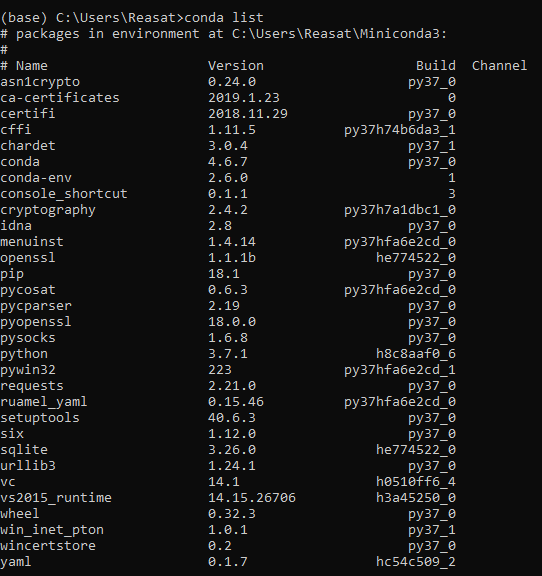
1. Task 1. Anaconda Set Up

i) *Install Anaconda https://www.anaconda.com/download/.*

*Show a screenshot of the results of the command “conda list anaconda”. (10*

*scores)*



ii) *Create virtual environment with Python 3.6 using the command “conda create –*

*name python36 python=3.6” to create the “python36” environment using Python*

*3.6.*

*Show a screenshot of the results of the command “source activate python36”. (10*

*scores)*

**My environment name is pytorch (instead of python36)**



2. Task 2. PyTorch Installation

i) *Install PyTorch using the conda command on https://pytorch.org/ in the python36*

*virtual environment.*

*Show a screenshot of the version of installed PyTorch the command “ python -c*

*"import torch; print(torch.\_\_version\_\_)" ”. This command can be varied among OS. I*

*only need to see the PyTorch is installed with the latest version. (10 scores)*

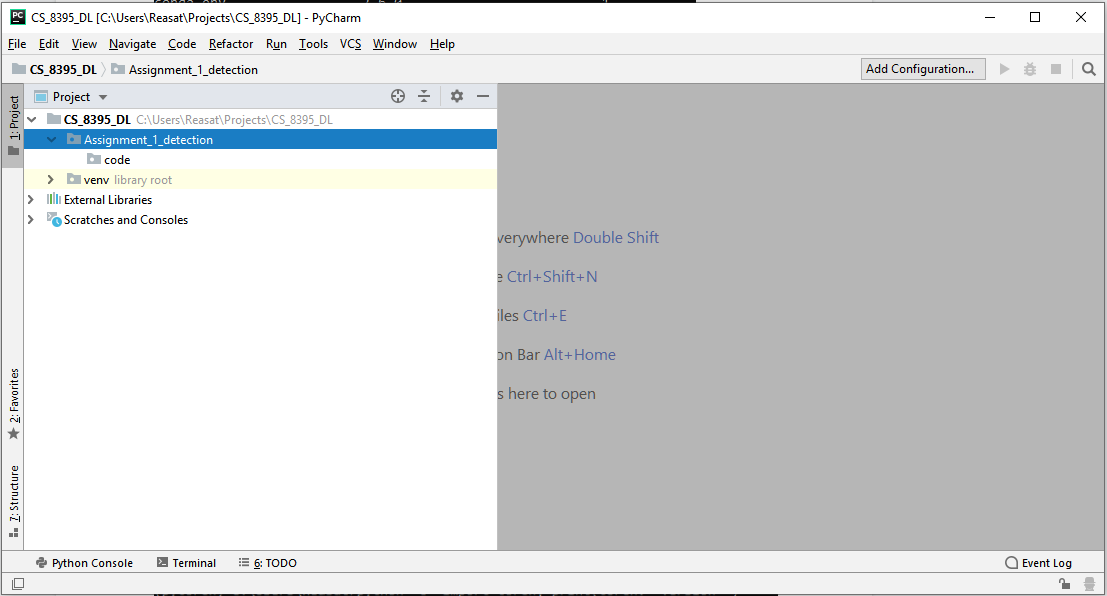


3. Task 3. PyCharm Installation and setup

i) Install the Python development GUI PyCharm from

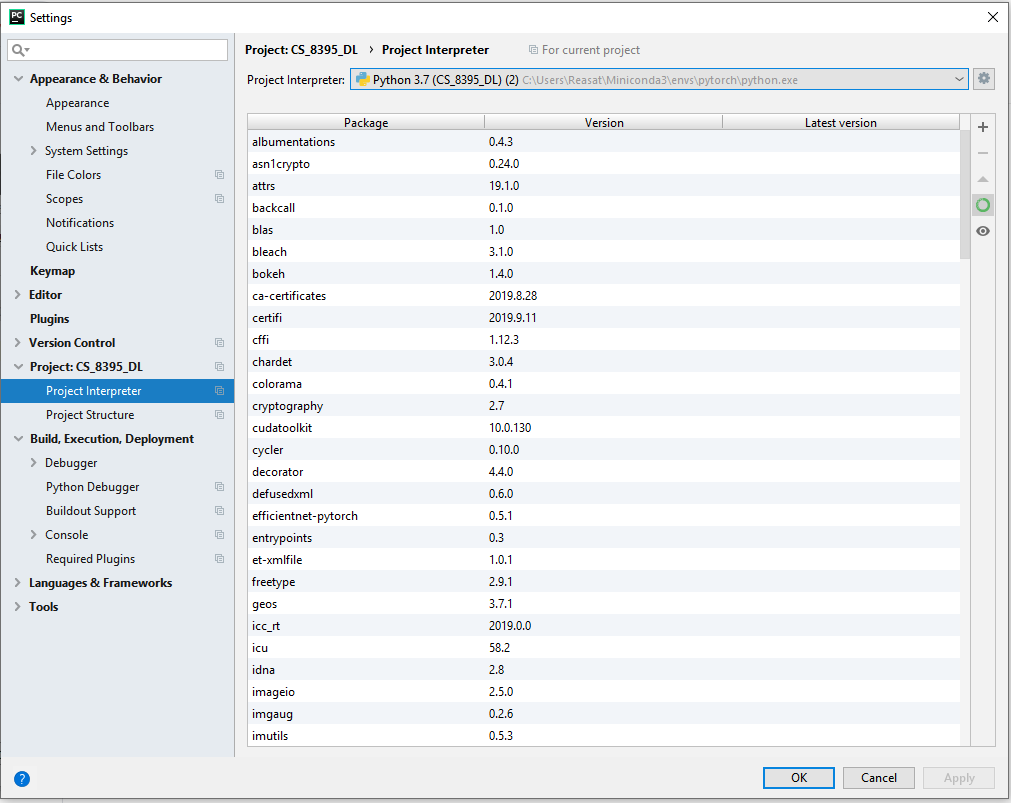
https://www.jetbrains.com/pycharm/download

Show a screenshot of the PyCharm workingspace (10 scores)



ii) *Set up the interpreter for PyCharm as the python36 conda environment.*

*Show a screenshot of the “project interpreter” in PyCharm (10 scores)*



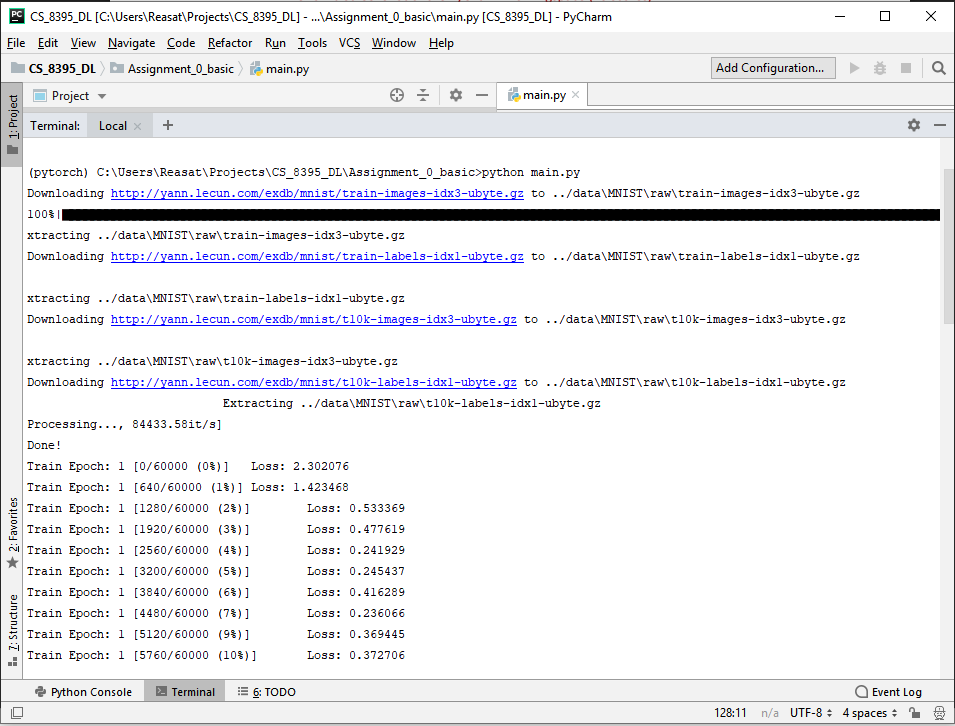
4. Task 4. Run the MNIST classification

i) *Run the “Basic MNIST example” on*

*https://github.com/pytorch/examples/tree/master/mnist*

*Show a screenshot of running results from PyCharm or command line window. (25*

*scores).*



5. Understand the code “main.py” you just run.

i) *Describe the major parts of the code using your own words.*

*You can put them as comments or write a separate paragraph to describe the code.*

*Both are fine for me. Please put one of them in the PDF report. (25 scores)*

*# loading all the necessary modules*

**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

**class** Net(nn.Module):

*# the Net class has the definition for the network architecture.*

**def** \_\_init\_\_(self):

*# the layer specifications are declared in the \_\_init\_\_ function*

super(Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 32, 3, 1)

self.conv2 = nn.Conv2d(32, 64, 3, 1)

self.dropout1 = nn.Dropout2d(0.25)

self.dropout2 = nn.Dropout2d(0.5)

self.fc1 = nn.Linear(9216, 128)

self.fc2 = nn.Linear(128, 10)

**def** forward(self, x):

*# the connectivity between the layers are defined by the forward function*

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):

*# takes a model, the train data loader and trains the model for specified number of epochs*

model.train()

**for** batch\_idx, (data, target) **in** enumerate(train\_loader):

data, target = data.to(device), target.to(device)

optimizer.zero\_grad()

output = model(data)

loss = F.nll\_loss(output, target)

loss.backward()

optimizer.step()

**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):

*# takes a model (usually trained beforehand), the test data loader and evaluates the model's performance on the test data*

model.eval() *# setting the model in evaluation mode*

test\_loss = 0

correct = 0

**with** torch.no\_grad(): *# during evaluation storing gradients is not necessary*

**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() *# sum up batch loss*

pred = output.argmax(dim=1, keepdim=**True**) *# get the index of the max log-probability*

correct += pred.eq(target.view\_as(pred)).sum().item()

test\_loss /= len(test\_loader.dataset)

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():

*# the main function which calls the train and test function defined above*

*# training and testing settings are fed using command line arguments*

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()

use\_cuda = **not** args.no\_cuda **and** torch.cuda.is\_available()

torch.manual\_seed(args.seed)

device = torch.device(**"cuda" if** use\_cuda **else "cpu"**)

*# defining training data loader*

kwargs = {**'num\_workers'**: 1, **'pin\_memory'**: **True**} **if** use\_cuda **else** {}

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)

*# defining testing data loader*

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)

*# defining model*

model = Net().to(device)

*# defining optimizer to optimize model weights*

optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

*# defining schedular for decreasing learning rate over training epochs*

scheduler = StepLR(optimizer, step\_size=1, gamma=args.gamma)

*# for each epoch perform training and testing*

**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 the model after training is done*

**if** args.save\_model:

torch.save(model.state\_dict(), **"mnist\_cnn.pt"**)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

*# start the simulation*

main()