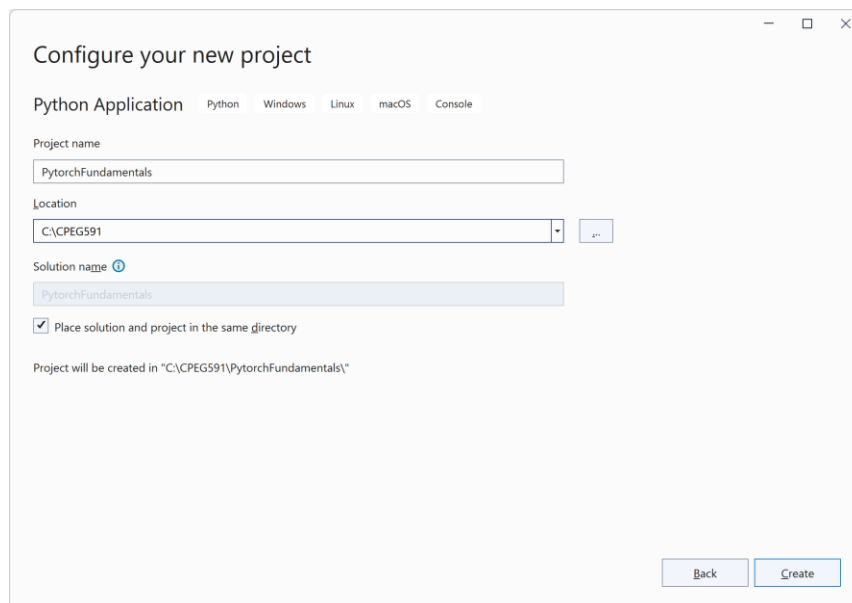
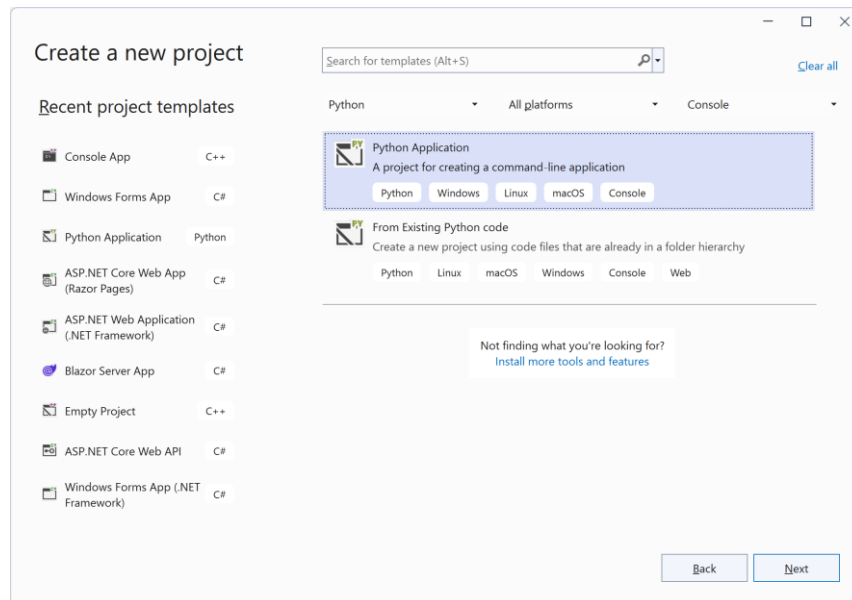


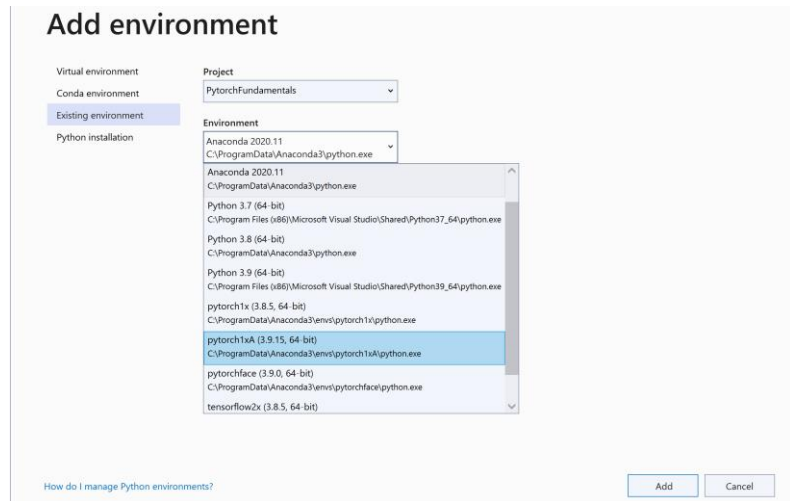
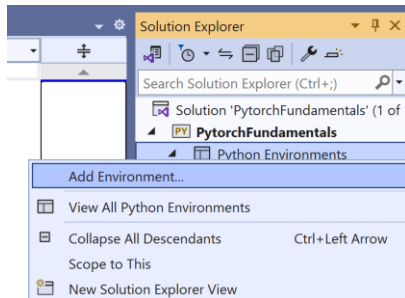
## CPEG 592- Assignment #1

### Pytorch Fundamentals for AI/Deep Learning

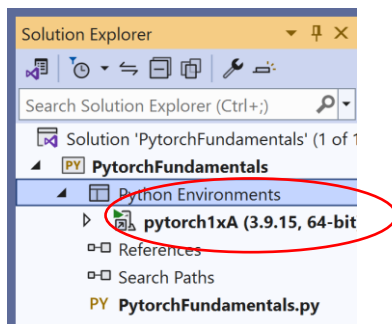
Create a new Python Application in Visual Studio called PytorchFundamentals as shown below.



Set the environment to be your pytorch environment that you had set up on your machine. I had called my environment as “pytorch1xA”. You will do this by right clicking on the environment and choosing “add existing environment as shown below.



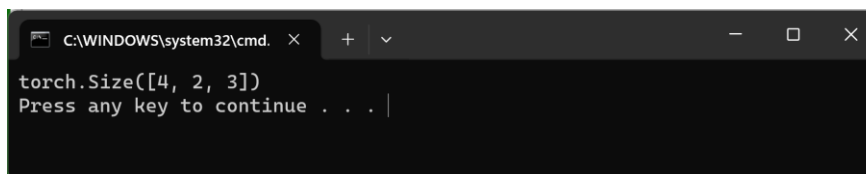
Now your environment will appear as:



Pytorch uses the concept of tensors. Each tensor has a shape (or size). For example, to create a three dimensional tensor with of size [4,2,3] and initialize it to zeros, you will declare as: `torch.zeros((4,2,3))`. The inner paranthesis of (4,2,3) indicate the shape or size of the tensor. The first dimension of 4 is shape[0], the second dimension of 2 is shape[1], and the last dimension of 3 is shape[3]. Note that in Python, the last part or dimension also be selected as shape[-1] Type the following program in `PytorchFundamentals.py` and run it.

```
import sys
import numpy as np
import torch
def main():
    t1 = torch.zeros((4,2,3))
    print(t1.shape)

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



Change the print statement to print `t1.shape[0]`, or `t1.shape[1]`, or `t1.shape[2]` to see you get the expected answer.

Some of the other concepts with pytorch are demonstrated by the following code. The modified `PytorchFundamentals.py` looks as shown below. Type the code to each print statement and then run the program to see if the purpose of code is becoming clear. Comments are provided to further clarify important concepts in pytorch.

```
import sys
import numpy as np
import torch
def main():
    t1 = torch.zeros((4,2,3))
    print(t1.shape) # try t1.shape[0], t1.shape[1], t1.shape[2], t1.shape[-1]

    # To add an extra dimension in the beginning i.e., to make the shape(1,4,2,3)
    t2 = t1.reshape(-1,4,2,3)
    print(t2.shape)

    # To add an extra dimension in the beginning i.e., to make the shape(1,4,2,3)
    t3 = t1.view(-1,4,2,3)
    print(t3.shape)

    # To add a dimension at the end, you can use view or reshape like before
    t4 = t1.view(4,2,3,-1)
    print(t4.shape)

    # To add a dimension in the beginning or end, you can also use unsqueeze
    # dim=0 to add a dimension in the beginning, dim=-1 to add at the end
    t5 = t1.unsqueeze(dim=0)
    print(t5.shape)

    # you can squeeze to remove the dimension
    t6 = t5.squeeze(dim=0)
    print(t6.shape)

    # when you reshape, or change the view, make sure the total number of
    # elements in the tensor stays the same, e.g., if original tensor is (4,2,3)
    # you can reshape it to (4,6) as the number of elements is still 4x6=24
    # or you can reshape it to (8,3) or (1,8,3) which is still 24 elements
    t7 = t1.reshape(4,6)
    print('t7', t7.shape)
    # In reshaping, one of the dimensions can be left as -1, then it will
    # automatically figure out based on other dimensions as to what that will be
    t8 = t1.reshape(4,-1) # -1 will become 6 as t1 has 24 elements
    print('t8', t8.shape)

    # * to unpack a tuple or a list, zip operation to combine two tuples or lists
    aa = [(2,3,5),(6,7,8)]
    print('unpacked list:', *aa) # unpacks the above list aa as two tuples
    bb, cc, dd = zip(*aa) # zip(*aa)=((2,6), (3,7), (5,8)), bb=(2,6)
    print('bb:', bb)

    a = np.array([[5,3],[6,7]])
    b = torch.tensor(a, dtype=torch.int64)[None] # like unsqueeze, adds dimension
    print('After None shape:', b.shape)
```

```

# gather allows us to select some of the elements from a tensor
# and further put them in a different order
t = torch.tensor([[1,2],[3,4]]) # dim=1 selects by col
r = torch.gather(t, dim=1, index=torch.tensor([[1,0],[0,1]]))
print('r:',r)
w = t.view(-1) # will create 1-d tensor
print('w:',w)
v = t.view(-1)[: , None]
print('v:',v.shape) # 4x1

r2 = torch.gather(t, dim=0, index=torch.tensor([[1,0],[0,1]]))
print('r2:',r2) # dim=0, selects row wise, r2=[[3,2],[1,4]]

r3 = torch.gather(t, dim=0, index=torch.tensor([[1],[0]]))
print('r3:',r3) # dim=0, selects row wise, r2=[[3],[1]]

# initialize
out1 = torch.tensor([
    [0.10, 0.50, 0.40], # correct
    [0.55, 0.20, 0.25], # wrong
    [0.60, 0.10, 0.30], # correct
    [0.15, 0.65, 0.20]]) # correct

print('out1:',out1.shape) # [4,3]
y = torch.tensor([1, 2, 0, 1], dtype=torch.int64) # indices, can vary 0-2
y = y.reshape(4,1) # to match out1
probs = out1.gather(dim=1, index=y) # dim=1 , selects index on each row
print(probs)

y2 = torch.tensor([[1,1], [2,0], [0,1], [1,2]], dtype=torch.int64) # targets
probs2 = out1.gather(dim=1, index=y2)
print(probs2)

# taking mean along a dim, we can also use axis=1 in code below
out1_mean = out1.mean(dim=1, keepdim=True) # keepdim, makes the output 4x1
print(out1_mean)

# masking
state = torch.tensor([[0,0,0],[0,0,0],[0,0,1]])
mask = ~(state != 0) # 0 cells will be True, 1 cells will be False
mask = ~(state != 0)*1 # True cells will be 1, False will be 0
mask2 = (state != 0)*-1000 # non zero cells will become -1000
mask3 = (state > 0).float()
print(mask3)

#np.eye
board = [0,0,1,0,2,0,1,2,0] # example
# above board state produces: [1 0 0 1 0 0 0 0 1 1 0 0 0 1 0 1 0 0 0 0 1 0 1 0 1
0 0]
print(np.eye(3))
print(np.eye(3)[board]) # eye is diagonal matrix, select rows by board
print('-----')
print(np.eye(3)[board][:,[0,2,1]]) # switch 2nd and 3rd column

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

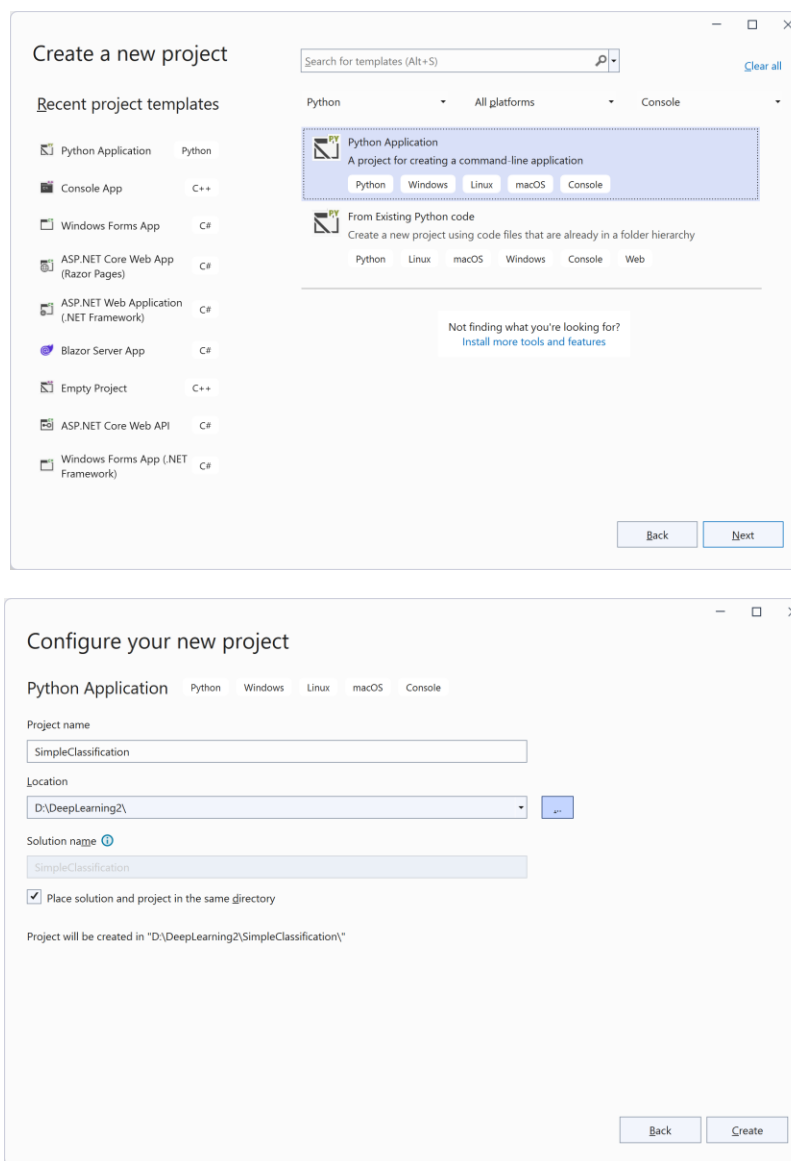
```

## Creating a simple Classifier using pytorch on custom DataSet:

We will review creating a simple CNN based classifier for classifying the AirplanesCarsShips dataset using pytorch. This dataset contains color images of airplanes, cars and ships, so we will create a classifier to classify an image into one of these three categories. The dataset is available as a zip file on the kiwi web site for the course. After downloading the zip file, extract it to a folder. It will contain a train and a test folder with further three folders for images belonging to each category. The number of images in each category for the train and test parts is shown below.

	<u>airplanes</u>	<u>ships</u>	<u>cars</u>
<b>train</b>	1000	1000	1000
<b>test</b>	189	200	193

Create a Python application called SimpleClassification as shown below.



Make sure to add the proper Python Environment for Pytorch for the project.

By right clicking on the project name, add a class to the project called MyDataSet.py with the following code in it. The purpose of the DataSet object is to contain the training and test data. If the total training data is small, we can read all the image pixels and store all images in multi-dimensional tensors in the dataset. However, if the number of images is large and images happen to be relatively high resolution, then one option is to store the image filenames (with full path info) in the dataset. The following code stores all images in the train or test folders as list of filenames and their labels. It will also apply a transform to the images. Transform usually resizes the image and changes the 0-255 pixel scale to either 0 to 1, or -1 to 1.

The `__getitem__` function in the DataSet class returns one data item and its label.

```
import torch
import torchvision
import numpy as np
import os
from PIL import Image

class MyDataset(torch.utils.data.Dataset):
    # for Airplanes, Cars and Ships dataset
    # train or test folders further have subfolders containing images
    # for each category

    def __init__(self, data_dir, transform=None):
        self.data_dir = data_dir
        self.transform = transform
        self.all_image_paths = []
        self.all_labels = []

        for planecarship_dir in os.listdir(data_dir): # each category folder
            category_path = os.path.join(data_dir, planecarship_dir)
            if not os.path.isdir(category_path):
                continue
            if planecarship_dir == "airplanes":
                label = 0
            elif planecarship_dir == "cars":
                label = 1
            elif planecarship_dir == "ships":
                label = 2
            image_paths = [os.path.join(category_path, f) for f in
os.listdir(category_path) if f.endswith('.jpg')]
            # list of image filenames for a category, e.g., airplane, car, or ship
            labels = [label for i in range(len(image_paths))]
            self.all_image_paths += image_paths
            self.all_labels += labels
        self.num_classes = len(set(self.all_labels))

    def __len__(self):
        return len(self.all_image_paths)

    def __getitem__(self, index):
        img_path = self.all_image_paths[index]
        label = self.all_labels[index]

        img = Image.open(img_path).convert('RGB')
        if self.transform is not None:
            img = self.transform(img)
        return img, label
```

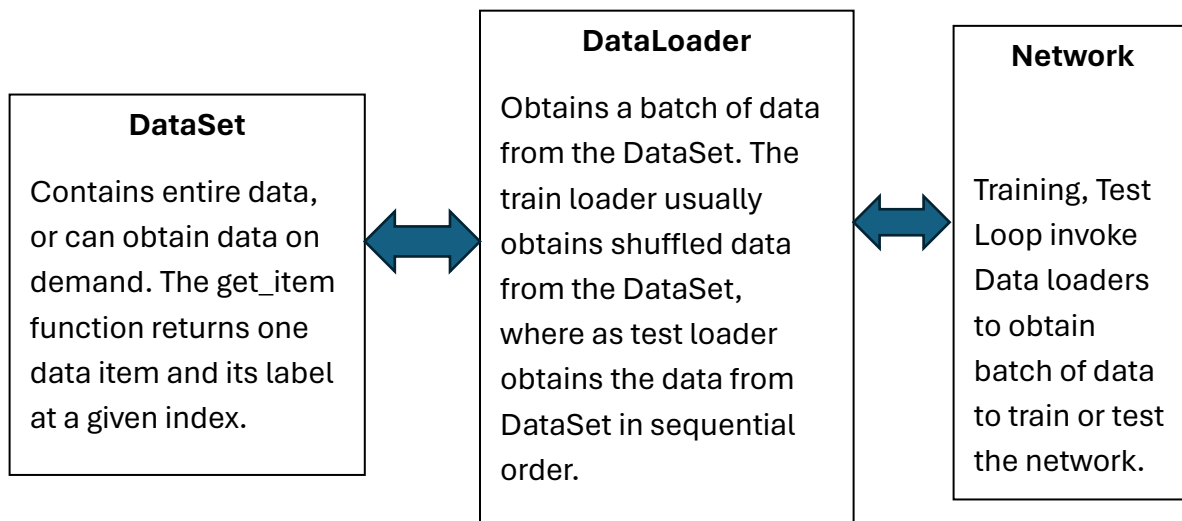
The DataSet is tied to a DataLoader which obtains a batch of data from the dataset and feeds to the training or test loop in the code. Add a file called Utils to the project with the following code in it.

```
import torch
from MyDataSet import MyDataset
import torchvision
import matplotlib.pyplot as plt

def get_train_loader(data_dir, batch_size, transform=None):
    dataset = MyDataset(data_dir, transform=transform)
    data_loader = torch.utils.data.DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=4,
        pin_memory=True,
        drop_last=True
    )
    return data_loader

def get_test_loader(data_dir, batch_size, transform=None):
    dataset = MyDataset(data_dir, transform=transform)
    data_loader = torch.utils.data.DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=4,
        pin_memory=True
    )
    return data_loader

def plot_images(images, labels):
    # normalise=True below shifts [-1,1] to [0,1]
    img_grid = torchvision.utils.make_grid(images, nrow=4, normalize=True)
    np_img = img_grid.numpy().transpose(1,2,0) # pytorch has the order, c,w,h
    # to be able to view an image, we need to change the order and
    # put it in width, height, color order
    plt.imshow(np_img)
    plt.show()
```



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

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class NetworkLinear(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(3*224*224, 100)
        self.fc2 = nn.Linear(100, 3)
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        x = x.view(-1, 3*224*224)
        x = F.relu(self.fc1(x))
        x = self.sm(self.fc2(x)) # softmax activation on final layer
        return x
```

As you can see, it uses two linear layers. The first one matches its input to the input image size i.e., 3x224x224 and has 100 neurons in it. It uses the Relu activation function. The second layer has 3 neurons in it as we have three categories of images. The input specification of the second layer has to match the output of the first layer e.g., 100 in our example. We will test the difference between using a linear network versus a CNN based network.

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

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class NetworkCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, 5) # input is color image with 3 channels
        # 32 is the number of feature maps and the kernel size is 5x5

        self.pool = nn.MaxPool2d(2,2)
        # maxpool will be used multiple times, but defined once
        # in_channels = 32 because self.conv1 output is 32 channels
        self.conv2 = nn.Conv2d(32,6,5)
        # 53*53 comes from the dimension of the last conv layer
        self.fc1 = nn.Linear(6*53*53, 100)
        self.fc2 = nn.Linear(100, 3)
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 6*53*53)
        x = F.relu(self.fc1(x))
        x = self.sm(self.fc2(x)) # softmax activation on final layer
        return x
```



Type the following code in SimpleClassification.py.

```
import sys
from torchvision import transforms
import torch
import Utils
from NetworkCNN import NetworkCNN
import torch.optim as optim
from NetworkLinear import NetworkLinear

def main():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    data_dir_train = "D:/DeepLearning2/data/Dataset_PlanesCarsShips/train"
    data_dir_test = "D:/DeepLearning2/data/Dataset_PlanesCarsShips/test"
    image_transforms = {
        "train": transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5],
                                [0.5, 0.5, 0.5])
        ]),
        "test": transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5],
                                [0.5, 0.5, 0.5])
        ])
    }
    # ToTensor converts a PIL Image or numpy.ndarray (HxWxC) in the range [0, 255]
    # to a torch.FloatTensor of shape (CxHxW) in the range [0.0, 1.0]
    batch_size = 16

    num_epochs = 25
    train_loader = Utils.get_train_loader(data_dir_train, batch_size,
    image_transforms["train"])
    test_loader = Utils.get_test_loader(data_dir_test, batch_size,
    image_transforms["test"])
    train_iter = iter(train_loader)
    images, labels = next(train_iter) # get a batch of data e.g., 16x3x224x224
    print(images[0].shape)

    Utils.plot_images(images, labels) # plot images

    net = NetworkLinear() # create the simple linear model
    #net = NetworkCNN() # create the CNN model
    loss_criterion = torch.nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    running_loss = 0
    print_freq = 100
    for epoch in range(num_epochs):
        for i, data in enumerate(train_loader):
            inputs, labels = data
            optimizer.zero_grad()
            outputs = net(inputs) # forward pass
            loss = loss_criterion(outputs, labels)
            loss.backward()
```

```

        optimizer.step()
        running_loss += loss.item()
        #if i % print_freq == print_freq-1:
    print('epoch:', epoch, i+1, running_loss/print_freq)
    running_loss = 0

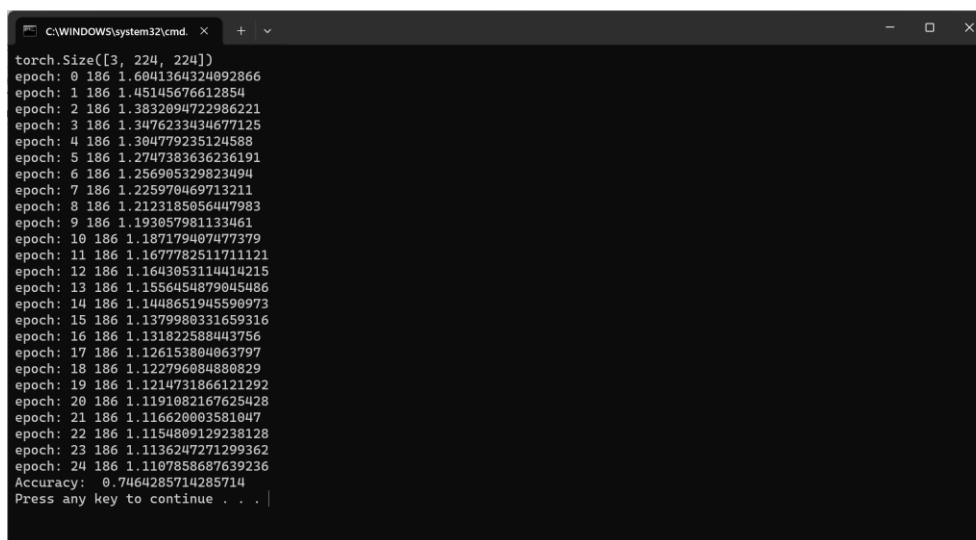
#-----compute accuracy on trained model-----
total = 0 # keeps track of how many images we have processed
correct = 0 # keeps track of how many correct images our net predicts
with torch.no_grad():
    for i, data in enumerate(test_loader):
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size()[0]
        correct += (predicted == labels).sum().item()

    print("Accuracy: ", correct/total)

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

```

Study the above code to see how the training and test loops are set up. If you run the program, you will get a classification accuracy of about 75% when using the linear network as the classifier.



```

C:\WINDOWS\system32\cmd  x  +  v
torch.Size([3, 224, 224])
epoch: 0 186 1.6041364324092866
epoch: 1 186 1.45145676612854
epoch: 2 186 1.3832094722986221
epoch: 3 186 1.3476233434677125
epoch: 4 186 1.304779235124588
epoch: 5 186 1.2747383636236191
epoch: 6 186 1.256905329823494
epoch: 7 186 1.225970469713211
epoch: 8 186 1.2123185056447983
epoch: 9 186 1.193057981133461
epoch: 10 186 1.187179407477379
epoch: 11 186 1.1677782511711121
epoch: 12 186 1.1643053114414215
epoch: 13 186 1.1556454879045486
epoch: 14 186 1.1448651945590973
epoch: 15 186 1.1379980331659316
epoch: 16 186 1.131822580443756
epoch: 17 186 1.126153804063797
epoch: 18 186 1.122796084880829
epoch: 19 186 1.1214731866121292
epoch: 20 186 1.1191082167625428
epoch: 21 186 1.116620003581047
epoch: 22 186 1.1154809129238128
epoch: 23 186 1.1136247271299362
epoch: 24 186 1.1107858687639236
Accuracy:  0.7464285714285714
Press any key to continue . . . |

```

If you comment and uncomment the network creation code as:

```

# net = NetworkLinear() # create the simple linear model
net = NetworkCNN() # create the CNN model

```

Run the project. Output accuracy for training for 25 epochs when using the CNN network as a classifier is around 80% on the test data set.

```
C:\WINDOWS\system32\cmd. x + v
torch.Size([3, 224, 224])
epoch: 0 186 1.859407696723938
epoch: 1 186 1.5797230231761932
epoch: 2 186 1.4703900408744812
epoch: 3 186 1.4194057138813598
epoch: 4 186 1.386071907877922
epoch: 5 186 1.3599673428190812
epoch: 6 186 1.3461096044015384
epoch: 7 186 1.3240294872714447
epoch: 8 186 1.2954773958576781
epoch: 9 186 1.279093211889267
epoch: 10 186 1.2672061729431152
epoch: 11 186 1.2612799179553986
epoch: 12 186 1.2419124114513398
epoch: 13 186 1.2313503617048263
epoch: 14 186 1.2225194263458252
epoch: 15 186 1.2048227697618856
epoch: 16 186 1.1998516416549683
epoch: 17 186 1.1924272161722183
epoch: 18 186 1.1931206250190736
epoch: 19 186 1.188189521431923
epoch: 20 186 1.1712018638849258
epoch: 21 186 1.1601939100027083
epoch: 22 186 1.161590760946274
epoch: 23 186 1.1510717695951462
epoch: 24 186 1.1457022005319595
Accuracy: 0.7982142857142858
Press any key to continue . . . |
```

While we have used DataSets and DataLoaders to obtain training data, in Reinforcement learning, most of the time training data will be obtained from the Experience Store (some times called the Relay Memory) as you will see in the next assignment.