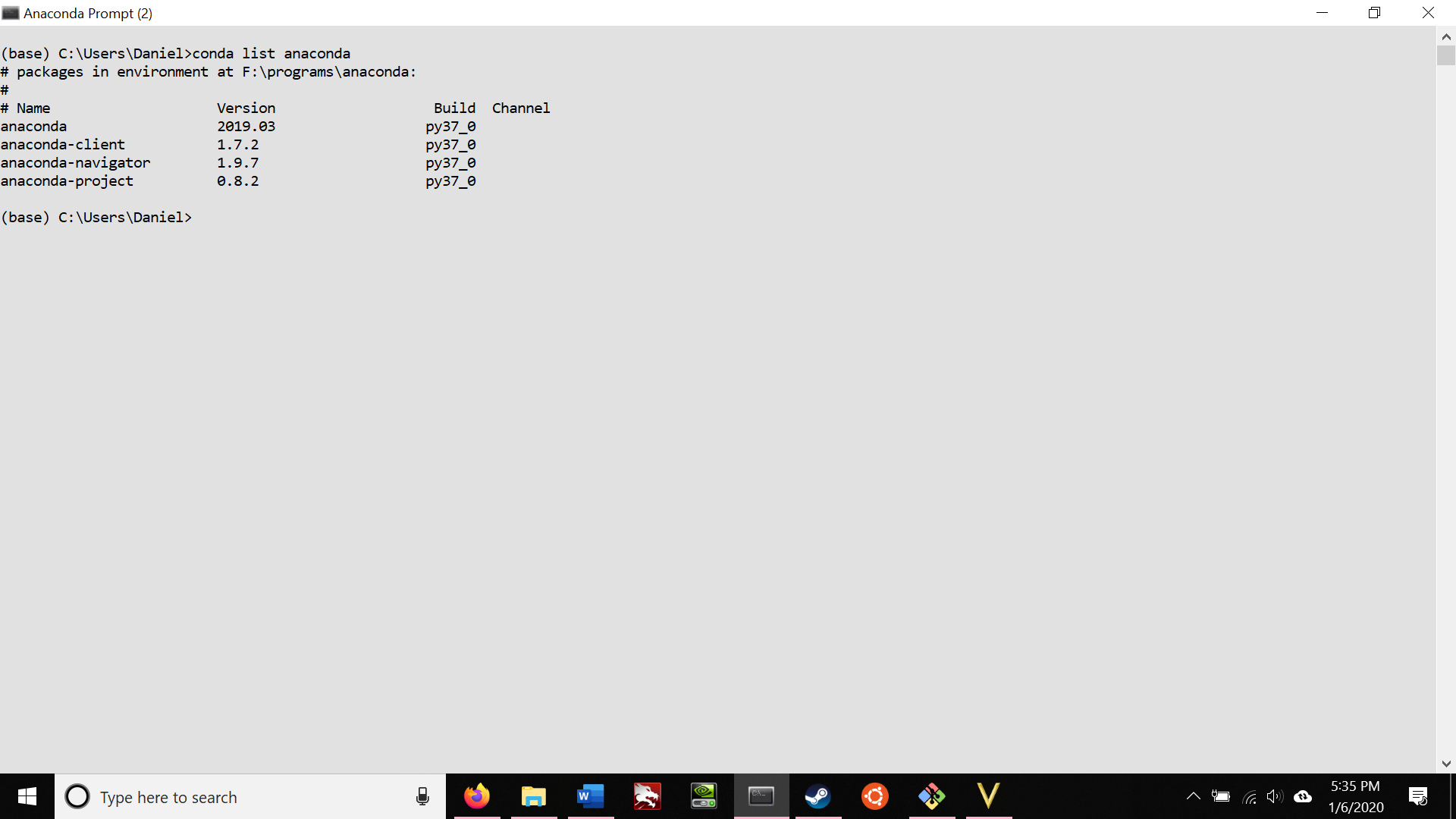
CS8395 Assignment 0

Name: Daniel Yan

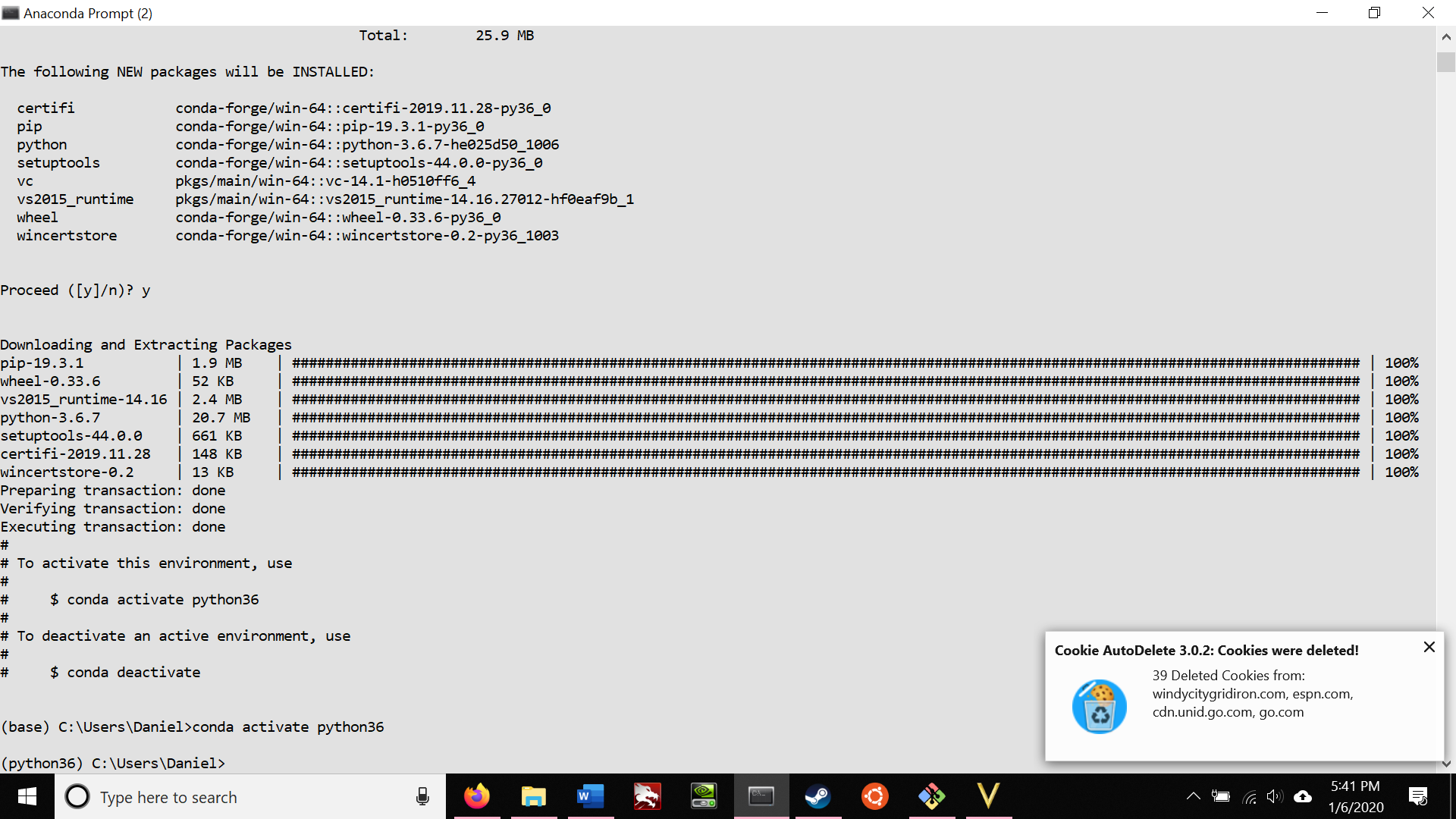
VUNet ID: yand1

Task 1

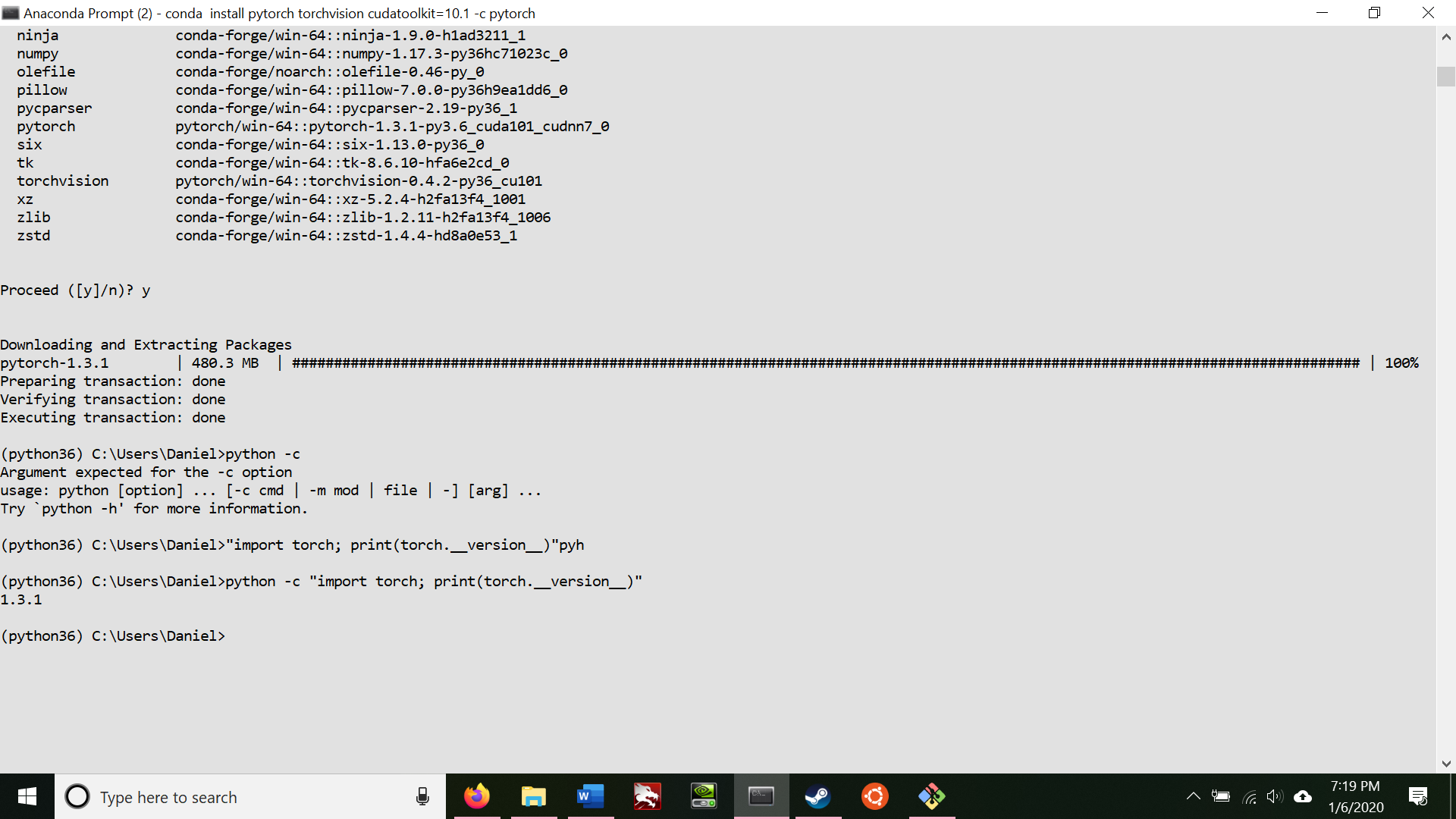
i).



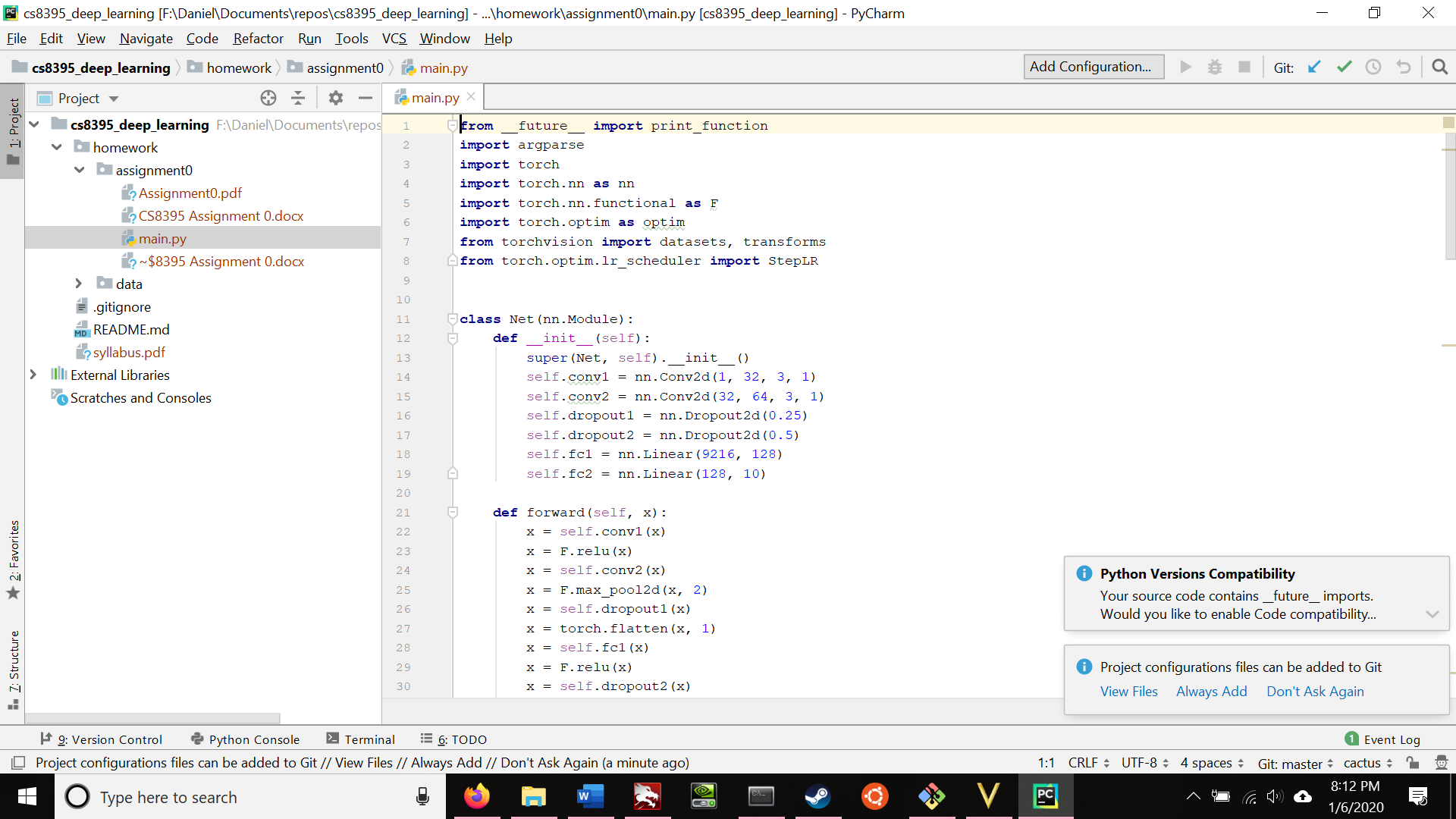
ii).



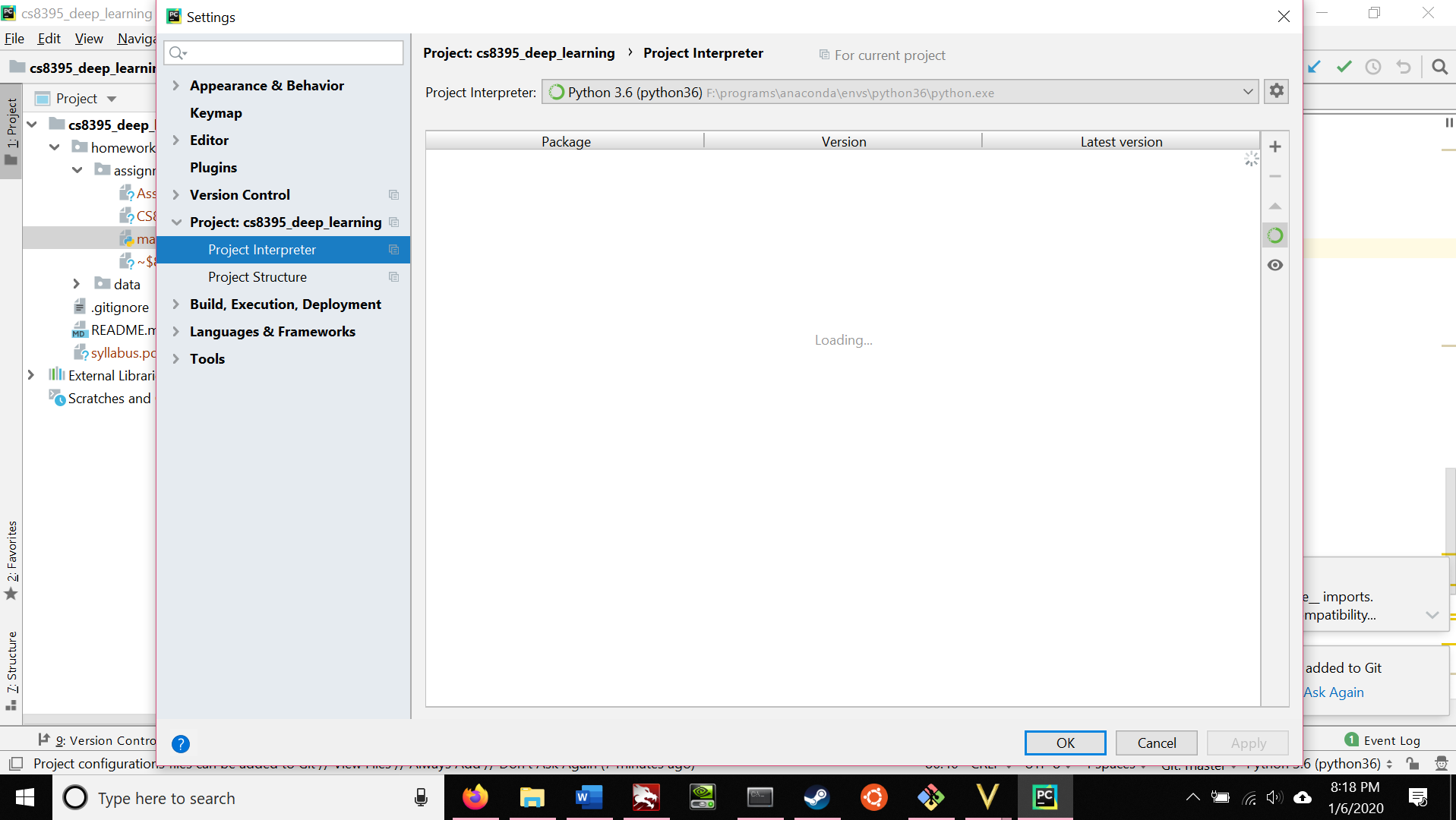
Task 2



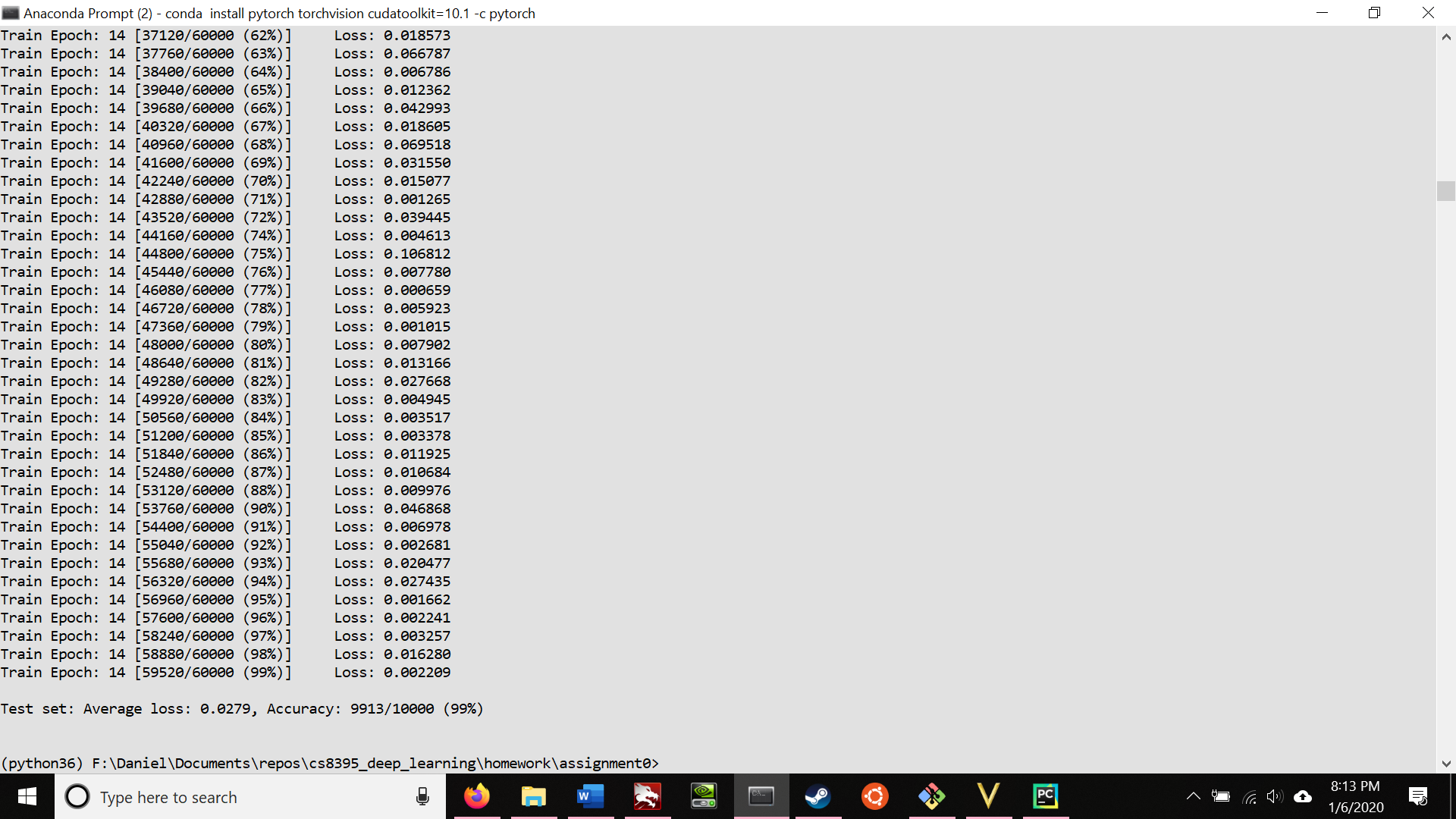
Task 3

i). 

ii).



Task 4



Task 5 (Description as comments in Code)

*# Imports for Pytorch***from** \_\_future\_\_ **import** print\_function  
**import** argparse  
**import** torch  
**import** torch.nn **as** nn  
**import** torch.nn.functional **as** F  
**import** torch.optim **as** optim  
**from** torchvision **import** datasets, transforms  
**from** torch.optim.lr\_scheduler **import** StepLR  
  
*# Define the neural network***class** Net(nn.Module):  
 *# Define the dimensions for each layer.* **def** \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 *# First convolutional layer has 1 input channel, 32 output channels,  
 # a 3x3 square kernel, and a stride of 1.* self.conv1 = nn.Conv2d(1, 32, 3, 1)  
 *# Second convolutional layer has 32 input channels  
 # since the first layer has 32 output channels.  
 # The second layer has 64 output channels, uses  
 # a 3x3 square kernel, and has a stride of 1.* self.conv2 = nn.Conv2d(32, 64, 3, 1)  
 *# Dropout is performed twice in the network,  
 # with the first time set to 0.25 and the  
 # second time set to 0.5.* self.dropout1 = nn.Dropout2d(0.25)  
 self.dropout2 = nn.Dropout2d(0.5)  
 *# Two fully connected layers. The input shape to the  
 # first fully connected layer is 64x12x12 = 9216. This is  
 # because the MNIST image is 28x28, so the first  
 # convolutional layer changes it to 26x26 since the kernel  
 # is 3x3. The second convolutional layer changes it to 24x24.  
 # We then have a maxpool layer that changes  
 # the dimensions to 12x12. Since we have 64 channels as  
 # the output from the second convolutional layer,  
 # we get a total of 64x12x12 = 9216. The output from  
 # the first fully connected layer is size 128.* self.fc1 = nn.Linear(9216, 128)  
 *# Second fully connected layer takes in shape of  
 # 128 from the output of the first fully connected layer  
 # and then has 10 outputs because we have 10 classes for MNIST.* self.fc2 = nn.Linear(128, 10)  
  
 *# Define the structure for forward propagation.* **def** forward(self, x):  
 *# We begin with a convolutional layer with a  
 # Relu activation function. We then use a second  
 # convolutional layer and perform max pooling  
 # and dropout on the output. We then flatten the  
 # 64 channels from the output of the second  
 # convolutional layer to pass to the first fully  
 # connected layer, and use a Relu activation  
 # function for the output. We then perform dropout  
 # a second time and send the output for the  
 # softmax function, since we are performing classification.* x = self.conv1(x)  
 x = F.relu(x)  
 x = self.conv2(x)  
 x = F.max\_pool2d(x, 2)  
 x = self.dropout1(x)  
 x = torch.flatten(x, 1)  
 x = self.fc1(x)  
 x = F.relu(x)  
 x = self.dropout2(x)  
 x = self.fc2(x)  
 output = F.log\_softmax(x, dim=1)  
 **return** output  
  
  
**def** train(args, model, device, train\_loader, optimizer, epoch):  
 *# Specify that we are in training phase* model.train()  
 *# Iterate through all minibatches.* **for** batch\_idx, (data, target) **in** enumerate(train\_loader):  
 *# Send training data and the training labels to GPU/CPU* data, target = data.to(device), target.to(device)  
 *# Zero the gradients carried over from previous step* optimizer.zero\_grad()  
 *# Obtain the predictions from forward propagation* output = model(data)  
 *# Compute the negative log likelihood of the loss function* loss = F.nll\_loss(output, target)  
 *# Perform backward propagation to compute the negative gradient, and  
 # update the gradients with optimizer.step()* loss.backward()  
 optimizer.step()  
 *# Send output to log if logging is needed* **if** batch\_idx % args.log\_interval == 0:  
 print(**'Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'**.format(  
 epoch, batch\_idx \* len(data), len(train\_loader.dataset),  
 100. \* batch\_idx / len(train\_loader), loss.item()))  
  
  
**def** test(args, model, device, test\_loader):  
 *# Specify that we are in evaluation phase* model.eval()  
 *# Set the loss and number of correct instances initially to 0.* test\_loss = 0  
 correct = 0  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# For each testing example, we run forward  
 # propagation to calculate the  
 # testing prediction. Update the total loss  
 # and the number of correct predictions  
 # with the counters from above.* **for** data, target **in** test\_loader:  
 data, target = data.to(device), target.to(device)  
 output = model(data)  
 test\_loss += F.nll\_loss(output, target, reduction=**'sum'**).item()  
 pred = output.argmax(dim=1, keepdim=**True**)  
 correct += pred.eq(target.view\_as(pred)).sum().item()  
  
 *# Average the loss by dividing by the total number of testing instances.* test\_loss /= len(test\_loader.dataset)  
  
 *# Print out the statistics for the testing set.* print(**'\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'**.format(  
 test\_loss, correct, len(test\_loader.dataset),  
 100. \* correct / len(test\_loader.dataset)))  
  
  
**def** main():  
 *# Command line arguments for hyperparameters of  
 # training and testing batch size, the number of  
 # epochs, the learning rate, gamma, and other  
 # settings such as whether to use a GPU device, the  
 # random seed, how often to log, and  
 # whether we should save the model.* parser = argparse.ArgumentParser(description=**'PyTorch MNIST Example'**)  
 parser.add\_argument(**'--batch-size'**, type=int, default=64, metavar=**'N'**,  
 help=**'input batch size for training (default: 64)'**)  
 parser.add\_argument(**'--test-batch-size'**, type=int, default=1000, metavar=**'N'**,  
 help=**'input batch size for testing (default: 1000)'**)  
 parser.add\_argument(**'--epochs'**, type=int, default=14, metavar=**'N'**,  
 help=**'number of epochs to train (default: 14)'**)  
 parser.add\_argument(**'--lr'**, type=float, default=1.0, metavar=**'LR'**,  
 help=**'learning rate (default: 1.0)'**)  
 parser.add\_argument(**'--gamma'**, type=float, default=0.7, metavar=**'M'**,  
 help=**'Learning rate step gamma (default: 0.7)'**)  
 parser.add\_argument(**'--no-cuda'**, action=**'store\_true'**, default=**False**,  
 help=**'disables CUDA training'**)  
 parser.add\_argument(**'--seed'**, type=int, default=1, metavar=**'S'**,  
 help=**'random seed (default: 1)'**)  
 parser.add\_argument(**'--log-interval'**, type=int, default=10, metavar=**'N'**,  
 help=**'how many batches to wait before logging training status'**)  
  
 parser.add\_argument(**'--save-model'**, action=**'store\_true'**, default=**False**,  
 help=**'For Saving the current Model'**)  
 args = parser.parse\_args()  
 *# Command to use gpu depending on command line arguments and if there is a cuda device* use\_cuda = **not** args.no\_cuda **and** torch.cuda.is\_available()  
  
 *# Random seed to use* torch.manual\_seed(args.seed)  
  
 *# Set to either use gpu or cpu* device = torch.device(**"cuda" if** use\_cuda **else "cpu"**)  
  
 *# GPU keywords.* kwargs = {**'num\_workers'**: 1, **'pin\_memory'**: **True**} **if** use\_cuda **else** {}  
 *# Load in the training and testing datasets. Convert to  
 # pytorch tensor and normalize.* train\_loader = torch.utils.data.DataLoader(  
 datasets.MNIST(**'../data'**, train=**True**, download=**True**,  
 transform=transforms.Compose([  
 transforms.ToTensor(),  
 transforms.Normalize((0.1307,), (0.3081,))  
 ])),  
 batch\_size=args.batch\_size, shuffle=**True**, \*\*kwargs)  
 test\_loader = torch.utils.data.DataLoader(  
 datasets.MNIST(**'../data'**, train=**False**, transform=transforms.Compose([  
 transforms.ToTensor(),  
 transforms.Normalize((0.1307,), (0.3081,))  
 ])),  
 batch\_size=args.test\_batch\_size, shuffle=**True**, \*\*kwargs)  
  
 *# Run model on GPU if available* model = Net().to(device)  
 *# Specify Adadelta optimizer* optimizer = optim.Adadelta(model.parameters(), lr=args.lr)  
  
 *# Run for the set number of epochs. For each epoch, run the training  
 # and the testing steps. Scheduler is used to specify the learning rate.* scheduler = StepLR(optimizer, step\_size=1, gamma=args.gamma)  
 **for** epoch **in** range(1, args.epochs + 1):  
 train(args, model, device, train\_loader, optimizer, epoch)  
 test(args, model, device, test\_loader)  
 scheduler.step()  
  
 *# Save model if specified by the command line argument* **if** args.save\_model:  
 torch.save(model.state\_dict(), **"mnist\_cnn.pt"**)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()