notes_25

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, 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
```

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 AdaGrad Optimizer

Different weights should have different learning rates. If one weight affects the loss much more strongly than the other, then consider using smaller learning rates with it. We can do this by maintaining a "cache" of the last gradients and normalizing based on this.

A downside is that as the cache keeps accumulating, some neurons will have such a small learning rate that the neuron basically becomes fixed.

[6]:
```
class Optimizer_Adagrad():
    def __init__(self, learning_rate=0.5, decay=0.0, epsilon=1e-7):
        self.initial_learning_rate = learning_rate
        self.current_learning_rate = self.initial_learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.initial_learning_rate / (1 + self.
 ↪decay * self.iterations)

    def update_params(self, layer):
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        layer.weight_cache += layer.dweights**2
        layer.bias_cache += layer.dbiases**2

        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)

    def post_update_params(self):
```

```
        self.iterations += 1
```

3.1 Testing the AdagGrad Optimizer

[12]:
```
# 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_Adagrad(learning_rate=1.0, decay=1e-4)

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

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

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

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

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

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

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

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

```
epoch: 0, acc: 0.353, loss: 1.099, lr: 1.0
epoch: 100, acc: 0.497, loss: 0.986, lr: 0.9901970492127933
epoch: 200, acc: 0.527, loss: 0.936, lr: 0.9804882831650161
epoch: 300, acc: 0.513, loss: 0.918, lr: 0.9709680551509855
epoch: 400, acc: 0.580, loss: 0.904, lr: 0.9616309260505818
epoch: 500, acc: 0.550, loss: 0.910, lr: 0.9524716639679969
epoch: 600, acc: 0.563, loss: 0.860, lr: 0.9434852344560807
epoch: 700, acc: 0.600, loss: 0.838, lr: 0.9346667912889054
epoch: 800, acc: 0.603, loss: 0.815, lr: 0.9260116677470135
epoch: 900, acc: 0.643, loss: 0.787, lr: 0.9175153683824203
epoch: 1000, acc: 0.617, loss: 0.810, lr: 0.9091735612328392
epoch: 1100, acc: 0.637, loss: 0.750, lr: 0.9009820704567978
epoch: 1200, acc: 0.677, loss: 0.744, lr: 0.892936869363336
epoch: 1300, acc: 0.670, loss: 0.722, lr: 0.8850340738118416
epoch: 1400, acc: 0.693, loss: 0.704, lr: 0.8772699359592947
epoch: 1500, acc: 0.680, loss: 0.708, lr: 0.8696408383337683
epoch: 1600, acc: 0.697, loss: 0.664, lr: 0.8621432882145013
epoch: 1700, acc: 0.650, loss: 0.699, lr: 0.8547739123001966
epoch: 1800, acc: 0.707, loss: 0.646, lr: 0.8475294516484448
epoch: 1900, acc: 0.693, loss: 0.633, lr: 0.8404067568703253
epoch: 2000, acc: 0.713, loss: 0.620, lr: 0.8334027835652972
epoch: 2100, acc: 0.710, loss: 0.610, lr: 0.8265145879824779
epoch: 2200, acc: 0.710, loss: 0.599, lr: 0.8197393228953193
epoch: 2300, acc: 0.713, loss: 0.588, lr: 0.8130742336775347
epoch: 2400, acc: 0.733, loss: 0.576, lr: 0.8065166545689169
epoch: 2500, acc: 0.763, loss: 0.579, lr: 0.8000640051204096
epoch: 2600, acc: 0.800, loss: 0.558, lr: 0.7937137868084768
epoch: 2700, acc: 0.803, loss: 0.551, lr: 0.7874635798094338
epoch: 2800, acc: 0.800, loss: 0.545, lr: 0.7813110399249941
```

```
epoch: 2900, acc: 0.807, loss: 0.541, lr: 0.7752538956508256
epoch: 3000, acc: 0.757, loss: 0.538, lr: 0.7692899453804138
epoch: 3100, acc: 0.757, loss: 0.532, lr: 0.7634170547370028
epoch: 3200, acc: 0.740, loss: 0.532, lr: 0.7576331540268202
epoch: 3300, acc: 0.763, loss: 0.514, lr: 0.7519362358072035
epoch: 3400, acc: 0.770, loss: 0.512, lr: 0.7463243525636241
epoch: 3500, acc: 0.757, loss: 0.507, lr: 0.7407956144899621
epoch: 3600, acc: 0.780, loss: 0.496, lr: 0.735348187366718
epoch: 3700, acc: 0.787, loss: 0.497, lr: 0.7299802905321557
epoch: 3800, acc: 0.767, loss: 0.491, lr: 0.7246901949416624
epoch: 3900, acc: 0.783, loss: 0.483, lr: 0.7194762213108857
epoch: 4000, acc: 0.790, loss: 0.484, lr: 0.7143367383384527
epoch: 4100, acc: 0.783, loss: 0.477, lr: 0.7092701610043266
epoch: 4200, acc: 0.780, loss: 0.487, lr: 0.7042749489400663
epoch: 4300, acc: 0.787, loss: 0.467, lr: 0.6993496048674733
epoch: 4400, acc: 0.793, loss: 0.465, lr: 0.6944926731022988
epoch: 4500, acc: 0.787, loss: 0.460, lr: 0.6897027381198704
epoch: 4600, acc: 0.777, loss: 0.477, lr: 0.6849784231796698
epoch: 4700, acc: 0.783, loss: 0.454, lr: 0.6803183890060548
epoch: 4800, acc: 0.800, loss: 0.448, lr: 0.6757213325224677
epoch: 4900, acc: 0.800, loss: 0.441, lr: 0.6711859856366199
epoch: 5000, acc: 0.797, loss: 0.437, lr: 0.6667111140742716
epoch: 5100, acc: 0.800, loss: 0.433, lr: 0.6622955162593549
epoch: 5200, acc: 0.813, loss: 0.429, lr: 0.6579380222383051
epoch: 5300, acc: 0.810, loss: 0.426, lr: 0.6536374926465782
epoch: 5400, acc: 0.813, loss: 0.423, lr: 0.649392817715436
epoch: 5500, acc: 0.823, loss: 0.420, lr: 0.6452029163171817
epoch: 5600, acc: 0.820, loss: 0.416, lr: 0.6410667350471184
epoch: 5700, acc: 0.820, loss: 0.413, lr: 0.6369832473405949
epoch: 5800, acc: 0.820, loss: 0.410, lr: 0.6329514526235838
epoch: 5900, acc: 0.820, loss: 0.407, lr: 0.6289703754953141
epoch: 6000, acc: 0.823, loss: 0.404, lr: 0.6250390649415589
epoch: 6100, acc: 0.823, loss: 0.401, lr: 0.6211565935772407
epoch: 6200, acc: 0.827, loss: 0.398, lr: 0.6173220569170937
epoch: 6300, acc: 0.833, loss: 0.395, lr: 0.6135345726731701
epoch: 6400, acc: 0.830, loss: 0.390, lr: 0.6097932800780536
epoch: 6500, acc: 0.827, loss: 0.387, lr: 0.6060973392326807
epoch: 6600, acc: 0.827, loss: 0.384, lr: 0.6024459304777396
epoch: 6700, acc: 0.830, loss: 0.381, lr: 0.5988382537876519
epoch: 6800, acc: 0.830, loss: 0.378, lr: 0.5952735281862016
epoch: 6900, acc: 0.830, loss: 0.375, lr: 0.5917509911829102
epoch: 7000, acc: 0.833, loss: 0.373, lr: 0.5882698982293076
epoch: 7100, acc: 0.837, loss: 0.370, lr: 0.5848295221942803
epoch: 7200, acc: 0.833, loss: 0.368, lr: 0.5814291528577243
epoch: 7300, acc: 0.833, loss: 0.366, lr: 0.5780680964217585
epoch: 7400, acc: 0.837, loss: 0.364, lr: 0.5747456750387954
epoch: 7500, acc: 0.833, loss: 0.362, lr: 0.5714612263557918
epoch: 7600, acc: 0.833, loss: 0.360, lr: 0.5682141030740383
```

```
epoch: 7700, acc: 0.837, loss: 0.358, lr: 0.5650036725238714
epoch: 7800, acc: 0.840, loss: 0.357, lr: 0.5618293162537221
epoch: 7900, acc: 0.840, loss: 0.355, lr: 0.5586904296329404
epoch: 8000, acc: 0.840, loss: 0.353, lr: 0.5555864214678593
epoch: 8100, acc: 0.843, loss: 0.351, lr: 0.5525167136305873
epoch: 8200, acc: 0.843, loss: 0.350, lr: 0.5494807407000385
epoch: 8300, acc: 0.843, loss: 0.348, lr: 0.5464779496147331
epoch: 8400, acc: 0.843, loss: 0.346, lr: 0.5435077993369205
epoch: 8500, acc: 0.847, loss: 0.345, lr: 0.5405697605275961
epoch: 8600, acc: 0.847, loss: 0.343, lr: 0.5376633152320017
epoch: 8700, acc: 0.850, loss: 0.342, lr: 0.5347879565752179
epoch: 8800, acc: 0.847, loss: 0.340, lr: 0.5319431884674717
epoch: 8900, acc: 0.847, loss: 0.338, lr: 0.5291285253188
epoch: 9000, acc: 0.847, loss: 0.337, lr: 0.5263434917627243
epoch: 9100, acc: 0.850, loss: 0.336, lr: 0.5235876223886068
epoch: 9200, acc: 0.850, loss: 0.335, lr: 0.5208604614823689
epoch: 9300, acc: 0.847, loss: 0.334, lr: 0.5181615627752734
epoch: 9400, acc: 0.847, loss: 0.333, lr: 0.5154904892004742
epoch: 9500, acc: 0.850, loss: 0.331, lr: 0.5128468126570593
epoch: 9600, acc: 0.850, loss: 0.330, lr: 0.5102301137813153
epoch: 9700, acc: 0.850, loss: 0.329, lr: 0.5076399817249606
epoch: 9800, acc: 0.850, loss: 0.328, lr: 0.5050760139400979
epoch: 9900, acc: 0.850, loss: 0.326, lr: 0.5025378159706518
epoch: 10000, acc: 0.850, loss: 0.325, lr: 0.5000250012500626
```

4 RMSProp Optimizer

Root Meas Square Propagation optimizer. It is similar to AdaGrad in that you apply different learning rates to different weights. However, the way you change the learning rate focuses more on the most recent cache than the sum of all of the last gradients.

Notably, RMSProp does not use momentum. While it does have information of the past gradient in the cache, it uses this to scale the learning rate, not to correct for overshooting.

[15]:
```python
class Optimizer_RMSProp():
    def __init__(self, learning_rate=1e-3, decay=0.0, epsilon=1e-7, rho=0.9):
        self.initial_learning_rate = learning_rate
        self.current_learning_rate = self.initial_learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.rho = rho

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.initial_learning_rate / (1 + self.
 ↪decay * self.iterations)
```

```
    def update_params(self, layer):
        if not hasattr(layer, 'weight_cache'):
            layer.weight_cache = np.zeros_like(layer.weights)
            layer.bias_cache = np.zeros_like(layer.biases)

        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

        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)

    def post_update_params(self):
        self.iterations += 1
```

4.1 Testing the RMSProp Optimizer

[16]:
```
# 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_RMSProp(learning_rate=0.02, decay=1e-5, rho=0.999)

# 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: 0.413, loss: 1.099, lr: 0.02
epoch: 100, acc: 0.467, loss: 0.980, lr: 0.01998021958261321
epoch: 200, acc: 0.480, loss: 0.904, lr: 0.019960279044701046
epoch: 300, acc: 0.507, loss: 0.864, lr: 0.019940378268975763
epoch: 400, acc: 0.490, loss: 0.861, lr: 0.01992051713662487
epoch: 500, acc: 0.510, loss: 0.843, lr: 0.01990069552930875
epoch: 600, acc: 0.613, loss: 0.779, lr: 0.019880913329158343
epoch: 700, acc: 0.633, loss: 0.741, lr: 0.019861170418772778
epoch: 800, acc: 0.577, loss: 0.722, lr: 0.019841466681217078
epoch: 900, acc: 0.627, loss: 0.698, lr: 0.01982180200001982
epoch: 1000, acc: 0.630, loss: 0.675, lr: 0.019802176259170884
epoch: 1100, acc: 0.657, loss: 0.661, lr: 0.01978258934311912
```

epoch: 1200, acc: 0.623, loss: 0.654, lr: 0.019763041136770 13
epoch: 1300, acc: 0.663, loss: 0.640, lr: 0.019743531525483964
epoch: 1400, acc: 0.627, loss: 0.672, lr: 0.01972406039507293
epoch: 1500, acc: 0.663, loss: 0.618, lr: 0.019704627631799327
epoch: 1600, acc: 0.693, loss: 0.607, lr: 0.019685233122373254
epoch: 1700, acc: 0.657, loss: 0.658, lr: 0.019665876753950384
epoch: 1800, acc: 0.730, loss: 0.587, lr: 0.019646558414129805
epoch: 1900, acc: 0.690, loss: 0.623, lr: 0.019627277990951823
epoch: 2000, acc: 0.730, loss: 0.573, lr: 0.019608035372895814
epoch: 2100, acc: 0.743, loss: 0.576, lr: 0.019588830448878047
epoch: 2200, acc: 0.740, loss: 0.560, lr: 0.019569663108249594
epoch: 2300, acc: 0.710, loss: 0.567, lr: 0.019550533240794143
epoch: 2400, acc: 0.740, loss: 0.548, lr: 0.019531440736725945
epoch: 2500, acc: 0.687, loss: 0.576, lr: 0.019512385486687673
epoch: 2600, acc: 0.710, loss: 0.550, lr: 0.01949336738174836
epoch: 2700, acc: 0.707, loss: 0.573, lr: 0.019474386313401298
epoch: 2800, acc: 0.760, loss: 0.523, lr: 0.019455442173562
epoch: 2900, acc: 0.770, loss: 0.525, lr: 0.019436534854566128
epoch: 3000, acc: 0.787, loss: 0.512, lr: 0.01941766424916747
epoch: 3100, acc: 0.763, loss: 0.517, lr: 0.019398830250535893
epoch: 3200, acc: 0.750, loss: 0.551, lr: 0.019380032752255354
epoch: 3300, acc: 0.803, loss: 0.498, lr: 0.01936127164832186
epoch: 3400, acc: 0.780, loss: 0.493, lr: 0.01934254683314152
epoch: 3500, acc: 0.780, loss: 0.514, lr: 0.019323858201528515
epoch: 3600, acc: 0.793, loss: 0.522, lr: 0.019305205648703173
epoch: 3700, acc: 0.780, loss: 0.516, lr: 0.01928658907028997
epoch: 3800, acc: 0.790, loss: 0.508, lr: 0.01926800836231563
epoch: 3900, acc: 0.750, loss: 0.523, lr: 0.019249463421207133
epoch: 4000, acc: 0.763, loss: 0.516, lr: 0.019230954143789846
epoch: 4100, acc: 0.787, loss: 0.499, lr: 0.019212480427285565
epoch: 4200, acc: 0.770, loss: 0.512, lr: 0.019194042169310647
epoch: 4300, acc: 0.790, loss: 0.496, lr: 0.019175639267874092
epoch: 4400, acc: 0.777, loss: 0.501, lr: 0.019157271621375684
epoch: 4500, acc: 0.810, loss: 0.479, lr: 0.0191389391286041
epoch: 4600, acc: 0.783, loss: 0.482, lr: 0.019120641688735073
epoch: 4700, acc: 0.797, loss: 0.463, lr: 0.019102379201329525
epoch: 4800, acc: 0.787, loss: 0.469, lr: 0.01908415156633174
epoch: 4900, acc: 0.777, loss: 0.497, lr: 0.01906595868406753
epoch: 5000, acc: 0.810, loss: 0.445, lr: 0.01904780045524243
epoch: 5100, acc: 0.793, loss: 0.450, lr: 0.019029676780939874
epoch: 5200, acc: 0.810, loss: 0.438, lr: 0.019011587562619416
epoch: 5300, acc: 0.797, loss: 0.452, lr: 0.01899353270211493
epoch: 5400, acc: 0.800, loss: 0.453, lr: 0.018975512101632844
epoch: 5500, acc: 0.820, loss: 0.419, lr: 0.018957525663750367
epoch: 5600, acc: 0.817, loss: 0.433, lr: 0.018939573291413745
epoch: 5700, acc: 0.763, loss: 0.533, lr: 0.018921654887936498
epoch: 5800, acc: 0.820, loss: 0.411, lr: 0.018903770356997703
epoch: 5900, acc: 0.817, loss: 0.424, lr: 0.01888591960264025

```
epoch: 6000, acc: 0.810, loss: 0.419, lr: 0.018868102529269144
epoch: 6100, acc: 0.827, loss: 0.403, lr: 0.018850319041649778
epoch: 6200, acc: 0.820, loss: 0.413, lr: 0.018832569044906263
epoch: 6300, acc: 0.820, loss: 0.414, lr: 0.018814852444519702
epoch: 6400, acc: 0.810, loss: 0.410, lr: 0.018797169146326564
epoch: 6500, acc: 0.830, loss: 0.389, lr: 0.018779519056516963
epoch: 6600, acc: 0.823, loss: 0.407, lr: 0.018761902081633038
epoch: 6700, acc: 0.823, loss: 0.403, lr: 0.018744318128567278
epoch: 6800, acc: 0.820, loss: 0.405, lr: 0.018726767104560903
epoch: 6900, acc: 0.823, loss: 0.386, lr: 0.018709248917202218
epoch: 7000, acc: 0.827, loss: 0.393, lr: 0.018691763474424996
epoch: 7100, acc: 0.800, loss: 0.428, lr: 0.018674310684506857
epoch: 7200, acc: 0.833, loss: 0.378, lr: 0.018656890456067686
epoch: 7300, acc: 0.820, loss: 0.382, lr: 0.01863950269806802
epoch: 7400, acc: 0.810, loss: 0.438, lr: 0.018622147319807447
epoch: 7500, acc: 0.833, loss: 0.379, lr: 0.018604824230923078
epoch: 7600, acc: 0.837, loss: 0.351, lr: 0.01858753334138793
epoch: 7700, acc: 0.827, loss: 0.397, lr: 0.018570274561509396
epoch: 7800, acc: 0.837, loss: 0.369, lr: 0.018553047801927663
epoch: 7900, acc: 0.787, loss: 0.458, lr: 0.018535852973614212
epoch: 8000, acc: 0.840, loss: 0.369, lr: 0.01851868998787026
epoch: 8100, acc: 0.863, loss: 0.336, lr: 0.018501558756325222
epoch: 8200, acc: 0.840, loss: 0.366, lr: 0.01848445919093522
epoch: 8300, acc: 0.833, loss: 0.369, lr: 0.018467391203981567
epoch: 8400, acc: 0.837, loss: 0.355, lr: 0.01845035470806926
epoch: 8500, acc: 0.840, loss: 0.357, lr: 0.018433349616125496
epoch: 8600, acc: 0.857, loss: 0.329, lr: 0.018416375841398172
epoch: 8700, acc: 0.843, loss: 0.352, lr: 0.018399433297454436
epoch: 8800, acc: 0.843, loss: 0.356, lr: 0.01838252189817921
epoch: 8900, acc: 0.797, loss: 0.447, lr: 0.018365641557773718
epoch: 9000, acc: 0.847, loss: 0.354, lr: 0.018348792190754044
epoch: 9100, acc: 0.840, loss: 0.349, lr: 0.0183319737119497
epoch: 9200, acc: 0.853, loss: 0.337, lr: 0.018315186036502167
epoch: 9300, acc: 0.847, loss: 0.350, lr: 0.018298429079863496
epoch: 9400, acc: 0.823, loss: 0.383, lr: 0.018281702757794862
epoch: 9500, acc: 0.853, loss: 0.338, lr: 0.018265006986365174
epoch: 9600, acc: 0.780, loss: 0.522, lr: 0.018248341681949654
epoch: 9700, acc: 0.840, loss: 0.335, lr: 0.018231706761228456
epoch: 9800, acc: 0.853, loss: 0.334, lr: 0.01821510214118526
epoch: 9900, acc: 0.723, loss: 0.696, lr: 0.018198527739105907
epoch: 10000, acc: 0.860, loss: 0.314, lr: 0.018181983472577025
```

5 Adam Optimizer

The Adam optimizer combines adaptive learning rate with momentum.

$$W_{i+1} = W_i - \frac{\alpha}{\sqrt{\frac{\text{cache}}{1-\beta_2^{i+1}}} + \epsilon} * \left(\beta_1 * \frac{\text{Momentum}_i}{1-\beta_1^{i+1}} + (1-\beta_2) * \frac{\delta L}{\delta W}_i \right)$$

From the equation, it is clear that the cache and momentum are "corrected" by dividing by some scalar that varies with the iteration value. This way, earlier iterations have larger corrected cache and momentum so local minima are harder to get stuck in.

[17]:
```
import numpy as np

# Adam optimizer
class Optimizer_Adam():
    def __init__(self, learning_rate=0.001, decay=0.0, epsilon=1e-7, beta_1=0.
 ↪9, beta_2=0.999):
        self.initial_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

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.initial_learning_rate * (1. / (1.␣
 ↪+ self.decay * self.iterations))

    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
        # use self.iteration + 1 because we start at iteration 0
        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
```

5.1 Testing the Adam Optimizer

[19]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

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

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

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

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

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

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

    # Perform a forward pass through activation function
```

```
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

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

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

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

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

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

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

```
epoch: 0, acc: 0.327, loss: 1.099, lr: 0.02
epoch: 100, acc: 0.523, loss: 0.941, lr: 0.01998021958261321
epoch: 200, acc: 0.517, loss: 0.854, lr: 0.019960279044701046
epoch: 300, acc: 0.630, loss: 0.753, lr: 0.019940378268975763
epoch: 400, acc: 0.693, loss: 0.699, lr: 0.01992051713662487
epoch: 500, acc: 0.693, loss: 0.658, lr: 0.01990069552930875
epoch: 600, acc: 0.707, loss: 0.632, lr: 0.019880913329158343
epoch: 700, acc: 0.707, loss: 0.612, lr: 0.019861170418772778
epoch: 800, acc: 0.720, loss: 0.572, lr: 0.019841466681217078
epoch: 900, acc: 0.737, loss: 0.558, lr: 0.01982180200001982
epoch: 1000, acc: 0.730, loss: 0.548, lr: 0.019802176259170884
```

epoch: 1100, acc: 0.730, loss: 0.540, lr: 0.01978258934311912
epoch: 1200, acc: 0.733, loss: 0.529, lr: 0.01976304113677013
epoch: 1300, acc: 0.733, loss: 0.521, lr: 0.019743531525483964
epoch: 1400, acc: 0.750, loss: 0.517, lr: 0.01972406039507293
epoch: 1500, acc: 0.750, loss: 0.518, lr: 0.019704627631799327
epoch: 1600, acc: 0.743, loss: 0.508, lr: 0.019685233122373254
epoch: 1700, acc: 0.723, loss: 0.505, lr: 0.019665876753950384
epoch: 1800, acc: 0.740, loss: 0.502, lr: 0.01964655841412981
epoch: 1900, acc: 0.740, loss: 0.498, lr: 0.019627277990951823
epoch: 2000, acc: 0.730, loss: 0.493, lr: 0.019608035372895814
epoch: 2100, acc: 0.753, loss: 0.491, lr: 0.01958883044887805
epoch: 2200, acc: 0.760, loss: 0.488, lr: 0.019569663108249594
epoch: 2300, acc: 0.757, loss: 0.487, lr: 0.01955053324079414
epoch: 2400, acc: 0.743, loss: 0.482, lr: 0.019531440736725945
epoch: 2500, acc: 0.743, loss: 0.479, lr: 0.019512385486687673
epoch: 2600, acc: 0.760, loss: 0.479, lr: 0.019493367381748363
epoch: 2700, acc: 0.797, loss: 0.462, lr: 0.019474386313401298
epoch: 2800, acc: 0.787, loss: 0.439, lr: 0.019455442173562
epoch: 2900, acc: 0.790, loss: 0.436, lr: 0.019436534854566128
epoch: 3000, acc: 0.800, loss: 0.435, lr: 0.01941766424916747
epoch: 3100, acc: 0.787, loss: 0.433, lr: 0.019398830250535893
epoch: 3200, acc: 0.793, loss: 0.430, lr: 0.019380032752255354
epoch: 3300, acc: 0.800, loss: 0.429, lr: 0.01936127164832186
epoch: 3400, acc: 0.790, loss: 0.427, lr: 0.01934254683314152
epoch: 3500, acc: 0.790, loss: 0.426, lr: 0.019323858201528515
epoch: 3600, acc: 0.783, loss: 0.428, lr: 0.019305205648703173
epoch: 3700, acc: 0.790, loss: 0.424, lr: 0.01928658907028997
epoch: 3800, acc: 0.773, loss: 0.428, lr: 0.01926800836231563
epoch: 3900, acc: 0.787, loss: 0.420, lr: 0.019249463421207133
epoch: 4000, acc: 0.797, loss: 0.417, lr: 0.019230954143789846
epoch: 4100, acc: 0.797, loss: 0.415, lr: 0.019212480427285565
epoch: 4200, acc: 0.807, loss: 0.414, lr: 0.019194042169310647
epoch: 4300, acc: 0.810, loss: 0.412, lr: 0.019175639267874092
epoch: 4400, acc: 0.807, loss: 0.413, lr: 0.019157271621375684
epoch: 4500, acc: 0.793, loss: 0.411, lr: 0.0191389391286041
epoch: 4600, acc: 0.810, loss: 0.413, lr: 0.019120641688735073
epoch: 4700, acc: 0.810, loss: 0.409, lr: 0.019102379201329525
epoch: 4800, acc: 0.790, loss: 0.408, lr: 0.01908415156633174
epoch: 4900, acc: 0.810, loss: 0.408, lr: 0.01906595868406753
epoch: 5000, acc: 0.800, loss: 0.406, lr: 0.01904780045524243
epoch: 5100, acc: 0.793, loss: 0.407, lr: 0.019029676780939874
epoch: 5200, acc: 0.787, loss: 0.405, lr: 0.019011587562619416
epoch: 5300, acc: 0.793, loss: 0.405, lr: 0.01899353270211493
epoch: 5400, acc: 0.797, loss: 0.404, lr: 0.018975512101632844
epoch: 5500, acc: 0.810, loss: 0.405, lr: 0.018957525663750367
epoch: 5600, acc: 0.793, loss: 0.403, lr: 0.018939573291413745
epoch: 5700, acc: 0.790, loss: 0.403, lr: 0.018921654887936498
epoch: 5800, acc: 0.790, loss: 0.402, lr: 0.018903770356997706

epoch: 5900, acc: 0.793, loss: 0.401, lr: 0.018885919602640248
epoch: 6000, acc: 0.813, loss: 0.404, lr: 0.018868102529269144
epoch: 6100, acc: 0.807, loss: 0.401, lr: 0.018850319041649778
epoch: 6200, acc: 0.807, loss: 0.402, lr: 0.018832569044906263
epoch: 6300, acc: 0.810, loss: 0.401, lr: 0.018814852444519702
epoch: 6400, acc: 0.810, loss: 0.399, lr: 0.018797169146326564
epoch: 6500, acc: 0.793, loss: 0.401, lr: 0.01877951905651696
epoch: 6600, acc: 0.797, loss: 0.399, lr: 0.018761902081633034
epoch: 6700, acc: 0.807, loss: 0.398, lr: 0.018744318128567278
epoch: 6800, acc: 0.790, loss: 0.399, lr: 0.018726767104560903
epoch: 6900, acc: 0.810, loss: 0.398, lr: 0.018709248917202218
epoch: 7000, acc: 0.807, loss: 0.398, lr: 0.018691763474424996
epoch: 7100, acc: 0.807, loss: 0.399, lr: 0.018674310684506857
epoch: 7200, acc: 0.790, loss: 0.398, lr: 0.01865689045606769
epoch: 7300, acc: 0.790, loss: 0.398, lr: 0.01863950269806802
epoch: 7400, acc: 0.803, loss: 0.396, lr: 0.018622147319807447
epoch: 7500, acc: 0.790, loss: 0.399, lr: 0.018604824230923075
epoch: 7600, acc: 0.793, loss: 0.398, lr: 0.01858753334138793
epoch: 7700, acc: 0.810, loss: 0.396, lr: 0.018570274561509396
epoch: 7800, acc: 0.810, loss: 0.395, lr: 0.018553047801927663
epoch: 7900, acc: 0.803, loss: 0.395, lr: 0.018535852973614212
epoch: 8000, acc: 0.790, loss: 0.395, lr: 0.01851868998787026
epoch: 8100, acc: 0.813, loss: 0.395, lr: 0.018501558756325222
epoch: 8200, acc: 0.790, loss: 0.395, lr: 0.01848445919093522
epoch: 8300, acc: 0.793, loss: 0.395, lr: 0.018467391203981567
epoch: 8400, acc: 0.793, loss: 0.395, lr: 0.018450354708069265
epoch: 8500, acc: 0.813, loss: 0.394, lr: 0.018433349616125496
epoch: 8600, acc: 0.813, loss: 0.395, lr: 0.018416375841398172
epoch: 8700, acc: 0.793, loss: 0.394, lr: 0.01839943329745444
epoch: 8800, acc: 0.783, loss: 0.398, lr: 0.01838252189817921
epoch: 8900, acc: 0.793, loss: 0.393, lr: 0.018365641557773718
epoch: 9000, acc: 0.797, loss: 0.393, lr: 0.018348792190754044
epoch: 9100, acc: 0.813, loss: 0.394, lr: 0.0183319737119497
epoch: 9200, acc: 0.813, loss: 0.393, lr: 0.018315186036502167
epoch: 9300, acc: 0.810, loss: 0.393, lr: 0.018298429079863496
epoch: 9400, acc: 0.793, loss: 0.395, lr: 0.018281702757794862
epoch: 9500, acc: 0.800, loss: 0.392, lr: 0.018265006986365174
epoch: 9600, acc: 0.797, loss: 0.393, lr: 0.018248341681949654
epoch: 9700, acc: 0.807, loss: 0.392, lr: 0.018231706761228456
epoch: 9800, acc: 0.817, loss: 0.393, lr: 0.018215102141185255
epoch: 9900, acc: 0.817, loss: 0.395, lr: 0.018198527739105907
epoch: 10000, acc: 0.790, loss: 0.392, lr: 0.018181983472577025