EECS 127 Optimization Models in Engineering

Project

Yunxuan Mao Xuan Jiang Shuai Liu

May 5, 2020

1 Exercise 1

Consider $\vec{x}' = \vec{x} + \varepsilon\vec{v}$, $\vec{x}$ is fixed.

So, we can get an inprovement over $\vec{v}$.

$$\|\vec{x} - \vec{x}'\|_\infty = \|\varepsilon\vec{v}\|_i nfty \leq \varepsilon \Longrightarrow \|\vec{v}\|_\infty \leq 1$$

In the optimization problem, because $\vec{x}$ is fixed, we can get an equivalent problem:

$$\arg\min_{\vec{v}} \bigtriangledown_{vecx} L(f_\theta(\vec{x}), \vec{y}_{true})^\top (\varepsilon\vec{v})$$

The optimal solution is $\vec{v} = sgn(L(f_\theta(\vec{x}), \vec{y}_{true}))$. Then we can get the optimal solution for the original problem:

$$\vec{x}_{FGSM} = \vec{x} + \varepsilon sgn(L(f_\theta(\vec{x}), \vec{y}_{true}))$$

2 Exercise 2

We can replace the constraints by adding some terms into the objective.

The first one is

$$\|\vec{z}_1 - \vec{x}\|_\infty \leq \varepsilon \quad \Longleftrightarrow \quad \min_{\vec{z}_1} \mathbf{1}_{B_\varepsilon(\vec{x})}(\vec{z}_1) = 0$$

The second one is

$$(z_{2j}, \hat{z_{2j}}) \in \mathcal{Z}_j \quad \Longleftrightarrow \quad \min_{\vec{z}_2, \hat{\vec{z}}_2} \sum_{j=1}^{n_2} \mathbf{1}_{\mathcal{Z}_j}(z_{2j}, \hat{z}_{2j}) = 0$$

Then, the convex relation can be expressed as

$$p^*(\vec{x}, \vec{c}) = \min_{\vec{z}} \vec{c}^\top \hat{\vec{z}}_3 + \mathbf{1}_{B_\varepsilon(\vec{x})}(\vec{z}_1) + \sum_{j=1}^{n_2} \mathbf{1}_{\mathcal{Z}_j}(z_{2j}, \hat{z}_{2j})$$

$$s.t. \quad \hat{\vec{z}}_2 = W_1\vec{z}_1 + \vec{b}_1$$

$$\hat{\vec{z}}_3 = W_2\vec{z}_2 + \vec{b}_2$$

3 Exercise 3

(a)

$$\mathcal{L}(\vec{z},\vec{\nu}) = \vec{c}^\top \vec{\hat{z}}_3 + \mathbf{1}_{B_\varepsilon(\vec{x})}(\vec{z}_1) + \sum_{j=1}^{n_2} \mathbf{1}_{\mathcal{Z}_j}(z_{2j},\hat{z}_{2j}) + \vec{\nu}_2^\top(\vec{\hat{z}}_2 - W_1\vec{z}_1 - \vec{b}_1) + \vec{\nu}_3^\top(\vec{\hat{z}}_3 - W_2\vec{z}_2 - \vec{b}_2)$$

$$= \vec{c}^\top \vec{\hat{z}}_3 + \vec{\nu_3}^\top \vec{\hat{z}}_3 + \mathbf{1}_{B_\varepsilon(\vec{x})}(\vec{z}_1) - \vec{\nu}_2^\top W_1\vec{z}_1 + \left(\sum_{j=1}^{n_2} \mathbf{1}_{\mathcal{Z}_j}(z_{2j},\hat{z}_{2j}) - \vec{\nu}_3^\top W_2\vec{z}_2 + \vec{\nu}_2^\top \vec{\hat{z}}_2\right) - \sum_{i=1}^{2} \vec{\nu}_{i+1}^\top \vec{b}_i$$

(b)

Because different terms in the lagrange had different variables, so we can separate the minimization problem:

$$g(\vec{\nu}_2,\vec{\nu}_3) = \min_{\vec{z}} \mathcal{L}(\vec{z},\vec{\nu})$$

$$= \min_{\vec{\hat{z}}_3}((\vec{c}^\top + \vec{\nu_3}^\top)\vec{\hat{z}}_3) + \min_{\vec{z}_1}(\mathbf{1}_{B_\varepsilon(\vec{x})}(\vec{z}_1) - \vec{\nu}_2^\top W_1\vec{z}_1) + \left(\sum_{j=1}^{n_2} \min_{z_{2j},\hat{z}_{2j}} (\mathbf{1}_{\mathcal{Z}_j}(z_{2j},\hat{z}_{2j}) - \vec{\nu}_3^\top W_2\vec{z}_2 + \vec{\nu}_2^\top \vec{\hat{z}}_2)\right) - \sum_{i=1}^{2} \vec{\nu}_{i+1}^\top \vec{b}_i$$

(c)

$$\min_{\vec{\hat{z}}_3}((\vec{c}^\top + \vec{\nu_3}^\top)\vec{\hat{z}}_3) = \begin{cases} 0, & \vec{c}^\top + \vec{\nu_3}^\top = 0 \\ -\infty, & otherwise \end{cases}$$

From the Fenchel conjugate, we know that $\min_{\vec{z}_1}(\mathbf{1}_{B_\varepsilon(\vec{x})}(\vec{z}_1) - \vec{\nu}_2^\top W_1\vec{z}_1) = -\min_{\vec{z}_1}(\mathbf{1}_{B_\varepsilon(\vec{x})}(W_1^\top\vec{\nu}_2))$

$$-\min_{\vec{z}_1}(\mathbf{1}_{B_\varepsilon(\vec{x})}(W_1^\top\vec{\nu}_2)) = -(\mathbf{1}^*_{B_\varepsilon(\vec{x})}(W_1^\top\vec{\nu}_2))$$

The same as before,

$$\left(\sum_{j=1}^{n_2} \min_{z_{2j},\hat{z}_{2j}} (\mathbf{1}_{\mathcal{Z}_j}(z_{2j},\hat{z}_{2j}) - \vec{\nu}_3^\top W_2\vec{z}_2 + \vec{\nu}_2^\top \vec{\hat{z}}_2)\right) = \sum_{j=1}^{n_2} -\mathbf{1}^*_{\mathcal{Z}_j}(\vec{\nu}_3^\top (W_2)_j, -\nu_{2j})$$

So, the dual is

$$d^*(\vec{x},\vec{c}) = \max_{\vec{\nu}} -(\mathbf{1}^*_{B_\varepsilon(\vec{x})}(W_1^\top\vec{\nu}_2)) + \sum_{j=1}^{n_2} -\mathbf{1}^*_{\mathcal{Z}_j}(\vec{\nu}_3^\top (W_2)_j, -\nu_{2j}) - \sum_{i=1}^{2} \vec{\nu}_{i+1}^\top \vec{b}_i$$

$$s.t. \vec{\nu}_3 = -\vec{c}$$

4 Exercise 4

Frenchel conjugate normal form:

$$\mathbf{1}^*_{B_\varepsilon(\vec{x})}(\vec{\nu}) = \max_{\vec{v}} \vec{v}^\top \vec{t} - \mathbf{1}_{B_\varepsilon(\vec{x})}(\vec{t})$$

which is equal to:

$$\max_{\vec{t}} \vec{v}^\top \vec{t}$$

$$s.t. \|\vec{t} - \vec{x}\|_\infty \le \varepsilon$$

The problem can be expressed as

$$\max_{\vec{t}} \vec{v}^\top \vec{t}$$

$$s.t. \vec{t} - \vec{x} \le \varepsilon$$

$$-\vec{t}+\vec{x}\le\varepsilon$$

If $\nu_i=0$, we have 0. If $\nu>0$, from condition one, we get

$$\nu_i(t_i-x_i)\le\nu_i\varepsilon \quad\Longrightarrow\quad \nu_i t_i\le\nu_i x_i+\nu_i\varepsilon$$

If $\nu<0$, from condition two, we get

$$\nu_i(t_i-x_i)\ge\nu_i\varepsilon \quad\Longrightarrow\quad \nu_i t_i\le\nu_i x_i-\nu_i\varepsilon$$

We can rewrite these inequation as $\nu_i x_i+\varepsilon|\nu_i|$, so

$$\mathbf{1}^*_{B_\varepsilon(\vec{x})}(\vec{\nu})=\vec{\nu}^\top\vec{x}+\varepsilon\|\vec{\nu}\|_1$$

5 Exercise 5

From exercise 4, we got $\mathbf{1}^*_{B_\varepsilon(\vec{x})}(\vec{\hat{\nu}_1})=\vec{\hat{\nu}}_1^\top\vec{x}+\varepsilon\|\vec{\hat{\nu}}_1\|_1$

From the propostion, we know that

$$\sum_{j=1}^{n_2}-\mathbf{1}^*_{\mathcal{Z}_j}(\nu_{2j},-\nu_{2j})=\begin{cases}0, & \nu_{2j}=0\\ 0, & \nu_{2j}=-\nu_{2j}\\ ReLU(-l_j\nu), & \nu_{2j}=\dfrac{\hat{\nu}_{2j}u_j}{u_j-l_j}\end{cases}=\sum_{j\in S}l_j ReLU(\nu_{2j})$$

So, the dual problem can be expressed as

$$d^*(\vec{x},\vec{c})=\max_{\vec{\nu}}-\vec{\nu}^\top\vec{x}-\varepsilon\|\vec{\nu}\|_1-\sum_{i=1}^{2}\vec{\nu}_{i+1}^\top\vec{b}_i+\sum_{j\in S}l_j ReLU(\nu_{2j})$$

$$\begin{aligned}s.t.\quad &\vec{\nu}_3=-\vec{c}\\ &\vec{\hat{\nu}}_2=W_2^\top\vec{\nu}_3\\ &\nu_{2j}=0\\ &\nu_{2j}=-\nu_{2j}\\ &\nu_{2j}=\frac{\hat{\nu}_{2j}u_j}{u_j-l_j}\\ &\vec{\hat{\nu}}_1=W_1^\top\vec{v}_2\end{aligned}$$

6 Exercise 6

We know that $\vec{\hat{z}}_2=W_1\vec{z}_1+\vec{b_1}=W_1\vec{x}+\vec{b_1}$.

For $\|W\|_{:1}$, every component is non-negative. Because $\varepsilon\ge0$,

$$\vec{u}=\vec{\hat{z}}_2+\varepsilon\|W_1\|_{:1}=W_1\vec{x}+\vec{b_1}+\varepsilon\|W_1\|_{:1}\succeq\vec{\hat{z}}_2$$

$$\vec{l}=\vec{\hat{z}}_2-\varepsilon\|W_1\|_{:1}=W_1\vec{x}+\vec{b_1}-\varepsilon\|W_1\|_{:1}\preceq\vec{\hat{z}}_2$$

So, $\vec{u}$ and $\vec{l}$ are the upper and lower bound for $\vec{z}_2$.

7 Exercise 7

$$L(\vec{y}, \vec{y}_{true}) = -\log(\frac{e^{y_{itrue}}}{\sum_{j=1}^{m} e^{y_j}})$$

$$L(\vec{y}, \vec{y}_{true}) - L(\vec{y}', \vec{y}_{true}) = -\log(\frac{e^{y_{itrue}}}{\sum_{j=1}^{m} e^{y_j}}) + \log(\frac{e^{y'_{itrue}}}{\sum_{j=1}^{m} e^{y'_j}}) = \log(\frac{e^{y'_{itrue}} \sum_{j=1}^{m} e^{y_j}}{e^{y_{itrue}} \sum_{j=1}^{m} e^{y'_j}})$$

Use the property that when $a/b \in (0,1)$ and for an positive c, $(a+c)/(b+c) \leq 1$.

$$\frac{e^{y'_{itrue}} \sum_{j=1}^{m} e^{y_j}}{e^{y_{itrue}} \sum_{j=1}^{m} e^{y'_j}} = \frac{e^{y'_{itrue}} (\sum_{j=1, j\neq itrue}^{m} e^{y_j} + e^{y_{itrue}})}{e^{y_{itrue}} (\sum_{j=1, j\neq itrue'}^{m} e^{y'_j} + e^{y_{itrue'}})}$$

Because $\frac{e^{y'_{itrue}} \sum_{j=1, j\neq itrue}^{m} e^{y_j}}{e^{y_{itrue}} \sum_{j=1, j\neq itrue'}^{m} e^{y_j}} \leq 1$,

$$\frac{e^{y'_{itrue}} \sum_{j=1}^{m} e^{y_j}}{e^{y_{itrue}} \sum_{j=1}^{m} e^{y'_j}} = \frac{e^{y'_{itrue}} (\sum_{j=1, j\neq itrue}^{m} e^{y_j} + e^{y_{itrue}})}{e^{y_{itrue}} (\sum_{j=1, j\neq itrue'}^{m} e^{y'_j} + e^{y_{itrue'}})} \leq 1$$

So, $L(\vec{y}, \vec{y}_{true}) - L(\vec{y}', \vec{y}_{true}) \leq 0$, cross-entropy loss is monotonic.

$$L(\vec{y} - a\mathbf{1}, \vec{y}_{true}) = -\log(\frac{e^{y_{itrue}-a}}{\sum_{j=1}^{m} e^{y_j - a}}) = -\log(\frac{e^{y_{itrue}}}{\sum_{j=1}^{m} e^{y_j}}) - \log(\frac{e^{-a}}{e^{-a}}) = -\log(\frac{e^{y_{itrue}}}{\sum_{j=1}^{m} e^{y_j}}) = L(\vec{y}, \vec{y}_{true})$$

So, cross-entropy loss is translation-invariant.

Provable Robustness for Deep Classifiers

In this notebook, we will implement the robustness certificate that we derived in the PDF. That is, we will first define and train a three-layer neural classifier; then, we will calculate its dual, and using this, check whether the classifier is dual at given input points.

Your task is to fill in any sections labeled TODO in the code and answer the bolded questions.

Torch

We are using torch here; for our purposes, we can think of torch as essentially numpy with GPU support and and automatic differentiation. That is, for any function we compute, torch automatically keeps track of the function's gradient with respect to inputs; this will make gradient descent much easier to implement.

The primary object you will need to manipulate here is `torch.Tensor` , which can be thought of as equivalent to `np.array` . Indexing, splicing, multiplication, etc. will work like you would expect them to in numpy.

Also, most of the numpy functions you are used to are present in torch, with the same name. E.g:

- `np.max(input, axis) --> torch.max(input, dim)` (Note that `torch.max` actually returns a tuple of the max and argmax).
- `np.zeros --> torch.zeros`
- `np.eye --> torch.eye`
- `np.linalg.norm(x, ord, axis) --> torch.norm(input, p, dim)`

For more information, refer to the torch documentation (https://pytorch.org/docs/stable/torch.html) or the torch tutorials (https://pytorch.org/tutorials/).

Setup

Here, we import the relevant libraries.

In [0]:
```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import copy
```

This line tells torch to use the GPU if available, and otherwise the CPU.

In [44]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

Here, we load in the MNIST dataset. The inputs are 28×28 images of handwritten digits, while the labels are the corresponding digit. Note that we split the data between a training set and a test set. In order to have an unbiased estimate of the classifier's performance, we must train the model only on the training set (**never the test set**), then test its accuracy on the test set.

In [0]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=T

trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

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

testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
```

This is a utility function to visualize torch Tensors as images.

In [0]:
```
def imshow(img):
    '''
    Visualizes IMG.
    IMG should be a 2D torch Tensor.
    '''
    img = img / 2 + 0.5     # unnormalize
    npimg = img.detach().numpy()
    plt.imshow(npimg, cmap='gray')
    plt.show()
```

Primal Network

Here, we define the neural classifier we will be using. Note that the network comprises three layers. The first layer has dimension 28^2 since this is the size of the input image. (The original inputs are square images, but we flatten them into a $28^2 \times 1$ vector in order to feed them into the network.) The output layer has dimension 10, since there are ten possible output classes (the digits 0-9). The hidden layer has dimension 256. (There isn't as much science behind choosing the dimensionality of input layers; we choose 256 because it is a round number, and is hopefully enough to the neccesary encode information about the input image.)

In [0]:
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        self.fc1 = nn.Linear(in_features=28*28, out_features = 256)
        self.fc2 = nn.Linear(in_features=256, out_features = 10)
        self.layers = [self.fc1, self.fc2]

  # define forward function
    def forward(self, t):
        '''
        On input T, performs a affine transformation, then
        a ReLU, then another affine transformation.
        '''
        self.z = []
        t = t.reshape(-1, 28*28)
        t = self.fc1(t)
        self.z.append(t)
        t = F.relu(t)
        t = self.fc2(t)
        self.z.append(t)
        return t
```

Here is the training code, which uses Adam, a variant of gradient descent. The actual optimization machinery is all abstracted away behind the torch library; all the work is being done by the `optimizer.step()` call.

In [0]:
```
def train(net, criterion, trainloader, lr=0.001):
    '''
    Uses the Adam optimization algorithm to train
    the classifier NET on training data from TRAINLOADER,
    on loss function CRITERION, with learning rate LR.

    Note that we half the learning rate each epoch.
    '''
    optimizer = optim.Adam(net.parameters(), lr=lr)

    for epoch in range(3):
        for i, data in enumerate(trainloader, 0):
            for param_group in optimizer.param_groups:
                param_group['lr'] = lr * 0.5 ** (epoch)

            inputs, labels = data[0].to(device), data[1].to(device)
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            if i % 500 == 0:
                print('Epoch', epoch, 'Iter:', i, 'Loss', loss.item())
```

We can now train the net on the training data, using cross entropy loss.

In [49]:
```
net = Net()
net.to(device)

criterion = nn.CrossEntropyLoss()

train(net, criterion, trainloader, 0.001)
```

```
Epoch 0 Iter: 0 Loss 2.289048194885254
Epoch 0 Iter: 500 Loss 0.12315428256988525
Epoch 0 Iter: 1000 Loss 0.27355486154556274
Epoch 0 Iter: 1500 Loss 0.19797652959823608
Epoch 0 Iter: 2000 Loss 0.11937451362609863
Epoch 0 Iter: 2500 Loss 1.3180737495422363
Epoch 0 Iter: 3000 Loss 0.5733668804168701
Epoch 0 Iter: 3500 Loss 0.022939205169677734
Epoch 0 Iter: 4000 Loss 0.10624265670776367
Epoch 0 Iter: 4500 Loss 0.32269287109375
Epoch 0 Iter: 5000 Loss 0.005548119544982910
Epoch 0 Iter: 5500 Loss 0.20479059219360352
Epoch 0 Iter: 6000 Loss 0.022185683250427246
Epoch 0 Iter: 6500 Loss 0.041847407817840576
Epoch 0 Iter: 7000 Loss 0.06954514980316162
Epoch 0 Iter: 7500 Loss 0.00995945930480957
Epoch 0 Iter: 8000 Loss 0.09848421812057495
Epoch 0 Iter: 8500 Loss 0.00448453426361084
Epoch 0 Iter: 9000 Loss 0.008963227272033691
Epoch 0 Iter: 9500 Loss 0.0257819294929504
Epoch 0 Iter: 10000 Loss 0.004914045333862305
Epoch 0 Iter: 10500 Loss 1.4364607334136963
Epoch 0 Iter: 11000 Loss 0.0404353141784668
Epoch 0 Iter: 11500 Loss 0.003002762794494629
Epoch 0 Iter: 12000 Loss 0.7548469305038452
Epoch 0 Iter: 12500 Loss 0.002043008804321289
Epoch 0 Iter: 13000 Loss 0.09219479560852051
Epoch 0 Iter: 13500 Loss 0.0067255496978759766
Epoch 0 Iter: 14000 Loss 0.409676194190979
Epoch 0 Iter: 14500 Loss 0.008940458297729492
Epoch 1 Iter: 0 Loss 0.13097059726715088
Epoch 1 Iter: 500 Loss 0.08075040578842163
Epoch 1 Iter: 1000 Loss 0.0789598822593689
Epoch 1 Iter: 1500 Loss 0.07522368431091309
Epoch 1 Iter: 2000 Loss 0.028467237949371338
Epoch 1 Iter: 2500 Loss 0.005944132804870605 5
Epoch 1 Iter: 3000 Loss 0.018410921096801758
Epoch 1 Iter: 3500 Loss 0.05136960744857788
Epoch 1 Iter: 4000 Loss 0.009801268577575684
Epoch 1 Iter: 4500 Loss 0.05542159080505371
Epoch 1 Iter: 5000 Loss 0.03602343797683716
Epoch 1 Iter: 5500 Loss 0.007740974426269531
Epoch 1 Iter: 6000 Loss 0.1637621521949768
Epoch 1 Iter: 6500 Loss 0.0008903741836547852
Epoch 1 Iter: 7000 Loss 0.004818558692932129
Epoch 1 Iter: 7500 Loss 0.06691169738769531
Epoch 1 Iter: 8000 Loss 0.0010306835174560547
Epoch 1 Iter: 8500 Loss 0.002672433853149414
Epoch 1 Iter: 9000 Loss 0.01766186952590942 4
Epoch 1 Iter: 9500 Loss 2.5033950805664062e-05
Epoch 1 Iter: 10000 Loss 0.013769865036010742
Epoch 1 Iter: 10500 Loss 0.0025058984756469727
Epoch 1 Iter: 11000 Loss 0.08055758476257324
Epoch 1 Iter: 11500 Loss 0.051490843296051025
Epoch 1 Iter: 12000 Loss 5.042552947998047e-05
Epoch 1 Iter: 12500 Loss 0.47020018100738525
Epoch 1 Iter: 13000 Loss 0.00245511531829834
Epoch 1 Iter: 13500 Loss 0.14051032066345215
Epoch 1 Iter: 14000 Loss 0.006453335285186768
Epoch 1 Iter: 14500 Loss 0.01865246891975403
Epoch 2 Iter: 0 Loss 0.0012974739074707031
Epoch 2 Iter: 500 Loss 0.022809267044067383
Epoch 2 Iter: 1000 Loss 0.2555362284183502
Epoch 2 Iter: 1500 Loss 0.0005691051483154297
Epoch 2 Iter: 2000 Loss 1.6739697456359863
Epoch 2 Iter: 2500 Loss 0.0011223554611206055
Epoch 2 Iter: 3000 Loss 0.017247498035430908
Epoch 2 Iter: 3500 Loss 0.1893690824508667
Epoch 2 Iter: 4000 Loss 0.20644605159759521
Epoch 2 Iter: 4500 Loss 0.00014829635620117188
Epoch 2 Iter: 5000 Loss 0.0031902194023132324
Epoch 2 Iter: 5500 Loss 6.508827209472656e-05
Epoch 2 Iter: 6000 Loss 0.00427478551864624
Epoch 2 Iter: 6500 Loss 0.0032018423080444336
Epoch 2 Iter: 7000 Loss 0.02210104465484619
Epoch 2 Iter: 7500 Loss 0.00044596195220947266
Epoch 2 Iter: 8000 Loss 0.04688990116119385
Epoch 2 Iter: 8500 Loss 0.0019412040710449219
Epoch 2 Iter: 9000 Loss 0.04879683256149292
Epoch 2 Iter: 9500 Loss 0.004187583923339844
Epoch 2 Iter: 10000 Loss 0.06578835844993591
Epoch 2 Iter: 10500 Loss 0.008633971214294434
Epoch 2 Iter: 11000 Loss 0.1966876983642578
Epoch 2 Iter: 11500 Loss 0.001170039176940918
Epoch 2 Iter: 12000 Loss 0.0003746151924133301
Epoch 2 Iter: 12500 Loss 0.006283760070800781
Epoch 2 Iter: 13000 Loss 0.0009174346923828125
Epoch 2 Iter: 13500 Loss 0.5940966606140137
Epoch 2 Iter: 14000 Loss 0.0009891986846923828
Epoch 2 Iter: 14500 Loss 0.023622632026672363
```

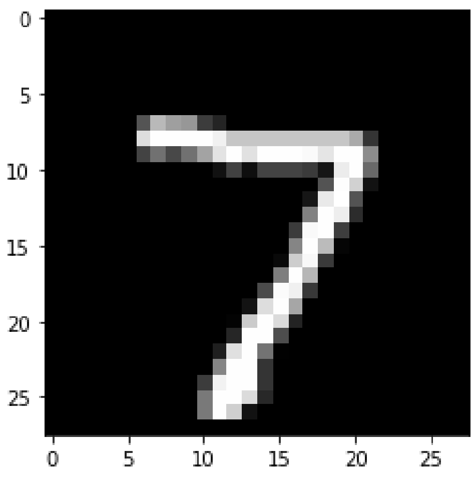
Let's load a sample image from the test dataset, and see what the classifier makes of it. Make sure to visualize the image using `imshow(x[0,0])` . Also, note that the line `test_iter.next()` pulls a new input image from the test set each time you run it; try running the next code block a few times to get a sense of what the MNIST dataset looks like, and how the classifier performs on it.

In [0]:
```
test_iter = iter(testloader)
```

In [51]:
```
x, labels = test_iter.next()
x = x[0].unsqueeze(0)
labels = labels[0].unsqueeze(0)
imshow(x[0,0])

x = x.to(device)
labels = labels.to(device)

out = net(x).data
print('Classifier output:', out)
print('Classifier prediction:', torch.argmax(out).item())
```



```
Classifier output: tensor([[-13.0860, -12.7495,  -6.1859,  -0.7290, -22.6
499,  -7.4789, -36.9544,
           7.4793, -10.2039,  -2.5041]], device='cuda:0')
Classifier prediction: 7
```

We can also measure the classifier's accuracy on the full test dataset. This function takes in a classifier we have trained and the loader for the test set, and outputs the classifier's accuracy. The accuracy is simply

$$\frac{\#\text{ correct}}{\#\text{ total}}.$$

In [0]:
```
def accuracy(net, testloader):
    '''
    Returns the accuracy of classifier NET
    on test data from TESTLOADER.
    '''
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return correct / total
```

In [53]:
```
print('Classifier accuracy on original test dataset:', accuracy(net, testlc
```

```
Classifier accuracy on original test dataset: 0.9712
```

Fast Gradient Sign Method

Here, we implement the Fast Gradient Sign Method, which takes in a batch of input images, their labels, a trained classifier, and the epsilon radius within which the perturbation should lie. This function should output the input image perturbed in the direction of the sign of the gradient with respect to the classifier's loss.

(Note that the output is not guaranteed to lie in the valid range for images, since here pixel values must be in $[-1, 1]$. You should use `torch.clamp` to fix the FGSM output to lie in the correct range.)

In [0]:
```
def FGSM(x, labels, net, eps):
    '''
    Given an input image X and its corresponding labels
    LABELS, as well as a classifier NET, returns X
    perturbed by EPS using the fast gradient sign method.
    '''
    net.zero_grad()    # Zero out any gradients from before
    x.requires_grad=True    # Keep track of gradients
    out = net(x)    # Output of classifier
    criterion = nn.CrossEntropyLoss()
    loss = criterion(out, labels)    # Classifier's loss
    loss.backward()
    grads = x.grad.data    # Gradient of loss w/r/t input
    x_next = torch.clamp(x + eps * torch.abs(grads), min = -1, max = 1)
    return x_next
```

Let's see how well the classifier does when the input is adversarially perturbed using FGSM. Try this for $\varepsilon \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$, and again remember to visualize the inputs with `imshow`.

In [0]:
```
#eps =  # TODO: Try eps = 0.05, 0.1, 0.2, 0.3, 0.4
```

In [67]:
```
# We are using the same sample input x as before.
for eps in [0.05, 0.1, 0.2, 0.3, 0.4]:
  x.requires_grad = True
  x_prime = FGSM(x, labels, net, eps)
  #imshow(x_prime[0,0].cpu())
  out = net(x_prime)
  print('=================')
  print('epsilon: '+str(eps))
  print('Classifier output:', out.data)
  print('Classifier prediction:', torch.argmax(out).item())
  for i in range(10):
    c = torch.zeros(10, 1).to(device)
    if i != labels:
        c[i] = -1
        c[labels] = 1
        print(i, (out @ c).item())
```

```
=================
epsilon: 0.05
Classifier output: tensor([[-13.0861, -12.7495,  -6.1858,  -0.7275, -22.6
481,  -7.4780, -36.9536,
           7.4780, -10.2011,  -2.5030]], device='cuda:0')
Classifier prediction: 7
0 20.564090728759766
1 20.227432250976562
2 13.663785934448242
3 8.205513000488281
4 30.126083374023438
5 14.956001281738281
6 44.43156814575195
8 17.67905616760254
9 9.980976104736328
=================
epsilon: 0.1
Classifier output: tensor([[-13.0862, -12.7494,  -6.1857,  -0.7259, -22.6
462,  -7.4771, -36.9527,
           7.4765, -10.1980,  -2.5017]], device='cuda:0')
Classifier prediction: 7
0 20.562711715698242
1 20.225910186767578
2 13.662158966064453
3 8.202377319335938
4 30.12264633178711
5 14.953545570373535
6 44.42918014526367
8 17.674474716186523
9 9.9782133102417
=================
epsilon: 0.2
Classifier output: tensor([[-13.0865, -12.7494,  -6.1854,  -0.7225, -22.6
421,  -7.4751, -36.9508,
           7.4733, -10.1915,  -2.4991]], device='cuda:0')
Classifier prediction: 7
0 20.559818267822266
1 20.22270393371582
2 13.658737182617188
3 8.19577693939209
```

```
4 30.115398406982422
5 14.948369979858398
6 44.42414093017578
8 17.66482925415039
9 9.97239875793457
=================
epsilon: 0.3
Classifier output: tensor([[-13.0868, -12.7493,  -6.1851,  -0.7187, -22.6
376,  -7.4728, -36.9487,
           7.4699, -10.1844,  -2.4962]], device='cuda:0')
Classifier prediction: 7
0 20.55663299560547
1 20.219175338745117
2 13.654973983764648
3 8.188516616821289
4 30.107425689697266
5 14.942681312561035
6 44.41859436035156
8 17.654218673706055
9 9.966001510620117
=================
epsilon: 0.4
Classifier output: tensor([[-13.0871, -12.7493,  -6.1848,  -0.7145, -22.6
327,  -7.4704, -36.9465,
           7.4661, -10.1766,  -2.4930]], device='cuda:0')
Classifier prediction: 7
0 20.55316162109375
1 20.215328216552734
2 13.65086841583252
3 8.180598258972168
4 30.098731994628906
5 14.936477661132812
6 44.41255569458008
8 17.64264678955078
9 9.959026336669922
```

We should evaluate the classifier's performance on FGSM-perturbed data by the same metric that we will later use in the primal adversarial problem. That is, for the classifier's output vector $\vec{\hat{z}}_3$, we want to compute

$$\vec{c}_j^\top \vec{\hat{z}}_3$$

where

$$\vec{c}_j = \vec{y}_{\text{true}} - \vec{e}_j$$

for each $j \in [10]$.

Recall that

$$\vec{c}_j^\top \vec{\hat{z}}_3 = \vec{\hat{z}}_{3i_{\text{true}}} - \vec{\hat{z}}_{3j},$$

i.e. $\vec{c}_j^\top \vec{\hat{z}}_3$ is the difference between the classifier's confidence on the true class and the jth (incorrect) class. If $\vec{c}_j^\top \vec{\hat{z}}_3$ is positive for all incorrect j, then the classifier was not fooled by the adversarial perturbation.

```
In [57]: for i in range(10):
             c = torch.zeros(10, 1).to(device)
             if i != labels:
                 c[i] = -1
                 c[labels] = 1
                 print(i, (out @ c).item())
```

```
0 20.562423706054688
1 20.2255859375
2 13.66181755065918
3 8.201717376708984
4 30.12192153930664
5 14.953027725219727
6 44.428672790527344
8 17.67350959777832
9 9.977631568908691
```

Q: What do the $\vec{c}_j^\top \hat{\vec{z}}_3$ scores tell you about the robustness of the classifier to different values of epsilon? For a given input digit, which output categories have higher/lower scores? Why?

A: The $\vec{c}_j^\top \hat{\vec{z}}_3$ score decreases with the value of epsilon increasing, which indicates that the robustness of the classifier to a smaller epsilon value is better than that to a larger epsilon value. Given the input digit is 7, 5 & 6 have higher scores and 2 & 9 have lower scores because 5 & 6 have features that are least similar to 7 and 2 & 9 have features that are most similar to 7. The larger the score is, the more confident the classifier is that the input digit is not of this output class.

Now that FGSM is defined, we can also measure a classifier's accuracy on a dataset where each input has been adversarially perturbed. That is, for each point in the original test dataset, we first perturb it using FGSM before feeding it to the classifier.

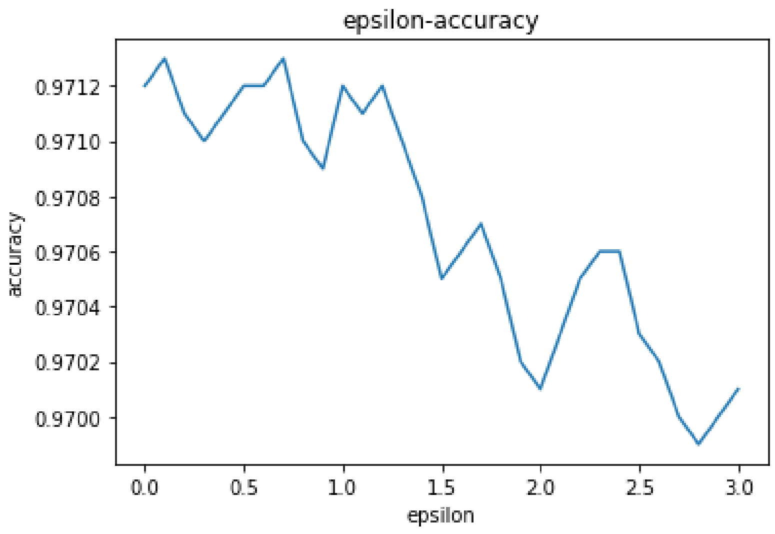
```
In [0]: def accuracy_on_FGSM(net, testloader, eps):
            '''
            Returns the accuracy of classifier NET on test
            data from TESTLOADER that has been perturbed by
            EPS using FSGM.
            '''
            correct = 0
            total = 0
            for data in testloader:
                x, labels = data[0].to(device), data[1].to(device)
                x_prime = FGSM(x, labels, net, eps)
                outputs = net(x_prime)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

            return correct / total
```

In [0]:
```
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
x_plot = []
y_plot = []
for eps in np.array(range(0,31), dtype=float)/10:
  #print('Classifier accuracy on test dataset perturbed with FGSM, eps is {
  cury = accuracy_on_FGSM(net, testloader, eps)
  x_plot.append(eps)
  y_plot.append(cury)
```

In [62]:
```
plt.title('epsilon-accuracy')
plt.xlabel('epsilon')
plt.ylabel('accuracy')
plt.plot(x_plot,y_plot)
```

Out[62]:
```
[<matplotlib.lines.Line2D at 0x7f68d50a44a8>]
```



Q: How does the classifier accuracy on data perturbed by FGSM compare to that on the original test dataset? How does this vary with epsilon?

A: It is plotted above.

Dual Network

Here, we will implement the dual network. First, we write the function to compute upper and lower bounds for the dual network. This function should take an input image, the trained classifier, and an epsilon value, and return the tuple

$$(\vec{l}, \vec{u}, S, S^-, S^+)$$

where $\vec{u}$ and $\vec{l}$ are the upper and lower bounds, respectively, for the input to the ReLU layer, and S^-, S^+, S are sets defined by

$$
\begin{aligned}
S &:= \{j \in [n_2] \mid l_j \le 0 \le u_j\} \\
S^- &:= \{j \in [n_2] \mid l_j \le u_j \le 0\} \\
S^+ &:= \{j \in [n_2] \mid 0 \le l_j \le u_j\}.
\end{aligned}
$$

See Section 6 of the PDF for more details.

In [0]:
```
def dual_bounds(x, net, eps):
    '''
    Given a classifier NET, an input image X,
    and the epsilon parameter EPS, returns the lower
    and upper bounds L and U respectively, as well as
    the corresponding sets S, S_MIN, S_PLUS.
    '''
    x = torch.tensor(x[0].reshape(-1, 1)).to(device)    # Reshape input to
    W = [layer.weight for layer in net.layers]    # Array of network weight
    b = [layer.bias.reshape(-1, 1) for layer in net.layers]    # Array of n
    n = W[1].shape[1]    # Dimensionality of hidden layer
    b0 = b[0].to(device)
    W0 = W[0].to(device)
    factor = torch.norm(W[0],p=1,dim=1).reshape(-1,1)
    #print(factor.shape)
    u = W[0].mm(x) + b[0]+eps *factor
    l = W[0].mm(x) + b[0] - eps *factor

    S = ((u>=0) & (l<=0)).clone().detach()
    S_plus = (l>=0).clone().detach()
    S_min = (u<=0).clone().detach()
    return l, u, S, S_min, S_plus
```

Given the tuple (l, u, S, S^-, S^+), we are ready to calculate the dual objective itself. This function should take in an input image, the classifier, a vector c, and the (l, u, S, S^-, S^+) from the previous function in order to output

$$
d^*(\vec{x}, \vec{c}) = -\vec{\hat{v}}_1^\top \vec{x} - \epsilon \|\vec{\hat{v}}_1\|_1 - \sum_{i=1}^{2} \vec{v}_{i+1}^\top \vec{b}_i + \sum_{j \in S} l_j \mathrm{ReLU}(v_{2j})
$$

Where the $\vec{v}$ vectors are computed as

$$
\begin{aligned}
\vec{v}_3 &= -\vec{c} & \\
\vec{\hat{v}}_2 &= W_2^\top \vec{v}_3 & \\
v_{2j} &= 0 & \forall j \in S^- \\
v_{2j} &= \hat{v}_{2j} & \forall j \in S^+ \, . \\
v_{2j} &= \frac{u_j}{u_j - l_j} \hat{v}_{2j} & \forall j \in S \\
\vec{\hat{v}}_1 &= W_1^\top \vec{v_2} &
\end{aligned}
$$

Again, see Section 6 of the PDF for more details.

One efficient way to compute $\vec{v}_2$ is to rewrite it as

$$
\vec{v}_2 = D\vec{\hat{v}}_2,
$$

where D is a diagonal matrix defined by

$$D_{jj} = \begin{cases} 0 & j \in S^- \\ \hat{v}_{2j} & j \in S^+ \\ \dfrac{u_j}{u_j - l_j}\hat{v}_{2j} & j \in S. \end{cases}$$

In [0]:
```
# Constructs the diagonal D matrix from the S sets, n (the dimensionality
# of the hidden layer), u, and l.
def StoD(S_min, S_plus, S, n, u, l):
    '''
    Given upper and lower bounds U and L, as well
    as the corresponding sets S_MIN, S_PLUS, and S,
    as well as the dimension of the hidden layer N,
    returns the corresponding diagonal matrix D.
    '''
    d = []
    for j in range(n):
        if S[j] and not (S_min[j] or S_plus[j]):
            d.append((u[j] / (u[j] - l[j])).item())
        elif S_plus[j] and not (S[j] or S_min[j]):
            d.append(1)
        elif S_min[j] and not (S[j] or S_plus[j]):
            d.append(0)
        else:
            assert False, 'StoD error.'
    return torch.diag(torch.Tensor(d)).to(device)

def dual_forward(x, net, c, eps, l, u, S, S_min, S_plus):
    '''
    Calculates the dual objective for classifier NET with input X
    and dual input C and epsilon parameter S. Depends on lower
    and upper bounds L and U, as well as the corresponding sets
    S, S_MIN, S_PLUS.
    '''
    x = x[0].reshape(-1, 1)    # Reshape input to more convenient dimensior
    W = [layer.weight for layer in net.layers]    # Array of network weight
    b = [layer.bias.reshape(-1, 1) for layer in net.layers]    # Array of r
    n = W[1].shape[1]    # Dimensionality of hidden layer
    D = StoD(S_min, S_plus, S, n, u, l)
    # TODO: Your code here!
    v3 = -1 * c.to(device)
    v2_hat = torch.transpose(W[1],0,1).to(device).mm(v3).to(device)
    #print(v2_hat)
    v2 = D.mm(v2_hat).to(device)
    v1_hat = torch.transpose(W[0],0,1).to(device).mm(v2).to(device)
    d_star = -1 * torch.transpose(v1_hat,0,1).mm(x).to(device)
    # print(torch.norm(v1_hat, 1, 0).shape)
    # print('pre'+str(d_star.shape))
    # print('norm'+str(torch.norm(v1_hat, 1, 0).shape))
    d_star = d_star - eps * (torch.norm(v1_hat, 1, 1)[0])
    # print(d_star.shape)
    d_star = d_star - torch.transpose(v2, 0, 1).mm(b[0].to(device)) - torch
    # print(d_star.shape)
    v2_reshape = v2[S].reshape(-1,1).to(device)
    v2_relu = torch.where(v2_reshape<0, torch.zeros(v2_reshape.shape).to(de
    d_star = d_star + l[S].reshape(1,-1).mm(v2_relu)
    #print(l[S].reshape(1,-1).mm(v2_relu))
    #print(torch.sum(l[S]))
    # print('dstar '+str(d_star.shape))

    return d_star
```

Now, we can use the dual network to check the robustness of the network we just trained on sample input images. We can do this for

$$\vec{c}_j = \vec{y}_{\text{true}} - \vec{e}_j$$

for each $j \in [10]$.

The output is a vector where the jth element is the difference between the model's confidence in the true class and the jth class; if $d^*(\vec{x}, \vec{c}_j)$ is nonnegative for every $j \in [10]$, then we know the model is robust to perturbations of size ε. See Section 8 of the PDF for more details.

Try running the following block of code for different values of $\varepsilon \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$, and compare the robustness guarantees here with the classifier's performance on the FGSM data from before.

```
In [0]: eps = 0.05 # TODO: Try eps = 0.05, 0.1, 0.2, 0.3, 0.4
```

In [73]:

```
def dual_estimate(eps):
  # We are still using the same sample input x as before.
  l, u, S, S_min, S_plus = dual_bounds(x, net, eps)

  # Here, we loop through each column c_j defined above, and output the

  # objective value for the dual function with input c.
  for i in range(10):
      c = torch.zeros(10, 1).to(device)
      if i != labels:
          c[i] = -1
          c[labels] = 1
          print(i, dual_forward(x, net, c, 0.1, l, u, S, S_min, S_plus).ite

for e in [0.05, 0.1, 0.2, 0.3, 0.4]:
  print(e)
  x.requires_grad = True
  x_prime = FGSM(x, labels, net, eps)
  #imshow(x_prime[0,0].cpu())
  out = net(x_prime)
  dual_estimate(e)
  print('===============')
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: UserWarni
ng: To copy construct from a tensor, it is recommended to use sourceTenso
r.clone().detach() or sourceTensor.clone().detach().requires_grad_(True),
rather than torch.tensor(sourceTensor).


0.05
0 19.53915786743164 20.56372833251953
1 19.347139358520508 20.227033615112305
2 12.508111953735352 13.663359642028809
3 7.082707405090332 8.204689025878906
4 28.81760597229004 30.12518310546875
5 13.947456359863281 14.955357551574707
6 43.09099578857422 44.43094253540039
8 16.27264976501465 17.677852630615234
9 8.88763427734375 9.980249404907227
===============
0.1
0 16.215076446533203 20.563655853271484
1 16.082265853881836 20.226951599121094
2 9.771034240722656 13.663272857666016
3 3.394176483154297 8.20452404022168
4 24.488967895507812 30.124998092651367
5 9.768722534179688 14.955225944519043
6 39.670955657958984 44.430816650390625
8 11.86845874786377 17.677610397338867
9 4.873041152954102 9.980103492736816
===============
0.2
0 1.580510139465332 20.563581466674805
1 4.184932708740234 20.22687339782715
2 -3.022064208984375 13.663187026977539
3 -11.380857467651367 8.204357147216797
```

```
4 8.692623138427734 30.124818801879883
5 -7.031081199645996 14.955095291137695
6 23.61827850341797 44.43069076538086
8 -3.647027015686035 17.6773681640625
9 -10.130411148071289 9.979957580566406
==============
0.3
0 -18.1629638671875 20.56351089477539
1 -14.541738510131836 20.226789474487305
2 -22.909034729003906 13.663101196289062
3 -32.05162811279297 8.204193115234375
4 -14.3287353515625 30.12463569641113
5 -28.885379791259766 14.95496654510498
6 -1.6692161560058594 44.430564880371094
8 -22.98923110961914 17.677125930786133
9 -30.045822143554688 9.979812622070312
==============
0.4
0 -36.52280807495117 20.563438415527344
1 -33.67796325683594 20.226709365844727
2 -43.184146881103516 13.663015365600586
3 -52.25334548950195 8.204028129577637
4 -36.942501068115234 30.124452590942383
5 -51.369293212890625 14.95483684539795
6 -26.28948211669922 44.4304313659668
8 -42.08064651489258 17.676883697509766
9 -49.66987991333008 9.979665756225586
==============
```

Q: What do the dual network outputs tell you about the robustness of the classifier? How does this compare to the classifier's performance (in particular, the $\vec{c}_j^\top \hat{\vec{z}}_3$ scores) on FGSM outputs? How does your answer change with epsilon?

A: The dual outputs show that the robustness of the classifier is very bad. With a small perturbation, the confidence difference would be negative with a large magnitude. Compared with the FGSM outputs, the classifier is much more sensitive to perturbations. The larger epsilon is, the more negative(negative number with larger maginitude) the values are.

Q: Suppose you have a deep neural classifier that you want to defend against adversarial attacks. That is, you want to detect and discard any input images that were possibly adversarially perturbed. How might you do this with the robustness certificate you have implemented?

A: Using c_j generated by all the classes that are different from the label to compute $d^*(\vec{x}, \vec{c_j})$. If the value is positive for all the classes, then there's no perturbation within the magnitude of ϵ that can fool the classifier.

Optional: Robust training

The following function should implement the robust loss from section A of the PDF. This loss is an upper bound on the worst-case loss within an ϵ ball of the original training input. Thus, training using this new loss should result in a classifier that is more robust than one trained on the original cross-entropy loss.

There are no mandatory questions here, but feel free to experiment with this robust training, and compare the performance here (measured by the dual objective certificate, as well as original/FGSM accuracy) to that of the original. You can also try training a model using the original loss first, then fine-tuning with the robust loss.

In [0]:
```
def robust_loss(x, label, net, eps, criterion):
    '''
    Given a batch of input images X, its corresponding lables LABEL,
    the classifier NET, epsilon value EPS, and original loss
    function CRITERION, returns the robust loss of NET w/r/t
    the original loss function, on the input image.
    '''
    l, u, S, S_min, S_plus = dual_bounds(x.to(device), net, eps)
    # We assume there are 10 classes.
    e_y = torch.zeros(10, 1)
    e_y[label] = 1
    c = e_y @ torch.ones(1, 10) - torch.eye(10)
    J = dual_forward(x, net, c, eps, l, u, S, S_min, S_plus).unsqueeze(0)
    return criterion(-J, label.unsqueeze(0))
```

In [0]:
```
def robust_train(net, criterion, trainloader, eps, lr=0.001):
    '''
    Trains the classifier NET using the robust version
    of the original loss function CRITERION with paramater EPS,
    using training data from TRAINLOADER and with learning rate LR.

    Note that we half the learning rate each epoch.
    '''
    optimizer = optim.Adam(net.parameters(), lr=lr)

    for epoch in range(3):
        for i, data in enumerate(trainloader, 0):
            for param_group in optimizer.param_groups:
                param_group['lr'] = lr * 0.5 ** (epoch)

            inputs, labels = data
            optimizer.zero_grad()
            loss = 0
            for i in range(inputs.shape[0]):
                x = inputs[i].unsqueeze(0).to(device)
                label = labels[i].unsqueeze(0)
                loss += robust_loss(x, label, net, eps, criterion)
            loss.backward()
            optimizer.step()

            if i % 500 == 0:
                print('Epoch', epoch, 'Iter:', i, 'Loss', loss.item())
```

In [0]: