Assignment #: 3

Venkata Sai Chelagamsetty

November 12th, 2022

1. (a) The security of any system is measured with respect to the adversarial goals and capabilities that it is designed to defend against–the system's threat model.Threat model is a model in which potential threats, such as structural vulnerabilities or the absence of appropriate safeguards, can be identified and enumerated, and countermeasures prioritized. Threat model answers questions like What kind of threats might a system face?, What kind of capabilities might we expect adversaries to have?,What are the limits on what the adversary might be able to do to us?.

   (b) An attack occurs when someone attempts to exploit a vulnerability. A compromise occurs when an attack is successful. In other words, when a compromise occurs, a vulnerability is exploited.

   (c) A security policy is a document that expresses clearly and concisely what the protection mechanisms are to achieve. Its a statement of the security we expect the system to enforce. A security model is a specification of a security policy:
   it describes the entities governed by the policy,
   it states the rules that constitute the policy.
   Essentially It ensures that the CIA-Triad is maintained. It answers questions like how data is accessed? What level of security is required? and what should happen when these requirements aren't met.

   (d) Confidentiality ensures that computer-related assets are accessed only by authorized parties
   Integrity requires that computer system assets and transmitted information be capable of modification only by authorized parties
   Availability: The degree to which data or systems are accessible and in functioning condition.

   (e) In ML systems, in terms of confidentiality, the attacks are classified as
   Model Extraction : An adversary aims to discover the structure or parameters of the model by observing its predictions.
   Whereas in terms of Integrity, the attacks are classified as
   Poisoning attacks : An adversary tries to manipulate the training dataset in order to control the prediction behavior of a trained model such that the model will label malicious examples into a desired classes e.g., labeling spam e-mails as safe).
   and , evasion attacks: The attacker manipulates input samples at test time to evade (cause a misclassification) a trained classifier at test time.

   (f) i. Accuracy = $100 * \frac{Attacksclassifiedasattacks+Benignclassifiedasbenign}{Totalno.oflogins}$
   $Accuracy = 100 * \frac{121806}{130200}$
   $Accuracy = 93$

   ii. $P(attack) = \frac{No.ofattacks}{Totallogins}$
   $P(Attack) = \frac{11250}{130200}$
   $P(attack) = 0.0864$

iii.

$$P(flag|attack) = \frac{Attackclassifiedasattack}{Totalactualattacks} \tag{1}$$
$$P(flag|attack) = \frac{11086}{11250} \tag{2}$$
$$P(flag|attack) = 0.9854 \tag{3}$$

iv.

$$P(flag) = \frac{trafficpredictedasattack}{Totallogins} \tag{4}$$
$$P(flag) = \frac{20036}{130200} \tag{5}$$
$$P(flag) = 0.1538 \tag{6}$$

v.

$$P(benign|flag) = \frac{P(flag|benign)P(benign)}{P(flag)} \tag{7}$$
$$P(benign) = 1 - P(attack) \tag{8}$$
$$P(benign) = 0.9136 \tag{9}$$
$$P(flag|benign) = \frac{Benignclassifiedasattack}{Totalactualbenign} \tag{10}$$
$$P(flag|benign) = \frac{8950}{110000 + 8950} \tag{11}$$
$$P(flag|benign) = 0.07524 \tag{12}$$
$$P(benign|flag) = \frac{0.07524 * 0.9136}{0.1538} \tag{13}$$
$$P(benign|flag) = 0.4469 \tag{14}$$

2. (a) The simplest and most accurate score function would be : $2a_1 + a_2 - a_3$, If the score goes above 10, then the score is 10. If the score goes below 0, then the score is 0. (As attribute a_1 is correlated twice as strongly with SPAM as a_2, and Attribute a3 is negatively correlated with SPAM exactly as strongly as a_3 is positively correlated with SPAM.

   (b) The scores would be as follows:

M1	Spam	4
M2	Not Spam	1
M3	Spam	9
M4	Spam	2
M5	Not Spam	7
M6	Not Spam	2
M7	Not Spam	8
M8	Spam	3
M9	Spam	7
M10	Not Spam	2

(c)

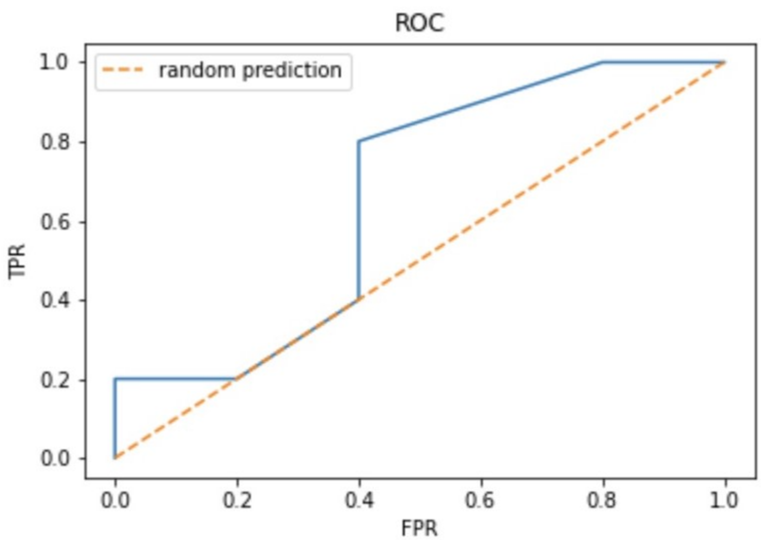
	Spam	Not Spam
Spam	4	2
Not Spam	1	3

i. False positive rate: $\frac{2}{5} = 40\%$

ii. True positive rate: $\frac{4}{5} = 80\%$

iii. False negative rate: $\frac{1}{5} = 20\%$

iv. True negative rate: $\frac{3}{5} = 60\%$



(d)

ROC is above.

3. (a) The size of the convoluted matrix is 22×22

   (b) We get the following 3×3 matrix :

4	6	5
6	6	8
9	8	8

4. Code

5. I used a subset of first 100 samples from the test dataset and used all the four methods given in the code i.e FGSM, Basic iterative method, Saliency map, and Universal Perturbation. The model summary is given in the code.
   The accuracies seem to be really good without much tuning, though I have tested 2 different CNN models and implemented the better one in that case.
   Finally, we can infer that the CNN and ANN models are trained on a dataset, but when we generate the adversarial samples, we basically generate data that has different distribution compared to the test and training set. Hence the CNN and ANN models perform so poorly. When we have the augmented dataset, almost half the dataset comes from the same distribution as the training dataset, therefore the accuracies are nearly 50 in such cases. And as the test data is from the same distribution, accuracy is very high.

```
In [1]:
# Imports.
import random
import os
print(os.getcwdb())
import network.network as Network
import network.mnist_loader as mnist_loader
import pickle
import matplotlib.pyplot as plt
import numpy as np

# Set the random seed. DO NOT CHANGE THIS!
seedVal = 41
random.seed(seedVal)
np.random.seed(seedVal)

%matplotlib inline
```

```
b'C:\\Users\\HP\\Desktop\\problem4'
```

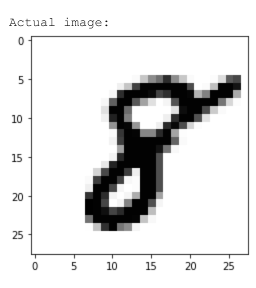
Use a pre-trained network. It has been saved as a pickle file. Load the model, and continue. The network has only one hidden layer of 30 units, 784 input units (MNIST images are $28 \times 28 = 784$ pixels large), and 10 output units. All the activations are sigmoidal.

```
In [2]:
# Load the pre-trained model.
with open('network/trained_network.pkl', 'rb') as f:
    u = pickle._Unpickler(f)
    u.encoding = 'latin1'
    net = u.load()

# Helpful function to load the MNIST data.
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
```

The neural network is pretrained, so it should already be set up to predict characters. Run `predict(n)` to evaluate the n^{th} digit in the test set using the network. You should see that even this relatively simple network works really well (~97% accuracy). The output of the network is a one-hot vector indicating the network's predictions:

```
In [9]:
def predict(n):
    # Get the data from the test set
    x = test_data[n][0]
    #print(x)
    # Print the prediction of the network
    print('Network output: \n' + str(np.round(net.feedforward(x), 2)) + '\n')
    print('Network prediction: ' + str(np.argmax(net.feedforward(x))) + '\n')
    print('Actual image: ')

    # Draw the image
    plt.imshow(x.reshape((28,28)), cmap='Greys')

# Replace the argument with any number between 0 and 9999
predict(8384)
```

```
Network output:
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]
 [0.]]

Network prediction: 8

Actual image:
```



To actually generate adversarial examples we solve a minimization problem. We do this by setting a "goal" label called $\vec y_{goal}$ (for instance, if we wanted the network to think the adversarial image is an 8, then we would choose $\vec y_{goal}$ to be a one-hot vector with the eighth entry being 1). Now we define a cost function:

$$ C = \frac{1}{2} \|\vec y_{goal} - \hat y(\vec x)\|^2_2 $$
where $\| \cdot \|^2_2$ is the squared Euclidean norm and $\hat y$ is the network's output. It is a function of $\vec x$, the input image to the network, so we write $\hat y(\vec x)$. Our goal is to find an $\vec x$ such that C is minimized. Hopefully this makes sense, because if we find an image $\vec x$ that minimizes C then that means the output of the network when given $\vec x$ is close to our desired output, $\vec y_{goal}$. So in full mathy language, our optimization problem is:

$$ \arg \min_{\vec x} C(\vec x) $$
that is, find the $\vec x$ that minimizes the cost C.

To actually do this we can do gradient descent on C. Start with an initially random vector $\vec x$ and take steps (changing $\vec x$) gradually in the direction opposite of the gradient $\nabla_x C$. To actually get these derivatives we can perform backpropagation on the network. In contrast to training a network, where we perform gradient descent on the weights and biases, when we create adversarial examples we hold the weights and biases constant (because we don't want to change the network!), and change the inputs to our network.

Helper functions to evaluate the non-linearity and it's derivative:

```
In [4]:
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

Also, a function to find the gradient derivatives of the cost function, $\nabla_x C$ with respect to the input $\vec x$, with a goal label of $\vec y_{goal}$. (Don't worry too much about the implementation, just know it calculates derivatives).

```
In [5]:
def input_derivative(net, x, y):
    """ Calculate derivatives wrt the inputs"""
    nabla_b = [np.zeros(b.shape) for b in net.biases]
    nabla_w = [np.zeros(w.shape) for w in net.weights]

    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(net.biases, net.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)

    # backward pass
    delta = net.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    for l in range(2, net.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(net.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())

    # Return derivatives WRT to input
    return net.weights[0].T.dot(delta)
```

The actual function that generates adversarial examples and a wrapper function:

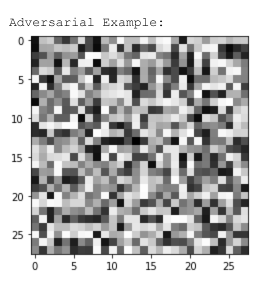
(a) Non Targeted Attack

```
In [18]:
def nonTargetedAdversarial(net, n, steps, eta):
    """
    net : network object
        neural network instance to use
    n : integer
        our goal label (just an int, the function transforms it into a one-hot vector)
    steps : integer
        number of steps for gradient descent
    eta : float
        step size for gradient descent
    """

    ####### Enter your code below #######

    # Set the goal output
    goal = np.zeros((10,1))
    goal[n] = 1


    # Create a random image to initialize gradient descent with
    x = np.random.rand(784, 1)
    # Gradient descent on the input
    for i in range(steps):
        # Calculate the derivative
        x = x - eta*(input_derivative(net, x, goal))
        # The GD update on x

    return x


# Wrapper function
def generate(n):
    """
    n : integer
        goal label (not a one hot vector)
    """

    ####### Enter your code below #######

    # Find the vector x with the above function that you just wrote.
    x = nonTargetedAdversarial(net, n, 1000, 0.2)
    # Pass the generated image (vector) to the neural network. Perform a forward pass

    #print('Network Output: \n' + str(x) + '\n')

    #print('Network Prediction: ' + str(np.argmax(x)) + '\n')

    #print('Adversarial Example: ')

    print('Network output: \n' + str(np.round(net.feedforward(x), 2)) + '\n')
    print('Network prediction: ' + str(np.argmax(net.feedforward(x))) + '\n')
    print('Adversarial Example: ')

    plt.imshow(x.reshape(28,28), cmap='Greys')
```

Now let's generate some adversarial examples! Use the function provided to mess around with the neural network. (For some inputs gradient descent doesn't always converge; 0 and 5 seem to work pretty well though. I suspect convergence is very highly dependent on our choice of random initial $\vec x$. We'll see later in the notebook if we force the adversarial example to "look like" a handwritten digit, convergence is much more likely. In a sense we will be adding regularization to our generation process).

```
In [19]:
generate(2)
```

```
Network output:
[[0.  ]
 [0.  ]
 [0.99]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]]

Network prediction: 2

Adversarial Example:
```



(b) Targeted Attack(s)

Sweet! We've just managed to create an image that looks utterly meaningless to a human, but the neural network thinks is a '5' with very high certainty. We can actually take this a bit further. Let's generate an image that looks like one number, but the neural network is certain is another. To do this we will modify our cost function a bit. Instead of just optimizing the input image, $\vec x$, to get a desired output label, we'll also optimize the input to look like a certain image, $\vec x_{target}$, at the same time. Our new cost function will be

$$ C = \|\vec y_{goal} - y_{hat}(\vec x)\|^2_2 + \lambda \|\vec x - \vec x_{target}\|^2_2 $$
The added term tells us the distance from our $\vec x$ and some $\vec x_{target}$ (which is the image we want our adversarial example to look like). Because we want to minimize C, we also want to minimize the distance between our adversarial example and this image. The λ is hyperparameter that we can tune; it determines which is more important: optimizing for the desired output or optimizing for an image that looks like $\vec x_{target}$.

If you are familiar with ridge regularization, the above cost function might look suspiciously like the ridge regression cost function. In fact, we can view this generation method as giving our model a prior, centered on our target image.

Here is a function that implements optimizing the modified cost function, called `sneaky_adversarial` (because it is very sneaky). Note that the only difference between this function and `adversarial` is an additional term on the gradient descent update for the regularization term:

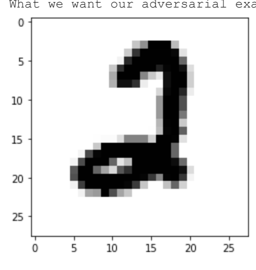
```
In [ ]:
# Probably include a question on why not generate an image of 5, instead of having to
# gradient based optimization methods. (as in the above case)
```

```
In [24]:
def targetedAdversarial(net, n, x_target, steps, eta, lam=.05):
    """
    net : network object
        neural network instance to use
    n : integer
        our goal label (just an int, the function transforms it into a one-hot vector)
    x_target : numpy vector
        our goal image for the adversarial example
    steps : integer
        number of steps for gradient descent
    eta : float
        step size for gradient descent
    lam : float
        lambda, our regularization parameter. Default is .05
    """

    # Set the goal output
    goal = np.zeros((10,1))
    goal[n] = 1
    # Create a random image to initialize gradient descent with
    x = np.random.rand(784, 1)
    # Gradient descent on the input
    for i in range(steps):

        # Calculate the derivative
        x = x - eta*(input_derivative(net, x, goal)+lam*(x-x_target))
        # The GD update on x, with an added penalty to the cost function

    return x

# Wrapper function
def generate_advSample(n, m):
    """
    n: int 0-9, the target number to match
    m: index of example image to use (from the test set)
    """

    # Find random instance of m in test set
    idx = np.random.randint(0,8000)
    while test_data[idx][1] != m:
        idx += 1

    # Hardcode the parameters for the wrapper function
    a = targetedAdversarial(net, n, test_data[idx][0], 100, 1)
    x = np.round(net.feedforward(a), 2)

    print('\nWhat we want our adversarial example to look like: ')
    plt.imshow(test_data[idx][0].reshape((28,28)), cmap='Greys')
    plt.show()

    print('\n')

    print('Adversarial Example: ')

    plt.imshow(a.reshape(28,28), cmap='Greys')
    plt.show()

    print('Network Prediction: ' + str(np.argmax(x)) + '\n')

    print('Network Output: \n' + str(x) + '\n')

    return a
```

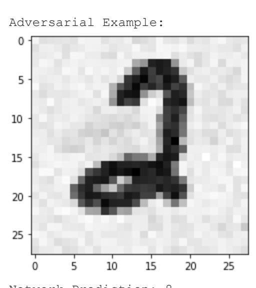
Play around with this function to make "sneaky" adversarial examples! (Again, some numbers converge better than others... try 0, 2, 3, 5, 6, or 8 as a target label. 1, 4, 7, and 9 still don't work as well... no idea why... We get more numbers that converge because we've added regularization term to our cost function. Perhaps changing λ will get more to converge?)

```
In [25]:
# generate_advSample(target label, target digit)
adv_ex = generate_advSample(8, 2)
```

```
What we want our adversarial example to look like:
```



```
Adversarial Example:
```



```
Network Prediction: 8

Network Output:
[[0.  ]
 [0.  ]
 [0.01]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]
 [1.  ]
 [0.  ]]
```

(c) Protection against adversarial attacks

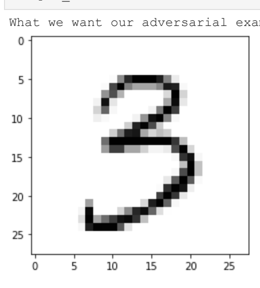
Awesome! We've just created images that trick neural networks. The next question we could ask is whether or not we could protect against these kinds of attacks. If you look closely at the original images and the adversarial examples you'll see that the adversarial examples have some sort of grey tinged background.

So how could we protect against these adversarial attacks? One very simple way would be to use binary thresholding. Set a pixel as completely black or completely white depending on a threshold. This should remove the "noise" that's always present in the adversarial images. Let's see if it works:

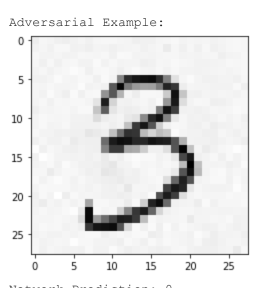
```
In [28]:
def simple_defense(n, m):
    """
    n: int 0-9, the target number to match
    m: index of example image to use (from the test set)
    """

    # Generate an adversarial sample.
    x = generate_advSample(n, m)

    # Perform binary thresholding on the generated sample. You can choose the thresho
    for k in range(len(x)):
        if x[k] > 0.5:
            x[k] = 1
        else:
            x[k] = 0
    print("With binary thresholding: ")

    # Plot a grayscale image of the binarized generated sample.
    plt.imshow(x.reshape(28,28), cmap='Greys')
    plt.show()
    # Print the network's predictions.
    print("Prediction with binary thresholding: " + str(np.argmax(net.feedforward(x)))

    # The output of the network.
    print("Network output: ")
    print( str(np.round(net.feedforward(x), 2)) )
```

```
In [39]:
# binary_thresholding(target digit, actual digit)
simple_defense(0, 3)
```

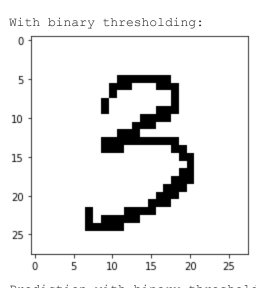
```
What we want our adversarial example to look like:
```



```
Adversarial Example:
```



```
Network Prediction: 0

Network Output:
[[0.96]
 [0.  ]
 [0.  ]
 [0.02]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]
 [0.  ]]

With binary thresholding:
```

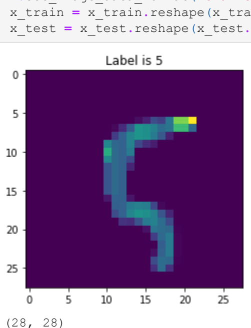


```
Prediction with binary thresholding: 3

Network output:
[[0.]
 [0.]
 [0.]
 [1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

Looks like it works pretty well! However, note that most adversarial attacks, especially on convolutional neural networks trained on massive full color image sets such as imagenet, can't be defended against by a simple binary threshold.

```
In [ ]:
```

Setting up things



Train CNN based model

You need to complete the following.

Train ANN based model (dense layers only)

You need to complete the following.

```
ann_based_classifier = KerasClassifier(model=ann_based_model, clip_values=(0,1))
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
ann_based_classifier.fit(x_train, y_train, nb_epochs=10, batch_size=BATCH_SIZE)

# Get the predictions on the test set.
predictions = ann_based_classifier.predict(x_test)

# Compute the accuracy on the test set.
ann_score = ann_based_model.evaluate(x_test,y_test,batch_size=BATCH_SIZE)[1]

print('Test accuracy:', ann_score)
```

```
Train on 60000 samples
Epoch 1/10
60000/60000 [==============================] - 3s 42us/sample - loss: 0.0123 - accurac
y: 0.9973
Epoch 2/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0102 - accurac
y: 0.9975
Epoch 3/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0072 - accurac
y: 0.9987
Epoch 4/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0058 - accurac
y: 0.9991
Epoch 5/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0044 - accurac
y: 0.9992
Epoch 6/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0045 - accurac
y: 0.9991
Epoch 7/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0030 - accurac
y: 0.9997
Epoch 8/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0045 - accurac
y: 0.9991
Epoch 9/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0028 - accurac
y: 0.9997
Epoch 10/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0010 - accurac
y: 0.9999
Test accuracy: 0.9769
```

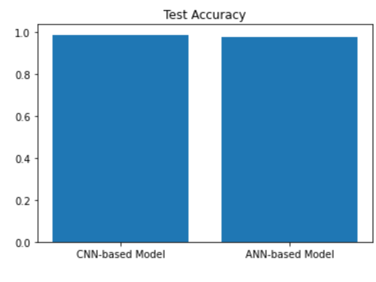
```
ann_based_model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 256)               200960
_________________________________________________________________
dense_8 (Dense)              (None, 10)                2570
=================================================================
Total params: 203,530
Trainable params: 203,530
Non-trainable params: 0
```

Compare the classification accuracy on the test data graphically

You need to complete the following.

```
X = ['CNN-based Model','ANN-based Model']
Y = [cnn_based_score1, ann_score]
plt.bar(X, Y)
plt.title('Test Accuracy')
plt.show()
plt.close()
```



Part 2: Generate adversarial examples using four methods

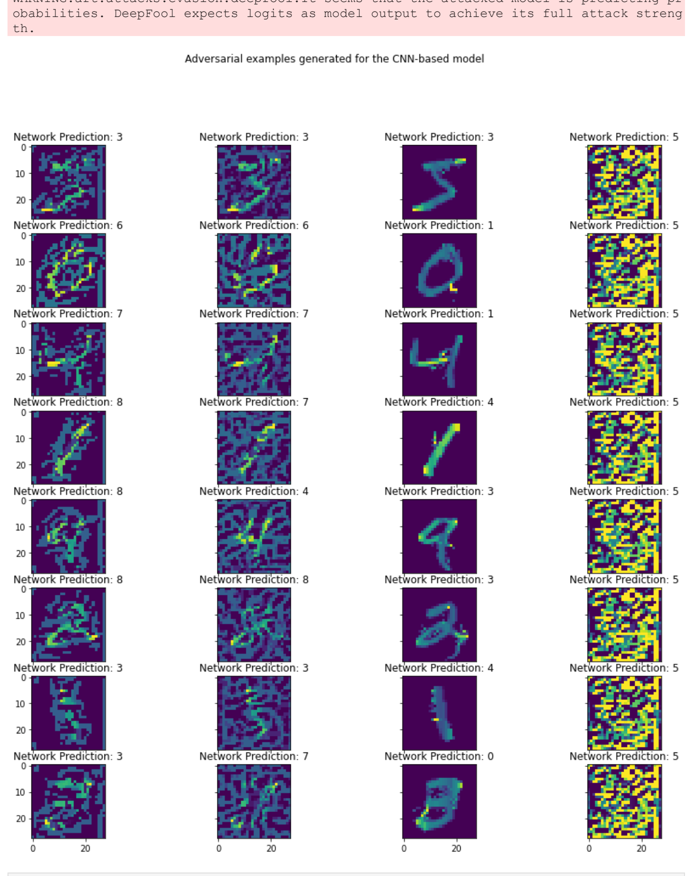
You need to complete the following.

In this question, we will use the following four attack methods: (1) FGSM, (2) Basic Iterative Method, (3) Saliency Map Method, (4) Universal Perturbation.

```
def generate_adv_examples_FGSM(classifier, x):
    # This attack is known as the "Fast Gradient Sign Method".
    attack = FastGradientMethod(classifier)
    x_adv = attack.generate(x)
    return x_adv

def generate_adv_examples_BasicIterativeMethod(classifier, x):
    # The Basic Iterative Method is the iterative version of FGM and FGSM.
    attack = BasicIterativeMethod(classifier)
    x_adv = attack.generate(x)
    return x_adv

def generate_adv_examples_SaliencyMapMethod(classifier, x):
    # attack from the Jacobian-based Saliency Map Attack (Papernot et al. 2016).
    # Paper link: https://arxiv.org/abs/1511.07528
    attack = SaliencyMapMethod(classifier)
    x_adv = attack.generate(x)
    return x_adv

def generate_adv_examples_UniversalPerturbation(classifier, x):
    attack = UniversalPerturbation(classifier)
    x_adv = attack.generate(x)
    return x_adv
```

```
# Generate examples for CNN-based model
x = x_train[0:100]
classifier = cnn_based_classifier

cnn_adv_examples_FGSM = generate_adv_examples_FGSM(classifier, x)
cnn_adv_examples_BasicIterativeMethod = generate_adv_examples_BasicIterativeMethod(cla
cnn_adv_examples_SaliencyMapMethod = generate_adv_examples_SaliencyMapMethod(classifie
cnn_adv_examples_UniversalPerturbation = generate_adv_examples_UniversalPerturbation(c

# Generate plot
fig, ax = plt.subplots(8, 4, sharex='col', sharey='row', figsize=(15,15))
fig.suptitle('Adversarial examples generated for the CNN-based model')
cnn_adv_examples = [cnn_adv_examples_FGSM,cnn_adv_examples_BasicIterativeMethod,cnn_a

for i in range(8):
    for j in range(4):
        ax[i, j].imshow(cnn_adv_examples[j][i].squeeze())
        predictions = cnn_based_classifier.predict(np.expand_dims(cnn_adv_examples[j]
        ax[i, j].set_title('Network Prediction: {}'.format(np.argmax(predictions)))
```

```
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
```

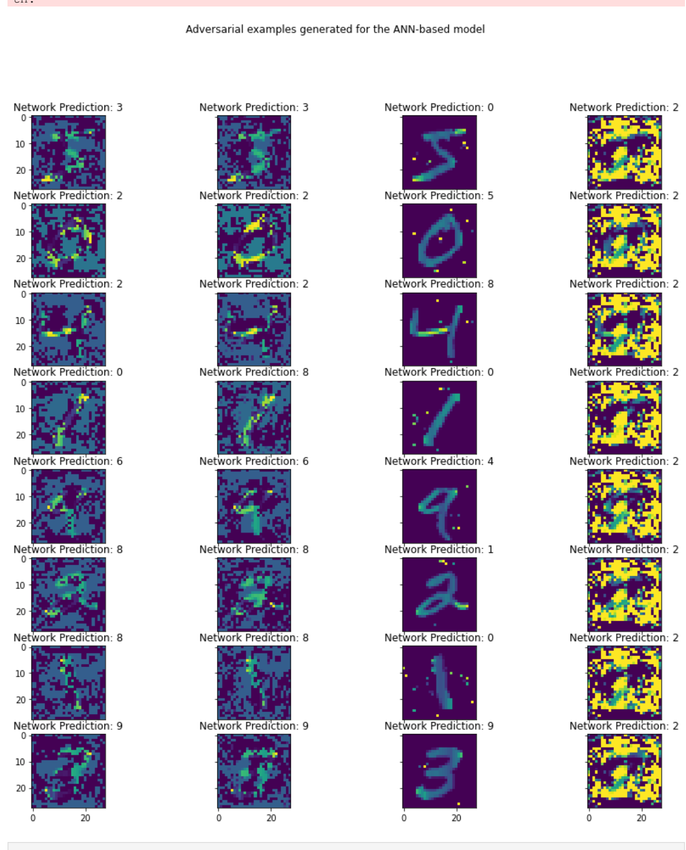



```
print(cnn_adv_examples_BasicIterativeMethod.shape )
```

```
(100, 1, 28, 28)
```

```
# Generate examples for ANN-based model
x = x_train[0:100]
x = x.reshape(100, 784)
classifier = ann_based_classifier

ann_adv_examples_FGSM = generate_adv_examples_FGSM(classifier, x)
ann_adv_examples_BasicIterativeMethod = generate_adv_examples_BasicIterativeMethod(cla
ann_adv_examples_SaliencyMapMethod = generate_adv_examples_SaliencyMapMethod(classifie
ann_adv_examples_UniversalPerturbation = generate_adv_examples_UniversalPerturbation(c

# Generate plots
fig, ax = plt.subplots(8, 4, sharex='col', sharey='row', figsize=(15,15))
fig.suptitle('Adversarial examples generated for the ANN-based model')
ann_adv_examples = [ann_adv_examples_FGSM.reshape(100, 28, 28),ann_adv_examples_Basic

for i in range(8):
    for j in range(4):
        ax[i, j].imshow(ann_adv_examples[j][i].squeeze())
        predictions = ann_based_classifier.predict(np.expand_dims(ann_adv_examples[j]
        ax[i, j].set_title('Network Prediction: {}'.format(np.argmax(predictions)))
```

```
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
WARNING:art.attacks.evasion.deepfool:It seems that the attacked model is predicting pr
obabilities. DeepFool expects logits as model output to achieve its full attack streng
th.
```




```
xe = ann_adv_examples_FGSM.reshape(100,28,28)
print(xe[0].shape)
```

```
(28, 28)
```

Part 3: Create a new test set, based entirely on the adversarial images generated previously. Test your classifiers performance on this test set.

You need to complete the following.

```
# We will be using the FGSM method in this part
x1 = x_test[0:10000]
classifier = cnn_based_classifier
cnn_adv_examples_FGSM = generate_adv_examples_FGSM(classifier, x1)
x2 = x_test[0:10000]
x2 = x2.reshape(10000, 784)
classifier2 = ann_based_classifier
ann_adv_examples_FGSM = generate_adv_examples_FGSM(classifier2, x2)
```

```
# Evaluate ANN-based-classifier on the newly generated adversarial test set
predictions = ann_based_classifier.predict(ann_adv_examples_FGSM)
ann_based_score = ann_based_model.evaluate(ann_adv_examples_FGSM, y_test)[1]
print('ANN-based Model Score: {}'.format(ann_based_score))
```

```
ANN-based Model Score: 0.016200000420212746
```

```
# Evaluate CNN-based-classifier on the newly generated adversarial test set
predictions = cnn_based_classifier.predict(cnn_adv_examples_FGSM)
cnn_based_score2 = cnn_based_model.evaluate(cnn_adv_examples_FGSM, y_test)[1]
print('CNN-based Model Score: {}'.format(cnn_based_score2))
```

```
CNN-based Model Score: 0.1542000025510788
```

```
print(ann_adv_examples_FGSM.shape)
print(x_train.shape)
```

```
(10000, 784)
(60000, 1, 28, 28)
```

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

Create a new augmented test set (original test images + adversarial images)

You need to complete the following.

```
# ANN-based-classifier
ann_augmented_examples = []
for i in range(10000):

    # Select adversarial samples.
    if random.randint(0, 1) == 0:
        ann_augmented_examples.append(ann_adv_examples_FGSM[i])
    # Select actual samples.
    else:
        ann_augmented_examples.append(x_test[i])

#ann_augmented_examples = np.concatenate(ann_augmented_examples, axis=0)
#print(ann_augmented_examples.shape)

predictions = ann_based_classifier.predict(ann_augmented_examples)
ann_based_score2 = ann_based_model.evaluate(x_test, predictions)[1]
print('ANN-based Model Score: {}'.format(ann_based_score2))
```

```
ANN-based Model Score: 0.503600001335144
```

```
# CNN-based-classifier
cnn_augmented_examples = []
for i in range(10000):

    # Select adversarial samples.
    if random.randint(0, 1) == 0:
        cnn_augmented_examples.append(cnn_adv_examples_FGSM[i])
    # Select actual samples.
    else:
        cnn_augmented_examples.append(x_test[i])

#cnn_augmented_examples = np.concatenate(cnn_augmented_examples, axis=0)

predictions = cnn_based_classifier.predict(cnn_augmented_examples)
cnn_based_score3 = cnn_based_model.evaluate(x_test, predictions)[1]
print('CNN-based Model Score: {}'.format(cnn_based_score3))
```

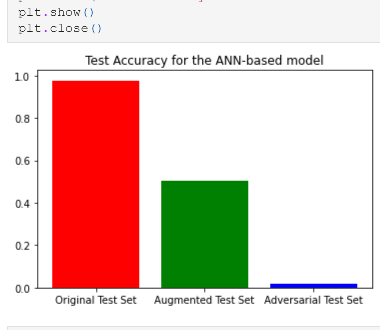
```
CNN-based Model Score: 0.5741000175476074
```

Make a single plot, wherein you compare the test accuracies of all the models, on the three types of test sets that you have

You need to complete the following.

```
cnn_models_score = [cnn_based_score1, cnn_based_score3, cnn_based_score2]
ann_models_score = [ann_score, ann_based_score2, ann_based_score]
```

```
# Plot for the ANN based model.
X = ['Original Test Set','Augmented Test Set', 'Adversarial Test Set']
plt.bar(X, ann_models_score, color=['r', 'g', 'b'])
plt.title('Test Accuracy for the ANN-based model')
plt.show()
plt.close()
```




```
# Plot for the CNN based model.
X = ['Original Test Set','Augmented Test Set', 'Adversarial Test Set']
plt.bar(X, cnn_models_score , color=['r', 'g', 'b'])
plt.title('Test Accuracy for the CNN-based model')
plt.show()
plt.close()
```