


Lab with PyTorch

Objective: By the end of this lab, you'll be able to:

1. Train a fully connected neural network model from scratch in PyTorch.
2. Save your trained model to your disc and load that from disc.
3. Classify the MNIST test data set with your trained model.
4. Realize the effect of different hyper-parameters (number of layers, number of neurons in each layer, learning rate, number of epochs, optimization technique, regularization technique(e.g., adding dropout layer, dropout rate), etc.) on model accuracy.
5. Realize the effect of pruning on model accuracy.

Software Used: You are suggested to use  PyCharm Python IDE. If you are installing the software in your laptop/PC, make sure to install the “free” community version. The software is already installed in Broun 123.

See Appendix at the end of this document on installing PyCharm and running the code.

Given Codes:

- [1] Lab1_Training_4layer_FC_NN.py
- [2] Lab1_Load_PreTrained_Prune_Inference.py

[1] Defining and Training a model:

a. Import necessary python packages (e.g., numpy, torch, torchvision, state_dict etc.). These are for directly using their builtin functions and modules throughout the code.

```
##-----Importing necessary packages-----  
import numpy as np  
import torch  
import torchvision  
import matplotlib.pyplot as plt  
from time import time  
from torchvision import datasets, transforms  
from torch import nn, optim  
  
##-----
```

b. Define the model & Loss function. You will create a blank skeleton of your architecture by defining how many layers, what type of layers, number of neurons in each layer, input size, output size, etc. you want to have in your model. During training, after each iterations the weights and biases will be updated and this skeleton will hold the learned parameters. In the example below, a model with two hidden layers, each having 64 neurons, input and output layer with 784 and 10 neurons respectively is defined. The example uses ReLU activation function for the internal linear layers and uses Softmax activation function for the output layer as it is a classification problem.

```
##-----Defining & Printing the model-----
input_size = 784
hidden_sizes = [64, 64]
output_size = 10

model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.LogSoftmax(dim=1))

print(model)
##-----
```

Counting the layers in NN while defining the model structure: When defining the model structure in Pytorch, the input layer is not defined as the *nn.Linear* layer. Only the hidden layers and output layer are defined as the *nn.Linear* layer. Because, the input layer is not actually a layer, it is just a set of inputs.

Defining Loss function: In this example, we are using NegativeLog-likelihood loss function. Together with LogSoftmax it works like a cross-entropy loss function.

```
##-----Defining the loss function and calculating the loss of input images-----
criterion = nn.NLLLoss()
images = images.view(images.shape[0], -1)      #flattening the image into a 1d tensor
logps = model(images)                          #log probabilities
loss = criterion(logps, labels)                 #calculate the NLL loss
##-----
```

c. **Downloading and processing the dataset.** We download the MNIST dataset from Yann Lecun's website (<http://yann.lecun.com/exdb/mnist/>) by using the *torchvision.datasets* class. The dataset is shuffled and loaded into the DataLoader according to the user defined batch size. The length of the loader will adapt to the *batch_size*. For example, If the train dataset has 60000 samples and the *batch_size* is 64, the loader will have the length of $60000/64 = 938$ where the last batch will contain 32 images.

Note: Replace the “set_path” in the given code with your computers download location.

```
##-----downloading the trainig and testing dataset and storing them in DataLoader-----
trainset = datasets.MNIST(r"set_path", download=True, train=True, transform=transform)
valset = datasets.MNIST(r"set_path", download=True, train=False, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True)
####-----
```

The downloaded data is processed by using the pytorch's builtin class *torchvision.transforms*.
transforms.ToTensor() — converts the image into Tensors of pixel values and
transforms.Normalize() — normalizes the tensors with a mean and standard deviation of user defined value.

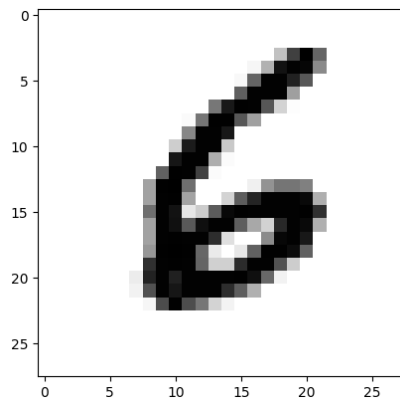
```
##transform function converts the image into tensor and normalize the tensor
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)),
                                ])

```

You can print the size of your processed data and display the image using the following lines of code:

```
##-----Printng the shape of the images & Labels and Displaying a single image from the dataset--
images, labels = next(iter(trainloader))
print(images.shape)
print(labels.shape)
plt.imshow(images[0].numpy().squeeze(), cmap='gray_r')
plt.show(block = False)
plt.pause(4)
plt.close()
##-----
```

Single training data:



You can also display a bunch of images from your training data set:

```
##-----Displaying 60 images at a time from the dataset-----
num_of_images = 60
for index in range(1, num_of_images + 1 ):
    plt.subplot(6, 10, index)
    plt.axis('off')
    plt.imshow(images[index].numpy().squeeze(), cmap='gray_r')
plt.show(block = False)
plt.pause(4)
plt.close()
##-----
```

Grid of training data:

```

1 2 4 7 9 3 2 4 7 9
7 7 8 9 7 5 0 9 1 8
7 2 5 6 9 3 5 6 2 7
4 3 7 7 4 0 3 0 5 4
4 8 2 5 8 7 6 3 0 1
1 6 0 0 9 0 4 1 4 3

```

d. Train the model. In this step you will define necessary hyperparameters to train your model. Define the epochs, learning rate, optimizer etc. and iterate the model over training data to learn and update the weights.

```
##-----Setting the hyperparameters-----
optimizer = optim.Adam(model.parameters(), lr = 0.0001, betas= (0.9, 0.999)) # setting the optimizer, le
time0 = time()
epochs = 10
E = []
L = []
for e in range(epochs):
    running_loss = 0
    c = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1) # Flatten MNIST images into a 784 long vector

        # Training pass
        optimizer.zero_grad() # setting the gradients to zero before starting backprop each time
        output = model(images)
        loss = criterion(output, labels)
        loss.backward() # This is where the model learns by backpropagating
        #print('Weights after each backprop:', model[0].weight)
        optimizer.step() # optimizes the weights

        running_loss += loss.item()
        c = c + 1
    else:
        print("Epoch {} - Training loss: {}".format(e, running_loss / len(trainloader)))
        E.append(e)
        L.append(running_loss / len(trainloader))

print("Iterations in one Epoch", c)
print("\nTraining Time (in minutes) =", (time() - time0) / 60)
```

Using an else block without an if: Python3 allows an else block without an if! The structure is: at the last iteration of the for loop the else block will be executed. In this case, the else block will be executed at the last iteration of the inner for loop. This prints the loss after each epoch.

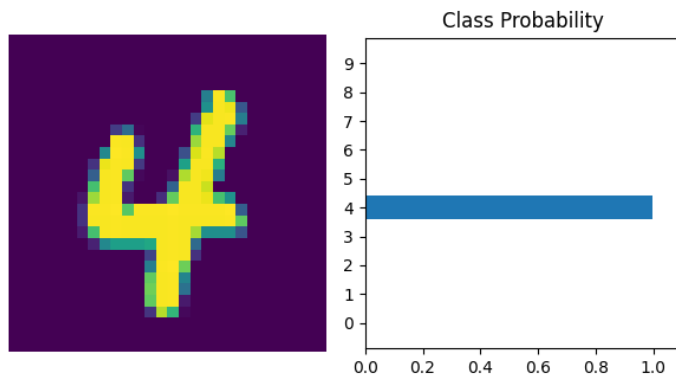
e. Testing the model. You evaluate how much your model learned testing over the test dataset. Now the model is in inference phase, so we do not need to calculate the gradient of loss. Therefore, we used `torch.no_grad()` and the model will return its prediction on the given input image.

```
## Finding the prediction of a single image-----
images, labels = next(iter(valloader))
img = images[0].view(1, 784)
with torch.no_grad():
    logps = model(img)
    ps = torch.exp(logps)
    probab = list(ps.numpy()[0])
```

You can also see the input image and the model's prediction on it using:

```
print("Predicted Digit =", probab.index(max(probab)))
view_classify(img.view(1, 28, 28), ps)
"
```

The image looks like:



Where, `view_classify()` is the function for viewing the image and its predicted classes

```
##view_classify function shows the image and model's prediction probability on it
def view_classify(img, ps):
    ''' Function for viewing an image and it's predicted classes.
    '''
    ps = ps.data.numpy().squeeze()

    fig, (ax1, ax2) = plt.subplots(figsize=(6,9), ncols=2)
    ax1.imshow(img.resize_(1, 28, 28).numpy().squeeze())
    ax1.axis('off')
    ax2.barh(np.arange(10), ps)
    ax2.set_aspect(0.1)
    ax2.set_yticks(np.arange(10))
    ax2.set_yticklabels(np.arange(10))
    ax2.set_title('Class Probability')
    ax2.set_xlim(0, 1.1)
    plt.tight_layout()
    plt.show(block = False)
    plt.pause(3)
    plt.close()
```

You can also find out how many images out of the whole testing data your model can correctly classify, i.e., the accuracy rate of your model.

```
##-----testing all images from the test folder & calculating model accuracy--
correct_count, all_count = 0, 0
for images, labels in valloader:
    for i in range(len(labels)):
        img = images[i].view(1, 784)
        with torch.no_grad():
            logps = model(img)

        ps = torch.exp(logps)
        probab = list(ps.numpy()[0])
        pred_label = probab.index(max(probab))
        true_label = labels.numpy()[i]
        if (true_label == pred_label):
            correct_count += 1
        all_count += 1

print("Number Of Images Tested =", all_count)
print("\nModel Accuracy =", (correct_count / all_count) * 100, '%')
####-----End of Validation Phase-----
```

f. Save the model. When you are satisfied with your trained model, you can save it and use it for future use. Usually, in pytorch the trained model is saved with *.pt* or *.pth* extension.

```
##-----Saving the tarined model-----  
torch.save(model, './trained_model.pt')  
##-----
```

The full code for training and saving the trained model is uploaded in canvas (*Training_4layer_FC_NN.py*). Change the path of trainset & valset to reflect your directory, run and understand the code.

g. Load the pretrained model. Loading the pretrained model. To load a pretrained model from your disc, at first you need to define the model architecture as in step b, then you can load the learned model by using the following command:

```
##-----Loading the pretrained model-----  
model = torch.load('trained_model.pt')  
print (model)  
###-----
```

However, you can download a pretrained model from pytorch repository just using the *models.model_name(pretrained = True)* command. In that case, defining the model architecture is not needed.

Change the directory of trainset & valset to reflect your directory and run *Lab1_Load_PreTrained_Prune_Inference.py* uploaded in canvas.

2. Pruning the model Parameter. The weights below a certain threshold are converted to 0 to reduce the computational complexity of the model in inference phase. To prune a model all at once, you need to define which parameters you want to prune and the pruning rate.

```
##-----Pruning the model-----  
parameters_to_prune = ((model[0], 'weight'), (model[2], 'weight'), (model[4], 'weight'))  
prune.global_unstructured(parameters_to_prune, pruning_method=prune.L1Unstructured, amount = 0.6)
```

This is known as Global Pruning, where lowest 60% of the weights will be removed from each layer. In case of Global Pruning, the pruning percentage of different layer may be different as the size of different layer is different. You can see the sparsity of each layer and total model sparsity using the following codes:

```

##-----Checking the sparsity of weight in each layer-----
print(
    "Sparsity in Linear Layer 1: {:.2f}%".format(
        100. * float(torch.sum(model[0].weight == 0))
        / float(model[0].weight.nelement())
    )
)
print(
    "Sparsity in Linear Layer 2: {:.2f}%".format(
        100. * float(torch.sum(model[2].weight == 0))
        / float(model[2].weight.nelement())
    )
)
print(
    "Sparsity in Linear Layer 3: {:.2f}%".format(
        100. * float(torch.sum(model[4].weight == 0))
        / float(model[4].weight.nelement())
    )
)
print(
    "Global Sparsity: {:.2f}%".format(100. * ((float(torch.sum(model[0].weight == 0)) +
        (float(torch.sum(model[2].weight == 0))) +
        (float(torch.sum(model[4].weight == 0))))
        / (float(model[0].weight.nelement()) +
        float(model[2].weight.nelement()) +
        float(model[4].weight.nelement()))
)
)
####

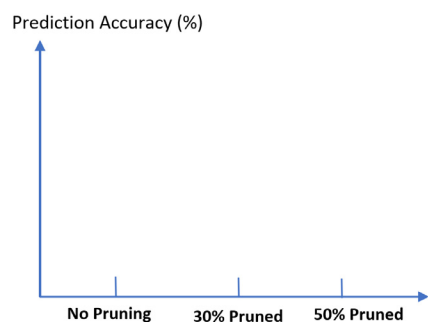
```

Run *Lab1_Load_PreTrained_Prune_Inference.py* uploaded in canvas.

Tasks

After completing the above steps, do the following tasks:

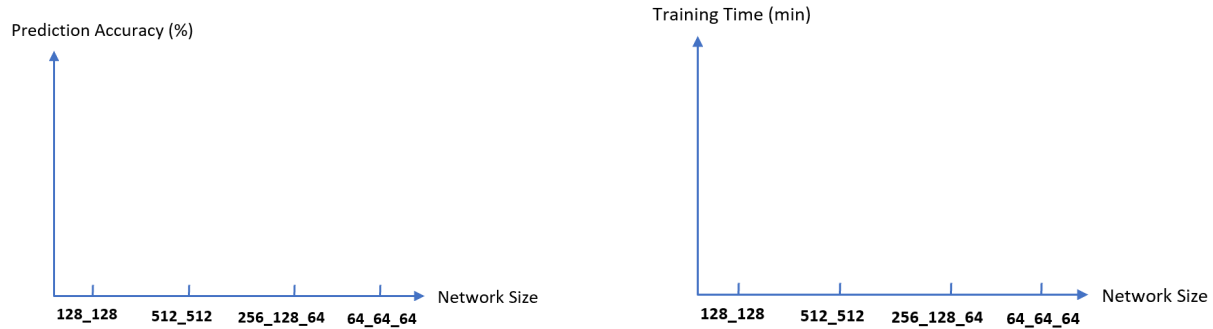
Task 1: Vary the pruning rate at inference time and generate a bar chart like below.



Task 2: Train multiple networks by varying the number of hidden layers and number of neurons in hidden layers, such as:

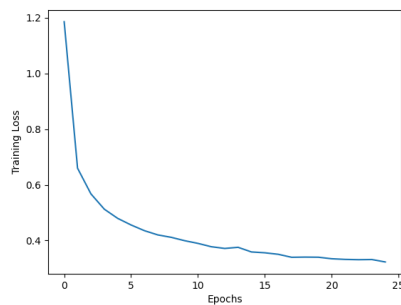
- ❖ 2 hidden layers each having 128 neurons,
- ❖ 2 hidden layers each having 512 neurons,
- ❖ 3 hidden layers: 1st, 2nd, and 3rd layer having 256, 128, and 64 neurons, respectively.
- ❖ 3 hidden layers: each having 64 neurons.

Train the above networks for 20 epochs. Record the **training time** and **accuracy** for each network and create bar chart like below.



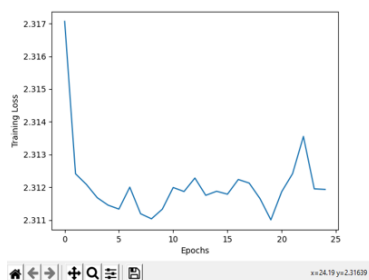
Comparing the accuracy and training time, among the networks you trained, which network you think is the best. Explain your answer.

Task 3: Pick the best network from task 2 and plot the training loss vs epoch like below.

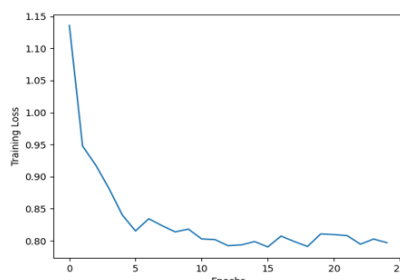


Task 4: So far, you were using Adam optimizer with learning rate 0.0001. Now use stochastic Gradient descent optimizer. Train your best model from task 2 for learning rate [0.3, 0.03, 0.003, 0.0003]. Plot these results separately. (Must use **stochastic gradient descent** optimizer in this task and **Adam** for rest tasks).

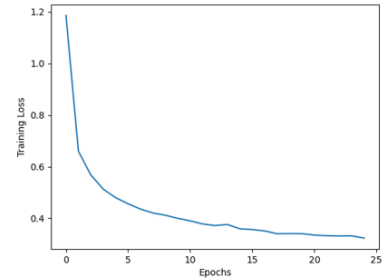
Sample figure:



Learning rate = 0.3, 2 hidden layers each having 64 neurons



Learning rate = 0.03, 2 hidden layers each having 64 neurons

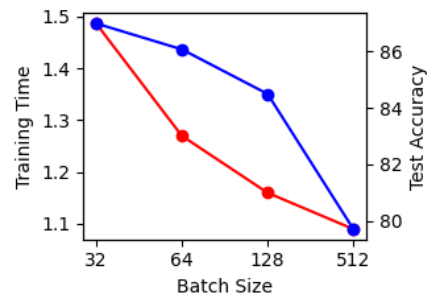


Learning rate = 0.003, 2 hidden layers each having 64 neurons

Explain your results.

Task 5: Again take the best model from task 2 and train this model with batch size [32, 128, 512]. Record the test accuracy and training time. Plot the results including the results for batch size of 64 (you

already have this result from previous tasks). Form a correlation between **batch-size, training time and test accuracy**. The plot might look like this:



Task 6: Notice, you did not use the dropout technique in this small network to prevent overfitting. Add dropout layer after the hidden layers for “256-256-256” size model, and continue the training process. Compare the results w/ and w/o dropout. Explain your results. You can add dropout layer like below:

```
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Dropout(p = 0.5, inplace = True),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Dropout(p = 0.5, inplace = True),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.LogSoftmax(dim=1))
```

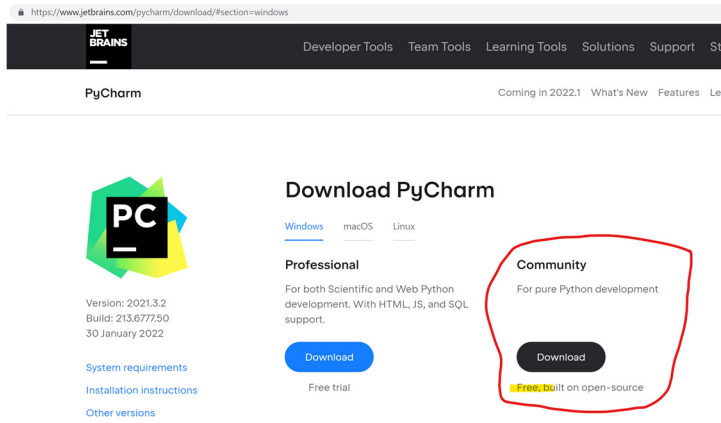
[N.B.: The plots of this sheet are sample plots. The results might vary in your cases.]

Deliverable: You should submit the plots and brief explanations for each task. And Screenshot of your codes output. [Submit a Single PDF in CANVAS.](#)

Appendix: Creating Project and Running Code in PyCharm

Install PyCharm:

For Windows Machine:



Install PyTorch:

Go to site: <https://pytorch.org/get-started/locally/>

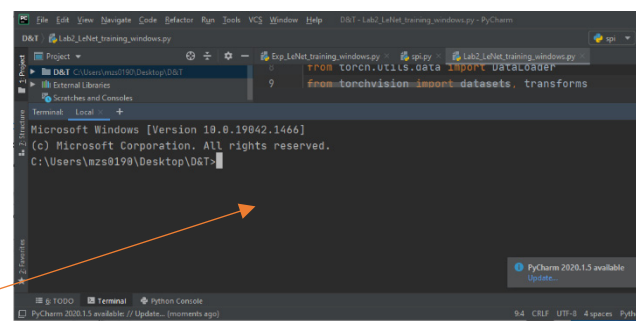
If you do not have GPU select CPU-only:

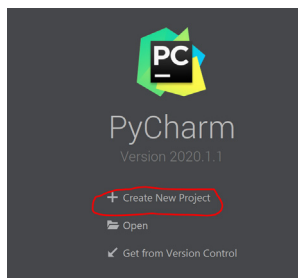
Additional support or warranty for some PyTorch Stable and LTS binaries are available through the [PyTorch Enterprise Support Program](#).

PyTorch Build	Stable (1.10.2)	Preview (Nightly)	LTS (1.8.2)
Your OS	Linux	Mac	Windows
Package	Conda	Pip	LibTorch
Language	Python	C++ / Java	Source
Compute Platform	CUDA 10.2	CUDA 11.3	ROCM 4.2 (beta)
Run this Command:	pip3 install torch torchvision torchaudio		

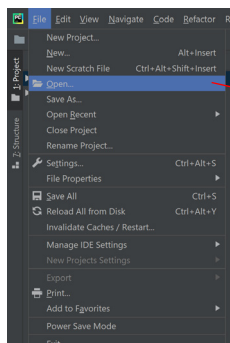
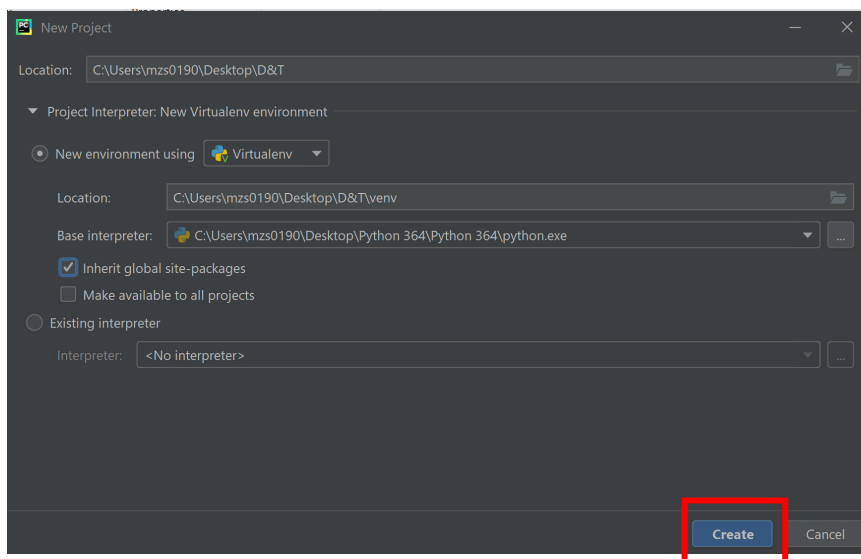
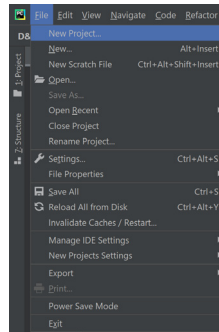
If your computer has GPU then select “CUDA”

PyTorch Build	Stable (1.10.2)	Preview (Nightly)	LTS (1.8.2)
Your OS	Linux	Mac	Windows
Package	Conda	Pip	LibTorch
Language	Python	C++ / Java	Source
Compute Platform	CUDA 10.2	CUDA 11.3	ROCM 4.2 (beta)
Run this Command:	pip3 install torch=1.10.2+cu113 torchvision=0.11.3+cu113 torchaudio==0.10.2+cu113 -f https://download.pytorch.org/whl/cu113/torch_stable.html		

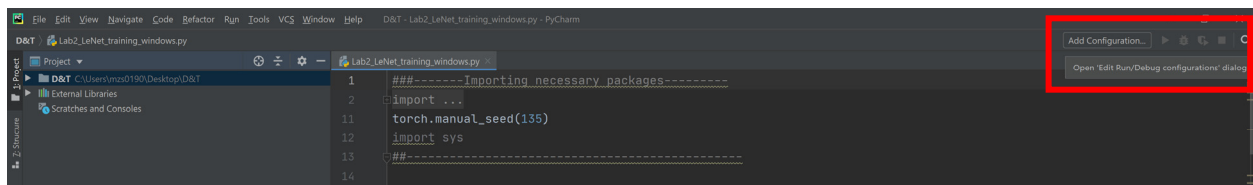


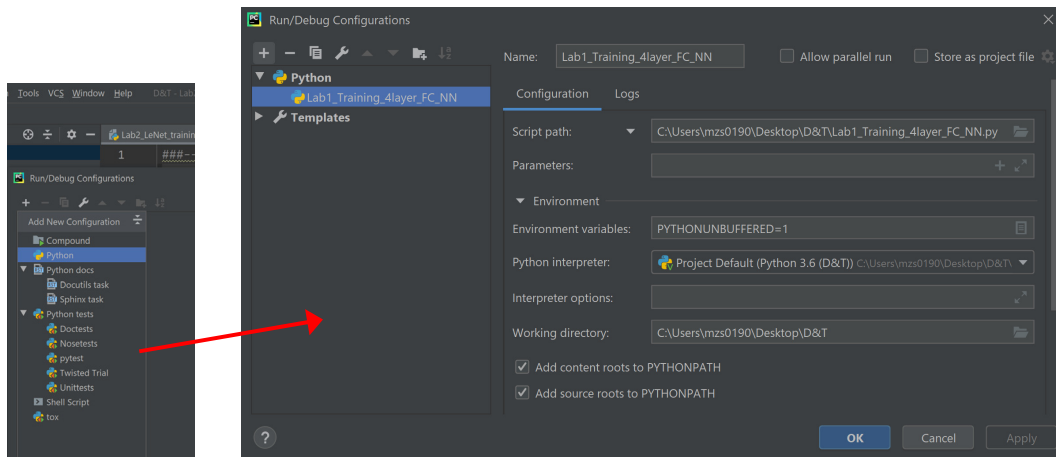


OR,



Open the "Python" file

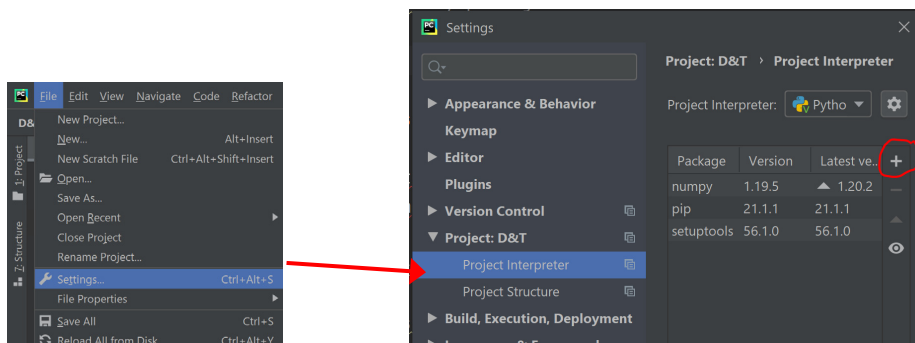




Adding packages in PyCharm:

```
C:\Users\mzs0190\Desktop\D&T\venv\Scripts\python.exe C:/Users/mzs0190/Desktop/D&T/Lab1_Training_4layer_FC_NN.py
Traceback (most recent call last):
  File "C:/Users/mzs0190/Desktop/D&T/Lab1_Training_4layer_FC_NN.py", line 3, in <module>
    import torch
ModuleNotFoundError: No module named 'torch'
```

If above error is observed about missing package, do the following:



Click the “+” button

Type the name of the missing package and “Install”

