PRACTICE INTRODUCTION

PRACTICE WITH PYTORCH #02

(*Keyword: PyTorch)*

I. Goals

- Students continue to use PyTorch to implement extensive basic artificial neural networks.

II. Installation Requirements

- Programming language: *Python*, recommend minimum version **3.6**.
- Library: *PyTorch*, *NumPy*, *OpenCV-Python* (+ *OpenCV_Contrib*).
- IDE / Text Editor: recommended use *JetBrains PyCharm Community* (*PyCharm*) or *Microsoft Visual Studio Code (VS Code)*.

III. Contents

Installing a basic *Feed Forward* (FF) neural network with PyTorch from each component is similar to the previous exercise, *but the network components need to be installed flexibly*.

Input: input layer

- Middle layer (hidden layer): hidden layer(s)
- Output: output layer
- Activation function: *sigmoid*, *tanh* or other activation functions

We will continue to use torch.nn to install these basic components.

```
# import PyTorch
import torch
# import PyTorch Neural Network module
import torch.nn as nn
```

In order to be able to use network components like the previous exercise but increase flexibility in use, we can install them into separate *modules* and define corresponding *classes* for the components, instead of only use the definition of individual functions via the *def* keyword.

It is possible to start with the *Module* for *Activation* functions, *Loss* error calculation functions and corresponding derivative functions...

```
class Activation:
    # sigmoid activation
    @staticmethod
```

```
    def sigmoid(s):
        return 1 / (1 + torch.exp(-s))

    # tanh activation
    @staticmethod
    def tanh(s):
        return torch.tanh(s)

class ActivationPrime:
    # derivative of sigmoid
    @staticmethod
    def sigmoid_derivative(s):
        return s * (1 - s)

    # derivative of tanh
    @staticmethod
    def tanh_derivative(s):
        return 1 - torch.tanh(s) ** 2

class Loss:
    # Mean Square Error loss function
    @staticmethod
    def mse(y_true, y_pred):
        return torch.mean(torch.pow(y_true - y_pred, 2))

class LossPrime:
    # derivative of Mean Square Error loss function
    @staticmethod
    def mse_prime(y_true, y_pred):
        return 2 * (y_pred - y_true) / y_true.numel()
```

Defines the *BaseLayer* class.

```
# abstract layer class
class BaseLayer:
    def __index__(self):
        pass

    def forward(self, in_data):
        pass

    def backward(self, out_error, rate):
        pass
```

From there define the *ActivationLayer* and *FullConnectedLayer* classes, inheriting from the base *BaseLayer* class, here it is necessary to pay attention to the parameter learning *rate* of the network.

```
from layer_simple.base_layer import BaseLayer


class ActivationLayer(BaseLayer):
    def __init__(self, activation, activation_derivative):
        self.in_data = None
        self.out_data = None

        self.activation = activation
        self.activation_derivative = activation_derivative

    def forward(self, in_data):
```

```
        self.in_data = in_data
        self.out_data = self.activation(in_data)

        return self.out_data

    def backward(self, out_error, rate):
        return self.activation_derivative(self.in_data) *
out_error
```

And

```
from layer_simple.base_layer import BaseLayer

import torch


class FCLayer(BaseLayer):
    def __init__(self, in_size, out_size):
        self.in_data = None
        self.out_data = None

        self.weights = torch.randn(in_size, out_size)
        self.bias = torch.randn(1, out_size)

    def forward(self, in_data):
        self.in_data = in_data
        self.out_data = torch.matmul(self.in_data,
self.weights) + self.bias

        return self.out_data

    def backward(self, out_error, rate):
        in_error = torch.matmul(out_error, self.weights.T)
        weights_error = torch.matmul(self.in_data.T, out_error)

        self.weights -= rate * weights_error
        self.bias -= rate * out_error

        return in_error
```

From the above basic functions and basic classes, we proceed to define the *NeuralNetwork* network, which is flexible for defining (adding) layers to the network structure, with the *Activation* function as well as the *Loss* and derivative error function. corresponding.

```
from function_simple import Loss, LossPrime
import torch


class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    def add(self, layer):
        self.layers.append(layer)

    def use(self, loss: Loss, loss_prime: LossPrime) -> None:
```

```
        self.loss = loss
        self.loss_prime = loss_prime
```

Set up a forward propagation step for the network, going through the *Layers* in the defined network topology.

```
    # forward propagation
    def predict(self, data):
        output = data
        for layer in self.layers:
            output = layer.forward(output)
        return output

    def predicts(self, data):
        samples = len(data)
        result = []

        # for every input vector data x_i do:
        for i in range(samples):
            # forward propagation
            output = data[i]
            for layer in self.layers:
                output = layer.forward(output)
            result.append(output)
        return result
```

And install the network training function, with the forward propagation algorithm that calculates the error and updates the parameter via backpropagation, with the number of iterations according to the required number of *epochs*, and the corresponding learning *rate*.

```
    def fit(self, x_train, y_train, epochs, alpha):
        samples = len(x_train)

        # for every training cycle -> forward() -> forward() ->
Loss() <- backward() <- backward() ...
        for i in range(epochs):
            error = 0
            for k in range(samples):
                # forward propagation
                output = x_train[k]
                for layer in self.layers:
                    # output of the previous layer -> input to
the current layer @l
                    output = layer.forward(output)
                # total error after current x_k has passed
training cycle.
                error += self.loss(y_train[k], output)

                # Backward propagation
                gradient = self.loss_prime(y_train[k], output)
                for layer in reversed(self.layers):
                    gradient = layer.backward(gradient, alpha)

            # the total average error after one epoch?
            error /= samples
            print('On epoch ' + str(i + 1) + ' an average error
= ' + str(error))
```

After installing the basic components, we can use the corresponding *modules* to define the network more flexibly than before. Sample data can be generated for training and prediction to test the installed network model.

```
import torch

from layer_simple import FCLayer, ActivationLayer
from function_simple import Activation, ActivationPrime
from function_simple import Loss, LossPrime
from network_simple import Network

# training data
x_train = torch.tensor([[[0, 0]], [[0, 1]], [[1, 0]], [[1,
1]]], dtype=torch.float)
y_train = torch.tensor([[[0]], [[1]], [[1]], [[0]]],
dtype=torch.float)

# network architecture
net = Network()
net.add(FCLayer(2, 3))
net.add(ActivationLayer(Activation.tanh,
ActivationPrime.tanh_derivative))
net.add(FCLayer(3, 1))
net.add(ActivationLayer(Activation.tanh,
ActivationPrime.tanh_derivative))

# train your network
net.use(Loss.mse, LossPrime.mse_prime)
net.fit(x_train, y_train, epochs=100, alpha=0.1)

# test
out = net.predicts(x_train)
print(out)
```

Students try to install and use the installed network, then change the basic parameters of the network: learning rate, class size, number of classes, class type, number of *epochs*... and conduct experiments to observe close to the results.

The reference result of the sample source code is as shown below.

```
On epoch 90 an average error = tensor(0.0060)
On epoch 91 an average error = tensor(0.0058)
On epoch 92 an average error = tensor(0.0057)
On epoch 93 an average error = tensor(0.0056)
On epoch 94 an average error = tensor(0.0055)
On epoch 95 an average error = tensor(0.0054)
On epoch 96 an average error = tensor(0.0053)
On epoch 97 an average error = tensor(0.0052)
On epoch 98 an average error = tensor(0.0051)
On epoch 99 an average error = tensor(0.0050)
On epoch 100 an average error = tensor(0.0050)
[tensor([[0.0117]]), tensor([[0.9108]]), tensor([[0.8962]]), tensor([[-0.0045]])]
```