Project Report

Junliang (Dexter) Lu, 3034087879
Xiaoyi Zhu, 3034994119

1. Our solutions to the problems codes are below:

```
def FGSM(x, labels, net, eps):
    ...
    return torch.clamp(x + eps*np.sign(grads), -1, 1)

def dual_bounds(x, net, eps):
    ...
    u = W[0]@x + b[0] + (eps*torch.norm(W[0], 1, 1)).reshape((b[0].shape[0], 1))
    l = W[0]@x + b[0] - (eps*torch.norm(W[0], 1, 1)).reshape((b[0].shape[0], 1))
    S = {j for j in range(n) if l[j]<=0 and u[j]>=0} #
    S_plus = {j for j in range(n) if l[j]>=0 and u[j]>=0} #
    S_min = {j for j in range(n) if l[j]<=0 and u[j]<=0} #
    ...

def dual_forward(x, net, c, eps, l, u, S, S_min, S_plus):
    ...
    v3 = -c
    v2_hat = W[1].T @ v3
    v2 = D @ v2_hat
    v1_hat = W[0].T @ v2
    last_term = 0
    for j in S:
        last_term += l[j] * max(0, v2[j])
    result = -v1_hat.T@x - eps*torch.norm(v1_hat,1) - (v2.T@b[0] + v3.T@b[1]) + last
    return result
```

2. Discuss the two primal and dual problem on fooling the classifer.

We are using the two methods FGSM and dual network separately to find x_prime. FGSM needs bigger eps in order to find a x_prime to fool the classifier, compared to the dual network, dual network needs lesser increase on Epsilon to find a x_prime that can maximize the classifier's losses. In other words, dual network works better on destroying the robustness of the classifier without much change on the input image. Finally, we summarize by getting a conclusion that if we want to make our classifer robust, we need to replace or remove any input images that have negative dual_forward output given a specific Epsilon.

3. What we learn from this project and the post-project reflection.

By completing this project, we first have a preliminary understanding of the neural network of machine learning, understand how the various layers of the neural network coordinate and

cooperate to perform classification prediction, and also understand the limitations of the neural network and how to optimize and improve it.

Secondly, we use this model to examine the theory we have learned, specifically convert the primal problem into a convex and solvable problem by modifying the nonlinear and non-convex RELU, and then simplifying the solution of the primal problem by solving the dual network.

Finally, we can check the results of our optimization and analyze the robustness of classifier by comparing different Epsilon parameters.

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 [94]:
```
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
import copy
```

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

In [95]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cpu
```

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 [96]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, tra

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 [97]:
```
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 [98]:
```
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 [99]:
```
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 [100]:
```
net = Net()
net.to(device)

criterion = nn.CrossEntropyLoss()

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

```
Epoch 0 Iter: 0 Loss 2.2041940689086914
Epoch 0 Iter: 500 Loss 0.08916271477937698
Epoch 0 Iter: 1000 Loss 0.027027402073144913
Epoch 0 Iter: 1500 Loss 0.15971067547798157
Epoch 0 Iter: 2000 Loss 0.19439055025577545
Epoch 0 Iter: 2500 Loss 0.0031414416152983904
Epoch 0 Iter: 3000 Loss 0.01799686998128891
Epoch 0 Iter: 3500 Loss 0.013792837038636208
Epoch 0 Iter: 4000 Loss 0.5312321186065674
Epoch 0 Iter: 4500 Loss 0.8572578430175781
Epoch 0 Iter: 5000 Loss 0.07473927736282349
Epoch 0 Iter: 5500 Loss 0.14251531660556793
Epoch 0 Iter: 6000 Loss 0.054336145520210266
Epoch 0 Iter: 6500 Loss 0.25754550099372864
Epoch 0 Iter: 7000 Loss 0.01948939450085163
Epoch 0 Iter: 7500 Loss 0.15049906075000763
Epoch 0 Iter: 8000 Loss 1.0912201404571533
Epoch 0 Iter: 8500 Loss 0.011114726774394512
Epoch 0 Iter: 9000 Loss 0.5738998055458069
Epoch 0 Iter: 9500 Loss 0.18750426173210144
Epoch 0 Iter: 10000 Loss 0.0404948927462101
Epoch 0 Iter: 10500 Loss 0.003113528247922659
Epoch 0 Iter: 11000 Loss 0.042347367852926254
Epoch 0 Iter: 11500 Loss 0.698900043964386
Epoch 0 Iter: 12000 Loss 0.043122708797454834
Epoch 0 Iter: 12500 Loss 0.6916748881340027
Epoch 0 Iter: 13000 Loss 0.03003678098320961
Epoch 0 Iter: 13500 Loss 0.0706872045993805
Epoch 0 Iter: 14000 Loss 0.001115485210902989
Epoch 0 Iter: 14500 Loss 0.010740955360233784
Epoch 1 Iter: 0 Loss 0.0010148435831069946
Epoch 1 Iter: 500 Loss 0.0012061131419613957
Epoch 1 Iter: 1000 Loss 0.07154256850481033
Epoch 1 Iter: 1500 Loss 0.0010887323878705502
Epoch 1 Iter: 2000 Loss 0.04060843586921692
Epoch 1 Iter: 2500 Loss 0.006883807014673948
Epoch 1 Iter: 3000 Loss 0.002232938539236784
Epoch 1 Iter: 3500 Loss 0.0023308969102799892
Epoch 1 Iter: 4000 Loss 0.000814545142930001
Epoch 1 Iter: 4500 Loss 0.002287180395796895
Epoch 1 Iter: 5000 Loss 0.030390823259949684
Epoch 1 Iter: 5500 Loss 0.11858262121677399
Epoch 1 Iter: 6000 Loss 0.09670558571815491
Epoch 1 Iter: 6500 Loss 0.0051245116628706455
Epoch 1 Iter: 7000 Loss 0.004933973308652639
Epoch 1 Iter: 7500 Loss 0.000630564580205828
Epoch 1 Iter: 8000 Loss 0.8106819987297058
Epoch 1 Iter: 8500 Loss 9.166501695290208e-05
Epoch 1 Iter: 9000 Loss 0.009554932825267315
Epoch 1 Iter: 9500 Loss 0.006679096724838018
Epoch 1 Iter: 10000 Loss 0.04626922681927681
```

```
Epoch 1 Iter: 10500 Loss 0.04573384299874306
Epoch 1 Iter: 11000 Loss 0.30732280015945435
Epoch 1 Iter: 11500 Loss 0.061380550265312195
Epoch 1 Iter: 12000 Loss 0.4142983555793762
Epoch 1 Iter: 12500 Loss 0.06236504018306732
Epoch 1 Iter: 13000 Loss 0.010364633053541183
Epoch 1 Iter: 13500 Loss 0.030467109754681587
Epoch 1 Iter: 14000 Loss 0.006845845375210047
Epoch 1 Iter: 14500 Loss 0.33165329694747925
Epoch 2 Iter: 0 Loss 0.0005439358064904809
Epoch 2 Iter: 500 Loss 0.007708691991865635
Epoch 2 Iter: 1000 Loss 0.02435087412595749
Epoch 2 Iter: 1500 Loss 0.027454400435090065
Epoch 2 Iter: 2000 Loss 0.14177247881889343
Epoch 2 Iter: 2500 Loss 0.012358015403151512
Epoch 2 Iter: 3000 Loss 0.07224833965301514
Epoch 2 Iter: 3500 Loss 0.05075689032673836
Epoch 2 Iter: 4000 Loss 0.003478949423879385
Epoch 2 Iter: 4500 Loss 0.0031556389294564724
Epoch 2 Iter: 5000 Loss 0.19360190629959106
Epoch 2 Iter: 5500 Loss 0.06727398186922073
Epoch 2 Iter: 6000 Loss 0.006381921470165253
Epoch 2 Iter: 6500 Loss 0.00038437877083197236
Epoch 2 Iter: 7000 Loss 0.0023238486610352993
Epoch 2 Iter: 7500 Loss 0.0001228542096214369
Epoch 2 Iter: 8000 Loss 0.0022666931618005037
Epoch 2 Iter: 8500 Loss 0.31095021963119507
Epoch 2 Iter: 9000 Loss 0.208926260471344
Epoch 2 Iter: 9500 Loss 0.0009874847019091249
Epoch 2 Iter: 10000 Loss 0.0020193662494421005
Epoch 2 Iter: 10500 Loss 0.00432209949940443
Epoch 2 Iter: 11000 Loss 0.012996421195566654
Epoch 2 Iter: 11500 Loss 0.005554873496294022
Epoch 2 Iter: 12000 Loss 0.006324805784970522
Epoch 2 Iter: 12500 Loss 0.01972530409693718
Epoch 2 Iter: 13000 Loss 0.0019252300262451172
Epoch 2 Iter: 13500 Loss 2.3215263354359195e-05
Epoch 2 Iter: 14000 Loss 0.007914301939308643
Epoch 2 Iter: 14500 Loss 0.019509119912981987
```

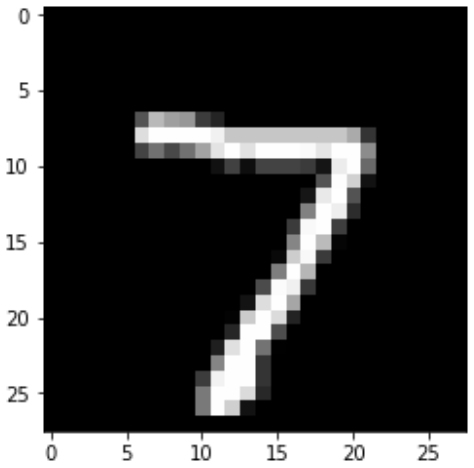
Let's load a sample image from the test dataset, and see what the classifier makes of it. Make sure to visualize the image using `imshow(x[0,0])` . Also, note that the line `test_iter.next()` pulls a new input image from the test set each time you run it; try running the next code block a few times to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

In [101]:
```
test_iter = iter(testloader)
```

In [102]:
```
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([[-12.7679,  -6.5708,  -1.2490,  -2.0271, -21.9014,  -6.
4082, -29.7661,
           8.3182,  -9.8423,  -8.8766]])
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 [103]:
```
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 [104]:
```
print('Classifier accuracy on original test dataset:', accuracy(net, testloader))
```

```
Classifier accuracy on original test dataset: 0.9704
```

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.)

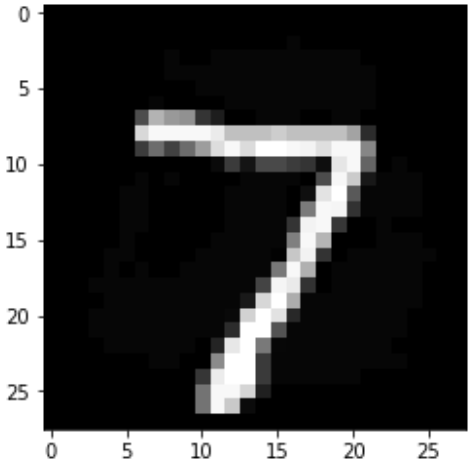
In [105]:
```
def FGSM(x, labels, net, eps):
    '''
    Given an input image X and its corresponding labels
    LABELS, as well as a classifier NET, returns X
    perturbed by EPS using the fast gradient sign method.
    '''
    net.zero_grad()    # Zero out any gradients from before
    x.requires_grad=True    # 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
    return torch.clamp(x + eps*np.sign(grads), -1, 1)     # TODO: Your code here!
```

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 [193]:
```
eps = 0.05 # TODO: Try eps = 0.05, 0.1, 0.2, 0.3, 0.4
```

In [194]:
```
# We are using the same sample input x as before.
x.requires_grad = True
x_prime = FGSM(x, labels, net, eps)
imshow(x_prime[0,0].cpu())
out = net(x_prime)

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




```
Classifier output: tensor([[-11.2999,  -5.1188,  -0.5740,  -1.5143, -20.6092,  -5.
8353, -26.2738,
           6.9757,  -8.2360,  -8.8247]])
Classifier prediction: 7
```

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 [195]:
```
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 18.275604248046875
1 12.094446182250977
2 7.549692153930664
3 8.489974021911621
4 27.58486557006836
5 12.81100845336914
6 33.24948501586914
8 15.211647987365723
9 15.800315856933594
```

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: Your answer here: higher epsilon will make the robustness of the classifier less robust. Digit 1,2,3,8 have negative scores because they look like 7 so much, and Digits like 4 doesn't look like 7 so it has higher scores.

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 [109]:
```
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 [110]:
```
print('Classifier accuracy on test dataset perturbed with FGSM:', accuracy_on_FGSM(
```

```
Classifier accuracy on test dataset perturbed with FGSM: 0.0714
```

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: Your answer here: It gets lower as eps get higher.

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

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

See Section 6 of the PDF for more details.

In [182]:
```
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 = x[0].reshape(-1, 1)    # Reshape input to more convenient dimensions
    W = [layer.weight for layer in net.layers]    # Array of network weights (W mat
    b = [layer.bias.reshape(-1, 1) for layer in net.layers]    # Array of network b
    n = W[1].shape[1]    # Dimensionality of hidden layer
    u = W[0]@x + b[0] + (eps*torch.norm(W[0], 1, 1)).reshape((b[0].shape[0], 1))
    l = W[0]@x + b[0] - (eps*torch.norm(W[0], 1, 1)).reshape((b[0].shape[0], 1))
    S = {j for j in range(n) if l[j]<=0 and u[j]>=0}    # TODO
    S_plus = {j for j in range(n) if l[j]>=0 and u[j]>=0}    # TODO
    S_min = {j for j in range(n) if l[j]<=0 and u[j]<=0}    # TODO
    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

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

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

$$
\begin{aligned}
\vec{\nu}_3 &= -\vec{c} \\
\vec{\hat{\nu}}_2 &= W_2^\top \vec{\nu}_3 \\
\nu_{2j} &= 0 && \forall j \in S^- \\
\nu_{2j} &= \hat{\nu}_{2j} && \forall j \in S^+ \\
\nu_{2j} &= \frac{u_j}{u_j - l_j} \hat{\nu}_{2j} && \forall j \in S \\
\vec{\hat{\nu}}_1 &= W_1^\top \vec{\nu_2}
\end{aligned}
.
$$

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

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

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

where D is a diagonal matrix defined by

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

In [186]:
```
# 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 j in S:
            d.append((u[j] / (u[j] - l[j])).item())
        elif j in S_plus:
            d.append(1)
        elif j in S_min:
            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 dimensions
    W = [layer.weight for layer in net.layers]    # Array of network weights (W mat
    b = [layer.bias.reshape(-1, 1) for layer in net.layers]    # Array of network b
    n = W[1].shape[1]    # Dimensionality of hidden layer
    D = StoD(S_min, S_plus, S, n, u, l)
    # TODO: Your code here!
    v3 = -c
    v2_hat = W[1].T @ v3
    v2 = D @ v2_hat
    v1_hat = W[0].T @ v2

    last_term = 0
    for j in S:
        last_term += l[j] * max(0, v2[j])
    result = -v1_hat.T@x - eps*torch.norm(v1_hat,1) - (v2.T@b[0] + v3.T@b[1]) + las
    return result     # TODO
```

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 $\epsilon \in \{0.05, 0.1, 0.2, 0.3, 0.4\}$, and compare the robustness guarantees here with the classifier's performance on the FGSM data from before.

In [206]:
```
eps = 0.1 # TODO: Try eps = 0.05, 0.1, 0.2, 0.3, 0.4
```

In [207]:
```
# 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).item())
```

```
0 3.4582228660583496
1 1.483102560043335
2 -2.2668371200561523
3 -1.5152201652526855
4 11.202545166015625
5 -1.0364689826965332
6 14.044443130493164
8 1.5146164894104004
9 3.2478199005126953
```

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: Your answer here: the lesser the score of an given digit gets, the less robust of the classifier on that digit. This scores here are easier affects by eps, and it takes lesser eps to achieve more loss. The scores get more negative when eps increases.

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: Your answer here: Given an specific eps, check the scores of the dual network outputs for each digit. All digits with negative scores should be handled; all digits with positive scores closed to 0 shoule also be handled.

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 [208]:
```
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, 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 [209]:
```
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)
                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())
```