# CPSC 552 – Assignment #10

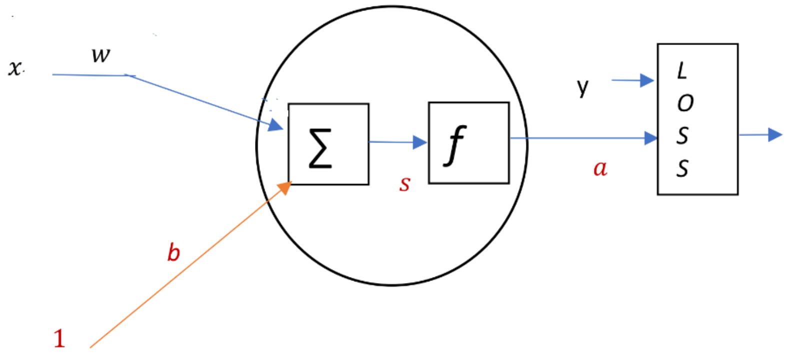
## Programming Neural Networks in PyTorch

PyTorch is one of the most popular libraries for programming deep neural networks. The typical program structure is divided into three files. One is the Utils.py where we write the code for creating the data loaders that return a batch of training or test data. The second file describes the neural network architecture, and the third file has the main with training and test code in it. We will create a few different type of networks to demonstrate the PyTorch concepts.

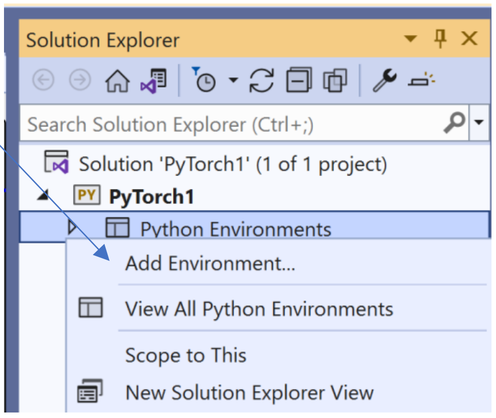
## Installing PyTorch:

Use the handout on "Getting Stated - Software Installation Instructions for Python, Pytorch in Visual Studio". The link is available under the handouts menu in the CPSC552 web site on kiwi.

**Problem 1**: Creating a simple one neuron program in pytorch. The neuron will learn to do regression for the equation of a line, y = 2 x + 0.3. Even though the weight and the bias will be initialized randomly, through the training data, the neuron will be able to learn weight of 2 and the bias of 0.3.

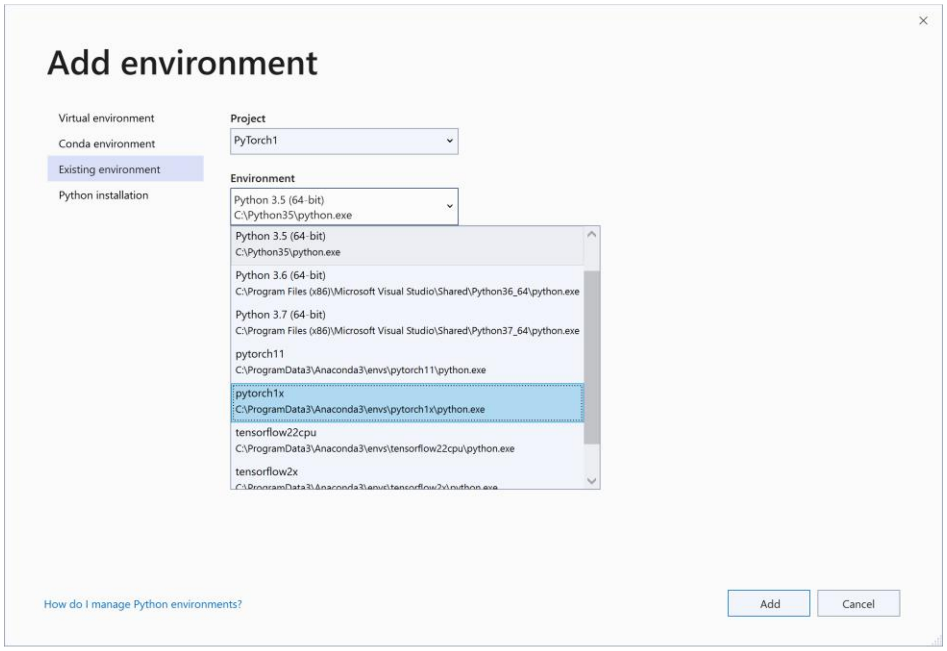


Create a Python application called "Pytorch1". Select the environment to be "pytorch1x" by right clicking on Python Environments and selecting "Add Environment" as shown below.



Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'PyTorch1' (1 of 1 project)
PyTorch1
Python Environments
Add Environment...
View All Python Environments
Scope to This
New Solution Explorer View

Then select "pytorch1x" and click the Add button.



Add environment

Virtual environment
Conda environment
Existing environment
Python installation

Project
PyTorch1

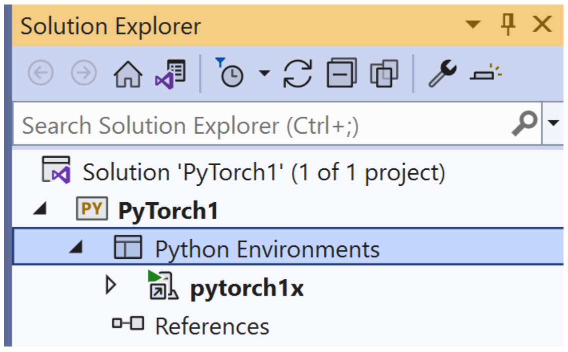
Environment
Python 3.5 (64-bit)
C:\Python35\python.exe

Python 3.5 (64-bit)
C:\Python35\python.exe
Python 3.6 (64-bit)
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Python 3.7 (64-bit)
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
pytorch11
C:\ProgramData3\Anaconda3\envs\pytorch11\python.exe
pytorch1x
C:\ProgramData3\Anaconda3\envs\pytorch1x\python.exe
tensorflow22cpu
C:\ProgramData3\Anaconda3\envs\tensorflow22cpu\python.exe
tensorflow2x
C:\ProgramData3\Anaconda3\envs\tensorflow2x\python.exe

How do I manage Python environments?

Add
Cancel

Now in Visual Studio, the pytorch1x environment will be highlighted, as shown below.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'PyTorch1' (1 of 1 project)
PyTorch1
Python Environments
pytorch1x
References

Add a file called MyDataSet.py to the project with the following code.

```python
from torch.utils.data import Dataset, TensorDataset
import torch

class MyDataSet(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)
```

The purpose of the dataset is to store the training or the test data in it, and return the input x, and the corresponding expected output y as needed by the associated data loader.

Add a file called SimpleModel.py with the following code in it. This class contains the code for the w and b parameters of the neuron. The forward function indicates how the neuron produces the output if an input x is given to it. The requires_grad=True indicates to pytorch, that it should compute the gradients of the loss with respect to the w and the b parameters.

```
import torch

class SimpleModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.w = torch.nn.Parameter(torch.randn(1, requires_grad=True,
dtype=torch.float))
        self.b = torch.nn.Parameter(torch.randn(1, requires_grad=True,
dtype=torch.float))

    def forward(self, x):
        return self.w * x + self.b
```

Add a file called Utils.py with the following code in it.

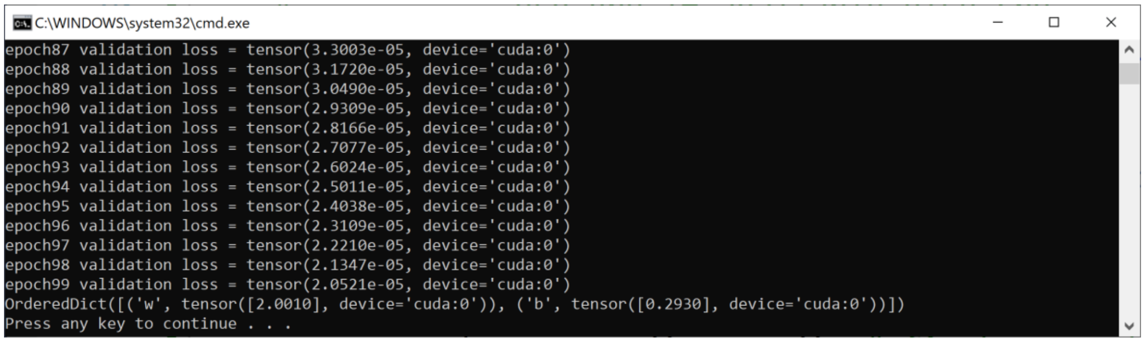
```
import torch
from MyDataSet import MyDataSet
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
import numpy as np

def get_train_val_loaders(batch_size):
    x_train = np.arange(0,10)
    y_train = x_train * 2 + 0.3  # y = 2x + 0.3
    x_train_tensor = torch.from_numpy(x_train).float()
    y_train_tensor = torch.from_numpy(y_train).float()
    mydataset = MyDataSet(x_train_tensor, y_train_tensor)
    train_dataset, val_dataset = random_split(mydataset, [8, 2])

    train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size)
    val_loader = DataLoader(dataset=val_dataset, batch_size=batch_size)
    return train_loader, val_loader
```

The above file loads the x and y data in the dataset (total of 10 pairs), then creates two data loaders by splitting the data into 8 and 2 items.

The code in the main file pytorch1.py appears as:

```
import sys
import numpy as np
import torch
import random
from SimpleModel import SimpleModel
import Utils

def main():
    z = np.arange(100)
    #----------- use gpu if available else cpu------------
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    train_loader, val_loader = Utils.get_train_val_loaders(2)

    losses = []
    val_losses = []
    lr = 1e-2
    n_epochs = 100
    loss_fn = torch.nn.MSELoss(reduction='mean')
    model = SimpleModel().to(device)
    model.train() # set the model in train mode

    # tell optimizer to optimize the model paramaters, specify learning rate
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    for epoch in range(n_epochs):
        for x_batch, y_batch in train_loader:
            x_batch = x_batch.to(device)  # load data in GPU if available
            y_batch = y_batch.to(device)
            aout = model(x_batch) # implicitly calls forward function in model
            loss = loss_fn(y_batch, aout)

            loss.backward()  # compute gradients
            optimizer.step() # update weights, biases
            optimizer.zero_grad() # clear gradients
            losses.append(loss)

        # do validation on the learned model so far
        with torch.no_grad():  # turn off gradient calculation
            for x_val, y_val in val_loader:
                x_val = x_val.to(device)
                y_val = y_val.to(device)
                model.eval() # set model to evaluation mode

                aout = model(x_val)
                val_loss = loss_fn(y_val, aout)
                val_losses.append(val_loss.item())
                print('epoch' + str(epoch) + ' validation loss = ' + str(val_loss))

    print(model.state_dict()) # print final model parameters


if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

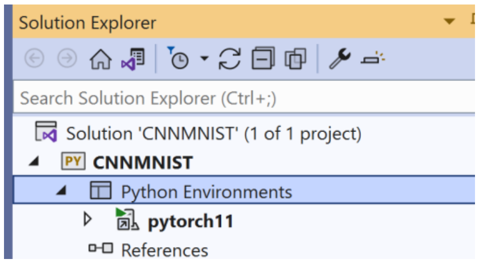
If you run the program, you will see that the neuron is able to learn a value of close to 2 for the ***w*** and close to 0.3 for the ***b***.



```
epoch87 validation loss = tensor(3.3003e-05, device='cuda:0')
epoch88 validation loss = tensor(3.1720e-05, device='cuda:0')
epoch89 validation loss = tensor(3.0490e-05, device='cuda:0')
epoch90 validation loss = tensor(2.9309e-05, device='cuda:0')
epoch91 validation loss = tensor(2.8166e-05, device='cuda:0')
epoch92 validation loss = tensor(2.7077e-05, device='cuda:0')
epoch93 validation loss = tensor(2.6024e-05, device='cuda:0')
epoch94 validation loss = tensor(2.5011e-05, device='cuda:0')
epoch95 validation loss = tensor(2.4038e-05, device='cuda:0')
epoch96 validation loss = tensor(2.3109e-05, device='cuda:0')
epoch97 validation loss = tensor(2.2210e-05, device='cuda:0')
epoch98 validation loss = tensor(2.1347e-05, device='cuda:0')
epoch99 validation loss = tensor(2.0521e-05, device='cuda:0')
OrderedDict([('w', tensor([2.0010], device='cuda:0')), ('b', tensor([0.2930], device='cuda:0'))])
Press any key to continue . . .
```

**Problem 2**: Creating a Neural network with layers of neurons to do classification of data. The first version will use two hidden layers and one output layer. The second version will use two CNN layers, followed by a two layer neural network to do the classification. MNIST dataset has 60000 training images of 0-9 digits with each didgit being 28x28 pixels gray scale image. There are 10000 test images.

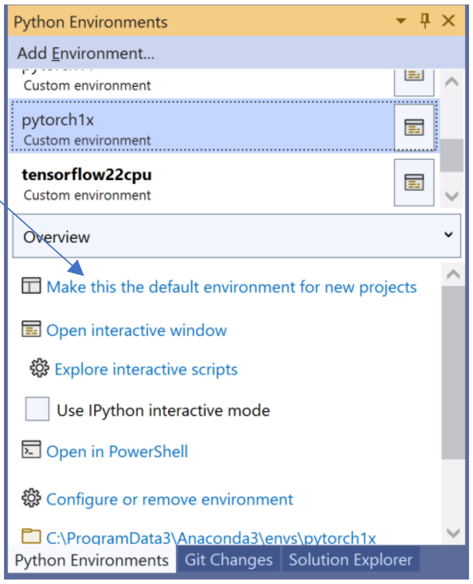
Create a python application called CNNMNIST. Make sure the environment for the project is set to "pytorch1x". Otherwise, right click on the Python Environments, and choose add environment, then existing environment to select the pytorch1x. If multiple environments appear under python environment, then right click on the unwanted environment and choose rmove.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'CNNMNIST' (1 of 1 project)
CNNMNIST
Python Environments
pytorch11
References

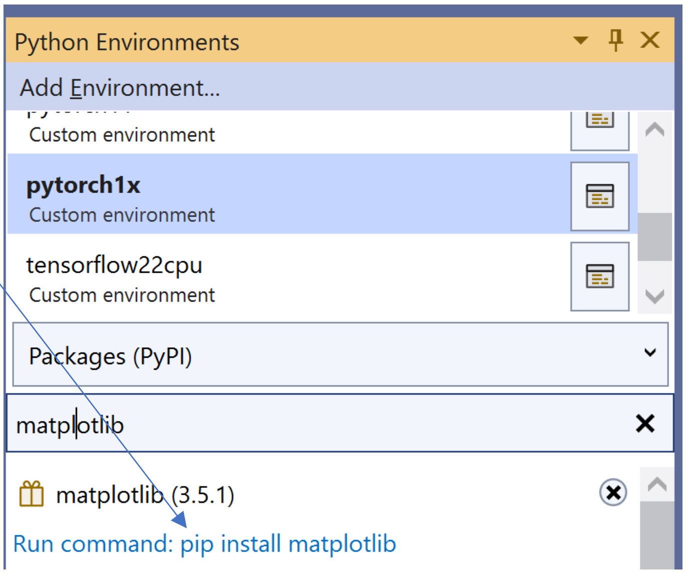
For MNIST digit classification, we will use the standard dataset and dataloader provided by the pytorch library. Add a file called Utils.py to the project with the following code in it.

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

def get_loaders(batch_size):
    train_dataset = torchvision.datasets.MNIST(root='./data',
                                        train=True,
                                    transform=transforms.ToTensor(),
                                        download=True)
    test_dataset = torchvision.datasets.MNIST(root='./data',
                                        train=False,
                                        transform=transforms.ToTensor())
    # Data loader
    train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                        batch_size=batch_size,
                                        shuffle=True)
    test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                        batch_size=batch_size,
                                        shuffle=False)
    return train_loader, test_loader

def plot_images(loader):
    mnistdata = iter(loader)
    digit, label = mnistdata.next()
    for i in range(6):
        plt.subplot(2,3,i+1)
        plt.imshow(digit[i][0], cmap='gray')
    plt.show()
```

Note: You will need to install any missing packages such as matplotlib into the pytorch1x environment. You can do this by going to the View-> Other windows-> Python Environments menu in Visual Studio, then first making the pytorch1x as the default environment as shown below.



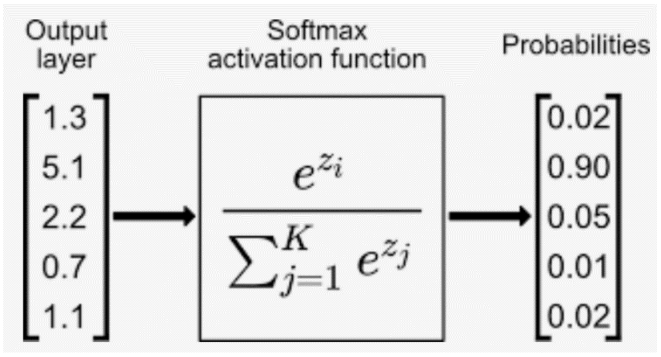
Once pytorch1x is highlighted, then you can change the dropdown to packages (PyPI), and search for matplotlib, then click on the link to install the package as shown below.



Add a file called LinearNetwork.py with the following code in it.

```
import torch
import sys
import torch.nn as nn
import torchvision

# fully connected neural network with two hidden layers
class LinearNetwork(nn.Module):
     def __init__(self, input_size, hidden_size1, hidden_size2, num_classes):
         super(LinearNetwork, self).__init__()
         self.input_size = input_size
         self.l1 = nn.Linear(input_size, hidden_size1)
         self.relu = nn.ReLU()
         self.l2 = nn.Linear(hidden_size1, hidden_size2)
         self.l3 = nn.Linear(hidden_size2, num_classes)
         self.smax = nn.Softmax(dim=1)

     def forward(self, x):
         out = self.l1(x)
         out = self.relu(out)
         out = self.l2(out)
         out = self.relu(out)
         out = self.l3(out)
         out = self.smax(out)
         return out
```

As you can see from the above code, the init function declares the building blocks needed to create the network, and the forward function indicates how the output will be produced when a batch of input data is passed through the network. Because we are using linear networks on the input, the 2-d MNIST digit data (28x28) has be to be converted to a 28x28 = 784x1 vector before passing to the above network. Thus the input_size in creating the above network has to be 784, and the num_classes to be 10 for MNIST. For multiclass classification, we often use softmax function on the output layer to convert the outputs to a probability distribution so that the sum of all outputs is one, and the output that is being recognized is enhanced further as compared to the rest of the outputs. For example, if the network produces 5 outputs, and as we pass these five outputs through the softmax function, the sum of the outputs will be 1, and the output that is being recognized will be made larger with respect to the other outputs. The formula for generating each output is a simple exponential function as shown below.

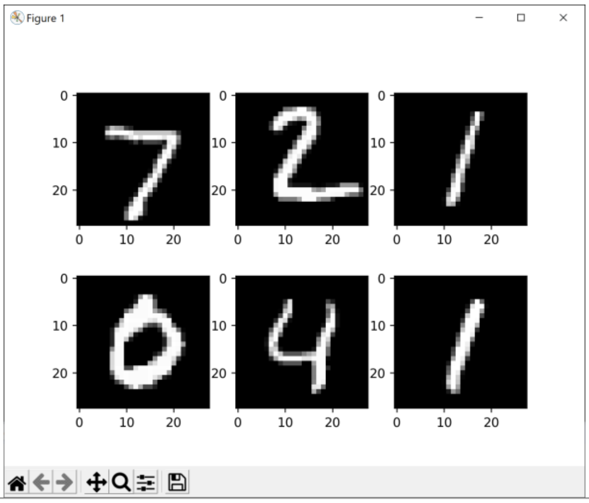

Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

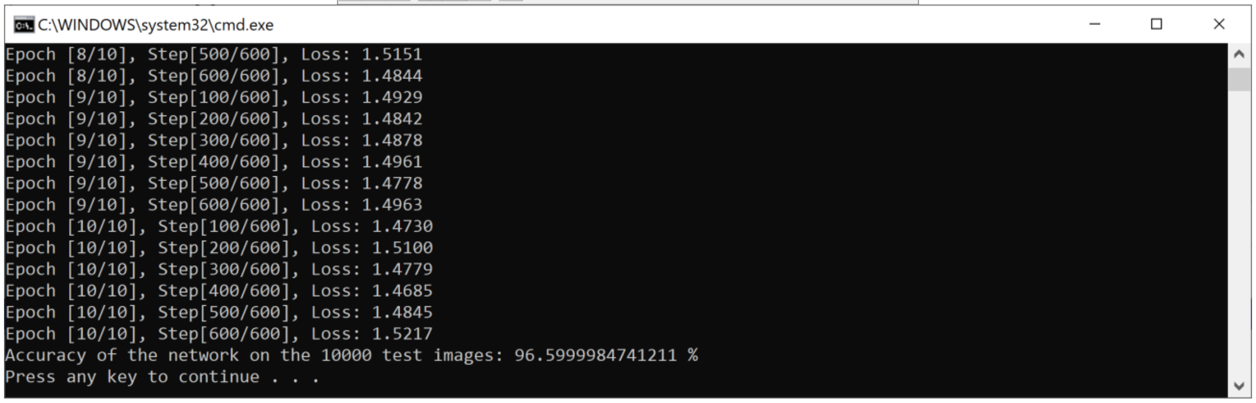
Add a file to the project called LinearMNISTTrain.py. Then type the following code in it.

```
import torch
import sys
import torch.nn as nn
from LinearNetwork import LinearNetwork
import Utils


def main():
    input_size = 784 # 28x28
    hidden_size1 = 100
    hidden_size2 = 50
    num_classes = 10
    num_epochs = 10
    batch_size = 100
    learning_rate = 0.001

    train_loader, test_loader = Utils.get_loaders(batch_size)
    Utils.plot_images(test_loader)

    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    model = LinearNetwork(input_size, hidden_size1, hidden_size2,
num_classes).to(device)

    # Loss and optimizer
    criterion = nn.CrossEntropyLoss() # for multiclass
    # classification, cross entropy loss works better
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    n_total_steps = len(train_loader)
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_loader):
            # original shape: [100, 1, 28, 28]
            # resized: [100, 784]
            images = images.reshape(-1, 28*28).to(device)
            labels = labels.to(device)

            outputs = model(images) # calls forward function

            loss = criterion(outputs, labels)
            optimizer.zero_grad() # clear gradients
            loss.backward() # compute gradients
            optimizer.step() # update weights and biases
            if (i+1) % 100 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}],
Step[{i+1}/{n_total_steps}], Loss: {loss.item():.4f}')

    # compute accuracy on test set
    with torch.no_grad():
        num_correct = 0
        num_samples = 0
        for xt, labels in test_loader:
            xt = xt.reshape(-1, 28*28).to(device)
            labels = labels.to(device)
            outputs = model(xt)
            _, predicted = torch.max(outputs, 1) # returns max,max_indices
            num_samples += labels.size(0)
            num_correct += (predicted == labels).sum()

    acc = 100.0 * num_correct / num_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Study the above code carefully to understand how the training and testing of the trained network is carried out.

Set the LinearMNISTTrain.py as the start file, and run the program. The output will appear as:




C:\WINDOWS\system32\cmd.exe

```
Epoch [8/10], Step[500/600], Loss: 1.5151
Epoch [8/10], Step[600/600], Loss: 1.4844
Epoch [9/10], Step[100/600], Loss: 1.4929
Epoch [9/10], Step[200/600], Loss: 1.4842
Epoch [9/10], Step[300/600], Loss: 1.4878
Epoch [9/10], Step[400/600], Loss: 1.4961
Epoch [9/10], Step[500/600], Loss: 1.4778
Epoch [9/10], Step[600/600], Loss: 1.4963
Epoch [10/10], Step[100/600], Loss: 1.4730
Epoch [10/10], Step[200/600], Loss: 1.5100
Epoch [10/10], Step[300/600], Loss: 1.4779
Epoch [10/10], Step[400/600], Loss: 1.4685
Epoch [10/10], Step[500/600], Loss: 1.4845
Epoch [10/10], Step[600/600], Loss: 1.5217
Accuracy of the network on the 10000 test images: 96.5999984741211 %
Press any key to continue . . .
```

The accuracy of recognition can be improved by using convolutions in learning distinguishable features before passing to a classifier. Since MNIST digits are 2-D images, we need to use 2-D convolutions. Add a class CNNNetwork with the following code in it.

```
import torch
import sys
import torch.nn as nn
import torchvision

# CNN network followed by a with one hidden layer classifier
class CNNNetwork(nn.Module):
     def __init__(self, hidden_size1,  num_classes):
        super(CNNNetwork, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5) # 6 feature maps, 5x5 conv.
        # 1 is the number of channels. This is because of gray scale image
        self.pool = nn.MaxPool2d(2,2)
        # we use the maxpool multiple times, but define it once
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(6,8,3)
        # 6 channels, 8 feature maps, and 3x3 convolution

        self.fc1 = nn.Linear(8*5*5, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, num_classes)
        self.smax = nn.Softmax(dim=1)

     def forward(self, x):
         out = self.conv1(x)
         out = self.pool(self.relu(out))
         out = self.conv2(out)
         out = self.pool(self.relu(out))
         flatten = out.view(-1,8*5*5)
         out = self.fc1(flatten)
         out = self.relu(out)
         out = self.fc2(out)
         out = self.smax(out)
         return out
```

Input of 28x28 => Conv2d(1,6, 5)    will produce 6 features maps, each of 28-5+1 x 28-5+1 = 24x24 size

MaxPool2d(2,2) => will result in 6 feature maps being reduced to 12x12 size

Conv2d(6, 8, 3) => The first number 6 indicates that there are 6 input channels. This is because the previous 2-DConv produced 6 feature maps. Here, we are producing 8 feature maps and doing 3x3 convolutions on each previous feature map. Since the previous feature map is 12x12 size, as we do a 3x3 convolution, it will result in 10x10 feature maps.

MaxPool2d(2,2) will then reduced each feature map to 5x5. Since there are 8 feature maps being produced, as we flatten these into a column of numbers, it will come out to be 8*5*5 numbers i.e., 200 numbers. This is accomplished by the following line:

```
flatten = out.view(-1,8*5*5)
```

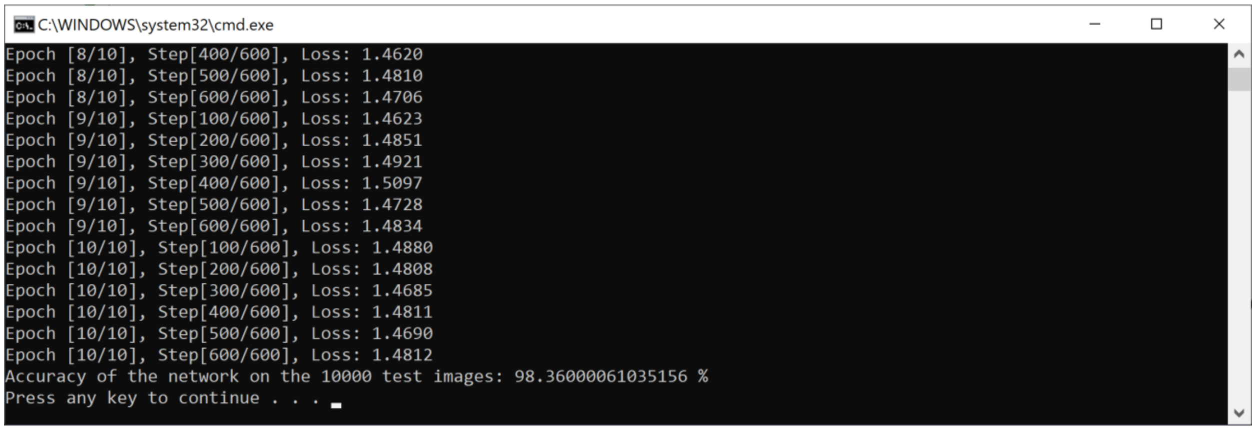
After doing the convolutions, we flatten the resulting feature maps before feeding the latent representation to a linear network based classifier.

Add a file called CNNMNISTTrain.py with the following code in it.

```
import torch
import sys
import torch.nn as nn
from CNNNetwork import CNNNetwork
import Utils

def main():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    hidden_size1 = 100
    num_classes = 10
    num_epochs = 10
    batch_size = 100
    learning_rate = 0.001
    train_loader, test_loader = Utils.get_loaders(batch_size)
    Utils.plot_images(test_loader)

    model = CNNNetwork(hidden_size1, num_classes).to(device)

    criterion = nn.CrossEntropyLoss() # for multiclass
    # classification, cross entropy loss works better
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    num_total_steps = len(train_loader)
    for epoch in range(num_epochs):
        for i, (x, labels) in enumerate(train_loader):
            # x shape: [batchsize, 1, 28, 28]
            x = x.to(device) # convert to CPU or GPU tensor
            labels = labels.to(device)

            pred_outputs = model(x) # calls forward function

            loss = criterion(pred_outputs, labels)
            optimizer.zero_grad() # clear gradients
            loss.backward() # compute gradients
            optimizer.step() # update weights and biases
            if (i+1) % 100 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}],
Step[{i+1}/{num_total_steps}], Loss: {loss.item():.4f}')

    # compute accuracy on test set
    with torch.no_grad():
        num_correct = 0
        num_samples = 0
        for xt, labels in test_loader:
            xt = xt.to(device)
            labels = labels.to(device)
            outputs = model(xt)
            _, predicted = torch.max(outputs, 1) # returns max,max_indices
            num_samples += labels.size(0)
            num_correct += (predicted == labels).sum()

    acc = 100.0 * num_correct / num_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Study the above code carefully to see how the data is fed to the CNNNetwork. For networks that have convolutions on the input side, the format of the data in pytorch is required to be:
[batch size, number of channels, input image width, input image height]. For example, for MNIST data, if the batch size is 100, the input data tensor to be fed to the network will be: [100,1,28,28]. If we were training a CNN network for color images, e.g., CIFAR10 dataset, then the input tensor will be: [100,3,32,32] because CIFAR10 has color images (so 3 channels), and each image is 32x32.

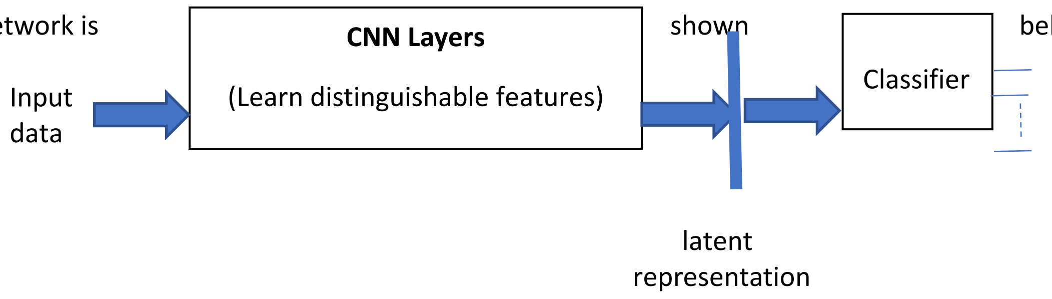
Set the CNNMNISTTrain.py as the start file and run the program. The output appears as:



```
Epoch [8/10], Step[400/600], Loss: 1.4620
Epoch [8/10], Step[500/600], Loss: 1.4810
Epoch [8/10], Step[600/600], Loss: 1.4706
Epoch [9/10], Step[100/600], Loss: 1.4623
Epoch [9/10], Step[200/600], Loss: 1.4851
Epoch [9/10], Step[300/600], Loss: 1.4921
Epoch [9/10], Step[400/600], Loss: 1.5097
Epoch [9/10], Step[500/600], Loss: 1.4728
Epoch [9/10], Step[600/600], Loss: 1.4834
Epoch [10/10], Step[100/600], Loss: 1.4880
Epoch [10/10], Step[200/600], Loss: 1.4808
Epoch [10/10], Step[300/600], Loss: 1.4685
Epoch [10/10], Step[400/600], Loss: 1.4811
Epoch [10/10], Step[500/600], Loss: 1.4690
Epoch [10/10], Step[600/600], Loss: 1.4812
Accuracy of the network on the 10000 test images: 98.36000061035156 %
Press any key to continue . . .
```

Note that if we use CNN layers and then flatten the latent space and feed it to a classifier, the classification accuracy is much better. In this example, we used only 2 layers of convolutions. Deeper CNN architectures can achieve an accuracy of > 99.8%.

The general CNN network is architecture shown of a deep below.



**Problem 3**:
For non-image data, the organization of the data is in rows and columns, where each row represents the data item, and the columns contain the number of features (or the dimensionality of data). For example, if we were to develop a classification system for the for the 801 patients where there are 20531 gene expression data for each patient, and there are five types of cancers, we will need to use 1-D CNNs in the feature learning stages.

Create a python application called CancerClassification. Add a file called MyDataSet.py with the following code in it.

```
from torch.utils.data import Dataset, TensorDataset
import torch

class MyDataSet(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)
```

Add a file called Utils.py with the following code in it.

```
import torch
from MyDataSet import MyDataSet
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
import numpy as np
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
#import tarfile
#import urllib

def get_train_test_loaders(batch_size):
    # download TCGA dataset from UCI
    uci_tcga_url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/00401/"
    archive_name = "TCGA-PANCAN-HiSeq-801x20531.tar.gz"

    # Build the url
    #full_download_url = urllib.parse.urljoin(uci_tcga_url, archive_name)

    # Download the file
    #r = urllib.request.urlretrieve (full_download_url, archive_name)

    # Extract the data from the archive
    #tar = tarfile.open(archive_name, "r:gz")
    #tar.extractall()
    #tar.close()

    datafile = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/data.csv"
    labels_file = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/labels.csv"

    data = np.genfromtxt(
        datafile,
        delimiter=",",
        usecols=range(1, 20532),
        skip_header=1
    )

    true_label_names = np.genfromtxt(
        labels_file,
        delimiter=",",
        usecols=(1,),
        skip_header=1,
        dtype="str"
    )
    print(type(data))
    print(data.shape)
    print(data[:5, :3])

    print(true_label_names[:5])
    # The data variable contains all the gene expression values
    #  from 20,531 genes. The true_label_names are the cancer
    #  types for each of the 801 samples.
    #  BRCA: Breast invasive carcinoma
    # COAD: Colon adenocarcinoma
    # KIRC: Kidney renal clear cell carcinoma
    # LUAD: Lung adenocarcinoma
    # PRAD: Prostate adenocarcinoma

    # we need to convert the labels to integers with LabelEncoder:
    label_encoder = LabelEncoder()
    true_labels = label_encoder.fit_transform(true_label_names)
    print(true_labels[:5])
    print(label_encoder.classes_)

    x_tensor = torch.from_numpy(data).float()
    y_tensor = torch.from_numpy(true_labels).int()
    mydataset = MyDataSet(x_tensor, y_tensor)
    train_dataset, test_dataset = random_split(mydataset, [len(mydataset)-150,
150])

    train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size)
    test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size)
    return train_loader, test_loader
```

Add a class called Network with the following code in it.

```
import torch
import sys
import torch.nn as nn
import torchvision

# CNN network followed by a with one hidden layer classifier
class Network(nn.Module):
     def __init__(self, hidden_size1,  num_classes):
        super(Network, self).__init__()
        self.conv1 = nn.Conv1d(1, 6, 15) # 15x1 conv.kernel
        # in_channels = 1 because of linear data
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv1d(6,4,18)
        # 20500 comes from the dimension of the last conv layer
        self.fc1 = nn.Linear(4*20500, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, num_classes)
        self.smax = nn.Softmax(dim=1)

     def forward(self, x):
         out = self.conv1(x)
         out = self.relu(self.conv2(self.relu(out)))
         flatten = out.view(-1,4*20500)
         out = self.fc1(flatten)
         out = self.relu(out)
         out = self.fc2(out)
         out = self.smax(out)
         return out
```

Type the following code in CancerClassification.py.

```
import sys
import Utils
from Network import Network
import torch
import torch.nn as nn

def main():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    hidden_size1 = 100
    num_classes = 5
    num_epochs = 10
    batch_size = 10
    learning_rate = 0.001

    train_loader, test_loader = Utils.get_train_test_loaders(batch_size)

    model = Network(hidden_size1, num_classes).to(device)

    criterion = nn.CrossEntropyLoss() # for multiclass
    # classification, cross entropy loss works better
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    num_total_steps = len(train_loader)
    for epoch in range(num_epochs):
        for i, (x, labels) in enumerate(train_loader):
            x = x.reshape(-1,1,x.size(1)).to(device) # convert to CPU or GPU
tensor
            labels = labels.type(torch.LongTensor)
            labels = labels.to(device)

            pred_outputs = model(x) # calls forward function

            loss = criterion(pred_outputs, labels)
            optimizer.zero_grad() # clear gradients
            loss.backward()  # compute gradients
            optimizer.step() # update weights and biases
            if (i+1) % 10 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}],
Step[{i+1}/{num_total_steps}], Loss: {loss.item():.4f}')

    # compute accuracy on test set
    with torch.no_grad():
        num_correct = 0
        num_samples = 0
        for xt, labels in test_loader:
            xt = xt.reshape(-1,1,xt.size(1)).to(device)
            labels = labels.type(torch.LongTensor)
            labels = labels.to(device)
            outputs = model(xt)
            _, predicted = torch.max(outputs, 1) # returns max,max_indices
            num_samples += labels.size(0)
            num_correct += (predicted == labels).sum()

    acc = 100.0 * num_correct / num_samples
    print(f'Accuracy of the network on the test set: {acc} %')

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

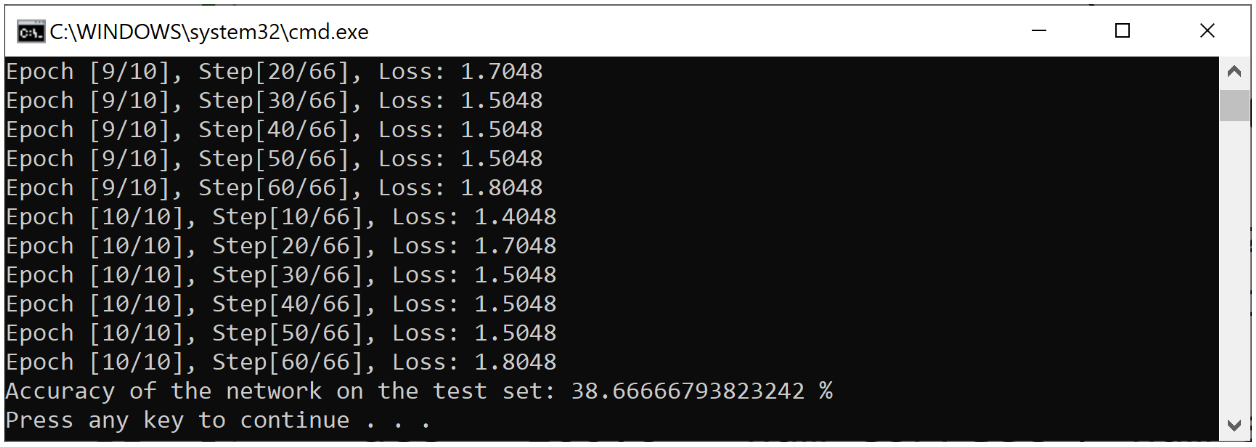
As mentioned earlier, for networks with CNNs, the input data has to include the channel, so we have to reshape the data to [batch size, channel, dimensionality of data]. The line in above code:

```
      x = x.reshape(-1,1,x.size(1)).to(device)
```

accomplishes this.

If you run the program, the accuracy is quite low. This is because the data is quite sparse and we are only using 6 feature maps in the first CNN layer:

```
self.conv1 = nn.Conv1d(1, 6, 15)
```

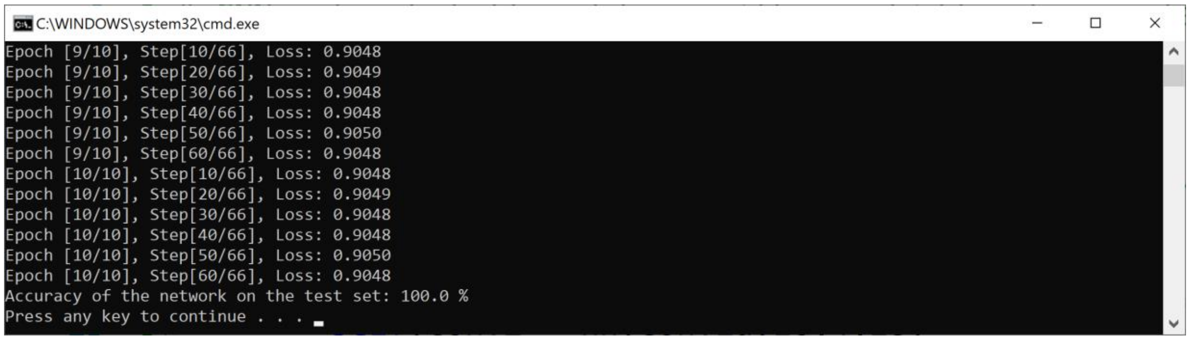


```
C:\WINDOWS\system32\cmd.exe
Epoch [9/10], Step[20/66], Loss: 1.7048
Epoch [9/10], Step[30/66], Loss: 1.5048
Epoch [9/10], Step[40/66], Loss: 1.5048
Epoch [9/10], Step[50/66], Loss: 1.5048
Epoch [9/10], Step[60/66], Loss: 1.8048
Epoch [10/10], Step[10/66], Loss: 1.4048
Epoch [10/10], Step[20/66], Loss: 1.7048
Epoch [10/10], Step[30/66], Loss: 1.5048
Epoch [10/10], Step[40/66], Loss: 1.5048
Epoch [10/10], Step[50/66], Loss: 1.5048
Epoch [10/10], Step[60/66], Loss: 1.8048
Accuracy of the network on the test set: 38.66666793823242 %
Press any key to continue . . .
```

Modify the Network.py to use 16 feature maps in the first CNN layer. The modified code is shown in bold.

```
    def __init__(self, hidden_size1,  num_classes):
        super(Network, self).__init__()
          self.conv1 = nn.Conv1d(1, 16, 15) # 15x1 conv.kernel
        # in_channels = 1 because linear data
        self.relu = nn.ReLU()
          self.conv2 = nn.Conv1d(16,4,18) # 18x1 conv.kernel
        # 20500 comes from the dimension of the last conv layer
        self.fc1 = nn.Linear(4*20500, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, num_classes)
        self.smax = nn.Softmax(dim=1)
```

Now if you run the program, you will get 100% accuracy on the test set.



```
Epoch [9/10], Step[10/66], Loss: 0.9048
Epoch [9/10], Step[20/66], Loss: 0.9049
Epoch [9/10], Step[30/66], Loss: 0.9048
Epoch [9/10], Step[40/66], Loss: 0.9048
Epoch [9/10], Step[50/66], Loss: 0.9050
Epoch [9/10], Step[60/66], Loss: 0.9048
Epoch [10/10], Step[10/66], Loss: 0.9048
Epoch [10/10], Step[20/66], Loss: 0.9049
Epoch [10/10], Step[30/66], Loss: 0.9048
Epoch [10/10], Step[40/66], Loss: 0.9048
Epoch [10/10], Step[50/66], Loss: 0.9050
Epoch [10/10], Step[60/66], Loss: 0.9048
Accuracy of the network on the test set: 100.0 %
Press any key to continue . . .
```

**Question:** If you keep the number of feature maps as 6 in the first CNN layer, and the number of feature maps to be 4 in the second CNN layer, what is the minimum convolution kernel size in the first and the second layers that can give you a 100% accuracy.