribution='uniform',
                           seed=seed)
```

This code is part of the Keras 2 library. The important part of the above is actually the comment section, which describes how it initializes weights. We can find there are important pieces of information to remember — the fraction that multiplies the draw from the uniform distribution depends on the number of inputs and the number of neurons and is not constant like in our case. This method of initialization is called **Glorot uniform**. We (the authors of this book) actually have had a very similar problem in one of our projects, and changing the way weights were initialized changed the model from not learning at all to a learning state.

For the purposes of this model, let's change the factor multiplying the draw from the normal distribution in the weight initialization of the *Dense* layer to 0.1 and re-run all four of the above attempts to compare results:

```
self.weights = 0.1 * np.random.randn(n_inputs, n_neurons)
```

And all above tests re-ran:

```
optimizer = Optimizer_Adam()
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.001
epoch: 100, acc: 0.005, loss: 0.114 (data_loss: 0.114, reg_loss: 0.000), lr:
0.001
...
epoch: 9900, acc: 0.869, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.001
epoch: 10000, acc: 0.883, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.001
```

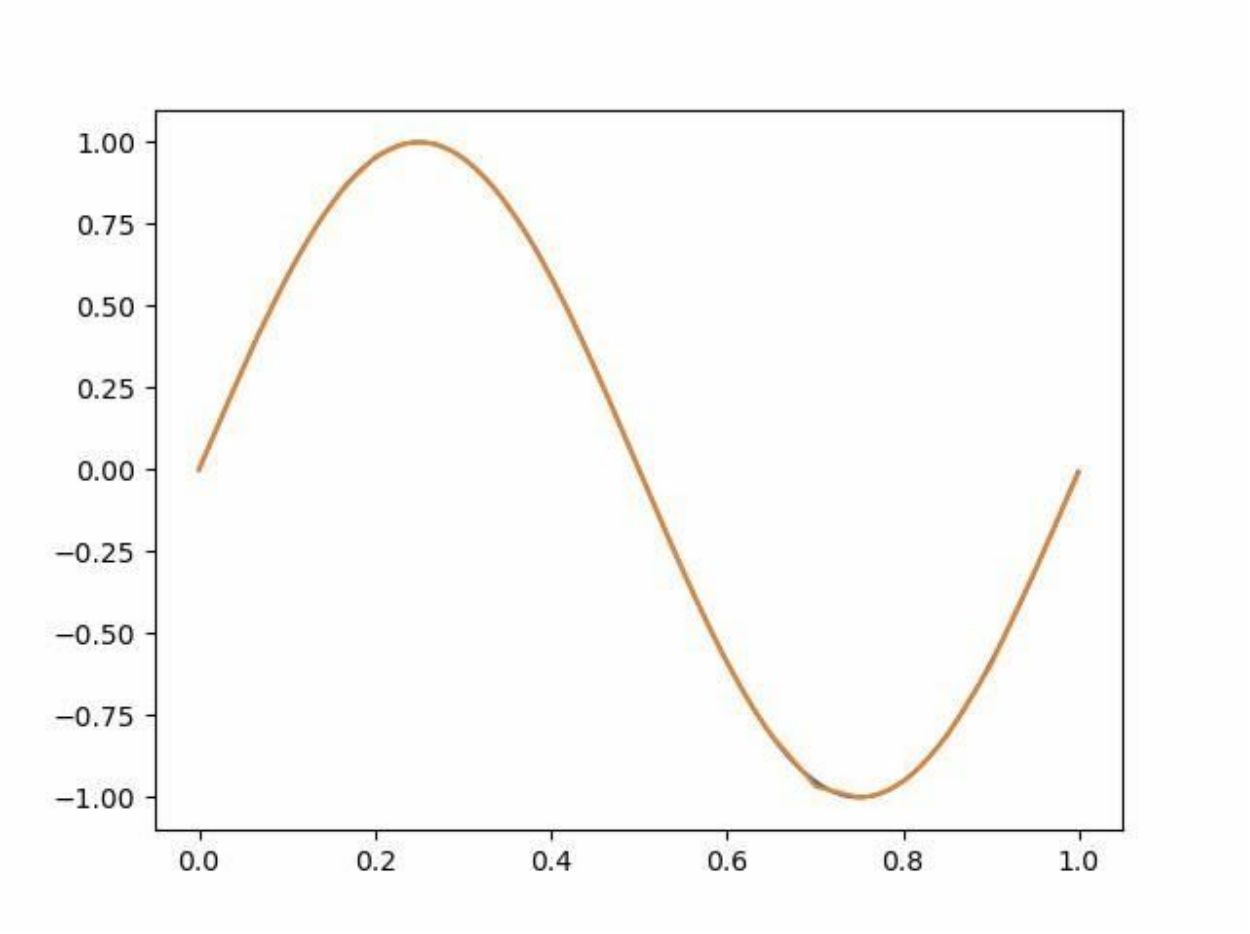

Fig 17.12: Model prediction - good fit to the data with different weight initialization.

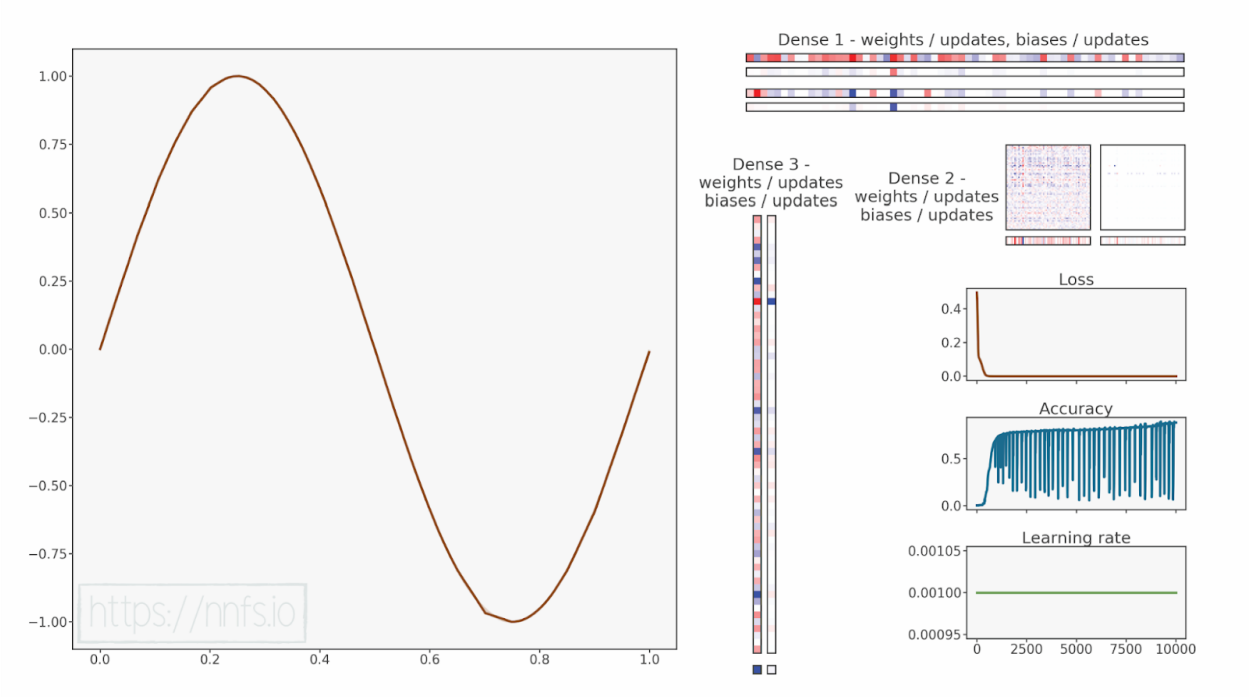

Fig 17.13: Model trained to fit the sine data after replacing weight initialization.

Anim 17.13: https://nnfs.io/lmn

This model was previously stuck and has now achieved high accuracy. There are some visible imperfections like at the bottom side of this sine, but the overall result is better.

```
optimizer = Optimizer_Adam(learning_rate=0.01, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.01
epoch: 100, acc: 0.065, loss: 0.011 (data_loss: 0.011, reg_loss: 0.000), lr:
0.009099181073703368
...
epoch: 9900, acc: 0.958, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.0009175153683824203
```

```
epoch: 10000, acc: 0.949, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00090917356123283 93
```

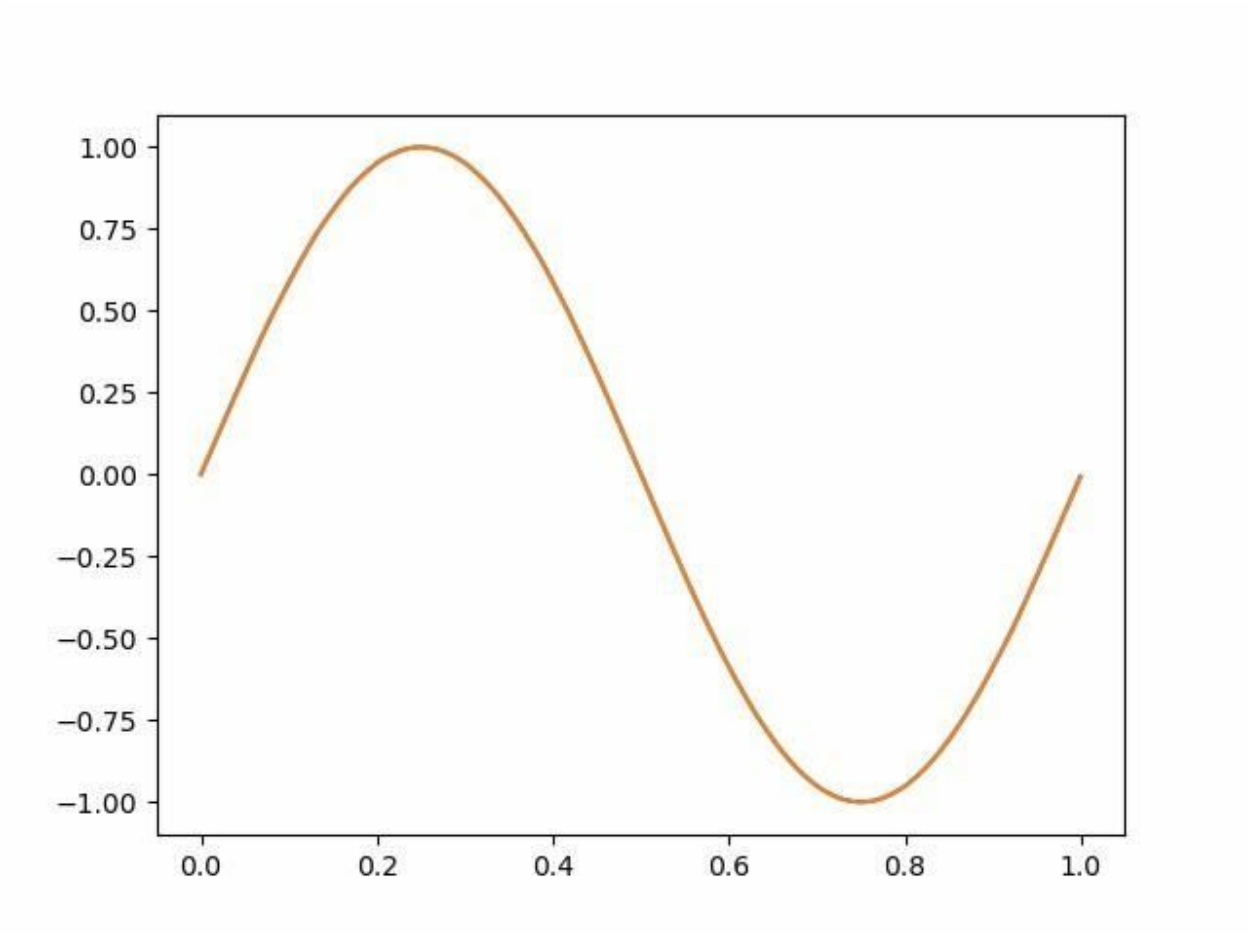

Fig 17.14: Model prediction - good fit to the data with different weight initialization.

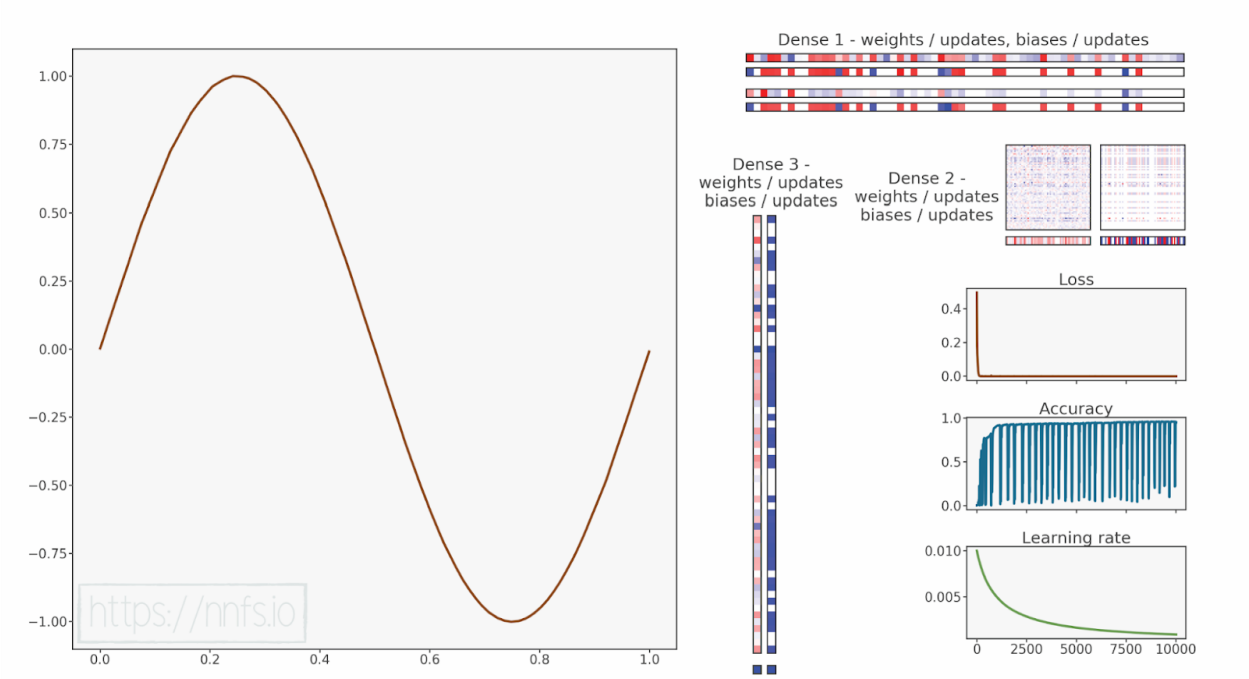

Fig 17.15: Model trained to fit the sine data after replacing weight initialization.

Anim 17.15: https://nnfs.io/mno

Another previously stuck model has trained very well this time, achieving very good accuracy.

```
optimizer = Optimizer_Adam(learning_rate=0.05, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.05
epoch: 100, acc: 0.016, loss: 0.008 (data_loss: 0.008, reg_loss: 0.000), lr:
0.04549590536851684
...
epoch: 9000, acc: 0.802, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.005000500050005001
epoch: 9100, acc: 0.233, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
```

```
lr: 0.004950985246063967
epoch: 9200, acc: 0.434, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004902441415825081
epoch: 9300, acc: 0.838, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.0048548402757549285
epoch: 9400, acc: 0.309, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004808154630252909
epoch: 9500, acc: 0.253, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004762358319839985
epoch: 9600, acc: 0.795, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004717426172280404
epoch: 9700, acc: 0.802, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004673333956444528
epoch: 9800, acc: 0.141, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004630058338735069
epoch: 9900, acc: 0.221, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.004587576841912101
epoch: 10000, acc: 0.631, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.0045458678061641965
```

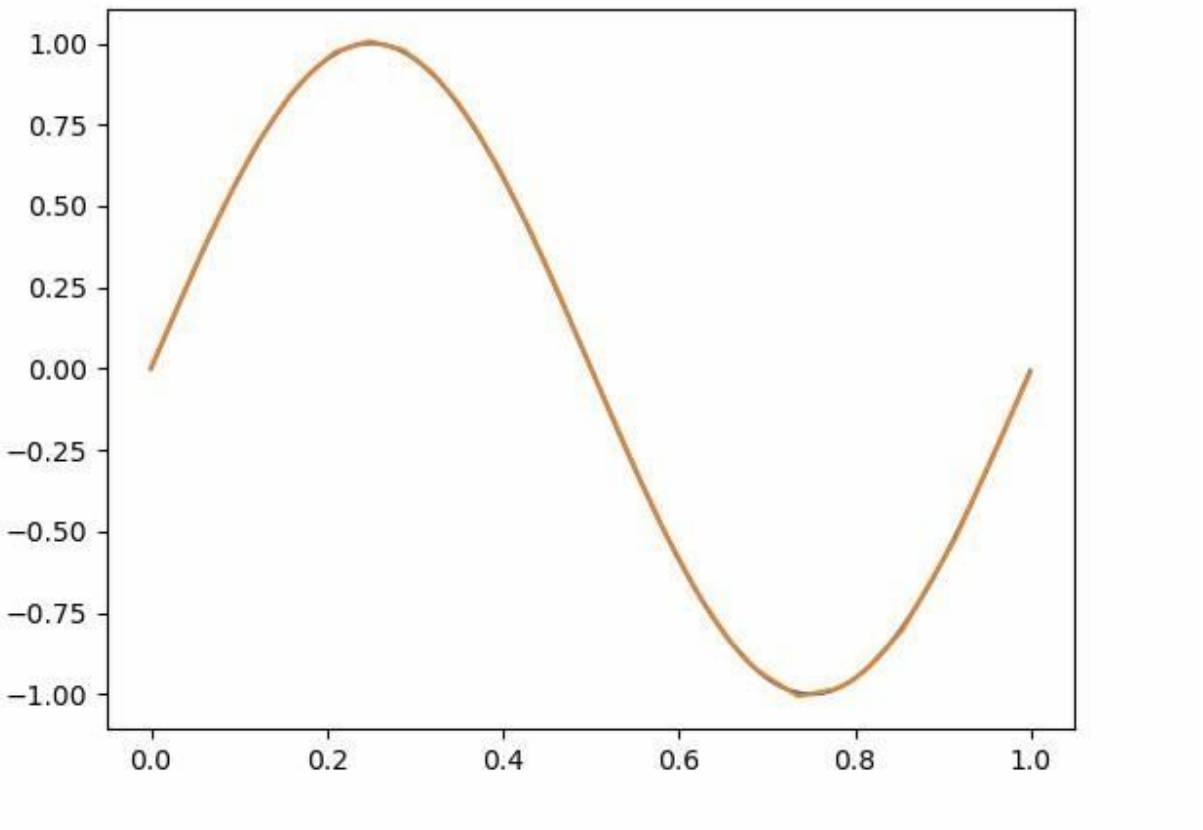

Fig 17.16: Model prediction - good fit to the data with different weight initialization.

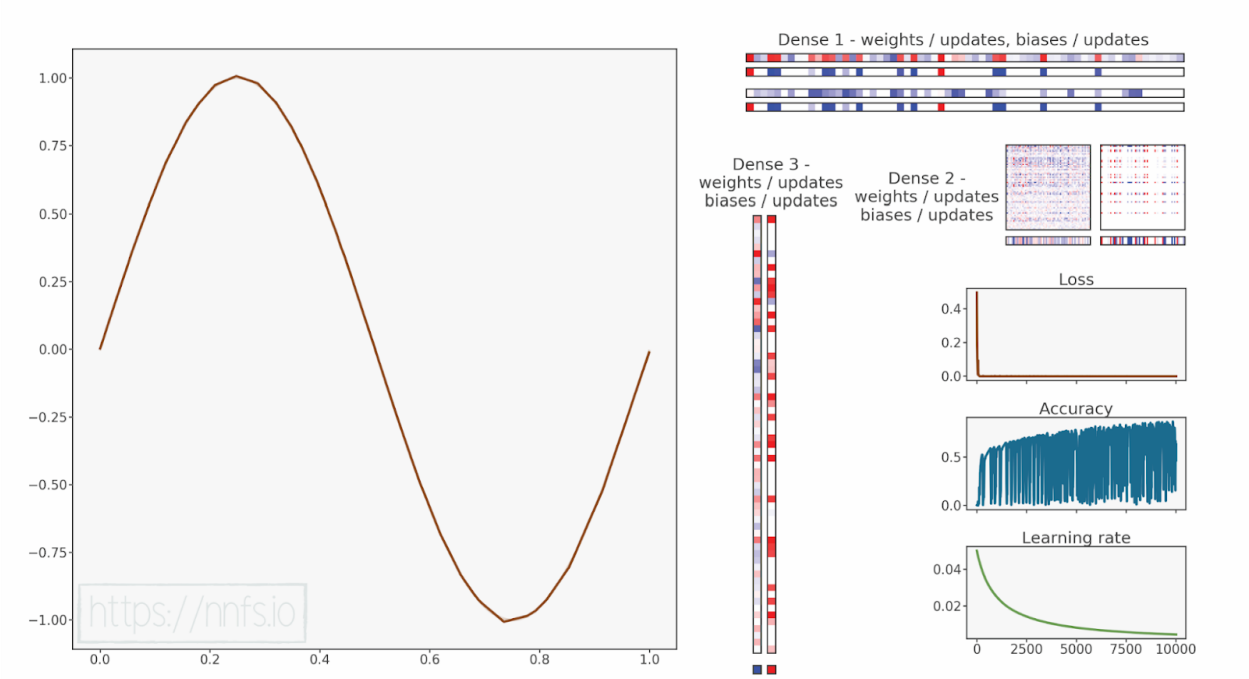

Fig 17.17: Model trained to fit the sine data after replacing weight initialization.

Anim 17.17: https://nnfs.io/nop

The "jumping" accuracy in the case of this set of the optimizer settings shows that the learning rate is way too big, but even then, the model learned the shape of the sine function considerably well.

```
optimizer = Optimizer_Adam(learning_rate=0.005, decay=1e-3)
```

```
>>>
epoch: 0, acc: 0.003, loss: 0.496 (data_loss: 0.496, reg_loss: 0.000), lr:
0.005
epoch: 100, acc: 0.017, loss: 0.048 (data_loss: 0.048, reg_loss: 0.000), lr:
0.004549590536851684
epoch: 200, acc: 0.242, loss: 0.001 (data_loss: 0.001, reg_loss: 0.000), lr:
0.004170141784820684
```

```
epoch: 300, acc: 0.786, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr:
0.003849114703618168
epoch: 400, acc: 0.885, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000), lr:
0.0035739814152966403
...
epoch: 9900, acc: 0.982, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045875768419121016
epoch: 10000, acc: 0.981, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045458678061641964
```

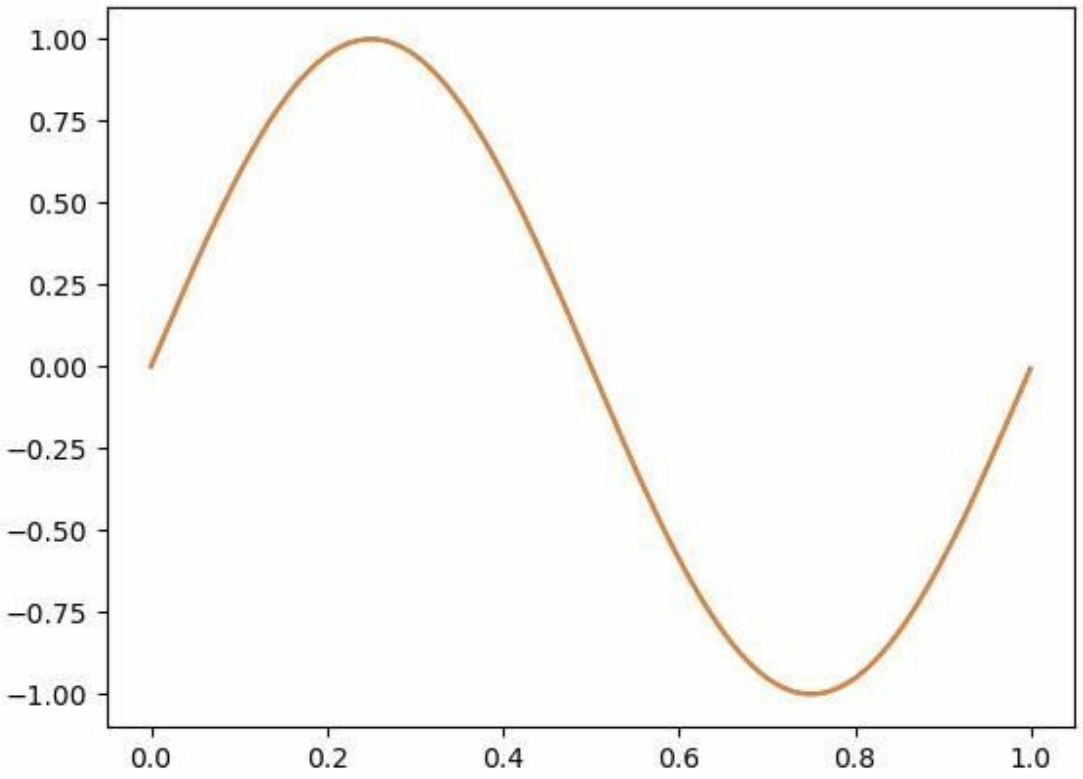

Fig 17.18: Model prediction - best fit to the data with different weight initialization.

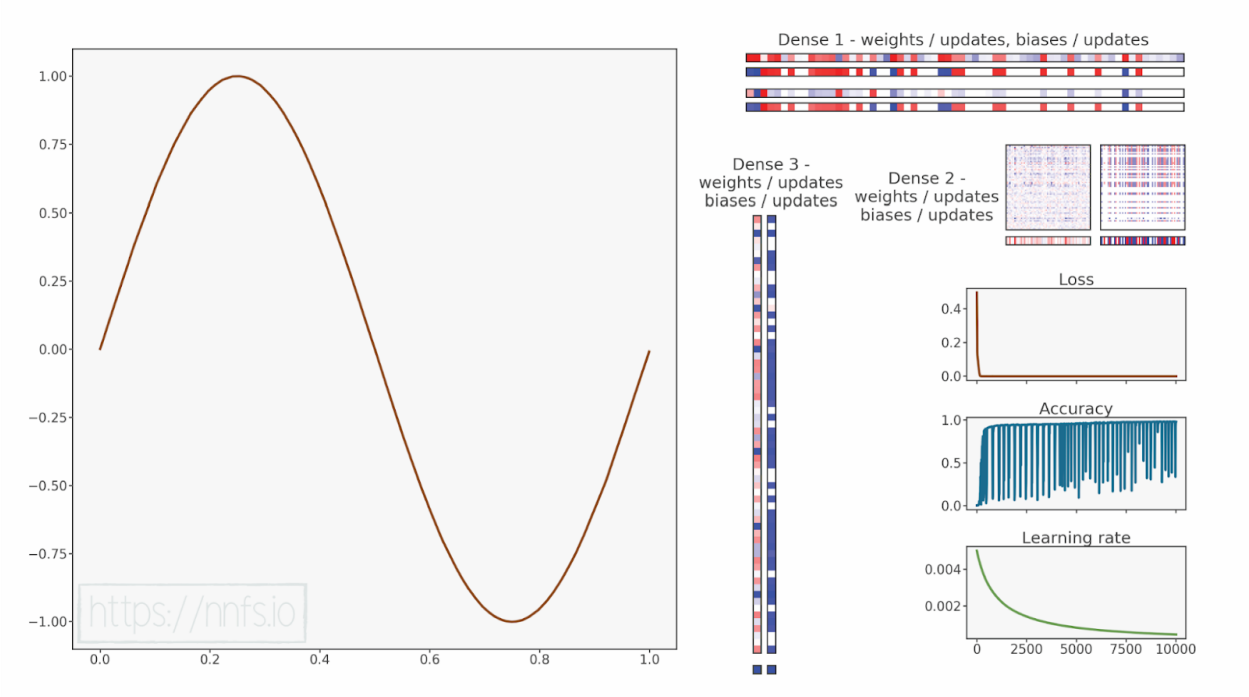

Fig 17.19: Model trained to best fit the sine data after replacing weight initialization.

Anim 17.19: https://nnfs.io/opq

These hyperparameters yielded the best results again, but not by much.

As we can see, this time, our model learned in all cases, using different learning rates, and did not get stuck if any of them. That's how much changing weight initialization can impact the training process.

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import sine_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 = 0.1 * 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
            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)


# 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


# Linear activation
class Activation_Linear:

    # Forward pass
    def forward(self, inputs):
        # Just remember values
        self.inputs = inputs
        self.output = inputs

    # Backward pass
    def backward(self, dvalues):
        # derivative is 1, 1 * dvalues = dvalues - the chain rule
        self.dinputs = dvalues.copy()


# 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


# Binary cross-entropy loss
class Loss_BinaryCrossentropy(Loss):

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

        # 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)

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

        # Return losses
        return sample_losses

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

        # 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])

        # 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)
```

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


# Mean Squared Error loss
class Loss_MeanSquaredError(Loss):  # L2 loss

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

        # Calculate loss
        sample_losses = np.mean((y_true - y_pred)**2, axis=-1)

        # Return losses
        return sample_losses

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

        # 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])

        # Gradient on values
        self.dinputs = -2 * (y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Mean Absolute Error loss
class Loss_MeanAbsoluteError(Loss):  # L1 loss

    def forward(self, y_pred, y_true):

        # Calculate loss
        sample_losses = np.mean(np.abs(y_true - y_pred), axis=-1)

        # Return losses
        return sample_losses
```

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

        # 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 = np.sign(y_true - dvalues) / outputs
        # Normalize gradient
        self.dinputs = self.dinputs / samples


# Create dataset
X, y = sine_data()

# Create Dense layer with 1 input feature and 64 output values
dense1 = Layer_Dense(1, 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 64 output values
dense2 = Layer_Dense(64, 64)

# Create ReLU activation (to be used with Dense layer):
activation2 = Activation_ReLU()

# Create third Dense layer with 64 input features (as we take output
# of previous layer here) and 1 output value
dense3 = Layer_Dense(64, 1)

# Create Linear activation:
activation3 = Activation_Linear()

# Create loss function
loss_function = Loss_MeanSquaredError()

# Create optimizer
optimizer = Optimizer_Adam(learning_rate=0.005, decay=1e-3)
```

```
# Accuracy precision for accuracy calculation
# There are no really accuracy factor for regression problem,
# but we can simulate/approximate it. We'll calculate it by checking
# how many values have a difference to their ground truth equivalent
# less than given precision
# We'll calculate this precision as a fraction of standard deviation
# of all the ground truth values
accuracy_precision = np.std(y) / 250

# 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 activation function
    # takes the output of second dense layer here
    activation2.forward(dense2.output)

    # Perform a forward pass through third Dense layer
    # takes outputs of activation function of second layer as inputs
    dense3.forward(activation2.output)

    # Perform a forward pass through activation function
    # takes the output of third dense layer here
    activation3.forward(dense3.output)

    # Calculate the data loss
    data_loss = loss_function.calculate(activation3.output, y)

    # Calculate regularization penalty
    regularization_loss = \
        loss_function.regularization_loss(dense1) + \
        loss_function.regularization_loss(dense2) + \
        loss_function.regularization_loss(dense3)

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

```
    # Calculate accuracy from output of activation2 and targets
    # To calculate it we're taking absolute difference between
    # predictions and ground truth values and compare if differences
    # are lower than given precision value
    predictions = activation3.output
    accuracy = np.mean(np.absolute(predictions - y) <
                       accuracy_precision)

    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_function.backward(activation3.output, y)
    activation3.backward(loss_function.dinputs)
    dense3.backward(activation3.dinputs)
    activation2.backward(dense3.dinputs)
    dense2.backward(activation2.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.update_params(dense3)
    optimizer.post_update_params()


import matplotlib.pyplot as plt

X_test, y_test = sine_data()

dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)
dense3.forward(activation2.output)
activation3.forward(dense3.output)

plt.plot(X_test, y_test)
plt.plot(X_test, activation3.output)
plt.show()
```

Supplementary Material: https://nnfs.io/ch17
Chapter code, further resources, and errata for this chapter

Chapter 18

Model Object

We built a model that can perform the forward pass, backward pass, and ancillary tasks like measuring accuracy. We have built all this by writing a fair bit of code and making modifications in some decently-sized blocks of code. It's beginning to make more sense to make our model an object itself, especially since we will want to do things like save and load this object to use for future prediction tasks. We will also use this object to cut down on some of the more common lines of code, making it easier to work with our current code base and build new models. To do this model object conversion, we'll use the last model we were working on, the regression model with sine data:

```
from nnfs.datasets import sine_data
X, y = sine_data()
```

Once we have the data, our first step for the model class is to add in the various layers we want. Thus, we can begin our model class by doing:

```
# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)
```

This allows us to use the `add` method of the model object to add layers. This alone will help with legibility considerably. Let's add some layers:

```
# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())
```

We can also query this model now:

```
print(model.layers)


>>>
[<__main__.Layer_Dense object at 0x0000015E9D504BC8>,
<__main__.Activation_ReLU object at 0x0000015E9D504C48>,
<__main__.Layer_Dense object at 0x0000015E9D504C88>,
<__main__.Activation_ReLU object at 0x0000015E9D504CC8>,
<__main__.Layer_Dense object at 0x0000015E9D504D08>,
<__main__.Activation_Linear object at 0x0000015E9D504D88>]
```

Besides adding layers, we also want to set a loss function and optimizer for the model. To do this, we'll create a method called `set`:

```
    # Set loss and optimizer
    def set(self, *, loss, optimizer):
        self.loss = loss
        self.optimizer = optimizer
```

The use of the asterisk in the parameter definitions notes that the subsequent parameters (*loss* and *optimizer* in this case) are keyword arguments. Since they have no default value assigned, they are required keyword arguments, which means that they have to be passed by names and values, making code more legible.

Now we can add a call to this method into our newly-created model object, and pass the loss and optimizer objects:

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())

# Set loss and optimizer objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
)
```

After we've set our model's layers, loss function, and optimizer, the next step is to train, so we'll add a train method. For now, we'll make it a placeholder and fill it in soon:

```
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Main training loop
        for epoch in range(1, epochs+1):

            # Temporary
            pass
```

We can then add a call to the train method in the model definition. We'll pass the training data, the number of epochs (10000, as we've used so far), and an indicator of how often to print a training summary. We do not need or want to print it every step, so we'll make it configurable:

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())

# Set loss and optimizer objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
)

model.train(X, y, epochs=10000, print_every=100)
```

To actually train, we need to perform a forward pass. Performing this forward pass in the object is slightly more complicated because we want to do this in a loop over the layers, and we need to know the previous layer's output to pass data properly. One issue with querying the previous layer is that the first layer doesn't have a "previous" layer. The first layer that we're defining is the first **hidden** layer. One option we have then is to create an "input layer." This is considered a layer in a neural network but doesn't have weights and biases associated with it. The input layer only contains the training data, and we'll only use it as a "previous" layer to the first layer during the iteration of the layers in a loop. We'll create a new class and call it in similarly as `Layer_Dense` class — `Layer_Input`:

```
# Input "layer"
class Layer_Input:

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

The `forward` method sets training samples as `self.output`. This property is common with other layers. There's no need for a backward method here since we'll never use it. It might seem silly right now to even have this class, but it should hopefully become clear how we're going to use this shortly. The next thing we're going to do is set the previous and next layer properties for each of the model's layers. We'll create a method called `finalize` in the `Model` class:

```
# Finalize the model
def finalize(self):

    # Create and set the input layer
    self.input_layer = Layer_Input()

    # Count all the objects
    layer_count = len(self.layers)

    # Iterate the objects
    for i in range(layer_count):

        # If it's the first layer,
        # the previous layer object is the input layer
        if i == 0:
            self.layers[i].prev = self.input_layer
            self.layers[i].next = self.layers[i+1]

        # All layers except for the first and the last
        elif i < layer_count - 1:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.layers[i+1]

        # The last layer - the next object is the loss
        else:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.loss
```

This code creates an input layer and sets `next` and `prev` references for each layer contained within the `self.layers` list of a model object. We wanted to create the `Layer_Input` class to set the `prev` property of the first hidden layer in a loop since we are going to call all of the layers in a uniform way. The `next` layer for the final layer will be the loss, which we already have created.

Now that we have the necessary layer information for our model object to perform a forward pass, let's add a forward method. We will use this forward method both for when we train and later when we just want to predict, which is also called **model inference**. Continuing the code within the `Model` class:

```
# Forward pass
class Model:
    ...
    # Performs forward pass
    def forward(self, X):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

In this case, we take in *X* (input data), then simply pass this data through the `input_layer` in the `Model` object, which creates an `output` attribute in this object. From here, we begin iterating over the `self.layers`, the layers starting with the first hidden layer. We perform a forward pass on the `layer.prev.output`, the output data of the previous layer, for each layer. For the first hidden layer, the `layer.prev` is `self.input_layer`. The `output` attribute is created for each layer when we call the `forward` method, which is then used as input to a `forward` method call on the next layer. Once we've iterated over all of the layers, we return the final layer's output. That's a forward pass, and now let's go ahead and add this forward pass method call to the `train` method in the `Model` class:

```
# Forward pass
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Temporary
            print(output)
            exit()
```

Full `Model` class up to this point:

```
# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)

    # Set loss and optimizer
    def set(self, *, loss, optimizer):
        self.loss = loss
        self.optimizer = optimizer

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]

            # The last layer - the next object is the loss
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
```

```
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Temporary
            print(output)
            exit()

    # Performs forward pass
    def forward(self, X):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

Finally, we can add in the `finalize` method call to the main code (recall this method makes, among other things, the model's layers aware of their previous and next layers).

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())
```

```
# Set loss and optimizer objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, epochs=10000, print_every=100)
```

Running this:

```
>>>
[[ 0.00000000e+00]
 [-1.13209149e-08]
 [-2.26418297e-08]
...
 [-1.12869511e-05]
 [-1.12982725e-05]
 [-1.13095930e-05]]
```

At this point, we've covered the forward pass of our model in the `Model` class. We still need to calculate loss and accuracy along with doing backpropagation. Before doing this, we need to know which layers are "trainable," which means layers with weights and biases that we can tweak. To do this, we need to check if the layer has either a `weights` or `biases` attribute. We can check this with the following code:

```
# If layer contains an attribute called "weights,"
# it's a trainable layer -
# add it to the list of trainable layers
# We don't need to check for biases -
# checking for weights is enough
if hasattr(self.layers[i], 'weights'):
    self.trainable_layers.append(self.layers[i])
```

Where `i` is the index for the layer in the list of layers. We'll put this code into the `finalize` method. The full code for that method so far:

```
# Finalize the model
def finalize(self):

    # Create and set the input layer
    self.input_layer = Layer_Input()

    # Count all the objects
    layer_count = len(self.layers)

    # Initialize a list containing trainable layers:
    self.trainable_layers = []

    # Iterate the objects
    for i in range(layer_count):

        # If it's the first layer,
        # the previous layer object is the input layer
        if i == 0:
            self.layers[i].prev = self.input_layer
            self.layers[i].next = self.layers[i+1]

        # All layers except for the first and the last
        elif i < layer_count - 1:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.layers[i+1]

        # The last layer - the next object is the loss
        # Also let's save aside the reference to the last object
        # whose output is the model's output
        else:
            self.layers[i].prev = self.layers[i-1]
            self.layers[i].next = self.loss
            self.output_layer_activation = self.layers[i]

        # If layer contains an attribute called "weights",
        # it's a trainable layer -
        # add it to the list of trainable layers
        # We don't need to check for biases -
        # checking for weights is enough
        if hasattr(self.layers[i], 'weights'):
            self.trainable_layers.append(self.layers[i])
```

Next, we'll modify the common `Loss` class to contain the following:

```
# Common loss class
class Loss:
    ...
    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers

    # 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 the data and regularization losses
        return data_loss, self.regularization_loss()
```

The `remember_trainable_layers` method in the common `Loss` class "tells" the loss object which layers in the `Model` object are trainable. The `calculate` method was modified to also return the `self.regularization_loss()` during a single call. The `regularization_loss` method currently requires a layer object, but with the `self.trainable_layers` property set in `remember_trainable_layers`, method we can now iterate over the trainable layers to compute regularization loss for the entire model, rather than one layer at a time:

```
class Loss:
    ...
    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # 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
```

For calculating accuracy, we need predictions. So far, predicting has required different code depending on the type of model. For a softmax classifier, we do a `np.argmax()`, but for regression, the prediction is the direct output because of the linear activation function being used in an output layer. Ideally, we'd have a prediction method that would choose the appropriate method for our model. To do this, we'll add a `predictions` method to each activation function class:

```
# Softmax activation
class Activation_Softmax:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return np.argmax(outputs, axis=1)


# Sigmoid activation
class Activation_Sigmoid:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return (outputs > 0.5) * 1
```

```
# Linear activation
class Activation_Linear:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

All the computations made inside the `predictions` functions are the same as those performed with appropriate models in previous chapters. While we have no plans for using the ReLU activation function for an output layer's activation function, we'll include it here for completeness:

```
# ReLU activation
class Activation_ReLU:
    ...
    # Calculate predictions for outputs
    def predictions(self, outputs):
        return outputs
```

We still need to set a reference to the activation function for the final layer in the `Model` object. We can later call the `predictions` method, which will return predictions calculated from the outputs. We'll set this in the `Model` class' `finalize` method:

```
# Model class
class Model:
    ...
    def finalize(self):
        ...
            # The last layer - the next object is the loss
            # Also let's save aside the reference to the last object
            # whose output is the model's output
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
                self.output_layer_activation = self.layers[i]
```

Just like the different prediction methods, we also calculate accuracy in different ways. We're going to implement this in a way similar to the specific loss class' objects implementation — we'll create specific accuracy classes and their objects, which we'll associate with models.

First, we'll write a common `Accuracy` class containing (for now) just a single method, `calculate`, returning an accuracy calculated from comparison results. We've already added a call to the `self.compare` method that does not exist yet, but we'll create it soon in other classes that will inherit from this `Accuracy` class. For now, it's enough to know that it will return a list of *True* and *False* values, indicating if a prediction matches the ground-truth value. Next, we calculate the mean value (which treats *True* as *1* and *False* as *0*) and return it as an accuracy. The

code:

```
# Common accuracy class
class Accuracy:

    # Calculates an accuracy
    # given predictions and ground truth values
    def calculate(self, predictions, y):

        # Get comparison results
        comparisons = self.compare(predictions, y)

        # Calculate an accuracy
        accuracy = np.mean(comparisons)

        # Return accuracy
        return accuracy
```

Next, we can work with this common `Accuracy` class, inheriting from it, then building further for specific types of models. In general, each of these classes will contain two methods: `init` (not to be confused with a Python class' `__init__` method) for initialization from inside the model object and `compare` for performing comparison calculations. For regression, the `init` method will calculate an accuracy precision, the same as we have written previously for the regression model, and have been running before the training loop. The `compare` method will contain the actual comparison code we have implemented in the training loop itself, which uses `self.precision`. Note that initialization won't recalculate precision unless forced to do so by setting the *`reinit`* parameter to `True`. This allows for multiple use-cases, including setting `self.precision` independently, calling `init` whenever needed (e.g., from outside of the model during its creation), and even calling it multiple times (which will become handy soon):

```
# Accuracy calculation for regression model
class Accuracy_Regression(Accuracy):

    def __init__(self):
        # Create precision property
        self.precision = None

    # Calculates precision value
    # based on passed in ground truth
    def init(self, y, reinit=False):
        if self.precision is None or reinit:
            self.precision = np.std(y) / 250

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        return np.absolute(predictions - y) < self.precision
```

We can then set the accuracy object from within the `set` method in our `Model` class the same way as the loss and optimizer currently:

```
# Model class
class Model:
    ...
    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy
```

Then we can finally add the loss and accuracy calculations to our model right after the completed forward pass' code. Note that we also initialize the accuracy with `self.accuracy.init(y)` at the beginning of the `train` method, which can be called multiple times — as noted earlier. In the case of regression accuracy, this will invoke a precision calculation a single time during the first call. The code of the `train` method with implemented loss and accuracy calculations:

```
# Model class
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y)
```

Finally, we'll add a call to the previously created method `remember_trainable_layers` with the `Loss` class' object, which we'll do in the `finalize` method (`self.loss.remember_trainable_layers(self.trainable_layers)`). The full model class code so far:

```
# Model class
class Model:

    def __init__(self):
        # Create a list of network objects
        self.layers = []

    # Add objects to the model
    def add(self, layer):
        self.layers.append(layer)

    # Set loss, optimizer and accuracy
    def set(self, *, loss, optimizer, accuracy):
        self.loss = loss
        self.optimizer = optimizer
        self.accuracy = accuracy

    # Finalize the model
    def finalize(self):

        # Create and set the input layer
        self.input_layer = Layer_Input()

        # Count all the objects
        layer_count = len(self.layers)

        # Initialize a list containing trainable layers:
        self.trainable_layers = []

        # Iterate the objects
        for i in range(layer_count):

            # If it's the first layer,
            # the previous layer object is the input layer
            if i == 0:
                self.layers[i].prev = self.input_layer
                self.layers[i].next = self.layers[i+1]

            # All layers except for the first and the last
            elif i < layer_count - 1:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.layers[i+1]
```

```
            # The last layer - the next object is the loss
            # Also let's save aside the reference to the last object
            # whose output is the model's output
            else:
                self.layers[i].prev = self.layers[i-1]
                self.layers[i].next = self.loss
                self.output_layer_activation = self.layers[i]

            # If layer contains an attribute called "weights",
            # it's a trainable layer -
            # add it to the list of trainable layers
            # We don't need to check for biases -
            # checking for weights is enough
            if hasattr(self.layers[i], 'weights'):
                self.trainable_layers.append(self.layers[i])

        # Update loss object with trainable layers
        self.loss.remember_trainable_layers(
            self.trainable_layers
        )

    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y)
```

```
    # Performs forward pass
    def forward(self, X):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output)

        # "layer" is now the last object from the list,
        # return its output
        return layer.output
```

Full code for the `Loss` class:

```
# Common loss class
class Loss:

    # Regularization loss calculation
    def regularization_loss(self):

        # 0 by default
        regularization_loss = 0

        # Calculate regularization loss
        # iterate all trainable layers
        for layer in self.trainable_layers:

            # 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
            # only calculate 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

    # Set/remember trainable layers
    def remember_trainable_layers(self, trainable_layers):
        self.trainable_layers = trainable_layers

    # 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 the data and regularization losses
        return data_loss, self.regularization_loss()
```

Now that we've done a full forward pass and have calculated loss and accuracy, we can begin the backward pass. The `backward` method in the `Model` class is structurally similar to the `forward` method, just in reverse and using different parameters. Following the backward pass in our previous training approach, we need to call the `backward` method of a loss object to create the `dinputs` property. Next, we'll loop through all the layers in reverse order, calling their `backward` methods with the `dinputs` property of the next layer (in normal order) as a parameter, effectively backpropagating the gradient returned by that next layer. Remember that we have set the loss object as a `next` layer in the last, output layer.

```
# Model class
class Model:
    ...
    # Performs backward pass
    def backward(self, output, y):

        # First call backward method on the loss
        # this will set dinputs property that the last
        # layer will try to access shortly
        self.loss.backward(output, y)

        # Call backward method going through all the objects
        # in reversed order passing dinputs as a parameter
        for layer in reversed(self.layers):
            layer.backward(layer.next.dinputs)
```

Next, we'll add a call of this `backward` method to the end of the `train` method:

```
            # Perform backward pass
            self.backward(output, y)
```

After this backward pass, the last action to perform is to optimize. We have previously been calling the optimizer object's `update_params` method as many times as we had trainable layers. We have to make this code universal as well by looping through the list of trainable layers and calling `update_params()` in this loop:

```
            # Optimize (update parameters)
            self.optimizer.pre_update_params()
            for layer in self.trainable_layers:
                self.optimizer.update_params(layer)
            self.optimizer.post_update_params()
```

Then we can output useful information — here's where this last parameter to the `train` method becomes handy:

```
            # Print a summary
            if not epoch % print_every:
                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: {self.optimizer.current_learning_rate}')
```

```
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y)

            # Perform backward pass
            self.backward(output, y)

            # Optimize (update parameters)
            self.optimizer.pre_update_params()
            for layer in self.trainable_layers:
                self.optimizer.update_params(layer)
            self.optimizer.post_update_params()

            # Print a summary
            if not epoch % print_every:
                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: {self.optimizer.current_learning_rate}')
```

We can now pass the accuracy class' object into the model and test our model's performance:

```
# Create dataset
X, y = sine_data()

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(1, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 64))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Linear())

# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_MeanSquaredError(),
    optimizer=Optimizer_Adam(learning_rate=0.005, decay=1e-3),
    accuracy=Accuracy_Regression()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, epochs=10000, print_every=100)


>>>
epoch: 100, acc: 0.006, loss: 0.085 (data_loss: 0.085, reg_loss: 0.000), lr:
0.004549590536851684
epoch: 200, acc: 0.032, loss: 0.035 (data_loss: 0.035, reg_loss: 0.000), lr:
0.004170141784820684
...
epoch: 9900, acc: 0.934, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045875768419121016
epoch: 10000, acc: 0.970, loss: 0.000 (data_loss: 0.000, reg_loss: 0.000),
lr: 0.00045458678061641964
```

Our new model is behaving well, and now we're able to make new models more easily with our `Model` class. We have to continue to modify these classes to handle for entirely new models. For example, we haven't yet handled for binary logistic regression. For this, we need to add two things. First, we need to calculate the categorical accuracy:

```
# Accuracy calculation for classification model
class Accuracy_Categorical(Accuracy):

    # No initialization is needed
    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predictions, y):
        if len(y.shape) == 2:
            y = np.argmax(y, axis=1)
        return predictions == y
```

This is the same as the accuracy calculation for classification, just wrapped into a class and with an additional switch parameter. This switch disables one-hot to sparse label conversion while this class is used with the binary cross-entropy model, since this model always require the groundtrue values to be a 2D array and they're not one-hot encoded. Note that we do not perform any initialization here, but the method needs to exist since it's going to be called from the `train` method of the `Model` class. The next thing that we need to add is the ability to validate the model using validation data. Validation requires only a forward pass and calculation of loss (just data loss). We'll modify the `calculate` method of the `Loss` class to let it calculate the validation loss as well:

```
# Common loss class
class Loss:
    ...
    # Calculates the data and regularization losses
    # given model output and ground truth values
    def calculate(self, output, y, *, include_regularization=False):

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

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

        # If just data loss - return it
        if not include_regularization:
            return data_loss

        # Return the data and regularization losses
        return data_loss, self.regularization_loss()
```

We've added a new parameter and condition to return just the data loss, as regularization loss is not being used in this case. To run it, we'll pass predictions and targets the same way as with the training data. We will not return regularization loss by default, which means we need to update the call to this method in the `train` method to include regularization loss during training:

```
            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y,
                                    include_regularization=True)
```

Then we can add the validation code to the `train` method in the `Model` class. We added the *`validation_data`* parameter to the function, which takes a tuple of validation data (samples and targets), the `if` statement to check if the validation data is present, and if it is — the code to perform a forward pass over these data, calculate loss and accuracy in the same way as during training and print the results:

```
# Model class
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1,
              validation_data=None):
        ...
        # If there is the validation data
        if validation_data is not None:

            # For better readability
            X_val, y_val = validation_data

            # Perform the forward pass
            output = self.forward(X_val)

            # Calculate the loss
            loss = self.loss.calculate(output, y_val)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                                output)
            accuracy = self.accuracy.calculate(predictions, y_val)

            # Print a summary
            print(f'validation, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f}')
```

The full `train` method for the `Model` class:

```
# Model class
class Model:
    ...
    # Train the model
    def train(self, X, y, *, epochs=1, print_every=1,
              validation_data=None):

        # Initialize accuracy object
        self.accuracy.init(y)

        # Main training loop
        for epoch in range(1, epochs+1):

            # Perform the forward pass
            output = self.forward(X)

            # Calculate loss
            data_loss, regularization_loss = \
                self.loss.calculate(output, y,
                                    include_regularization=True)
            loss = data_loss + regularization_loss

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                                output)
            accuracy = self.accuracy.calculate(predictions, y)

            # Perform backward pass
            self.backward(output, y)

            # Optimize (update parameters)
            self.optimizer.pre_update_params()
            for layer in self.trainable_layers:
                self.optimizer.update_params(layer)
            self.optimizer.post_update_params()

            # Print a summary
            if not epoch % print_every:
                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: {self.optimizer.current_learning_rate}')
```

```
        # If there is the validation data
        if validation_data is not None:

            # For better readability
            X_val, y_val = validation_data

            # Perform the forward pass
            output = self.forward(X_val)

            # Calculate the loss
            loss = self.loss.calculate(output, y_val)

            # Get predictions and calculate an accuracy
            predictions = self.output_layer_activation.predictions(
                              output)
            accuracy = self.accuracy.calculate(predictions, y_val)

            # Print a summary
            print(f'validation, ' +
                  f'acc: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f}')
```

Now we can create the test data and test the binary logistic regression model with the following code:

```
# Create train and test dataset
X, y = spiral_data(samples=100, classes=2)
X_test, y_test = spiral_data(samples=100, classes=2)

# Reshape labels to be a list of lists
# Inner list contains one output (either 0 or 1)
# per each output neuron, 1 in this case
y = y.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)

# Instantiate the model
model = Model()

# Add layers
model.add(Layer_Dense(2, 64, weight_regularizer_l2=5e-4,
                          bias_regularizer_l2=5e-4))
model.add(Activation_ReLU())
model.add(Layer_Dense(64, 1))
model.add(Activation_Sigmoid())
```

```
# Set loss, optimizer and accuracy objects
model.set(
    loss=Loss_BinaryCrossentropy(),
    optimizer=Optimizer_Adam(decay=5e-7),
    accuracy=Accuracy_Categorical()
)

# Finalize the model
model.finalize()

# Train the model
model.train(X, y, validation_data=(X_test, y_test),
            epochs=10000, print_every=100)
```

```
>>>
epoch: 100, acc: 0.625, loss: 0.675 (data_loss: 0.674, reg_loss: 0.001), lr:
0.0009999505024501287
epoch: 200, acc: 0.630, loss: 0.669 (data_loss: 0.668, reg_loss: 0.001), lr:
0.0009999005098992651
...
epoch: 9900, acc: 0.905, loss: 0.312 (data_loss: 0.276, reg_loss: 0.037),
lr: 0.0009950748768967994
epoch: 10000, acc: 0.905, loss: 0.312 (data_loss: 0.275, reg_loss: 0.036),
lr: 0.0009950253706593885
validation, acc: 0.775, loss: 0.423
```

Now that we're streamlining the forward and backward pass code, including validation, this is a good time to reintroduce dropout. Recall that dropout is a method to disable, or filter out, certain neurons in an attempt to regularize and improve our model's ability to generalize. If dropout is employed in our model, we want to make sure to leave it out when performing validation and inference (predictions); in our previous code, we left it out by not calling its `forward` method during the forward pass during validation. Here we have a common method for performing a forward pass for both training and validation, so we need a different approach for turning off dropout — to inform the layers if we are during the training and let them "decide" on calculation to include. The first thing we'll do is include a *`training`* boolean argument for the `forward` method in all the layer and activation classes, since we are calling them in a unified form:

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

When we're not training, we can set the output to the input directly in the `Layer_Dropout` class and return from the method without changing outputs:

```
        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return
```

When we are training, we will engage the dropout:

```
# Dropout
class Layer_Dropout:
    ...
    # Forward pass
    def forward(self, inputs, training):
        # Save input values
        self.input = inputs

        # If not in the training mode - return values
        if not training:
            self.output = inputs.copy()
            return

        # 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
```

Next, we modify the `forward` method of our `Model` class to add the *`training`* parameter and a call to the `forward` methods of the layers to take this parameter's value:

```
# Model class
class Model:
    ...
    # Performs forward pass
    def forward(self, X, training):

        # Call forward method on the input layer
        # this will set the output property that
        # the first layer in "prev" object is expecting
        self.input_layer.forward(X, training)

        # Call forward method of every object in a chain
        # Pass output of the previous object as a parameter
        for layer in self.layers:
            layer.forward(layer.prev.output, training)
```