HƯỚNG DẪN THỰC HÀNH

THỰC HÀNH VỚI PYTORCH #01

(*Keyword: PyTorch*)

I. Mục tiêu

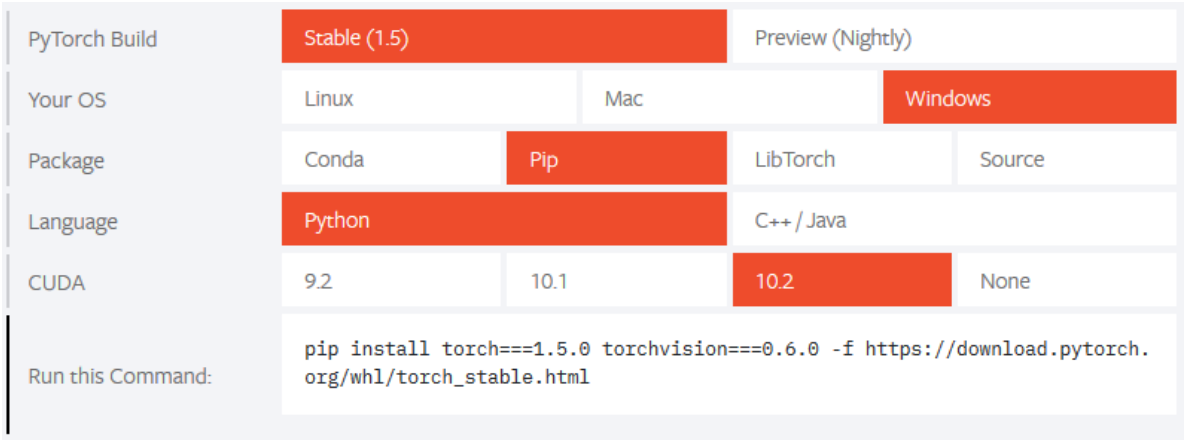
- Sinh viên làm quen với PyTorch để cài đặt các mạng nơ ron nhân tạo cơ bản.

II. Yêu cầu cài đặt

- Ngôn ngữ lập trình: Python, phiên bản khuyến nghị tối thiểu 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

1. Cài đặc các thành phần cần thiết:
   - *PyTorch*: https://pytorch.org/
   → Có thể cài đặt qua câu lệnh pip tương tự như các hướng dẫn thực hành khác trước đó (nếu có). Chú ý kiểm tra câu lệnh với môi trường (ảo) tương ứng.
   Lưu ý: nếu có GPU NVIDIA hỗ trợ CUDA, cần cài đặt đầy đủ driver cần thiết, cũng như CUDA tại https://developer.nvidia.com/cuda-downloads và cuDNN tại https://developer.nvidia.com/rdp/cudnn-download.



2. *Tensor*
   Đây là kiểu dữ liệu chính được sử dụng trong *PyTorch*, về mặt trực quan có thể hình dung *Tensor* giống như là một ma trận nhiều chiều. *Tensor* và *Numpy* có thể chosen đổi qua lại rất dễ dàng.

```
# import required libraries
# import PyTorch
import torch
# import NumPy
import numpy as np

# numpy array
x_np = np.array([[1, 0, 2], [2, 0, 1]])
# PyTorch tensor from numpy array
x_torch = torch.from_numpy(x_np)

print('x_np', x_np)
print('x_torch', x_torch)

x_np += 1

print('x_np', x_np)
print('x_torch', x_torch)

x_torch += 1

print('x_np', x_np)
print('x_torch', x_torch)

# PyTorch tensor
y_torch = torch.tensor(([0, 8], [0, 4], [20, 20]),
dtype=torch.float)
# numpy array from PyTorch tensor
y_np = y_torch.numpy()
# or more explicit
y_np_cpu = y_torch.detach().cpu().numpy()

print('y_torch', y_torch)
print('y_np', y_np)

y_np += 1

print('y_torch', y_torch)
print('y_np', y_np)

y_torch += 1

print('y_torch', y_torch)
print('y_np', y_np)
```

3. *PyTorch Neural Network*

   Cài đặt mạng nơ-ron nhân tạo cơ bản *Feed Forward* (FF) với PyTorch từ từng thành phần cấu thành.

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

   Chúng ta sẽ 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
```

Định nghĩa hàm *sigmoid* và đạo hàm *derivative* tương ứng.

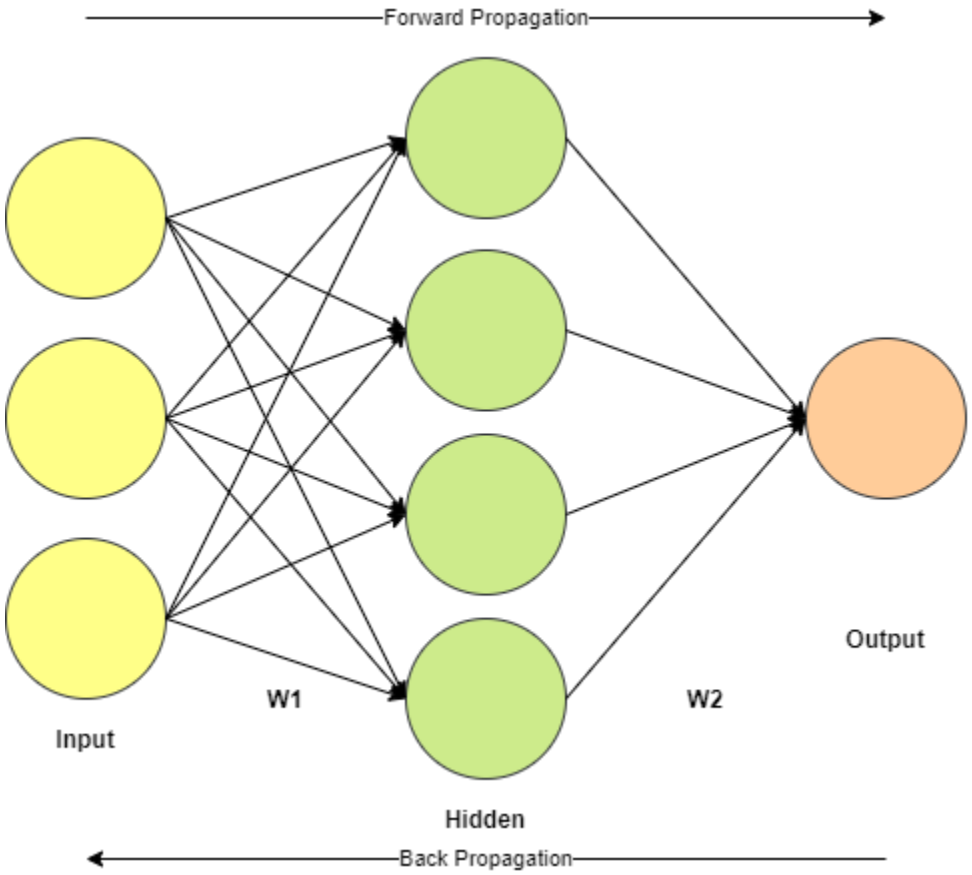
```
# sigmoid activation
def sigmoid(s):
    return 1 / (1 + torch.exp(-s))


# derivative of sigmoid
def sigmoid_derivative(s):
    return s * (1 - s)
```

Định nghĩa mạng *Feed Forward Neural Network* (FFNN) thông qua lớp đối tượng mới kế thừa từ lớp nn.Module của PyTorch.

```
# Feed Forward Neural Network class
class FFNN(nn.Module):
```

Ở đây, lấy ví dụ kích thước của lớp đầu vào là 3, có 1 lớp ẩn duy nhất với kích thước là 4, và kích thước lớp đầu ra là 1.

Ta cần định nghĩa các thông số tương ứng, và bộ trọng số *weight* ngẫu nhiên từ một phân phối chuẩn với kích thước tương ứng cho từng bước *Feed Forward*. Các định nghĩa này được cài đặt trong hàm khởi tạo *init* của lớp đối tượng chính.




```
    # initialization function
    def __init__(self, ):
        # init function of base class
        super(FFNeuralNetwork, self).__init__()

        # corresponding size of each layer
        self.inputSize = 3
```

```
        self.hiddenSize = 4
        self.outputSize = 1

        # random weights from a normal distribution
        self.W1 = torch.randn(self.inputSize, self.hiddenSize)
# 3 X 4 tensor
        self.W2 = torch.randn(self.hiddenSize, self.outputSize)
# 4 X 1 tensor
```

Định nghĩa hàm kích hoạt *activation* và đạo hàm tương ứng sử dụng *sigmoid* và *sigmoid_derivative* đã cài đặt ở trên.

```
    # activation function using sigmoid
    def activation(self, z):
        self.z_activation = sigmoid(z)
        return self.z_activation

    # derivative of activation function
    def activation_derivative(self, z):
        self.z_activation_derivative = sigmoid_derivative(z)
        return self.z_activation_derivative
```

Chúng ta có thể cài đặt phần lan truyền xuôi (*forward propagation*) một cách cơ bản như sau (chưa xét tới *bias*).

```
    # forward propagation
    def forward(self, X):
        # multiply input X and weights W1 from input layer to
hidden layer
        self.z = torch.matmul(X, self.W1)
        self.z2 = self.activation(self.z)  # activation
function
        # multiply current tensor and weights W2 from hidden
layer to output layer
        self.z3 = torch.matmul(self.z2, self.W2)
        o = self.activation(self.z3)  # final activation
function
        return o
```

Tương ứng là phần lan truyền ngược (*backward propagation*) với tốc độ học tương ứng *learning rate* là tham số *rate*.

```
    # backward propagation
    def backward(self, X, y, o, rate):
        self.out_error = y - o  # error in output
        self.out_delta = self.out_error *
self.activation_derivative(o)  # derivative of activation to
error

        # error and derivative of activation to error of next
layer in backward propagation
        self.z2_error = torch.matmul(self.out_delta,
torch.t(self.W2))
        self.z2_delta = self.z2_error *
self.activation_derivative(self.z2)

        # update weights from delta of error and learning rate
        self.W1 += torch.matmul(torch.t(X), self.z2_delta) *
rate
        self.W2 += torch.matmul(torch.t(self.z2),
self.out_delta) * rate
```

Mỗi lần huấn luyện tương ứng với một lần lan truyền xuôi và cập nhật tham số với lan truyền ngược.

```
    # backward propagation
    # training function with learning rate parameter
    def train(self, X, y, rate):
        # forward + backward pass for training
        o = self.forward(X)
        self.backward(X, y, o, rate)
```

Cài đặt thêm hàm để lưu và nạp bộ trọng số *weights*.

```
    # save weights of model
    @staticmethod
    def save_weights(model, path):
        # use the PyTorch internal storage functions
        torch.save(model, path)

    # load weights of model
    @staticmethod
    def load_weights(path):
        # reload model with all the weights
        torch.load(path)
```

Cài đặt hàm dự đoán nhận đầu vào *x* phù hợp và cho ra kết quả dự đoán tương ứng qua lan truyền xuôi.

```
    # predict function
    def predict(self, x_predict):
        print("Predicted data based on trained weights: ")
        print("Input: \n" + str(x_predict))
        print("Output: \n" + str(self.forward(x_predict)))
```

Để hỗ trợ cho phần cài đặt truyền xuôi và lan truyền ngược cơ bản trong mạng nơ-ron, ta có thể khai báo sẵn các biến trung gian tại hàm khởi tạo, phục vụ cho quá trình tính toán từng bước qua từng lớp tương ứng.

```
class FFNeuralNetwork(nn.Module):
    def __init__(self, ):
        # init function of base class
        super(FFNeuralNetwork, self).__init__()

        # corresponding size of each layer
        self.inputSize = 3
        self.hiddenSize = 4
        self.outputSize = 1

        # random weights from a normal distribution
        self.W1 = torch.randn(self.inputSize, self.hiddenSize)
# 3 X 4 tensor
        self.W2 = torch.randn(self.hiddenSize, self.outputSize)
# 4 X 1 tensor

        self.z = None
        self.z_activation = None
        self.z_activation_derivative = None

        self.z2 = None
        self.z3 = None

        self.out_error = None
        self.out_delta = None
```

```
        self.z2_error = None
        self.z2_delta = None
```

Sử dụng lớp đối tượng mạng nơ-ron đã cài đặt để tiến hành huấn luyện mạng 1000 lần, với tốc độ học là 0.1, lưu lại trọng số sau khi quá trình huấn luyện hoàn tất.

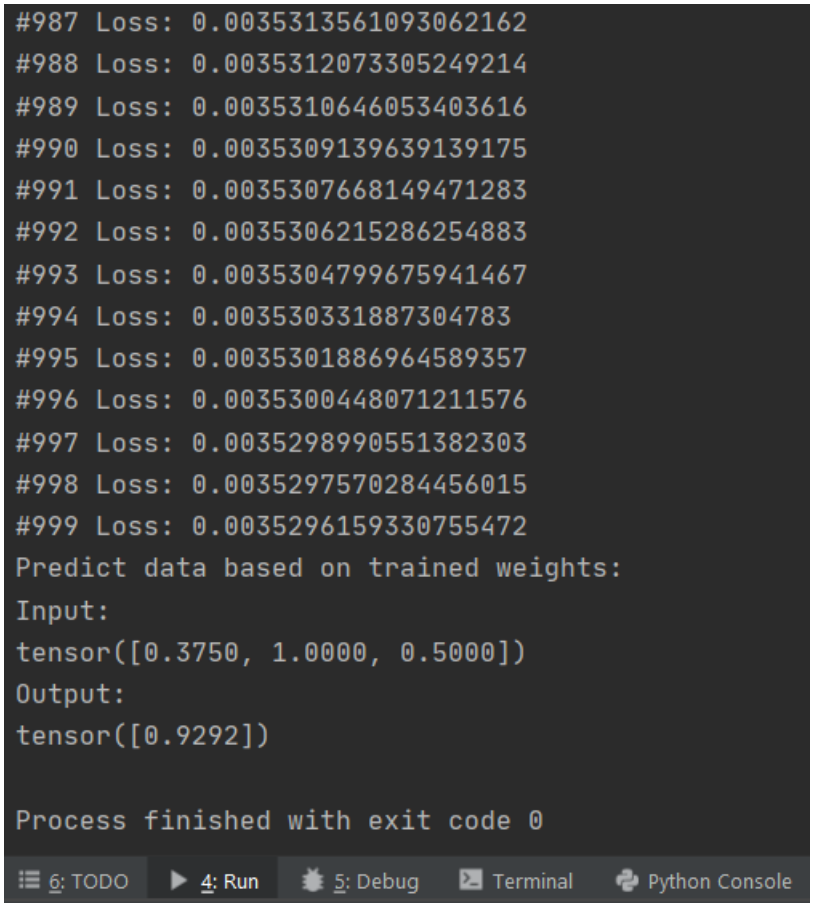
```
# create new object of implemented class
NN = nn.FFNeuralNetwork()

# trains the NN 1,000 times
for i in range(1000):
    # print mean sum squared loss
    print("#" + str(i) + " Loss: " + str(torch.mean((y - NN(X))
** 2).detach().item()))
    # training with learning rate = 0.1
    NN.train(X, y, 0.1)
# save weights
NN.save_weights(NN, "NN")
```

Tạo các dữ liệu mẫu để huấn luyện và dự đoán nhằm thử nghiệm mô hình mạng đã cài đặt, có thể tiền xử lý dữ liệu đơn giản bằng việc chuẩn hóa các giá trị theo tỉ lệ so với giá trị lớn nhất.

```
# sample input and output value for training
X = torch.tensor(([2, 9, 0], [1, 5, 1], [3, 6, 2]),
dtype=torch.float)  # 3 X 3 tensor
y = torch.tensor(([90], [100], [88]), dtype=torch.float)  # 3 X
1 tensor

# scale units by max value
X_max, _ = torch.max(X, 0)
X = torch.div(X, X_max)
y = y / 100  # for max test score is 100

# sample input x for predicting
x_predict = torch.tensor(([3, 8, 4]), dtype=torch.float)  # 1 X
3 tensor

# scale input x by max value
x_predict_max, _ = torch.max(x_predict, 0)
x_predict = torch.div(x_predict, x_predict_max)

# load saved weights
NN.load_weights("NN")
# predict x input
NN.predict(x_predict)
```

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 ẩn… và tiến hành thực nghiệm để quan sát các kết quả.

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


#987 Loss: 0.0035313561093062162
#988 Loss: 0.0035312073305249214
#989 Loss: 0.0035310646053403616
#990 Loss: 0.0035309139639139175
#991 Loss: 0.0035307668149471283
#992 Loss: 0.0035306215286254883
#993 Loss: 0.0035304799675941467
#994 Loss: 0.003530331887304783
#995 Loss: 0.0035301886964589357
#996 Loss: 0.0035300448071211576
#997 Loss: 0.0035298990551382303
#998 Loss: 0.0035297570284456015
#999 Loss: 0.0035296159330755472
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.9292])
Process finished with exit code 0
6: TODO
4: Run
5: Debug
Terminal
Python Console