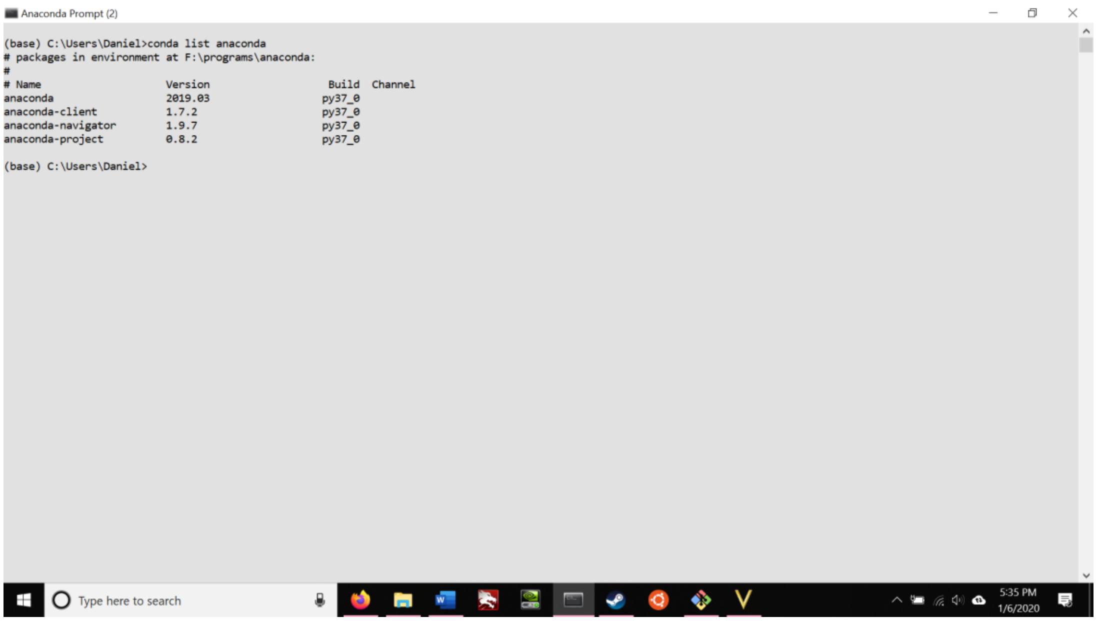
CS8395 Assignment 0

Name: Daniel Yan

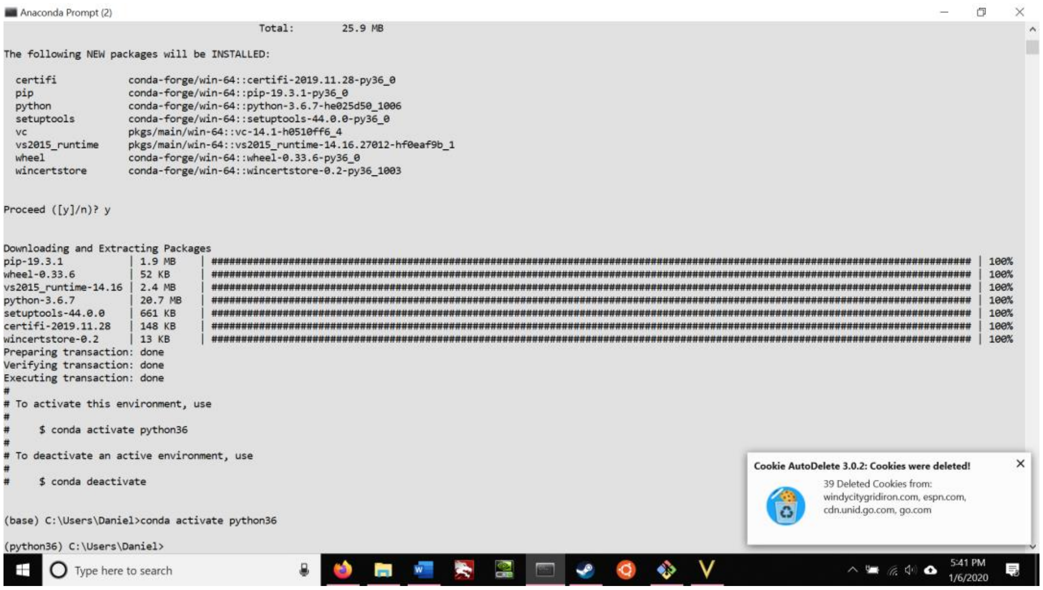
VUNet ID: yand1

Task 1

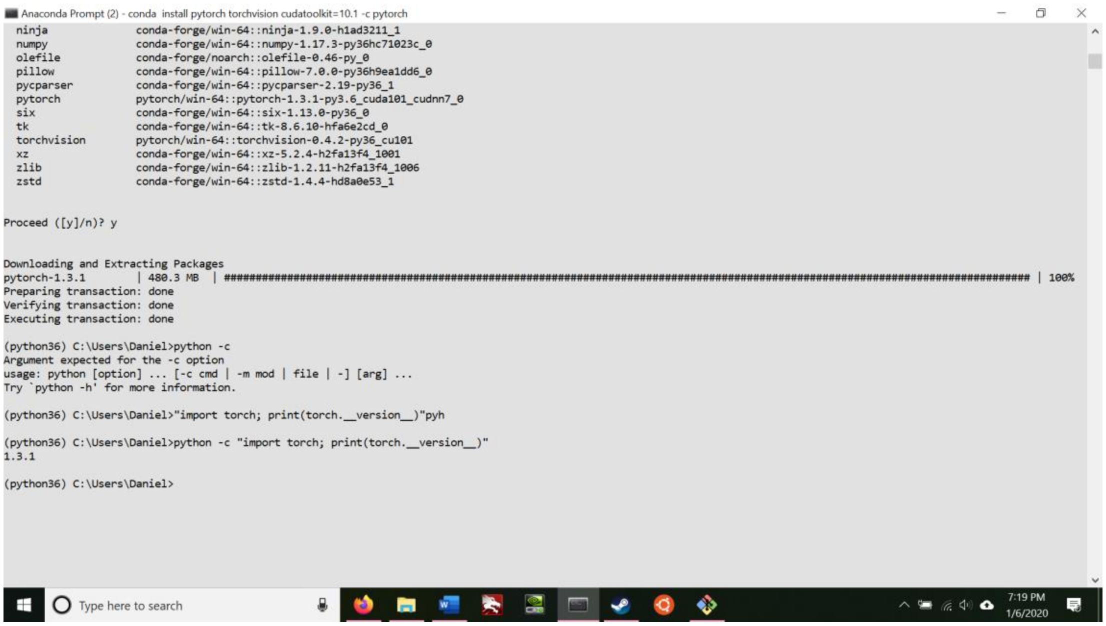
i).



ii).

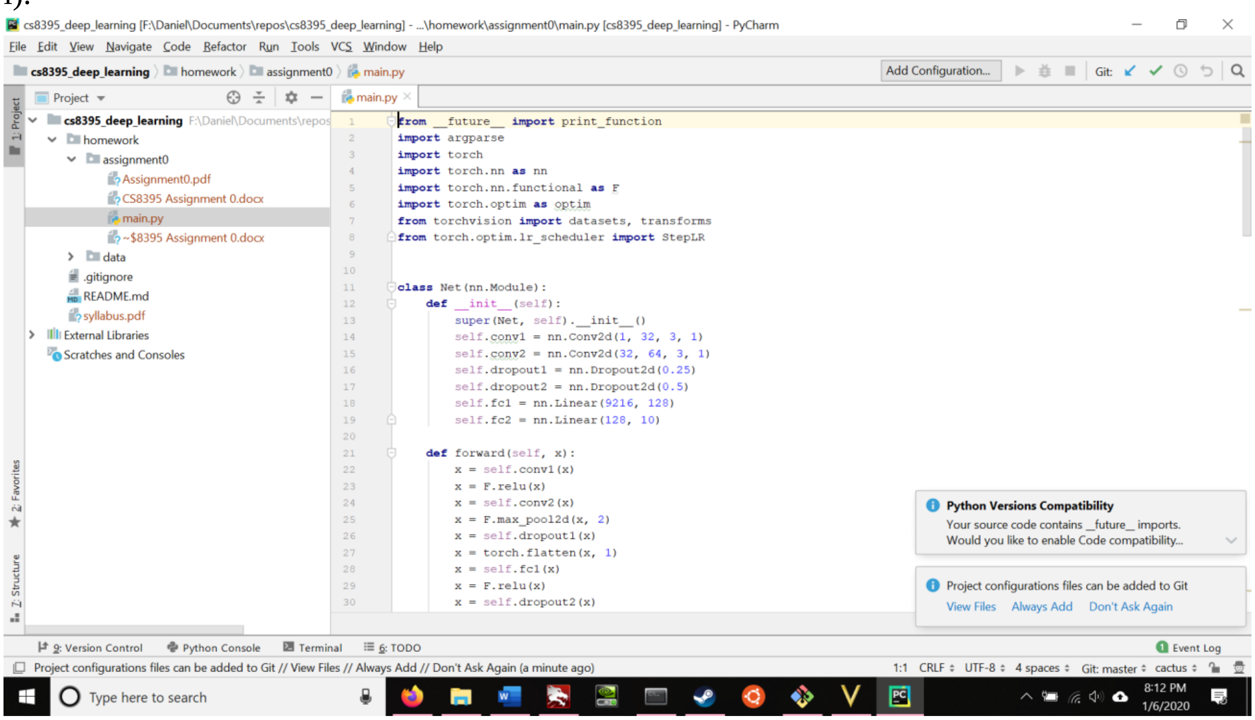


Task 2

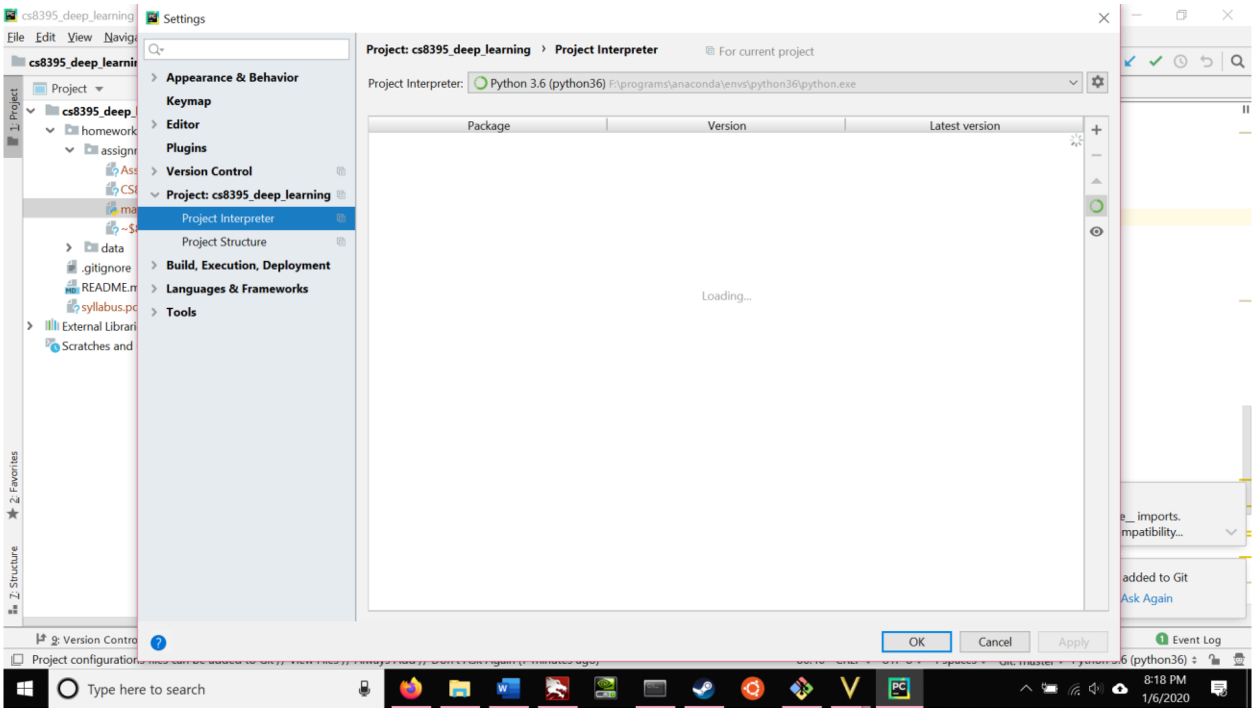


Task 3

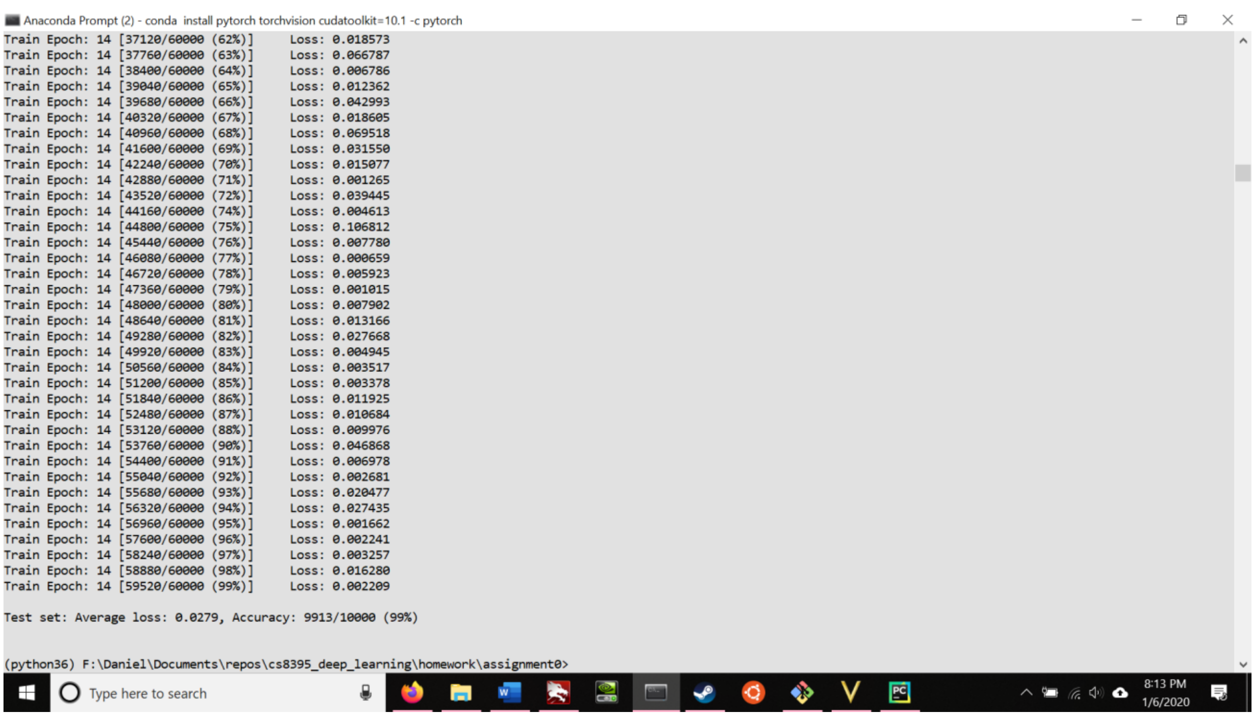
i).



ii).


Settings
Project: cs8395_deep_learning › Project Interpreter
For current project
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Project: cs8395_deep_learning
Project Interpreter
Project Structure
Build, Execution, Deployment
Languages & Frameworks
Tools
Project Interpreter: Python 3.6 (python36) F:\programs\anaconda\envs\python36\python.exe
Package
Version
Latest version
Loading...
OK
Cancel
Apply
Type here to search
8:18 PM
1/6/2020

Task 4


Anaconda Prompt (2) - conda install pytorch torchvision cudatoolkit=10.1 -c pytorch
Train Epoch: 14 [37120/60000 (62%)] Loss: 0.018573
Train Epoch: 14 [37760/60000 (63%)] Loss: 0.066787
Train Epoch: 14 [38400/60000 (64%)] Loss: 0.006786
Train Epoch: 14 [39040/60000 (65%)] Loss: 0.012362
Train Epoch: 14 [39680/60000 (66%)] Loss: 0.042993
Train Epoch: 14 [40320/60000 (67%)] Loss: 0.018605
Train Epoch: 14 [40960/60000 (68%)] Loss: 0.069518
Train Epoch: 14 [41600/60000 (69%)] Loss: 0.031550
Train Epoch: 14 [42240/60000 (70%)] Loss: 0.015077
Train Epoch: 14 [42880/60000 (71%)] Loss: 0.001265
Train Epoch: 14 [43520/60000 (72%)] Loss: 0.039445
Train Epoch: 14 [44160/60000 (74%)] Loss: 0.004613
Train Epoch: 14 [44800/60000 (75%)] Loss: 0.106812
Train Epoch: 14 [45440/60000 (76%)] Loss: 0.007780
Train Epoch: 14 [46080/60000 (77%)] Loss: 0.000659
Train Epoch: 14 [46720/60000 (78%)] Loss: 0.005923
Train Epoch: 14 [47360/60000 (79%)] Loss: 0.001015
Train Epoch: 14 [48000/60000 (80%)] Loss: 0.007902
Train Epoch: 14 [48640/60000 (81%)] Loss: 0.013166
Train Epoch: 14 [49280/60000 (82%)] Loss: 0.027668
Train Epoch: 14 [49920/60000 (83%)] Loss: 0.004945
Train Epoch: 14 [50560/60000 (84%)] Loss: 0.003517
Train Epoch: 14 [51200/60000 (85%)] Loss: 0.003378
Train Epoch: 14 [51840/60000 (86%)] Loss: 0.011925
Train Epoch: 14 [52480/60000 (87%)] Loss: 0.010684
Train Epoch: 14 [53120/60000 (88%)] Loss: 0.009976
Train Epoch: 14 [53760/60000 (90%)] Loss: 0.046868
Train Epoch: 14 [54400/60000 (91%)] Loss: 0.006978
Train Epoch: 14 [55040/60000 (92%)] Loss: 0.002681
Train Epoch: 14 [55680/60000 (93%)] Loss: 0.020477
Train Epoch: 14 [56320/60000 (94%)] Loss: 0.027435
Train Epoch: 14 [56960/60000 (95%)] Loss: 0.001662
Train Epoch: 14 [57600/60000 (96%)] Loss: 0.002241
Train Epoch: 14 [58240/60000 (97%)] Loss: 0.003257
Train Epoch: 14 [58880/60000 (98%)] Loss: 0.016280
Train Epoch: 14 [59520/60000 (99%)] Loss: 0.002209
Test set: Average loss: 0.0279, Accuracy: 9913/10000 (99%)
(python36) F:\Daniel\Documents\repos\cs8395_deep_learning\homework\assignment0>
Type here to search
8:13 PM
1/6/2020

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)
```

```python
        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.
```

```python
    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()
```