HW2 - Q3: Evaluating Robustness of Neural Networks (35 points)

Keywords: Adversarial Robustness, FGSM/PGD Attack, Certification

About the dataset: \ The [MNIST](#) database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.\ The MNIST database contains 70,000 labeled images. Each datapoint is a 28×28 pixels grayscale image.\ Here we will be starting off with a pre-trained 2-hidden-layer model on the full MNIST dataset.

Agenda:

- **In this programming challenge, you will implement adversarial attack on an MNIST neural network model as well as visualize those attacks.**
- **You will do this by solving the inner maximization problem using FGSM (Fast Gradient Sign Method) and PGD (Projected Gradient Descent).**
- **You will then perform verification of the model using Interval-Bound -Propagation (IBP).**

Note:

- **It is important that you use GPU accelaration for this Question.**
- **A note on working with GPU:**
  - **Take care that whenever declaring new tensors, set `device=device` in parameters.**
  - **You can also move a declared torch tensor/model to device using `.to(device)`.**
  - **To move a torch model/tensor to cpu, use `.to('cpu')`**
  - **Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU).**
- **Run all the cells in order.**
- **Do not edit the cells marked with !!DO NOT EDIT!!**
- **Only add your code to cells marked with !!!! YOUR CODE HERE !!!!**
- **Do not change variable names, and use the names which are suggested.**

Preprocessing

In [1]:

```
# install this library
!pip install gdown
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: gdown in /usr/local/lib/python3.7/dist-packages (4.4.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages (from g
down) (3.7.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from gdown
) (4.64.0)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.7/dist-packages (
from gdown) (4.6.3)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from gdown)
(1.15.0)
Requirement already satisfied: requests[socks] in /usr/local/lib/python3.7/dist-packages
(from gdown) (2.23.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/
python3.7/dist-packages (from requests[socks]->gdown) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-package
s (from requests[socks]->gdown) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (fr
om requests[socks]->gdown) (2.10)
```

```
om requests[socks]->gdown) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packag
es (from requests[socks]->gdown) (2022.5.18.1)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in /usr/local/lib/python3.7/dist-pa
ckages (from requests[socks]->gdown) (1.7.1)
```

- **We will be using a pre-trained 2-hidden layer neural network model (`nn_model`) that takes as input features vectors of size 784, and ouputs logits vector of size 10. Each of the two hidden layers are of size 1024.**
- **This is a highly accurate model with train accuracy of approx 99.88% and test accuracy of approx 98.14%.**
- **We will also be loading and initializing a dummy model (`test_model`) for unit testing code implementation.**

In [2]:

```
# !!DO NOT EDIT!!
# imports
import os.path

import torch
import torch.nn as nn
import numpy as np
import requests
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
import gdown
from zipfile import ZipFile

# set hardware device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# loading the dataset full MNIST dataset
mnist_train = datasets.MNIST("./data", train=True, download=True, transform=transforms.To
Tensor())
mnist_test = datasets.MNIST("./data", train=False, download=True, transform=transforms.To
Tensor())

mnist_train.data = mnist_train.data.to(device)
mnist_test.data = mnist_test.data.to(device)

mnist_train.targets = mnist_train.targets.to(device)
mnist_test.targets = mnist_test.targets.to(device)

# number of target classes
num_classes = 10
num_classes_test = 2

# reshape and min-max scale
X_train =  (mnist_train.data.reshape((mnist_train.data.shape[0], -1))/255).to(device)
y_train = mnist_train.targets
X_test = (mnist_test.data.reshape((mnist_test.data.shape[0], -1))/255).to(device)
y_test = mnist_test.targets



if not os.path.exists("nn_model.pt"):
  # load pretrained and dummy model
  print("Downloading pretrained model")
  url_nn_model = 'https://bit.ly/3sKvyOs'
  url_models   = 'https://bit.ly/3lsVcDn'
  gdown.download(url_nn_model, 'nn_model.pt')
  gdown.download(url_models, 'models.zip')
  ZipFile("models.zip").extractall("./")


from model import NN_Model
from test_model import Test_Model

nn_model = torch.load("./nn_model.pt").to(device)
print('Pretrained model (nn_model):', nn_model)
```

```
test_model = Test_Model()
print('Dummy model (test_model):', test_model)

print("Done")
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/r
aw/train-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/r
aw/train-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/ra
w/t10k-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/ra
w/t10k-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading pretrained model
```

```
Downloading...
From: https://bit.ly/3sKvyOs
To: /content/nn_model.pt
100%|██████████| 7.46M/7.46M [00:00<00:00, 83.2MB/s]
Downloading...
From: https://bit.ly/3lsVcDn
To: /content/models.zip
100%|██████████| 1.55k/1.55k [00:00<00:00, 3.21MB/s]
```

```
Pretrained model (nn_model): NN_Model(
  (l1): Linear(in_features=784, out_features=1024, bias=True)
  (l2): Linear(in_features=1024, out_features=1024, bias=True)
  (l3): Linear(in_features=1024, out_features=10, bias=True)
)
Dummy model (test_model): Test_Model(
  (l1): Linear(in_features=2, out_features=3, bias=True)
  (l2): Linear(in_features=3, out_features=3, bias=True)
  (l3): Linear(in_features=3, out_features=2, bias=True)
)
Done
```

In this problem set you need to access the individual layers of the neural network. The below piece of code creates a list of ordered layers for each of the neural network models for easy access.

In [3]:

```
# This will save the linear layers of the neural network model in a ordered list
# Eg:
# to access weight of first layer: model_layers[0].weight
# to access bias of first layer: model_layers[0].bias
model_layers = [layer for layer in nn_model.children()] # for nn_model
test_model_layers = [layer for layer in test_model.children()] # for dummy model
```

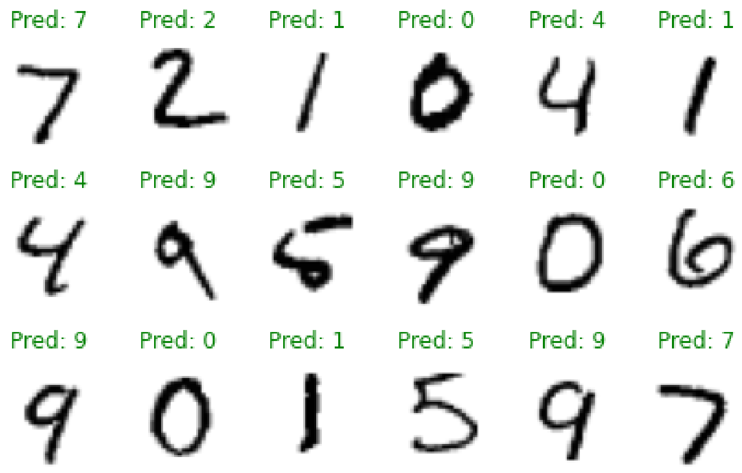
In [4]:

```
# !!DO NOT EDIT!!
```

```python
# utility function to plot the images
def plot_images(X,y,yp,M,N):
  f,ax = plt.subplots(M,N, sharex=True, sharey=True, figsize=(N,M*1.3))
  for i in range(M):
    for j in range(N):
      ax[i][j].imshow(1-X[i*N+j].cpu().detach().numpy(), cmap="gray")
      title = ax[i][j].set_title("Pred: {}".format(yp[i*N+j].max(dim=0)[1]))
      plt.setp(title, color=('g' if yp[i*N+j].max(dim=0)[1] == y[i*N+j] else 'r'))
      ax[i][j].set_axis_off()
    plt.tight_layout()
```

In [5]:

```python
# !!DO NOT EDIT!!
# let us visualize a few test examples
example_data = mnist_test.data[:18]/255
example_data_flattened  = example_data.view((example_data.shape[0], -1)).to(device) # ne
eded for training
example_labels = mnist_test.targets[:18].to(device)
plot_images(example_data, example_labels, nn_model(example_data_flattened), 3, 6)
```



In [6]:

```python
# device

# x = torch.tensor([1.0,-1.0, 4.0])
# x.sign()

# x.shape
#
# torch.zeros(x.shape)

# X_train.shape
# example_data_flattened.shape
#
# y_train.shape, example_labels.shape
```

In [7]:

```python
#######
# !!! YOUR CODE HERE !!!

loss = nn.CrossEntropyLoss()

def fgsm(model, x, y, epsilon = 0.05):

  delta = torch.zeros(x.shape, requires_grad=True, device=device)
  cost = loss(model(x+ delta), y)
  cost.backward()
  return epsilon * delta.grad.detach().sign()
```

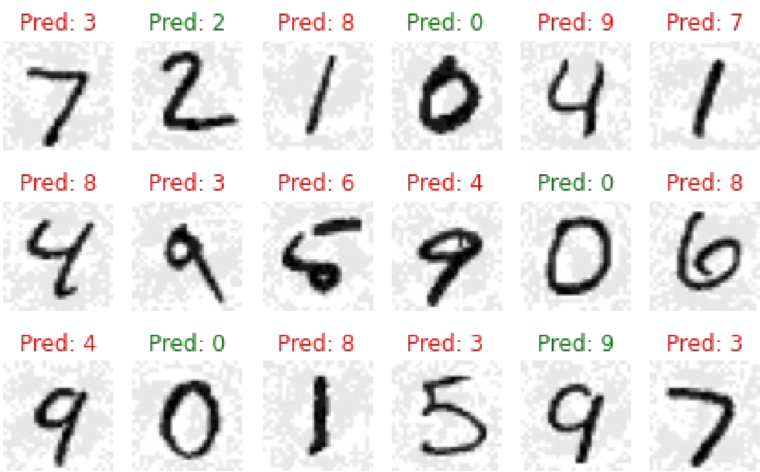
#2. Now, consider the first few examples from the training dataset which are already defined above as `example_data_flattened` and `example_labels`. Using the function `fgsm`, get the value of `delta` for these examples. Perform prediction on the modified

fgsm, get the value of `delta` for these examples. Perform prediction on the modified dataset (`example_data_flattened + delta`), and construct a similar plot of images as above. You may reuse the `plot_images` function. Is the attack successful?

In [8]:

```
#######
# !!! YOUR CODE HERE !!!

# example_data_flattened.shape, example_labels

delta = fgsm(nn_model, example_data_flattened, example_labels, 0.05)
perturbed_data_flat = example_data_flattened + delta
perturbed_images = (example_data_flattened + delta).reshape(example_data.shape)

fsgm_y_pred = nn_model(perturbed_data_flat)

fsgm_y_pred_labels= torch.argmax(fsgm_y_pred, dim=1)

print("------------------Perturbed Images | fsgm_pred---------------------")
plot_images(perturbed_images, example_labels, fsgm_y_pred, 3, 6)

attack_rate = torch.ne(example_labels, fsgm_y_pred_labels).sum() / len(example_labels)
torch.ne(example_labels, fsgm_y_pred_labels)
if attack_rate != 0.0:
  print("Attack successful!")
  print(f"Attack rate - {float(attack_rate)*100} %", )
else:
  print("Attack not successful")

#######
```

```
------------------Perturbed Images | fsgm_pred---------------------
Attack successful!
Attack rate - 72.22222089767456 %
```



(b) PGD attack: In this part you will create a few adversarial examples using PGD attack. Use an attack budget $\epsilon = 0.05$. (10 points)

Note: For the Projected Gradient Descent (PGD) attack, you create an adversarial example by iteratively performing gradient descent with a fixed step size α. The update rule is: $\delta := P(\delta + \alpha \nabla_\delta \ell(h_\theta(x+\delta), y))$, where δ is the perturbation, θ are the frozen DNN parameters, x and y is the training example and its ground truth label respectively, h_θ is the hypothesis function, ℓ denotes the loss function, and P denotes the projection onto a norm ball (l_∞, l_1, l_2, etc.) of interest. For l_∞ ball, this just means clamping the value of δ between $-\epsilon$ and ϵ.

#1. Instead of using FGSM, now use Projected Gradient Descent (PGD) with projection on l_∞ ball for the attack. Define a function `pgd` that takes as input the neural network model (`model`), training examples (`X`), target labels (`y`), step size (`alpha`), attack budget (`epsilon`), and number of iterations (`num_iter`). Return the perturbation (δ) after `num_iter` gradient descent steps.

In [9]:

```
#######
# !!! YOUR CODE HERE !!!

def pgd(model, x, y, alpha, epsilon, num_iter):
  delta = torch.zeros(x.shape, requires_grad=True, device=device)

  for i in range(num_iter):
    cost = loss(model(x+ delta), y)
    cost.backward()

    d = delta + alpha * delta.grad.detach().sign()
    d = d.clamp(-epsilon, epsilon)     # as we are using l infity norm hence [-eps,+eps]

    delta.data = d
    delta.grad.zero_()

  return delta.detach()

#######
```

#2. Now use the PGD attack for the examples from `example_data_flattened`. Use `alpha=1000`, `num_iter=1000`, and create a similar plot as before. Is the attack successful?

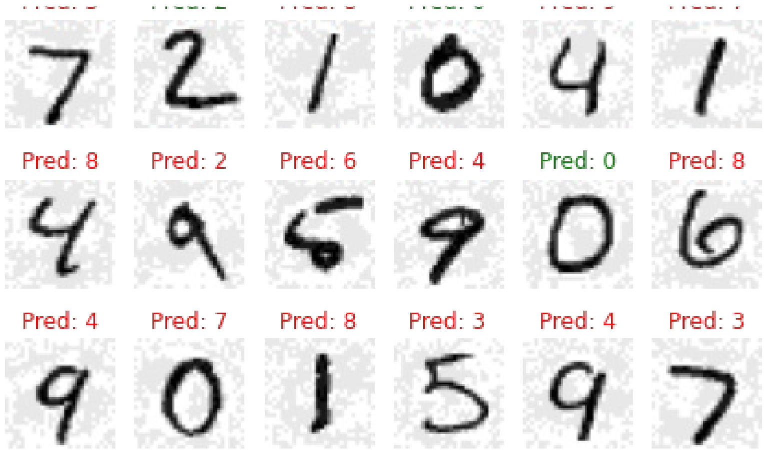
The value of `alpha` is large because the neural network model is pretrained and is therefore at the local minima. The value of gradients here is extremely small, and we therefore need a huge value of step size to have any hope of moving out of the local minima.

In [10]:

```
#######
# !!! YOUR CODE HERE !!!

delta = pgd(nn_model, example_data_flattened, example_labels, alpha = 1000, epsilon = 0.
05, num_iter=1000)
perturbed_data_flat = example_data_flattened + delta
perturbed_images = (example_data_flattened + delta).reshape(example_data.shape)

pgd_y_pred = nn_model(perturbed_data_flat)

pgd_y_pred_labels= torch.argmax(pgd_y_pred, dim=1)

print("------------------Perturbed Images | pgd ---------------------")
plot_images(perturbed_images, example_labels, pgd_y_pred, 3, 6)

attack_rate = torch.ne(example_labels, pgd_y_pred_labels).sum() / len(example_labels)
torch.ne(example_labels, pgd_y_pred_labels)
if attack_rate != 0.0:
  print("Attack successful!")
  print(f"Attack rate - {float(attack_rate)*100} %", )
else:
  print("Attack not successful")
######
```

```
------------------Perturbed Images | pgd ---------------------
Attack successful!
Attack rate - 83.33333134651184 %
```

Pred: 3 Pred: 2 Pred: 8 Pred: 0 Pred: 9 Pred: 7



(c) Use FGSM and PGD to create adversarial examples using the complete test dataset. Create the datasets with different values of `epsilon: [0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.12, 0.14, 0.16, 0.18, 0.2]`. For each of the dataset created with different `epsilon` values and attack type, get the model accuracies. Plot a (single) graph of accuracy vs. epsilon for both attack types. Note that `epsilon=0` means no attack, so you can just get accuracy on the original dataset. (10 points)

It is important that you use **GPU accelaration for this part.**

In [11]:

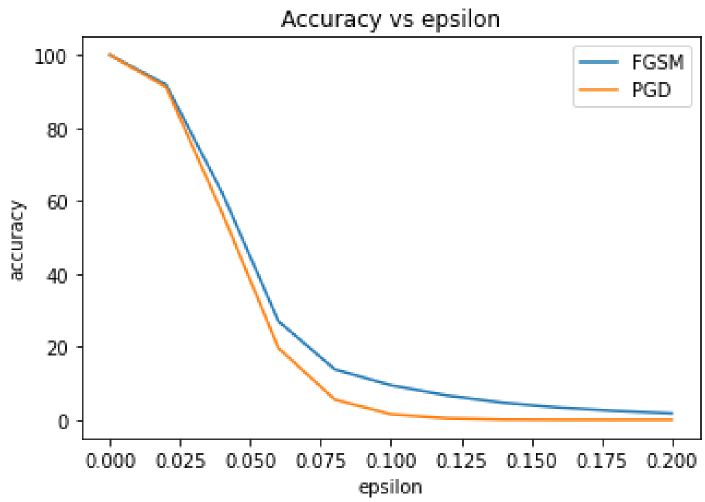
```
#######
# !!! YOUR CODE HERE !!!

epsilon = [0.02*i for i in range(11)]
accuracy_list_fgsm = []
accuracy_list_pgd = []


def get_accuracy(attack_method, X_train, y_train, epsilon):

    if attack_method == "fgsm":
        delta = fgsm(nn_model, X_train, y_train, epsilon=epsilon)
    else:
        delta = pgd(nn_model, X_train, y_train, alpha = 1000, epsilon = epsilon, num_ite
r=1000)

    perturbed_data_flat = X_train + delta
    fgsm_y_pred = nn_model(perturbed_data_flat)
    fgsm_y_pred_labels= torch.argmax(fgsm_y_pred, dim=1)

    accuracy = float(torch.eq(y_train, fgsm_y_pred_labels).sum() / len(y_train) * 100.0)
    return accuracy

# n = 1000
n = len(X_train)

for eps in epsilon:
    fgsm_acc = get_accuracy("fgsm", X_train[:n], y_train[:n], epsilon=eps)
    pgd_acc = get_accuracy("pgd", X_train[:n], y_train[:n], epsilon=eps)
    accuracy_list_fgsm.append(fgsm_acc)
    accuracy_list_pgd.append(pgd_acc)
    print(f"{eps} -> {fgsm_acc} - {pgd_acc}")
```

```
0.0 -> 99.97666931152344 - 99.97666931152344
0.02 -> 91.83999633789062 - 91.17500305175781
0.04 -> 62.13333511352539 - 56.61000061035156
0.06 -> 27.0049991607666 - 19.648332595825195
0.08 -> 13.844999313354492 - 5.563333034515381
0.1 -> 9.458333015441895 - 1.4900000095367432
0.12 -> 6.644999980926514 - 0.3916666805744171
0.14 -> 4.681666374206543 - 0.0833333358168602
0.16 -> 3.323333263397217 - 0.011666666716337204
0.18 -> 2.4183332920074463 - 0.0
```

```
0.2 -> 1.746666669845581 - 0.0
```

In [12]:

```
plt.title("Accuracy vs epsilon")
plt.ylabel("accuracy")
plt.xlabel("epsilon")

plt.plot(epsilon,accuracy_list_fgsm, label = "FGSM")
plt.plot(epsilon,accuracy_list_pgd, label = "PGD")
plt.legend()
plt.show()

# accuracy_list_fgsm
# accuracy_list_pgd
```



(d) Use the Interval-Bound-Propagation (IBP) technique to certify robustness of the model through lower bound with a given value of epsilon. (10 points)

- **In this section, you will find the lower and upper bounds for each neuron of each of the linear layers of the neural network model.**
- **Note that the initial bound is the bound of the first layer, which is the input example. For the l_∞ perturbation, the initial lower bound is simply $max(0, x-\epsilon)$, and the initial upper bound is $min(1, x+\epsilon)$, for an input example x (Note that each value of x must lie in between 0 and 1, thats why the min and max).**
- **In the function, propagate the initial bound across all layers of the neural network and return a list of tuples of *pre-activation* lower and upper bound for each layer. The *pre-activation* bounds are the bound before applying ReLU activation.**
- **Let's review a bit of the IBP bounds: let $z = Wx + b$ denote an intermediate linear layer of the model, and suppose $\hat{l} \leq x \leq \hat{u}, l \leq z \leq u$, we have:\ $l = W_+\hat{l} + W_-\hat{u} + b$\ $u = W_+\hat{u} + W_-\hat{l} + b$ \ Note l, u here are the *pre-activation* bounds**
- **If a non-linear ReLU activation function $\sigma(\cdot)$ is applied to the layer $z = Wx + b$, then the bounds of $\sigma(z)$ will be: $l = \sigma(\hat{l}), u = \sigma(\hat{u})$ as σ is a monotonically non-decreasing function. I.e. $l \leq \sigma(z) \leq u$. The l, u here are the *post-activation* bounds. Note, here we use $\hat{l}$ and $\hat{u}$ to denote the bounds of the previous layer: $\hat{l} \leq z \leq \hat{u}$.**

#1. Define a function `bound_propagation` which takes as input an ordered list of layers of the model (`model_layers`), a feature vector (`x`), and attack budget (`epsilon`). Return a list of tuples of `pre-activation` lower and upper bound tensors for each layer. Verify that your implementation is correct by verifying the results of your function on the unit tests given below.

In [13]:

```python
#######
# !!! YOUR CODE HERE !!!

import torch.nn.functional as fn

def bound_propagation(model_layers, x, epsilon):
    initial_bound = ((x - epsilon).clamp(min=0), (x + epsilon).clamp(max=1))
    l, u = initial_bound
    bounds = []
    bounds.append((l, u))
    for i,layer in enumerate(model_layers):
        layer = layer.to(device)
        if isinstance(layer, nn.Linear):
            if i < len(model_layers):
                l = torch.nn.functional.relu(l)
                u = torch.nn.functional.relu(u)

            l_ = (layer.weight.clamp(min=0) @ l.t() + layer.weight.clamp(max=0) @ u.t()
                  + layer.bias[:,None]).t()
            u_ = (layer.weight.clamp(min=0) @ u.t() + layer.weight.clamp(max=0) @ l.t()
                  + layer.bias[:,None]).t()
        elif isinstance(layer, fn.relu):
        # else:
            print("Clamping done")
            l_ = l.clamp(min=0)
            u_ = u.clamp(min=0)

        bounds.append((l_, u_))
        l,u = l_, u_
    return bounds


      #######
```

In [15]:

```python
# !!DO NOT EDIT!!
sample_epsilon = 0.2
# unit test - 1
x_1 = torch.tensor([[0.1, 0.9]], device=device)
test_bounds_1 = bound_propagation(test_model_layers, x_1, sample_epsilon)
assert torch.all(torch.eq(torch.round(test_bounds_1[0][0], decimals=2), torch.tensor([[0
.0000, 0.7000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_1[0][1], decimals=2), torch.tensor([[0
.3000, 1.0000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_1[1][0], decimals=2), torch.tensor([[0
.0000, 1.4000, 1.2000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_1[1][1], decimals=2), torch.tensor([[0
.4500, 2.6000, 1.5000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_1[2][0], decimals=2), torch.tensor([[2
.6500, -0.8000, 2.1000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_1[2][1], decimals=2), torch.tensor([[6
.7000, 0.1000, 4.3500]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_1[3][0], decimals=2), torch.tensor([[4
.2000, 1.4500]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_1[3][1], decimals=2), torch.tensor([[9
.4700, 11.900]], device=device)))

# unit test - 2
x_2 = torch.tensor([[0.4, 0.5]], device=device)
test_bounds_2 = bound_propagation(test_model_layers, x_2, sample_epsilon)
assert torch.all(torch.eq(torch.round(test_bounds_2[0][0], decimals=2), torch.tensor([[0
.2000, 0.3000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_2[0][1], decimals=2), torch.tensor([[0
.6000, 0.7000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_2[1][0], decimals=2), torch.tensor([[0
.4000, 1.0000, 0.8000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_2[1][1], decimals=2), torch.tensor([[1
.0000, 2.6000, 1.2000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_2[2][0], decimals=2), torch.tensor([[-
0.2000, -0.7000,  0.4000]], device=device))
```

```
assert torch.all(torch.eq(torch.round(test_bounds_2[2][1], decimals=2), torch.tensor([[5
.2000, 0.5000, 3.4000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_2[3][0], decimals=2), torch.tensor([[0
.7000, -2.9000]], device=device)))
assert torch.all(torch.eq(torch.round(test_bounds_2[3][1], decimals=2), torch.tensor([[7
.9000, 11.0000]], device=device)))

print("Unit tests successful")
```

```
Unit tests successful
```

#2. Let the lower and upper bounds of the final layer of the model be l^{final} and u^{final} respectively. Then we say that an input example x has a robustness certificate ϵ if the criteria: $l^{final}[c] - u^{final}[i]$, where c denotes the ground truth class of the input x.

$$> 0, \forall i \neq c$$

- **We need to determine the maximum value of epsilon for certified robustness against an adversarial attack for a given example. We can do the same using binary search over a few values of epsilon.**
- **Define a function `binary_search` that takes as input a sorted array of epsilon values (`epsilons`), an ordered list of neural network model layers (`model_layers`), examples (`X`), corresponding targets (`y`), the number of target classes (`num_classes`). It should return `certified_epsilons` which is a python list of the final values of epsilon certification for each example in input. You can use `None` when unable to find an epsilon value from epsilons.**
- **Verify that your implementation is correct by verifying the results of your function on the unit tests given below.**

In [16]:

```
#######
# !!! YOUR CODE HERE !!!
def _check(lf, uf, y):
    u_js = torch.cat((uf[0][:y], uf[0][y + 1:]))
    if torch.all(lf[0][y] > u_js):
        return True
    else:
        return False

def binary_search(epsilons, model_layers, X, y, num_classes):
    eps_f = []
    for i in range(len(y)):
        x = X[[i]]
        x = torch.reshape(x, (x.shape[0], x.shape[1]))
        j = y[i]
        if_found = False
        low = 0
        high = len(epsilons) - 1

        while(low <= high and not if_found):
            mid = (low + high) // 2
            bounds_mid = bound_propagation(model_layers, x, epsilon=epsilons[mid])
            lf_mid = bounds_mid[-1][0]
            uf_mid = bounds_mid[-1][1]
            bounds_hi = bound_propagation(model_layers, x, epsilon=epsilons[high])
            lf_hi = bounds_hi[-1][0]
            uf_hi = bounds_hi[-1][1]
            if _check(lf_hi, uf_hi, j):
                if_found = True
            else:
                if _check(lf_mid, uf_mid, j):
                    low = mid + 1
                else:
                    high = mid - 1
        if if_found==True:
            eps_f.append(epsilons[high])
        else:
```

```
            eps_f.append(epsilons[mid])

    return eps_f

#######
```

In [17]:

```
# !!DO NOT EDIT!!
epsilons = [x/10000 for x in range(1, 10000)]
# unit test - 1
sample_X = torch.tensor([[0.1, 0.9], [0.4, 0.5]], device=device)
sample_y = torch.tensor([0,0], device=device)
test_epsilons = binary_search(epsilons, test_model_layers, sample_X, sample_y, num_classe
s_test)
assert test_epsilons==[0.0028, 0.0067]
print("Unit tests sucscessful.")
```

```
Unit tests sucscessful.
```

#3. Report the certified values of epsilon on the first few examples (simply run the below cell).

In [18]:

```
# !!DO NOT EDIT!!
# finding epsilon for first few examples of MNIST dataset using IBP
epsilons = [x/10000 for x in range(1, 10000)]
X = example_data_flattened[0:2]
y = example_labels[0:2]
binary_search(epsilons, model_layers, X, y, num_classes)
```

Out[18]:

```
[0.0008, 0.0013]
```