PRACTICE INTRODUCTION

PRACTICE WITH PYTORCH #1

(*Keyword: PyTorch*)

I. Goals

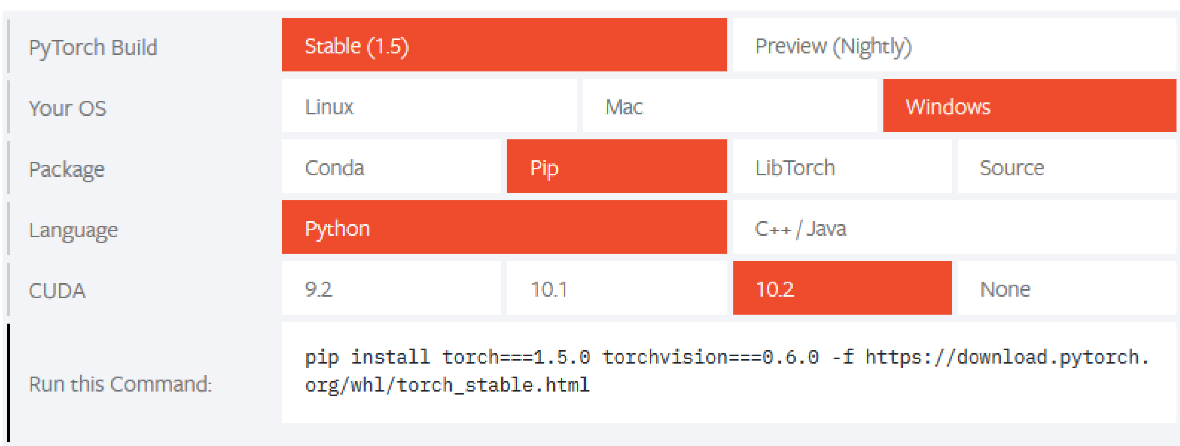
- Students become familiar with using PyTorch to implement basic neural networks.

II. Installation Requirements

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

III. Contents

1. Install the required components:
   - *PyTorch*: https://pytorch.org/
   → Can be installed via *pip*. Look closely at the *pip* command for the correct environment.



   Note: if you have an NVIDIA GPU that supports CUDA, you need to install all the necessary drivers, as well as CUDA at https://developer.nvidia.com/cuda-downloads and cuDNN at https://developer.nvidia.com/rdp/cudnn-download.

2. *Tensor*
   This is the main data type used in *PyTorch*, visually *Tensor* can be visualized as a multidimensional matrix. *Tensor* and *Numpy* can be converted very convenience.

```
# 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*

Install *Feed Forward* (FF) basic artificial neural network with PyTorch from each component.

- Input: input layer
- Middle layer (hidden layer): hidden layer(s)
- Input: output layer
- Activation function: *sigmoid* or other activation functions

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

```
# import PyTorch
import torch
```

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

Function definition *sigmoid* and corresponding *derivative*.

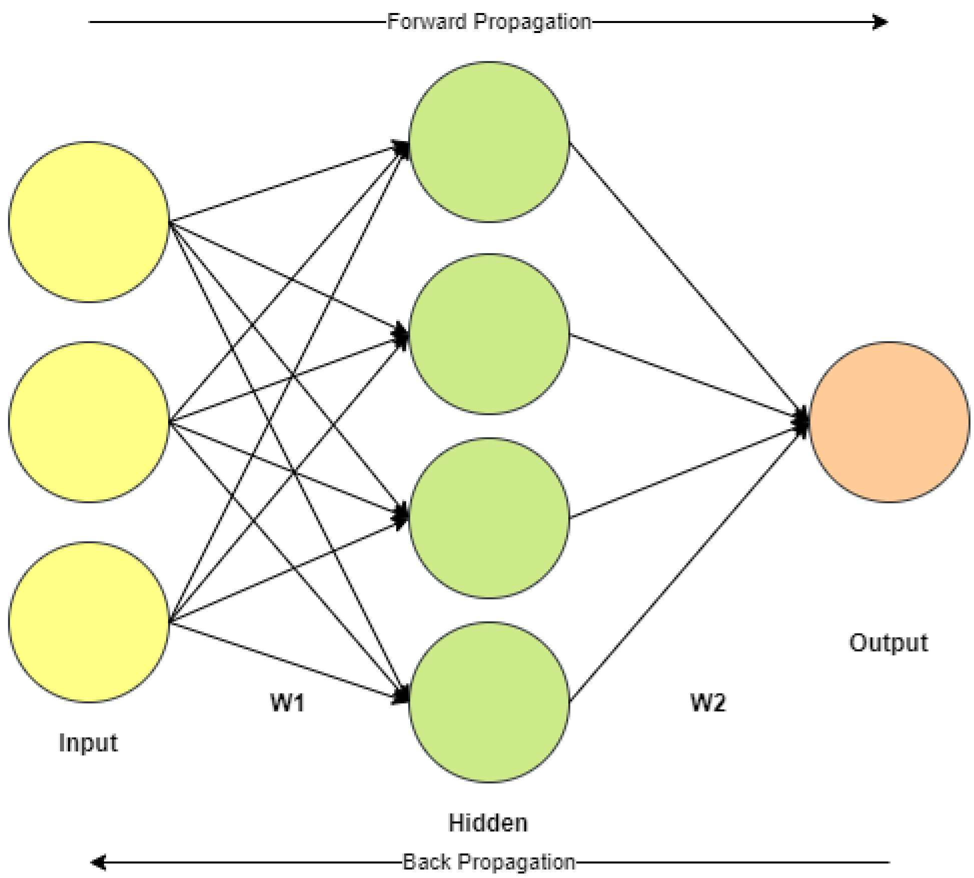
```
# sigmoid activation
def sigmoid(s):
    return 1 / (1 + torch.exp(-s))


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

Define a *Feed Forward Neural Network* (FFNN) through a new object class that inherits from PyTorch's nn.Module class.

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

Here, for example, the input layer size is 3, there is a single hidden layer of size 4, and the output layer size is 1.

We need to define the corresponding parameters and set of random weights from a



normal distribution with corresponding sizes for each *Feed Forward* step. These definitions are implemented in the main object's *init* constructor.

```
    # 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
```

Define the *activation* function and the corresponding derivate using the *sigmoid*, and *sigmoid_derivative* installed above.

```
    # 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
```

We can basically implement *forward propagation* as follows (without considering *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
```

The corresponding is the *backpropagation* with the corresponding *learning rate* the *rate* parameter.

```
    # 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
```

Each training corresponds to a forward propagation and parameter update with backpropagation.

```
    # 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)
```

Install additional functions to save and load the set of *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)
```

Implement a prediction function that takes the appropriate input *x* and outputs the corresponding prediction through forward propagation.

```
    # 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)))
```

To support the basic forward and backpropagation settings in neural networks, we can pre-declare intermediate variables at the constructor, serving the step-by-step computation through each respective layer.

```
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
```

Using the installed neural network object class to train the network 1000 times, with a learning rate of 0.1, save the weights after the training is complete.

```
# 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")
```

Generate sample data for training and prediction to test the installed network model, which can preprocess the data simply by normalizing the values to the ratio to the maximum value.

```
# 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)
```

Students try to install, use the installed network, then change the basic parameters of the network: learning speed, size of layers, number of hidden layers... and conduct experiments to observe the results.

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

```
#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