NEURAL NETWORKS FROM SCRATCH IN PYTHON

By Harrison Kinsley & Daniel Kukiela

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

Do not resolve comments that are not yours.

Links to chapters:

Neural Networks from Scratch in Python

Harrison Kinsley & Daniel Kukieła

Copyright

License for Code

Readme

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

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

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

Chapter 14

L1 and L2 Regularization

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

Forward Pass

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

L1 weight regularization:

$$L_{1w} = \lambda \sum_m |w_m|$$

L1 bias regularization:

$$L_{1b} = \lambda \sum_n |b_n|$$

L2 weight regularization:

$$L_{2w} = \lambda \sum_m w_m^2$$

L2 bias regularization:

$$L_{2b} = \lambda \sum_n b_n^2$$

Overall loss:

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

Using code notation:

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

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

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

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

This method sets the lambda hyperparameters. Now we update our loss class to include the additional penalty if we choose to set the lambda hyperparameter for any of the regularizers in the layer's initialization. We will implement this code into the *Loss* class as it is common for the hidden layers. What's more, the regularization calculation is the same, regardless of the type of loss used. It's only a penalty that is summed with the data loss value resulting in a final, overall loss value. For this reason, we're going to add a new method to a general loss class, which is inherited by all of our specific loss functions (such as our existing `Loss_CategoricalCrossentropy`). For the code of this method, we'll create the layer's regularization loss variable. We'll add to it each of the atomic regularization losses if its corresponding lambda value is greater than *0*. To perform these calculations, we read the lambda hyperparameters, weights, and biases from the passed-in layer object. For our general loss class:

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

    # 0 by default
    regularization_loss = 0

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

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

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

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

    return regularization_loss
```

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

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

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

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

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

Backward pass

The derivative of L2 regularization is relatively simple:

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

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

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

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

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

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

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

When we combine that:

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

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

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

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

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

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

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

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

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


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

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

```
import numpy as np

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

dL1 = np.ones_like(weights)

dL1[weights < 0] = -1

print(dL1)


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

This returned an array of the same shape containing values of 1 and -1 — the partial gradient of the `np.abs` function (we still have to multiply it by the lambda hyperparameter). We can now take these and update the backward pass method for the dense layer object. For L1 regularization, we'll take the code above and multiply it by *λ* for weights and perform the same operation for biases. For L2 regularization, as discussed at the beginning of this chapter, all we need to do is take the weights/biases, multiply them by *2λ*, and add that product to the gradients:

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

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

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

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

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

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

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

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

Full code up to this point:

```
import numpy as np
import nnfs
from nnfs.datasets import spiral_data

nnfs.init()


# Dense layer
class Layer_Dense:

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

    # Forward pass
    def forward(self, inputs):
        # Remember input values
        self.inputs = inputs
        # Calculate output values from inputs, weights and biases
        self.output = np.dot(inputs, self.weights) + self.biases
    # Backward pass
    def backward(self, dvalues):
        # Gradients on parameters
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

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

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


# ReLU activation
class Activation_ReLU:

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

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

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

```
# Softmax activation
class Activation_Softmax:

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

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

        self.output = probabilities

    # Backward pass
    def backward(self, dvalues):

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

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


# SGD optimizer
class Optimizer_SGD:

    # Initialize optimizer - set settings,
    # learning rate of 1. is default for this optimizer
    def __init__(self, learning_rate=1., decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum
    # Call once before any parameter updates
    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                (1. / (1. + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):

        # If we use momentum
        if self.momentum:

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

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

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

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

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

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

```
# Adagrad optimizer
class Optimizer_Adagrad:

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

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

    # Update parameters
    def update_params(self, layer):

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

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

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

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

```python
# 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, y_pred, y_true):

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

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

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


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

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

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

        # Number of samples
        samples = len(y_pred)

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

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


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

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

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

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

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

# Create optimizer
optimizer = Optimizer_Adam(learning_rate=0.02, decay=5e-7)

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

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

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)
    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

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

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

    # Calculate overall loss
    loss = data_loss + regularization_loss

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

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

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

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

```
# Validate the model

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

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

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

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

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

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

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

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

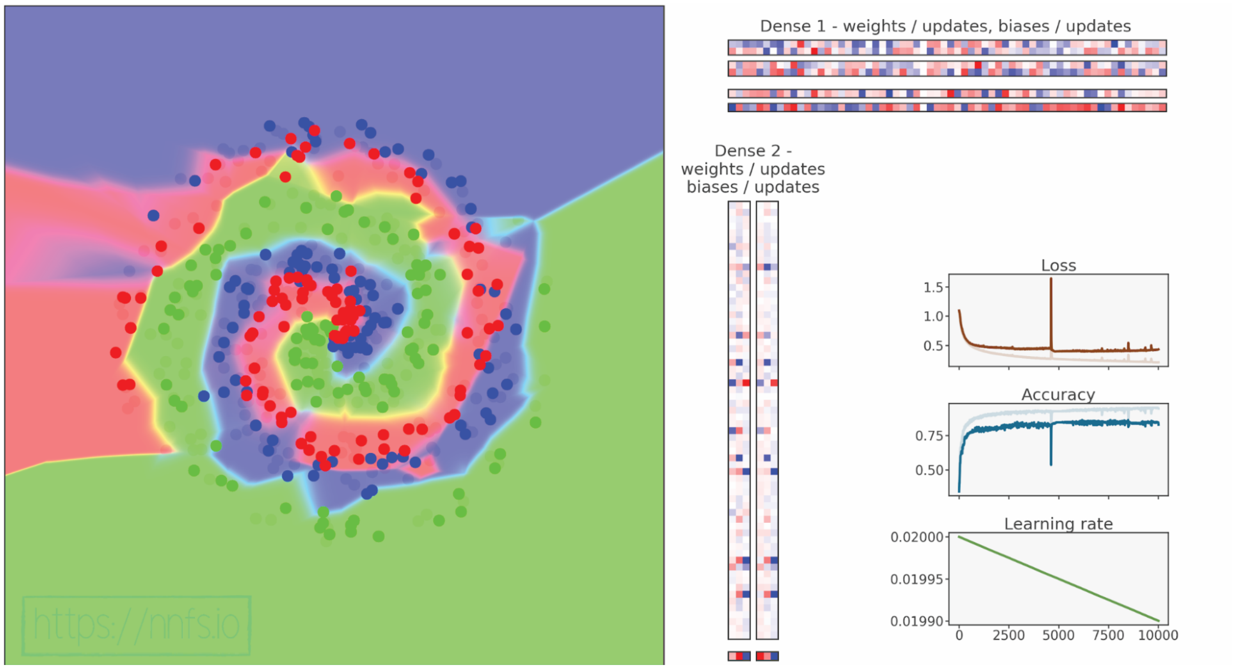

Fig 14.01: Training with regularization

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

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

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

And run the code again:

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

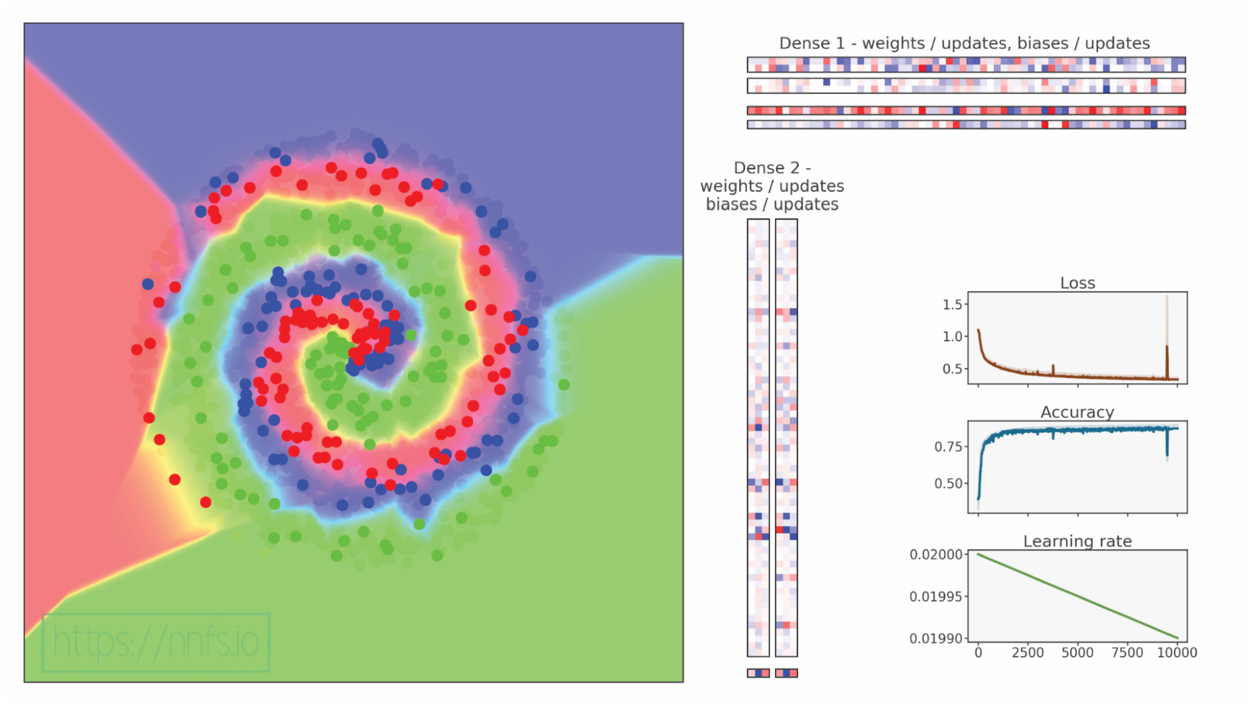

Fig 14.02: Training with regularization and more training data.

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

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

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

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

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

```
>>>
epoch: 10000, acc: 0.924, loss: 0.253 (data_loss: 0.210, reg_loss: 0.043),
lr: 0.019900507413187767
validation, acc: 0.887, loss: 0.331
```

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

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

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

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

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

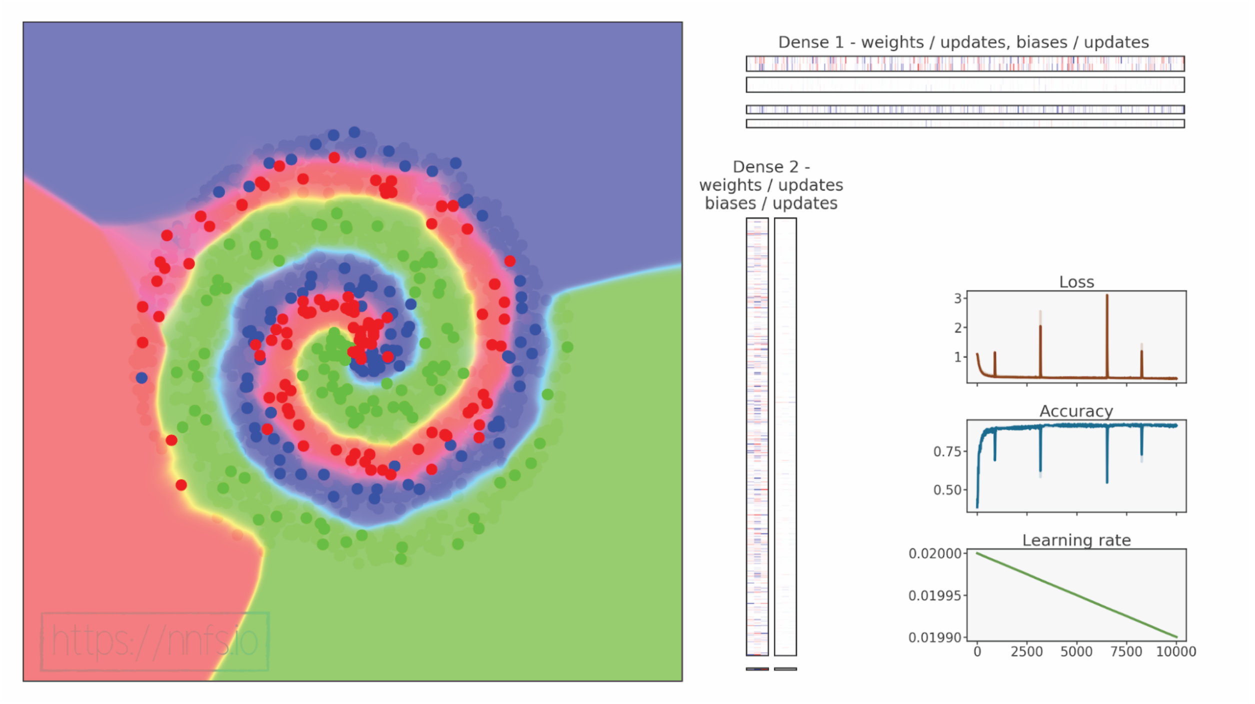

Fig 14.03: Training with regularization and more training data (tuned).

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

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

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

Chapter 15 - Dropout