HƯỚNG DẪN THỰC HÀNH

THỰC HÀNH VỚI PYTORCH #02

(*Keyword: PyTorch)*

I. Mục tiêu

- Sinh viên tiếp tục sử dụng PyTorch để cài đặt mạng nơ-ron nhân tạo cơ bản mở rộng.

II. Yêu cầu cài đặt

- Ngôn ngữ lập trình: *Python*, phiên bản tối thiểu khuyến nghị **3.6**.
- Thư viện: *PyTorch*, *NumPy*, *OpenCV-Python* (+ *OpenCV_Contrib*).
- IDE / Text Editor: khuyến nghị sử dụng *JetBrains PyCharm Community* (*PyCharm*) hoặc *Microsoft Visual Studio Code (VS Code)*.

III. Nội dung

Cài đặt mạng nơ-ron nhân tạo cơ bản *Feed Forward* (FF) với PyTorch từ từng thành phần cấu thành tương tự như bài thực hành trước, *nhưng các thành phần mạng cần được cài đặt một cách linh hoạt*.

- Đầu vào: input layer
- Lớp giữa (lớp ẩn): hidden layer(s)
- Đầu ra: output layer
- Hàm kích hoạt: *sigmoid*, *tanh* hoặc các hàm kích hoạt khác

Chúng ta sẽ tiếp tục sử dụng torch.nn để cài đặt các thành phần cơ bản này.

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

Nhằm có thể sử dụng các thành phần mạng như bài thực hành trước nhưng tăng khả năng linh hoạt khi sử dụng, chúng ta có thể cài đặt thành các *module* riêng biệt và định nghĩa các lớp *class* tương ứng cho các thành phần, thay vì chỉ sử dụng định nghĩa các hàm riêng lẽ qua từ khóa *def*.

Có thể bắt đầu với *Module* cho các hàm kích hoạt *Activation*, hàm tính độ lỗi *Loss* và các hàm lấy đạo hàm tương ứng...

```
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()
```

Định nghĩa lớp *BaseLayer* cơ sở.

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

    def forward(self, in_data):
        pass

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

Từ đó định nghĩa các lớp *ActivationLayer* và *FullConnectedLayer*, kế thừa từ lớp *BaseLayer* cơ sở, ở đây cần chú ý tham số tốc độ học *rate* của mạng.

```
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
```

Và

```
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
```

Từ các hàm cơ sở và các lớp cơ bản nói trên, chúng ta tiến hành định nghĩa mạng *NeuralNetwork*, linh hoạt cho định nghĩa (thêm) các Layer vào cấu trúc mạng, với hàm kích hoạt *Activation* cũng như hàm lỗi *Loss* và đạo hàm tương ứng.

```
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
```

Cài đặt bước lan truyền xuôi cho mạng, đi qua các *Layer* trong cấu trúc mạng đã định nghĩa.

```
    # 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
```

Và cài đặt hàm huấn luyện mạng, với thuật toán lan truyền xuôi tính toán độ lỗi và cập nhật tham số qua lan truyền ngược, với số lần lặp theo số *epoch* yêu cầu, cùng tốc độ học tương ứng *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))
```

Sau khi cài đặt xong các thành phần cơ bản, chúng ta có thể sử dụng các *module* tương ứng để định nghĩa mạng một cách linh hoạt hơn trước. Có thể tạo dữ liệu mẫu để huấn luyện và dự đoán nhằm thử nghiệm mô hình mạng đã cài đặt.

```
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)
```

Sinh viên thử cài đặt, sử dụng mạng được cài đặt, sau đó thay đổi các tham số cơ bản của mạng: tốc độ học, kích thước các lớp, số lớp, loại lớp, số *epoch*… và tiến hành thực nghiệm để quan sát các kết quả.

Kết quả tham khảo của mã nguồn mẫu như hình dưới.

```
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]])]
```