# robust_challenge(1)

April 12, 2022

## 0.1 Pytorch Implementation of "Towards Deep Neural Networks Resistant to Adversarial Attacks"

Model Details: Resnet18 \ CIFAR10 \ L_inf threat epsilon=8/255

[1]:
```
import torch
import numpy as np
from torch.utils.data import Dataset
import torchvision
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda
from torchvision import transforms
import matplotlib.pyplot as plt
from torch import nn
import torch.optim as optim
```

[2]:
```
cuda0 = torch.device('cuda:0')
cuda0
```

[2]:
```
device(type='cuda', index=0)
```

[3]:
```
#Scale to [0,1] and normalize
transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.
↪2010))]
        )

batch_size = 128

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
```

```
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

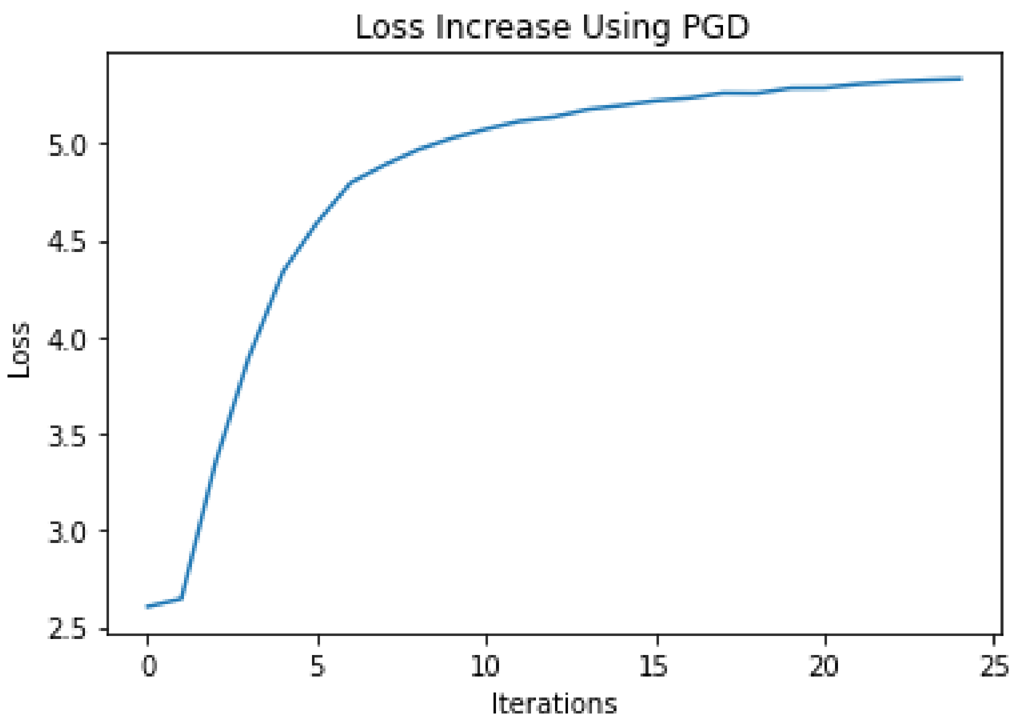
```
Files already downloaded and verified
Files already downloaded and verified
```

[4]:
```
#Resnet18 Model
model = torchvision.models.resnet18(pretrained=False)
model.fc = nn.Linear(model.fc.in_features,10,bias=True)
model.cuda()
loss_fn = nn.CrossEntropyLoss()
```

[5]:
```
class PGD:
  def __init__(self, network, loss_fn, steps, alpha, epsilon):
    """
    Initialize attack with parameters.
    """
    self.network = network
    self.loss_fn = loss_fn
    self.steps = steps
    self.alpha = alpha
    self.epsilon = epsilon
  def attack(self, X, y, device = None, log = False):
    """
    Projected gradient ascent on self.loss_fn w.r.t. X. If log==True, then
    vals is the evolution of the loss over iterations. Otherwise, vals = [].
    """
    x0 = X
    vals = []
    for _ in range(self.steps):
      X.requires_grad_()
      loss = self.loss_fn(self.network(X), y).to(device)
      loss.backward()
      if log:
        vals.append(loss)
      X = X + self.alpha*X.grad.sign()
      X = torch.clamp(X, min = x0-self.epsilon, max = x0+self.epsilon)
      X = torch.clamp(X, min=0.0,max=1.0).detach_()
    return X, vals
```

[6]:
```
pgd = PGD(model,loss_fn,steps=25,alpha=2/255,epsilon=8/255)
def test_PGD(i):
  """
  Example of PGD applied on an arbitrary batch indexed by i.
```

```
    """
    X,y = list(trainloader)[i]
    X,y = X.cuda(), y.cuda()
    model.cuda()
    a, v = pgd.attack(X,y, cuda0, True)
    plt.plot(range(len(v)),[x.item() for x in v])
    plt.title("Loss Increase Using PGD")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
test_PGD(0)
```




[7]:
```
optimizer = optim.SGD(model.parameters(), lr=0.0003)
model.cuda()
epochs = 6

def train(model, trainloader, epochs, optimizer, attacker, loss_fn):
    """
    Train the model and track the evolution of loss, adversarial accuracy (i.e.
    robust accuracy), and normal accuracy (non-perturbed inputs).
    """
    loss_history = []
    adv_accuracy_history = []
    normal_accuracy_history = []
```

```
  for epoch in range(epochs):
    for i, data in enumerate(trainloader, 0):
        #sample point(s)

        X,y = data

        X,y = X.to(cuda0), y.to(cuda0)

        optimizer.zero_grad()


        #compute maximizer of inner problem
        X_adv,_ = pgd.attack(X,y, device=cuda0)

        out = model(X_adv.cuda())

        #use approximate gradient for minimization
        loss = loss_fn(out, y)
        loss.backward()
        optimizer.step()

        loss_history.append(loss)

        adv_accuracy = torch.sum(torch.argmax(out, dim=1) == y).item()/
↪batch_size
        normal_accuracy = torch.sum(torch.argmax(model(X.cuda()), dim=1) == y).
↪item()/batch_size
        adv_accuracy_history.append(adv_accuracy)
        normal_accuracy_history.append(normal_accuracy)

        if i%10 == 0:
          print(f"iteration {i+1} with loss {loss.item()}",␣
↪f"{(i+1)*batch_size}/{len(trainloader.dataset)} and adv_accuracy␣
↪{adv_accuracy}, normal accuracy {normal_accuracy}")
    print("EPOCH", epoch+1, " DONE")

  print('Finished Training')
  return loss_history, adv_accuracy_history,normal_accuracy_history
```

[ ]:
```
loss_history, adv_accuracy_history, normal_accuracy_history = train(model,␣
↪trainloader, epochs, optimizer, pgd, loss_fn)
```

## 0.2 Training Statistics

Each iteration corresponds to a batch of 128 examples. Trends suggest that parameters can be improved further with more iterations.

[8]:
```
#Evaluation
def test(model):
  """
  Evaluate the model on the test set. As with training, distinguishes between
  and returns adversarial accuracy and normal accuracy.
  """
  correct = 0
  adv_correct = 0
  total = 0

  pgd_test = PGD(model, loss_fn, steps=20,alpha=2/255,epsilon=8/255)

  model.eval()
  for i,data in enumerate(testloader):
      X,y = data
      X,y = X.cuda(), y.cuda()
      adv_X,_ = pgd_test.attack(X,y,device=cuda0)

      adv_c = torch.sum(torch.argmax(model(adv_X), dim=1) == y).item()
      norm_c = torch.sum(torch.argmax(model(X), dim=1) == y).item()

      adv_correct += adv_c
      correct += norm_c

      total += batch_size
      print(f'Finished evaluating Batch {i+1} with Normal Accuracy {correct/
↪total} and Adversarial Accuracy {adv_correct/total}')

  print(f'Adv/Normal Accuracy on test images: {100 *  adv_correct// total} %,␣
↪{100*correct//total}%')
  return correct, adv_correct, total
```

[9]:
```
m = torch.load('full_model.pth')
m.eval()
```

[9]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
```

```
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
```

```
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

[10]:
```
test(m)
```

```
Finished evaluating Batch 1 with Normal Accuracy 0.4375 and Adversarial Accuracy
0.4296875
Finished evaluating Batch 2 with Normal Accuracy 0.41796875 and Adversarial
Accuracy 0.421875
Finished evaluating Batch 3 with Normal Accuracy 0.4114583333333333 and
Adversarial Accuracy 0.4296875
Finished evaluating Batch 4 with Normal Accuracy 0.400390625 and Adversarial
Accuracy 0.4296875
Finished evaluating Batch 5 with Normal Accuracy 0.4109375 and Adversarial
Accuracy 0.4375
Finished evaluating Batch 6 with Normal Accuracy 0.4140625 and Adversarial
Accuracy 0.41796875
Finished evaluating Batch 7 with Normal Accuracy 0.421875 and Adversarial
Accuracy 0.4140625
Finished evaluating Batch 8 with Normal Accuracy 0.4296875 and Adversarial
Accuracy 0.423828125
Finished evaluating Batch 9 with Normal Accuracy 0.4331597222222222 and
Adversarial Accuracy 0.4236111111111111
Finished evaluating Batch 10 with Normal Accuracy 0.4328125 and Adversarial
Accuracy 0.4265625
Finished evaluating Batch 11 with Normal Accuracy 0.4296875 and Adversarial
Accuracy 0.4190340909090909
Finished evaluating Batch 12 with Normal Accuracy 0.4348958333333333 and
Adversarial Accuracy 0.4192708333333333
Finished evaluating Batch 13 with Normal Accuracy 0.4338942307692308 and
Adversarial Accuracy 0.41646634615384615
Finished evaluating Batch 14 with Normal Accuracy 0.43582589285714285 and
```

Adversarial Accuracy 0.41573660714285715
Finished evaluating Batch 15 with Normal Accuracy 0.43645833333333334 and Adversarial Accuracy 0.41822916666666665
Finished evaluating Batch 16 with Normal Accuracy 0.43310546875 and Adversarial Accuracy 0.41455078125
Finished evaluating Batch 17 with Normal Accuracy 0.4296875 and Adversarial Accuracy 0.4099264705882353
Finished evaluating Batch 18 with Normal Accuracy 0.4296875 and Adversarial Accuracy 0.4127604166666667
Finished evaluating Batch 19 with Normal Accuracy 0.43379934210526316 and Adversarial Accuracy 0.4136513157894737
Finished evaluating Batch 20 with Normal Accuracy 0.433984375 and Adversarial Accuracy 0.412890625
Finished evaluating Batch 21 with Normal Accuracy 0.4322916666666667 and Adversarial Accuracy 0.40811011904761907
Finished evaluating Batch 22 with Normal Accuracy 0.43323863636363635 and Adversarial Accuracy 0.4069602272727273
Finished evaluating Batch 23 with Normal Accuracy 0.4313858695652174 and Adversarial Accuracy 0.4038722826086957
Finished evaluating Batch 24 with Normal Accuracy 0.4287109375 and Adversarial Accuracy 0.4020182291666667
Finished evaluating Batch 25 with Normal Accuracy 0.42875 and Adversarial Accuracy 0.4028125
Finished evaluating Batch 26 with Normal Accuracy 0.42788461538461536 and Adversarial Accuracy 0.40384615384615385
Finished evaluating Batch 27 with Normal Accuracy 0.42650462962962965 and Adversarial Accuracy 0.40335648148148145
Finished evaluating Batch 28 with Normal Accuracy 0.42606026785714285 and Adversarial Accuracy 0.404296875
Finished evaluating Batch 29 with Normal Accuracy 0.4261853448275862 and Adversarial Accuracy 0.4035560344827586
Finished evaluating Batch 30 with Normal Accuracy 0.42734375 and Adversarial Accuracy 0.40598958333333335
Finished evaluating Batch 31 with Normal Accuracy 0.42893145161290325 and Adversarial Accuracy 0.4075100806451613
Finished evaluating Batch 32 with Normal Accuracy 0.427978515625 and Adversarial Accuracy 0.4072265625
Finished evaluating Batch 33 with Normal Accuracy 0.42945075757575757 and Adversarial Accuracy 0.4090909090909091
Finished evaluating Batch 34 with Normal Accuracy 0.4287683823529412 and Adversarial Accuracy 0.4078584558823529
Finished evaluating Batch 35 with Normal Accuracy 0.4299107142857143 and Adversarial Accuracy 0.4095982142857143
Finished evaluating Batch 36 with Normal Accuracy 0.4312065972222222 and Adversarial Accuracy 0.4110243055555556
Finished evaluating Batch 37 with Normal Accuracy 0.43053209459459457 and Adversarial Accuracy 0.4123733108108108
Finished evaluating Batch 38 with Normal Accuracy 0.4315378289473684 and

```
Adversarial Accuracy 0.4144736842105263
Finished evaluating Batch 39 with Normal Accuracy 0.4312900641025641 and
Adversarial Accuracy 0.41326121794871795
Finished evaluating Batch 40 with Normal Accuracy 0.4298828125 and Adversarial
Accuracy 0.41171875
Finished evaluating Batch 41 with Normal Accuracy 0.4293064024390244 and
Adversarial Accuracy 0.4106326219512195
Finished evaluating Batch 42 with Normal Accuracy 0.4291294642857143 and
Adversarial Accuracy 0.41127232142857145
Finished evaluating Batch 43 with Normal Accuracy 0.4284156976744186 and
Adversarial Accuracy 0.4106104651162791
Finished evaluating Batch 44 with Normal Accuracy 0.4286221590909091 and
Adversarial Accuracy 0.4112215909090909
Finished evaluating Batch 45 with Normal Accuracy 0.43020833333333336 and
Adversarial Accuracy 0.41128472222222223
Finished evaluating Batch 46 with Normal Accuracy 0.42866847826086957 and
Adversarial Accuracy 0.40998641304347827
Finished evaluating Batch 47 with Normal Accuracy 0.42686170212765956 and
Adversarial Accuracy 0.4080784574468085
Finished evaluating Batch 48 with Normal Accuracy 0.4261067708333333 and
Adversarial Accuracy 0.408203125
Finished evaluating Batch 49 with Normal Accuracy 0.4257015306122449 and
Adversarial Accuracy 0.40800382653061223
Finished evaluating Batch 50 with Normal Accuracy 0.42546875 and Adversarial
Accuracy 0.40796875
Finished evaluating Batch 51 with Normal Accuracy 0.4253982843137255 and
Adversarial Accuracy 0.40900735294117646
Finished evaluating Batch 52 with Normal Accuracy 0.4247295673076923 and
Adversarial Accuracy 0.41060697115384615
Finished evaluating Batch 53 with Normal Accuracy 0.42497051886792453 and
Adversarial Accuracy 0.4111143867924528
Finished evaluating Batch 54 with Normal Accuracy 0.4252025462962963 and
Adversarial Accuracy 0.41030092592592593
Finished evaluating Batch 55 with Normal Accuracy 0.42485795454545455 and
Adversarial Accuracy 0.40852272727272726
Finished evaluating Batch 56 with Normal Accuracy 0.4252232142857143 and
Adversarial Accuracy 0.4086216517857143
Finished evaluating Batch 57 with Normal Accuracy 0.4244791666666667 and
Adversarial Accuracy 0.4074835526315789
Finished evaluating Batch 58 with Normal Accuracy 0.4252424568965517 and
Adversarial Accuracy 0.40759698275862066
Finished evaluating Batch 59 with Normal Accuracy 0.4258474576271186 and
Adversarial Accuracy 0.4075741525423729
Finished evaluating Batch 60 with Normal Accuracy 0.4256510416666667 and
Adversarial Accuracy 0.407421875
Finished evaluating Batch 61 with Normal Accuracy 0.4262295081967213 and
Adversarial Accuracy 0.40791495901639346
Finished evaluating Batch 62 with Normal Accuracy 0.42502520161290325 and
```

```
Adversarial Accuracy 0.4067540322580645
Finished evaluating Batch 63 with Normal Accuracy 0.4252232142857143 and
Adversarial Accuracy 0.4069940476190476
Finished evaluating Batch 64 with Normal Accuracy 0.4254150390625 and
Adversarial Accuracy 0.4061279296875
Finished evaluating Batch 65 with Normal Accuracy 0.4243990384615385 and
Adversarial Accuracy 0.4051682692307692
Finished evaluating Batch 66 with Normal Accuracy 0.4238873106060606 and
Adversarial Accuracy 0.4054214015151515
Finished evaluating Batch 67 with Normal Accuracy 0.42455690298507465 and
Adversarial Accuracy 0.40543376865671643
Finished evaluating Batch 68 with Normal Accuracy 0.4245174632352941 and
Adversarial Accuracy 0.4055606617647059
Finished evaluating Batch 69 with Normal Accuracy 0.42481884057971014 and
Adversarial Accuracy 0.4060235507246377
Finished evaluating Batch 70 with Normal Accuracy 0.42544642857142856 and
Adversarial Accuracy 0.40669642857142857
Finished evaluating Batch 71 with Normal Accuracy 0.425506161971831 and
Adversarial Accuracy 0.4065801056338028
Finished evaluating Batch 72 with Normal Accuracy 0.4258897569444444 and
Adversarial Accuracy 0.4069010416666667
Finished evaluating Batch 73 with Normal Accuracy 0.4254066780821918 and
Adversarial Accuracy 0.406785102739726
Finished evaluating Batch 74 with Normal Accuracy 0.42588682432432434 and
Adversarial Accuracy 0.40625
Finished evaluating Batch 75 with Normal Accuracy 0.42635416666666665 and
Adversarial Accuracy 0.4075
Finished evaluating Batch 76 with Normal Accuracy 0.4263980263157895 and
Adversarial Accuracy 0.40799753289473684
Finished evaluating Batch 77 with Normal Accuracy 0.4260349025974026 and
Adversarial Accuracy 0.4074675324675325
Finished evaluating Batch 78 with Normal Accuracy 0.42568108974358976 and
Adversarial Accuracy 0.4074519230769231
Finished evaluating Batch 79 with Normal Accuracy 0.4209849683544304 and
Adversarial Accuracy 0.40278876582278483
Adv/Normal Accuracy on test images: 40 %, 42%
```

[10]: 
```
(4257, 4073, 10112)
```

Overall robust test accuracy 40%, normal accuracy 42%. Can increase with more training.