notes_23

January 21, 2025

1 Previous Class Definitions

[1]:
```
# imports
import matplotlib.pyplot as plt
import numpy as np
import nnfs
from nnfs.datasets import spiral_data, vertical_data
nnfs.init()
```

[2]:
```
class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        # Initialize the weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)  # Normal␣
 ↪distribution of weights
        self.biases = np.zeros((1, n_neurons))

    def forward(self, inputs):
        # Calculate the output values from inputs, weights, and biases
        self.inputs = inputs
        self.output = np.dot(inputs, self.weights) + self.biases        #␣
 ↪Weights are already transposed

    def backward(self, dvalues):
        '''Calculated the gradient of the loss with respect to the weights and␣
 ↪biases of this layer.
        dvalues is equiavelent to a transposed dl_dZ. It is the gradient
        of the loss with respect to the outputs of this layer.'''
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=0)
        self.dinputs = np.dot(dvalues, self.weights.T)

class Activation_ReLU:
    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.maximum(0, inputs)

    def backward(self, dvalues):
```

```
        '''Calculated the gradient of the loss with respect to this layer's␣
 ↪activation function
        dvalues is equiavelent to a transposed dl_dZ. It is the gradient
        of the loss with respect to the outputs of this layer.'''
        self.dinputs = dvalues.copy()
        self.dinputs[self.inputs <= 0] = 0

class Activation_Softmax:
    def forward(self, inputs):
        # Get the unnormalized probabilities
        # Subtract max from the row to prevent larger numbers
        exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))

        # Normalize the probabilities with element wise division
        probabilities = exp_values / np.sum(exp_values, axis=1,keepdims=True)
        self.output = probabilities

# Base class for Loss functions
class Loss:
    '''Calculates the data and regularization losses given
    model output and ground truth values'''
    def calculate(self, output, y):
        sample_losses = self.forward(output, y)
        data_loss = np.average(sample_losses)
        return data_loss

class Loss_CategoricalCrossEntropy(Loss):
    def forward(self, y_pred, y_true):
        '''y_pred is the neural network output
        y_true is the ideal output of the neural network'''
        samples = len(y_pred)
        # Bound the predicted values
        y_pred_clipped = np.clip(y_pred, 1e-7, 1-1e-7)

        if len(y_true.shape) == 1:      # Categorically labeled
            correct_confidences = y_pred_clipped[range(samples), y_true]
        elif len(y_true.shape) == 2:    # One hot encoded
            correct_confidences = np.sum(y_pred_clipped*y_true, axis=1)

        # Calculate the losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    def backward(self, dvalues, y_true):
        samples = len(dvalues)

        # Number of lables in each sample
```

```
        labels = len(dvalues[0])

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

        # Calculate the gradient then normalize
        self.dinputs = -y_true / dvalues
        self.dinputs = self.dinputs / samples

class Activation_Softmax_Loss_CategoricalCrossentropy():
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossEntropy()

    def forward(self, inputs, y_true):
        self.activation.forward(inputs)
        self.output = self.activation.output
        return self.loss.calculate(self.output, y_true)

    def backward(self, dvalues, y_true):
        samples = len(dvalues)

        # if the samples 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 and normalize gradient
        self.dinputs[range(samples), y_true] -= 1
        self.dinputs = self.dinputs / samples

class Optimizer_SGD():
    def __init__(self, learning_rate=0.5):
        self.learning_rate = learning_rate

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

2 Previous Notes and Notation

The previous notation is clunky and long. From here forward, we will use the following notation for a layer with n inputs and i neurons. The neruon layer has is followed by an activation layer and then fed into a final value y with a computed loss l. There can be j batches of data.

$\vec{X}_j = \begin{bmatrix} x_{1j} & x_{2j} & \cdots & x_{nj} \end{bmatrix}$ -> Row vector for the layer inputs for the j batch of data.

$\overline{\overline{W}} = \begin{bmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \vdots \\ \vec{w}_i \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{in} \end{bmatrix}$ -> Matrix of weight values. Each row is a neuron's weights and each column is the weights for a given input.

$\vec{B} = \begin{bmatrix} b_1 & b_2 & \cdots & b_i \end{bmatrix}$ -> Row vector for the neuron biases

$\vec{Z}_j = \begin{bmatrix} z_{1j} & z_{2j} & \cdots & z_{ij} \end{bmatrix}$ -> Row vector for the neuron outputs for the j batch of data.

$\vec{A}_j = \begin{bmatrix} a_{1j} & a_{2j} & \cdots & a_{ij} \end{bmatrix}$ -> Row vector for the activation later outputs for the j batch of data.

y_j -> Final layer output for the j batch of data if the layer is the final layer (could be summation, probability, etc).

l_j -> Loss for the j batch of data.

The j is often dropped because we typically only need to think with 1 set of input data.

2.1 Gradient Descent Using New Notation

We will look at the weight that the i neuron applies for the n input.

$\frac{\delta l}{\delta w_{in}} = \frac{\delta l}{\delta y} \frac{\delta y}{\delta a_i} \frac{\delta a_i}{\delta z_i} \frac{\delta z_i}{\delta w_{in}}$

Similarly, for the bias of the i neuron, there is

$\frac{\delta l}{\delta b_i} = \frac{\delta l}{\delta y} \frac{\delta y}{\delta a_i} \frac{\delta a_i}{\delta z_i} \frac{\delta z_i}{\delta b_i}$

For the system we are using, where $l = (y - 0)^2$ and the activation layer is ReLU, we have

$\frac{\delta l}{\delta y} = 2y$

$\frac{\delta y}{\delta a_i} = 1$

$\frac{\delta a_i}{\delta z_i} = 1$ if $z_i > 0$ else 0

$\frac{\delta z_i}{\delta w_{in}} = x_n$

$\frac{\delta z_i}{\delta b_i} = 1$

2.2 Matrix Representation of Gradient Descent

We can simplify by seeing that $\frac{\delta l}{\delta y} \frac{\delta y}{\delta a_i} \frac{\delta a_i}{\delta z_i} = \frac{\delta l}{\delta z_i}$ is a common term.

We take $\frac{\delta l}{\delta z_i}$ and turn it into a 1 x i vector that such that

$\frac{\delta l}{\delta \vec{Z}} = \begin{bmatrix} \frac{\delta l}{\delta z_1} & \frac{\delta l}{\delta z_2} & \cdots & \frac{\delta l}{\delta z_i} \end{bmatrix}$

We than can get that the gradient matrix for all weights is a i x n matrix given by

$$\frac{\delta l}{\delta \overline{\overline{W}}} = \begin{bmatrix} \frac{\delta l}{\delta w_{11}} & \frac{\delta l}{\delta w_{12}} & \cdots & \frac{\delta l}{\delta w_{1n}} \\ \frac{\delta l}{\delta w_{21}} & w\frac{\delta l}{\delta w_{22}} & \cdots & \frac{\delta l}{\delta w_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta l}{\delta w_{i1}} & \frac{\delta l}{\delta w_{i2}} & \cdots & \frac{\delta l}{\delta w_{in}} \end{bmatrix} = \begin{bmatrix} \frac{\delta l}{\delta z_1} \\ \frac{\delta l}{\delta z_2} \\ \vdots \\ \frac{\delta l}{\delta z_n} \end{bmatrix} \begin{bmatrix} \frac{\delta z_1}{\delta w_{i1}} & \frac{\delta z_1}{\delta w_{i1}} & \cdots & \frac{\delta z_1}{\delta w_{in}} \end{bmatrix} = \begin{bmatrix} \frac{\delta l}{\delta z_1} \\ \frac{\delta l}{\delta z_2} \\ \vdots \\ \frac{\delta l}{\delta z_n} \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

Similarly, the gradient vector for the biases is given by $\frac{\delta l}{\delta \vec{B}} = \frac{\delta l}{\delta \vec{Z}} \frac{\delta \vec{Z}}{\delta \vec{B}} = \vec{1} \begin{bmatrix} \frac{\delta l}{\delta z_1} & \frac{\delta l}{\delta z_2} & \cdots & \frac{\delta l}{\delta z_i} \end{bmatrix}$

2.3 Gradients of the Loss with Respect to Inputs

When chaining multiple layers together, we will need the partial derivatives of the loss with respect to the next layers input (ie, the output of the current layer). This involves extra summation because the output of 1 layer is fed into every neuron of the next layer, so the total loss must be found.

The gradient of the loss with respect to the n input fed into i neurons is

$$\frac{\delta l}{\delta x_n} = \frac{\delta l}{\delta z_1}\frac{\delta z_1}{\delta x_n} + \frac{\delta l}{\delta z_2}\frac{\delta z_2}{\delta x_n} + ... + \frac{\delta l}{\delta z_i}\frac{\delta z_i}{\delta x_n}$$

Noting that $\frac{\delta z_i}{\delta x_n} = w_{in}$ allows us to have

$$\frac{\delta l}{\delta \vec{X}} = \begin{bmatrix} \frac{\delta l}{\delta x_1} & \frac{\delta l}{\delta x_2} & \cdots & \frac{\delta l}{\delta x_n} \end{bmatrix} = \begin{bmatrix} \frac{\delta l}{\delta z_1} & \frac{\delta l}{\delta z_2} & \cdots & \frac{\delta l}{\delta z_n} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{in} \end{bmatrix}$$

2.4 Note With Layer_Dense class

The Layer_Dense class has the weights stored in the transposed fashion for forward propagation. Therefore, the weight matrix must be transposed for the backpropagation.

3 Learning Rate Decay

Start with a larger learning rate and gradually decrease it.

$\alpha = \frac{\alpha_0}{1 + decay * t}$

[3]:
```
class Optimizer_SGD():
    def __init__(self, learning_rate=0.5, decay=0.0):
        self.initial_rate = learning_rate
        self.current_learning_rate = self.initial_rate
        self.decay = decay
        self.iterations = 0

    def pre_update_params(self):
        # Update the current_learning_rate before updating params
        if self.decay:
            self.current_learning_rate = self.initial_rate / (1 + self.decay *␣
 ↪self.iterations)

    def update_params(self, layer):
        layer.weights += -self.current_learning_rate * layer.dweights
```

```
        layer.biases += -self.current_learning_rate * layer.dbiases

    def post_update_params(self):
        # Update the self.iterations for use with decay
        self.iterations += 1
```

3.1 Testing the Learning Rate Decay

[4]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

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

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

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

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

# Create optimizer
optimizer = Optimizer_SGD(learning_rate=1.0, decay=1e-3)

# Train in loop
for epoch in range(10001):
    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

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

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

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

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

```
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions == y)

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

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

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

```
epoch: 0, acc: 0.360, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.400, loss: 1.088, lr: 0.9099181073703367
epoch: 200, acc: 0.423, loss: 1.078, lr: 0.8340283569641367
epoch: 300, acc: 0.423, loss: 1.076, lr: 0.7698229407236336
epoch: 400, acc: 0.420, loss: 1.076, lr: 0.7147962830593281
epoch: 500, acc: 0.403, loss: 1.074, lr: 0.66711140760507
epoch: 600, acc: 0.403, loss: 1.072, lr: 0.6253908692933083
epoch: 700, acc: 0.410, loss: 1.070, lr: 0.5885815185403178
epoch: 800, acc: 0.413, loss: 1.068, lr: 0.5558643690939411
epoch: 900, acc: 0.427, loss: 1.066, lr: 0.526592943654555
epoch: 1000, acc: 0.440, loss: 1.063, lr: 0.5002501250625312
epoch: 1100, acc: 0.437, loss: 1.059, lr: 0.4764173415912339
epoch: 1200, acc: 0.447, loss: 1.056, lr: 0.45475216007276037
epoch: 1300, acc: 0.440, loss: 1.052, lr: 0.43497172683775553
epoch: 1400, acc: 0.427, loss: 1.048, lr: 0.4168403501458941
epoch: 1500, acc: 0.417, loss: 1.041, lr: 0.4001600640256102
epoch: 1600, acc: 0.427, loss: 1.032, lr: 0.3847633705271258
epoch: 1700, acc: 0.440, loss: 1.025, lr: 0.3705075954057058
epoch: 1800, acc: 0.473, loss: 1.017, lr: 0.35727045373347627
epoch: 1900, acc: 0.460, loss: 1.008, lr: 0.3449465332873405
epoch: 2000, acc: 0.463, loss: 1.000, lr: 0.33344448149383127
epoch: 2100, acc: 0.493, loss: 1.006, lr: 0.32268473701193934
epoch: 2200, acc: 0.463, loss: 1.014, lr: 0.31259768677711786
epoch: 2300, acc: 0.480, loss: 1.015, lr: 0.3031221582297666
epoch: 2400, acc: 0.497, loss: 1.012, lr: 0.29420417769932333
```

```
epoch: 2500, acc: 0.493, loss: 1.010, lr: 0.2857959416976279
epoch: 2600, acc: 0.503, loss: 1.006, lr: 0.2778549597110308
epoch: 2700, acc: 0.487, loss: 1.003, lr: 0.2703433360367667
epoch: 2800, acc: 0.483, loss: 0.998, lr: 0.26322716504343247
epoch: 2900, acc: 0.483, loss: 0.996, lr: 0.25647601949217746
epoch: 3000, acc: 0.490, loss: 0.991, lr: 0.25006251562890724
epoch: 3100, acc: 0.490, loss: 0.988, lr: 0.2439619419370578
epoch: 3200, acc: 0.490, loss: 0.983, lr: 0.23815194093831865
epoch: 3300, acc: 0.490, loss: 0.980, lr: 0.23261223540358225
epoch: 3400, acc: 0.493, loss: 0.974, lr: 0.22732439190725165
epoch: 3500, acc: 0.510, loss: 0.969, lr: 0.22227161591464767
epoch: 3600, acc: 0.517, loss: 0.966, lr: 0.21743857360295715
epoch: 3700, acc: 0.520, loss: 0.961, lr: 0.21281123643328367
epoch: 3800, acc: 0.520, loss: 0.957, lr: 0.20837674515524068
epoch: 3900, acc: 0.520, loss: 0.951, lr: 0.20412329046744235
epoch: 4000, acc: 0.523, loss: 0.947, lr: 0.2000400080016003
epoch: 4100, acc: 0.537, loss: 0.942, lr: 0.19611688566385566
epoch: 4200, acc: 0.543, loss: 0.938, lr: 0.19234468166955185
epoch: 4300, acc: 0.543, loss: 0.934, lr: 0.18871485185884126
epoch: 4400, acc: 0.553, loss: 0.929, lr: 0.18521948508983144
epoch: 4500, acc: 0.553, loss: 0.924, lr: 0.18185124568103292
epoch: 4600, acc: 0.557, loss: 0.920, lr: 0.1786033220217896
epoch: 4700, acc: 0.567, loss: 0.916, lr: 0.1754693805930865
epoch: 4800, acc: 0.573, loss: 0.911, lr: 0.17244352474564578
epoch: 4900, acc: 0.577, loss: 0.907, lr: 0.16952025767079165
epoch: 5000, acc: 0.583, loss: 0.903, lr: 0.16669444907484582
epoch: 5100, acc: 0.580, loss: 0.898, lr: 0.16396130513198884
epoch: 5200, acc: 0.587, loss: 0.894, lr: 0.16131634134537828
epoch: 5300, acc: 0.590, loss: 0.890, lr: 0.15875535799333226
epoch: 5400, acc: 0.593, loss: 0.886, lr: 0.1562744178777934
epoch: 5500, acc: 0.600, loss: 0.882, lr: 0.15386982612709646
epoch: 5600, acc: 0.603, loss: 0.877, lr: 0.15153811183512653
epoch: 5700, acc: 0.617, loss: 0.873, lr: 0.14927601134497687
epoch: 5800, acc: 0.620, loss: 0.869, lr: 0.14708045300779526
epoch: 5900, acc: 0.627, loss: 0.865, lr: 0.14494854326714016
epoch: 6000, acc: 0.637, loss: 0.860, lr: 0.1428775539362766
epoch: 6100, acc: 0.640, loss: 0.857, lr: 0.1408649105507818
epoch: 6200, acc: 0.637, loss: 0.852, lr: 0.13890818169190167
epoch: 6300, acc: 0.640, loss: 0.848, lr: 0.13700506918755992
epoch: 6400, acc: 0.647, loss: 0.844, lr: 0.13515339910798757
epoch: 6500, acc: 0.653, loss: 0.840, lr: 0.13335111348179757
epoch: 6600, acc: 0.643, loss: 0.836, lr: 0.13159626266614027
epoch: 6700, acc: 0.647, loss: 0.833, lr: 0.12988699831146902
epoch: 6800, acc: 0.653, loss: 0.830, lr: 0.12822156686754713
epoch: 6900, acc: 0.657, loss: 0.827, lr: 0.126598303582732
epoch: 7000, acc: 0.657, loss: 0.823, lr: 0.12501562695336915
epoch: 7100, acc: 0.647, loss: 0.821, lr: 0.12347203358439313
epoch: 7200, acc: 0.647, loss: 0.818, lr: 0.12196609342602757
```

```
epoch: 7300, acc: 0.660, loss: 0.816, lr: 0.12049644535486204
epoch: 7400, acc: 0.663, loss: 0.813, lr: 0.11906179307060363
epoch: 7500, acc: 0.677, loss: 0.810, lr: 0.11766090128250381
epoch: 7600, acc: 0.670, loss: 0.806, lr: 0.11629259216187929
epoch: 7700, acc: 0.670, loss: 0.804, lr: 0.11495574203931487
epoch: 7800, acc: 0.670, loss: 0.803, lr: 0.11364927832708263
epoch: 7900, acc: 0.693, loss: 0.804, lr: 0.11237217664906168
epoch: 8000, acc: 0.687, loss: 0.803, lr: 0.11112345816201799
epoch: 8100, acc: 0.687, loss: 0.801, lr: 0.10990218705352237
epoch: 8200, acc: 0.680, loss: 0.799, lr: 0.10870746820306555
epoch: 8300, acc: 0.677, loss: 0.796, lr: 0.1075384449940854
epoch: 8400, acc: 0.670, loss: 0.793, lr: 0.10639429726566654
epoch: 8500, acc: 0.667, loss: 0.791, lr: 0.10527423939362038
epoch: 8600, acc: 0.667, loss: 0.789, lr: 0.10417751849150952
epoch: 8700, acc: 0.670, loss: 0.787, lr: 0.10310341272296113
epoch: 8800, acc: 0.667, loss: 0.785, lr: 0.1020512297173181
epoch: 8900, acc: 0.667, loss: 0.783, lr: 0.10102030508132134
epoch: 9000, acc: 0.670, loss: 0.781, lr: 0.1000100010001
epoch: 9100, acc: 0.670, loss: 0.778, lr: 0.09901970492127933
epoch: 9200, acc: 0.663, loss: 0.776, lr: 0.09804882831650162
epoch: 9300, acc: 0.663, loss: 0.774, lr: 0.09709680551509856
epoch: 9400, acc: 0.673, loss: 0.772, lr: 0.09616309260505818
epoch: 9500, acc: 0.680, loss: 0.769, lr: 0.09524716639679968
epoch: 9600, acc: 0.680, loss: 0.768, lr: 0.09434852344560807
epoch: 9700, acc: 0.680, loss: 0.766, lr: 0.09346667912889055
epoch: 9800, acc: 0.683, loss: 0.765, lr: 0.09260116677470137
epoch: 9900, acc: 0.680, loss: 0.763, lr: 0.09175153683824203
epoch: 10000, acc: 0.680, loss: 0.761, lr: 0.09091735612328393
```

4 Momentum in Training Neural Networks

Using the past information on how changing the weights and biases changes the loss, we can use the large dimensional vectors to try and cancel out the overshoot.

The new weight update = some factor * last weight update + learning rate * current gradient

[5]:
```python
class Optimizer_SGD():
    def __init__(self, learning_rate=0.5, decay=0.0, momentum=0.0):
        self.initial_rate = learning_rate
        self.current_learning_rate = self.initial_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    def pre_update_params(self):
        # Update the current_learning_rate before updating params
        if self.decay:
```

```
            self.current_learning_rate = self.initial_rate / (1 + self.decay *␣
↪self.iterations)

    def update_params(self, layer):
        if self.momentum:
            # For each layer, we need to use its last momentums

            # First check if the layer has a last momentum stored
            if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                layer.bias_momentums = np.zeros_like(layer.biases)

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

            bias_updates = self.momentum * layer.bias_momentums - \
                           self.current_learning_rate * layer.dbiases
            layer.bias_momentums = bias_updates

        # Not using momentum
        else:
            weight_updates = -self.current_learning_rate * layer.dweights
            bias_updates = -self.current_learning_rate * layer.dbiases

        layer.weights += weight_updates
        layer.biases += bias_updates

    def post_update_params(self):
        # Update the self.iterations for use with decay
        self.iterations += 1
```

4.1 Testing the Gradient Optimizer with Momentum

[6]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

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

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

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

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

# Create optimizer
optimizer = Optimizer_SGD(learning_rate=1.0, decay=1e-3, momentum=0.9)

# Train in loop
for epoch in range(10001):
    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

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

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

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

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

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

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

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

```
epoch: 0, acc: 0.343, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.443, loss: 1.047, lr: 0.9099181073703367
epoch: 200, acc: 0.397, loss: 1.013, lr: 0.8340283569641367
epoch: 300, acc: 0.540, loss: 0.905, lr: 0.7698229407236336
epoch: 400, acc: 0.553, loss: 0.844, lr: 0.7147962830593281
epoch: 500, acc: 0.560, loss: 0.813, lr: 0.66711140760507
epoch: 600, acc: 0.567, loss: 0.792, lr: 0.6253908692933083
epoch: 700, acc: 0.570, loss: 0.776, lr: 0.5885815185403178
epoch: 800, acc: 0.570, loss: 0.768, lr: 0.5558643690939411
epoch: 900, acc: 0.573, loss: 0.762, lr: 0.526592943654555
epoch: 1000, acc: 0.573, loss: 0.757, lr: 0.5002501250625312
epoch: 1100, acc: 0.573, loss: 0.753, lr: 0.4764173415912339
epoch: 1200, acc: 0.573, loss: 0.751, lr: 0.45475216007276037
epoch: 1300, acc: 0.577, loss: 0.749, lr: 0.43497172683775553
epoch: 1400, acc: 0.573, loss: 0.747, lr: 0.4168403501458941
epoch: 1500, acc: 0.577, loss: 0.745, lr: 0.4001600640256102
epoch: 1600, acc: 0.577, loss: 0.744, lr: 0.3847633705271258
epoch: 1700, acc: 0.580, loss: 0.743, lr: 0.3705075954057058
epoch: 1800, acc: 0.583, loss: 0.742, lr: 0.35727045373347627
epoch: 1900, acc: 0.583, loss: 0.741, lr: 0.3449465332873405
epoch: 2000, acc: 0.580, loss: 0.740, lr: 0.33344448149383127
epoch: 2100, acc: 0.583, loss: 0.740, lr: 0.32268473701193934
epoch: 2200, acc: 0.583, loss: 0.739, lr: 0.31259768677711786
epoch: 2300, acc: 0.583, loss: 0.738, lr: 0.3031221582297666
epoch: 2400, acc: 0.583, loss: 0.738, lr: 0.29420417769932333
epoch: 2500, acc: 0.583, loss: 0.737, lr: 0.2857959416976279
epoch: 2600, acc: 0.583, loss: 0.737, lr: 0.2778549597110308
epoch: 2700, acc: 0.583, loss: 0.737, lr: 0.2703433360367667
epoch: 2800, acc: 0.583, loss: 0.736, lr: 0.26322716504343247
epoch: 2900, acc: 0.583, loss: 0.736, lr: 0.25647601949217746
epoch: 3000, acc: 0.583, loss: 0.736, lr: 0.25006251562890724
epoch: 3100, acc: 0.583, loss: 0.735, lr: 0.2439619419370578
epoch: 3200, acc: 0.583, loss: 0.735, lr: 0.23815194093831865
epoch: 3300, acc: 0.583, loss: 0.735, lr: 0.23261223540358225
epoch: 3400, acc: 0.583, loss: 0.735, lr: 0.22732439190725165
epoch: 3500, acc: 0.583, loss: 0.734, lr: 0.2227161591464767
epoch: 3600, acc: 0.583, loss: 0.734, lr: 0.21743857360295715
epoch: 3700, acc: 0.580, loss: 0.734, lr: 0.21281123643328367
epoch: 3800, acc: 0.583, loss: 0.734, lr: 0.20837674515524068
epoch: 3900, acc: 0.583, loss: 0.734, lr: 0.20412329046744235
epoch: 4000, acc: 0.583, loss: 0.734, lr: 0.2000400080016003
epoch: 4100, acc: 0.583, loss: 0.733, lr: 0.19611688566385566
epoch: 4200, acc: 0.583, loss: 0.733, lr: 0.19234468166955185
epoch: 4300, acc: 0.580, loss: 0.733, lr: 0.18871485185884126
epoch: 4400, acc: 0.580, loss: 0.733, lr: 0.18521948508983144
epoch: 4500, acc: 0.583, loss: 0.733, lr: 0.18185124568103292
epoch: 4600, acc: 0.583, loss: 0.733, lr: 0.1786033220217896
epoch: 4700, acc: 0.583, loss: 0.733, lr: 0.1754693805930865
```

```
epoch: 4800, acc: 0.583, loss: 0.732, lr: 0.17244352474564578
epoch: 4900, acc: 0.580, loss: 0.732, lr: 0.16952025767079165
epoch: 5000, acc: 0.580, loss: 0.732, lr: 0.16669444907484582
epoch: 5100, acc: 0.580, loss: 0.732, lr: 0.16396130513198884
epoch: 5200, acc: 0.583, loss: 0.732, lr: 0.16131634134537828
epoch: 5300, acc: 0.583, loss: 0.731, lr: 0.15875535799333226
epoch: 5400, acc: 0.583, loss: 0.731, lr: 0.1562744178777934
epoch: 5500, acc: 0.583, loss: 0.731, lr: 0.15386982612709646
epoch: 5600, acc: 0.583, loss: 0.731, lr: 0.15153811183512653
epoch: 5700, acc: 0.583, loss: 0.731, lr: 0.14927601134497687
epoch: 5800, acc: 0.583, loss: 0.731, lr: 0.14708045300779526
epoch: 5900, acc: 0.580, loss: 0.730, lr: 0.14494854326714016
epoch: 6000, acc: 0.580, loss: 0.730, lr: 0.1428775539362766
epoch: 6100, acc: 0.583, loss: 0.730, lr: 0.1408649105507818
epoch: 6200, acc: 0.580, loss: 0.730, lr: 0.13890818169190167
epoch: 6300, acc: 0.583, loss: 0.730, lr: 0.13700506918755992
epoch: 6400, acc: 0.580, loss: 0.730, lr: 0.13515339910798757
epoch: 6500, acc: 0.583, loss: 0.729, lr: 0.13335111348179757
epoch: 6600, acc: 0.580, loss: 0.729, lr: 0.13159626266614027
epoch: 6700, acc: 0.583, loss: 0.729, lr: 0.12988699831146902
epoch: 6800, acc: 0.583, loss: 0.729, lr: 0.12822156686754713
epoch: 6900, acc: 0.580, loss: 0.729, lr: 0.126598303582732
epoch: 7000, acc: 0.580, loss: 0.729, lr: 0.12501562695336915
epoch: 7100, acc: 0.583, loss: 0.728, lr: 0.12347203358439313
epoch: 7200, acc: 0.580, loss: 0.728, lr: 0.12196609342602757
epoch: 7300, acc: 0.580, loss: 0.728, lr: 0.12049644535486204
epoch: 7400, acc: 0.583, loss: 0.728, lr: 0.11906179307060363
epoch: 7500, acc: 0.580, loss: 0.728, lr: 0.11766090128250381
epoch: 7600, acc: 0.580, loss: 0.728, lr: 0.11629259216187929
epoch: 7700, acc: 0.580, loss: 0.727, lr: 0.11495574203931487
epoch: 7800, acc: 0.580, loss: 0.727, lr: 0.11364927832708263
epoch: 7900, acc: 0.583, loss: 0.727, lr: 0.11237217664906168
epoch: 8000, acc: 0.580, loss: 0.727, lr: 0.11112345816201799
epoch: 8100, acc: 0.580, loss: 0.727, lr: 0.10990218705352237
epoch: 8200, acc: 0.580, loss: 0.727, lr: 0.10870746820306555
epoch: 8300, acc: 0.580, loss: 0.727, lr: 0.1075384449940854
epoch: 8400, acc: 0.580, loss: 0.726, lr: 0.10639429726566654
epoch: 8500, acc: 0.583, loss: 0.726, lr: 0.10527423939362038
epoch: 8600, acc: 0.580, loss: 0.726, lr: 0.10417751849150952
epoch: 8700, acc: 0.580, loss: 0.726, lr: 0.10310341272296113
epoch: 8800, acc: 0.580, loss: 0.726, lr: 0.1020512297173181
epoch: 8900, acc: 0.580, loss: 0.726, lr: 0.10102030508132134
epoch: 9000, acc: 0.580, loss: 0.726, lr: 0.1000100010001
epoch: 9100, acc: 0.580, loss: 0.726, lr: 0.09901970492127933
epoch: 9200, acc: 0.580, loss: 0.725, lr: 0.09804882831650162
epoch: 9300, acc: 0.580, loss: 0.725, lr: 0.09709680551509856
epoch: 9400, acc: 0.580, loss: 0.725, lr: 0.09616309260505818
epoch: 9500, acc: 0.580, loss: 0.725, lr: 0.09524716639679968
```

```
epoch: 9600, acc: 0.583, loss: 0.725, lr: 0.09434852344560807
epoch: 9700, acc: 0.583, loss: 0.725, lr: 0.09346667912889055
epoch: 9800, acc: 0.587, loss: 0.725, lr: 0.09260116677470137
epoch: 9900, acc: 0.587, loss: 0.725, lr: 0.09175153683824203
epoch: 10000, acc: 0.587, loss: 0.724, lr: 0.09091735612328393
```