Project 3 Write-up

Haotian Xu
hax030@ucsd.edu

Abstract

In this project, we investigate techniques for creating adversarial examples aimed at neural networks, specifically using the MNIST dataset and targeting a pretrained multi-layer perceptron (MLP) with three hidden layers. Our exploration starts with the Fast Gradient Sign Method (FGSM) as a foundational approach. Building on this, we examine two variations: the Basic Iterative Method and the Target Class Method. Additionally, we develop and evaluate a novel method inspired by these established techniques, adapting the FGSM framework to enhance its efficacy.

1 Algorithm Description

For all our methods, we create adversarial examples $\tilde{x}$ by adding a minor perturbation Δx to the original images x. We limit the perturbation to ϵ, ensuring that the infinity norm of the difference between x and $\tilde{x}$ does not exceed ϵ, as illustrated in the equation below:

$$\|x - \tilde{x}\|_\infty \leq \epsilon \tag{1}$$

This constraint allows us to fairly compare the effectiveness of different methods under similar conditions. We assume our neural network function is f, and we can compute the gradient of the loss function ∇L.

1.1 Fast Gradient Sign Method

The Fast Gradient Sign Method (FGSM) (Goodfellow et al., 2014) is designed to increase the loss of neural networks for a given image and label pair (x, y) by applying a perturbation in the direction of the gradient's sign. This approach ensures adherence to the perturbation constraint. Below is the pseudocode for FGSM as presented in Algorithm 1. We will present the test accuracy of this method later.

Algorithm 1 FGSM

Function: FGSM(x, y, ϵ):
  $\tilde{x} = x + \epsilon \cdot \text{sign}(\nabla L(f, x, y))$
  Clip $\tilde{x}$ to the valid range
  return $\tilde{x}$

1.2 Basic Iterative Method

The Basic Iterative Method (BIM) (Kurakin et al., 2016) extends FGSM by iteratively applying small FGSM perturbations. This approach enhances control over the perturbation process and ensures compliance with the perturbation limit through clipping. Algorithm 2 outlines the procedure for BIM.

Algorithm 2 Basic Iterative Method

Function: BIM($x, y, \epsilon, \alpha, num\_iterations$):
  $\Delta x = 0$
  For $num\_iterations$:
    $\Delta x = \Delta x + \alpha \cdot \text{sign}(\nabla L(f, x + \Delta x, y))$
    Clip Δx to the range $[-\epsilon, \epsilon]$
  $\tilde{x} = x + \Delta x$
  Clip $\tilde{x}$ to the valid range
  return $\tilde{x}$

1.3 Target Class Method

The Target Class Method (Kurakin et al., 2016) adapts the FGSM approach to not just increase the loss for the actual label but to specifically move the sample towards a chosen target class. This is done by applying a perturbation in the opposite direction of the gradient's sign with respect to the target class, often selecting the least likely class as the target for this purpose. Algorithm 3 outlines the steps for implementing this method.

1.4 Combined Method

Building on the insights from previous methods, we propose a hybrid approach that integrates the

Algorithm 3 Target Class Method

Function: TCM($x, y, \epsilon, target$)
  $\tilde{x} = x - \epsilon \cdot \text{sign}(\nabla L(f, x, target))$
  Clip $\tilde{x}$ to the valid range
  return $\tilde{x}$

principles of the Target Class Method and FGSM. In this method, we refine the selection criteria for the target class. Rather than choosing the least likely class, we opt for the class most similar to the actual class. To achieve this, we compute the cosine similarity scores between samples of each class and identify the most similar class for each. This adjustment is based on the understanding that the MNIST dataset's balanced nature may not favor the selection of the least likely class. The cosine similarity function used in this process is defined as:

$$\text{cosine\_similarity}(a, b) = \frac{a \cdot b}{\|a\|_2 \times \|b\|_2} \quad (2)$$

In our composite method, we calculate adversarial examples $\tilde{x}_1$ using the modified Target Class Method and $\tilde{x}_2$ using FGSM. We then evaluate the effectiveness of each adversarial example by calculating the probability of the actual class using the neural network's forward function. The method that more significantly reduces this probability is deemed more effective, and its result is chosen. Algorithm 4 outlines this process, where $target\_list$ contains the most similar class for each actual class.

Algorithm 4 Combined Method

Function: Combined($x, y, \epsilon, target\_list$)
  $\tilde{x}_1 = x + \epsilon \cdot \text{sign}(\nabla L(f, x, y))$
  Clip $\tilde{x}_1$ to the valid range
  $target = target\_list[y]$
  $\tilde{x}_2 = x - \epsilon \cdot \text{sign}(\nabla L(f, x, target))$
  Clip $\tilde{x}_2$ to the valid range
  $p_1 = \text{forward}(\tilde{x}_1, y)$
  $p_2 = \text{forward}(\tilde{x}_2, y)$
  $\tilde{x} = \tilde{x}_1$
  if $p_2[y] < p_1[y]$ **then**
    $\tilde{x} = \tilde{x}_2$
  return $\tilde{x}$

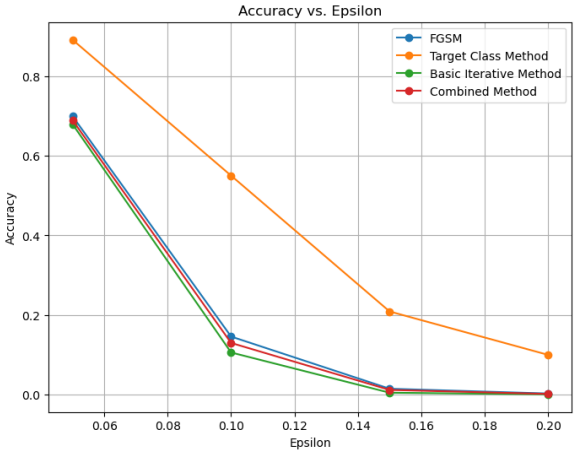


Figure 1: Accuracy vs. ϵ

2 Experiment and Results

2.1 Experiment Settings

Our evaluation encompasses four methods for generating adversarial examples, applied to a subset of the MNIST test set and analyzed using a pretrained neural network with three hidden layers. We experiment with various ϵ values: 0.05, 0.1, 0.15, and 0.2. After we generate adversarial examples, we measure the accuracy of predicting the true class of the adversarial examples. For the Basic Iterative Method, we set $num\_iterations$ to 5 and align α with ϵ for consistency. In applying the Target Class Method, we identify the digit 0 as the least likely class, represented by 85 out of 1,000 test samples. For our combined method, we select a representative sample from each class to calculate similarity scores.

2.2 Results

Figure 1 and Table 1 present the neural network's accuracy on adversarial examples against various ϵ values for different methods. The figure illustrates accuracy trends, while the table details both accuracy and run times.

Analysis reveals that both the Basic Iterative Method and the combined method slightly outperform FGSM, with the Target Class Method lagging in effectiveness. As ϵ increases, the accuracy of the Basic Iterative Method, the combined method, and FGSM approaches zero, indicating that larger perturbations more effectively degrade performance. However, the impact of increasing ϵ diminishes at higher values.

The table shows run times are not directly correlated with ϵ. The Target Class Method is the

Method	Epsilon	Accuracy	Run Time (s)
FGSM	0.05	0.699	150.981
	0.1	0.146	152.629
	0.15	0.015	151.526
	0.2	0.003	156.816
Basic Iterative Method	0.05	0.679	640.813
	0.1	0.106	639.291
	0.15	0.005	636.889
	0.2	0.001	642.296
Target Class Method	0.05	0.891	138.975
	0.1	0.55	138.348
	0.15	0.209	137.886
	0.2	0.1	138.536
Combined Method	0.05	0.689	285.512
	0.1	0.13	286.385
	0.15	0.012	284.510
	0.2	0.002	285.655

Table 1: Adversarial Attack Methods Performance

fastest but least effective. The Basic Iterative Method demonstrates the highest accuracy at the cost of longer run times. Conversely, the combined method offers a balanced solution, achieving near-optimal performance with significantly reduced computational demands.

2.3 Analysis

The empirical outcomes show that an increase in computational effort correlates with enhanced performance across the evaluated adversarial methods. Specifically, the Target Class Method, which employs the least likely target strategy, does not surpass the effectiveness of other approaches. This observation is likely attributable to the balanced nature of the test set. Notably, maintaining an accuracy rate of 0.1 at $\epsilon = 0.2$ supports the notion that a subset of samples remains accurately classified even when targeting the least likely class. However, it's important to clarify that this method's performance might improve in scenarios involving imbalanced datasets.

The Basic Iterative Method, which iteratively applies the principles of FGSM, exhibited expectedly strong performance. Despite this, the marginal gains over FGSM may not justify the additional computational resources required. This outcome suggests a diminishing return on investment for the increased complexity and run time associated with this method.

Meanwhile, the performance of the combined method, which strategically oscillates between FGSM and a modified version of the Target Class Method, occupies a middle ground. Its modest advantage over FGSM validates our approach to dynamically select between the two strategies based on their effectiveness. This finding highlights the potential benefits of adaptive techniques in adversarial example generation, balancing performance improvement with computational efficiency.

3 Critical Evaluation

The incremental improvement of our combined method over the Fast Gradient Sign Method (FGSM) underscores the challenge in significantly surpassing the effectiveness of well-established adversarial attack techniques. FGSM's simplicity and efficiency set a high benchmark. Nevertheless, our analysis indicates potential areas for enhancement.

First, the optimization of hyperparameters for the Basic Iterative Method was constrained by computational resources. It is plausible that an optimal set of parameters exists, which could offer a more favorable balance between accuracy and computational overhead. Future efforts will be dedicated to exploring this possibility through extensive parameter tuning.

Moreover, our approach to calculating similarity scores was based on a singular sample per class. To fortify the foundation of our similarity-based decisions, incorporating a larger set of samples per class is a logical next step. This expansion,

however, introduces the challenge of managing the increased computational demand, particularly in datasets with a broad array of classes. Identifying efficient methodologies to compute class similarities without disproportionately inflating the computational budget will be a focus of subsequent research.

Given the adaptive nature of our combined method, which strategically chooses between different attack strategies, integrating additional sophisticated adversarial techniques could enhance its efficacy. This direction promises an intriguing avenue for elevating the performance of adversarial attack methods through diversity and strategic selection.

References

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*.

In [9]:
```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import time
```

First, we define some functions for computing the output of the multilayer perceptron.

In [2]:
```
def softmax(x):
    '''
    Input
         x: a vector in ndarray format,
            typically the raw score of prediction.
    Output
         a vector in ndarray format,
         typically representing the predicted class probability.
    '''
    res = np.exp(x-np.max(x))
    return res/np.sum(res)

def cross_entropy(y, p):
    '''
    Input
        y: an int representing the class label
        p: a vector in ndarray format showing the predicted
           probability of each class.

    Output
        the cross entropy loss.
    '''
    log_likelihood = -np.log(p)
    return log_likelihood[y]

def relu(x):
    '''
    Input
        x: a vector in ndarray format
    Output
        a vector in ndarray format,
        representing the ReLu activation of x.
    '''
    return np.maximum(x, 0)
```

Next, we define the structure and some utility functions of our multi-layer perceptron.

In [73]:
```
class MultiLayerPerceptron():
    '''
    This class defines the multi-layer perceptron we will be using
    as the attack target.

    '''
    def __init__(self):
        self.eps = 0.1

    def load_params(self, params):
        '''
        This method loads the weights and biases of a trained model.
```

```
        '''
        self.W1 = params["fc1.weight"]
        self.b1 = params["fc1.bias"]
        self.W2 = params["fc2.weight"]
        self.b2 = params["fc2.bias"]
        self.W3 = params["fc3.weight"]
        self.b3 = params["fc3.bias"]
        self.W4 = params["fc4.weight"]
        self.b4 = params["fc4.bias"]

    def set_attack_budget(self, eps):
        '''
        This method sets the maximum L_infty norm of the adversarial
        perturbation.
        '''
        self.eps = eps

    def forward(self, x):
        '''
        This method finds the predicted probability vector of an input
        image x.

        Input
            x: a single image vector in ndarray format
        Ouput
            a vector in ndarray format representing the predicted class
            probability of x.

        Intermediate results are stored as class attributes.
        You might need them for gradient computation.
        '''
        W1, W2, W3, W4 = self.W1, self.W2, self.W3, self.W4
        b1, b2, b3, b4 = self.b1, self.b2, self.b3, self.b4

        self.z1 = np.matmul(x,W1)+b1
        self.h1 = relu(self.z1)
        self.z2 = np.matmul(self.h1,W2)+b2
        self.h2 = relu(self.z2)
        self.z3 = np.matmul(self.h2,W3)+b3
        self.h3 = relu(self.z3)
        self.z4 = np.matmul(self.h3,W4)+b4
        self.p = softmax(self.z4)

        return self.p

    def predict(self, x):
        '''
        This method takes a single image vector x and returns the
        predicted class label of it.
        '''
        res = self.forward(x)
        return np.argmax(res)

    def gradient(self,x,y):
        '''
        This method finds the gradient of the cross-entropy loss
        of an image-label pair (x,y) w.r.t. to the image x.

        Input
            x: the input image vector in ndarray format
```

```
            y: the true label of x

        Output
            a vector in ndarray format representing
            the gradient of the cross-entropy loss of (x,y)
            w.r.t. the image x.
        '''

        self.p = self.forward(x)
        y_one_hot = np.zeros(self.p.shape)
        y_one_hot[y] = 1
        self.dl_dz4 = self.p - y_one_hot
        self.dz4_dh3 = self.W4.T
        #Similarly fill the others
        self.dh1_dz1 = np.diag(1 * (self.h1 > 0))
        self.dh2_dz2 = np.diag(1 * (self.h2 > 0))
        self.dh3_dz3 = np.diag(1 * (self.h3 > 0))
        self.dz3_dh2 = self.W3.T
        self.dz2_dh1 = self.W2.T
        self.dz1_dh0 = self.W1.T
        self.gradients = self.dl_dz4@self.dz4_dh3@((self.dh3_dz3@self.dz3_dh2)\
                        @(self.dh2_dz2@self.dz2_dh1)@(self.dh1_dz1@self.dz1_dh0))
        return self.gradients

    def attack(self,x,y):
        '''
        This method generates the adversarial example of an
        image-label pair (x,y).

        Input
            x: an image vector in ndarray format, representing
               the image to be corrupted.
            y: the true label of the image x.

        Output
            a vector in ndarray format, representing
            the adversarial example created from image x.
        '''

        newX = x + self.eps * np.sign(self.gradient(x,y))
        np.clip(newX, 0.0, 1.0, out=newX)
        return newX


    def one_fgsm(self, x, y, label): # Target Class Method
        newX = x - self.eps * np.sign(self.gradient(x,label))
        np.clip(newX, 0.0, 1.0, out=newX)
        return newX
    def it_fgsm(self, x, y):# Basic Iterative Method
        newX = x
        delta = 0
        for i in range(5):
            delta = delta + self.eps * np.sign(self.gradient(newX + delta,y))
            np.clip(delta, -self.eps, self.eps, out=delta)
        newX = newX + delta
        np.clip(newX, 0.0, 1.0, out=newX)
        return newX
    def cust_fgsm(self, x, y): # Revised Target Class Method
        most_similar = [3,8,6,7,6,8,4,3,1,3]
        label = most_similar[y]
```

```
        newX = x - self.eps * np.sign(self.gradient(x,label))
        np.clip(newX, 0.0, 1.0, out=newX)
        return newX
    def cust_fgsm2(self, x, y): # Combined Method
        most_similar = [3,8,6,7,6,8,4,3,1,3]
        label = most_similar[y]
        newX1 = x - self.eps * np.sign(self.gradient(x,label))
        newX2 = x + self.eps * np.sign(self.gradient(x,y))
        np.clip(newX1, 0.0, 1.0, out=newX1)
        np.clip(newX2, 0.0, 1.0, out=newX2)
        y1 = self.forward(newX1)
        y2 = self.forward(newX2)
        newX = newX2
        if y1[y] < y2[y]:
            newX = newX1
        return newX
```

Now, let's load the pre-trained model and the test data.

In [74]:
```
X_test = np.load("./data/X_test.npy")
Y_test = np.load("./data/Y_test.npy")

params = {}
param_names = ["fc1.weight", "fc1.bias",
               "fc2.weight", "fc2.bias",
               "fc3.weight", "fc3.bias",
               "fc4.weight", "fc4.bias"]

for name in param_names:
    params[name] = np.load("./data/"+name+'.npy')

clf = MultiLayerPerceptron()
clf.load_params(params)
```

In [36]:
```
for i in range(10): # Find the number of samples in each class
    print(np.sum(np.array(Y_test==i)))
```

```
85
126
116
107
110
87
87
99
89
94
```

In [58]:
```
num_classes = np.unique(Y_test).size
samples_per_class = {}

for i in range(len(Y_test)):
    label = Y_test[i]
    if label not in samples_per_class:
        samples_per_class[label] = X_test[i]
    if len(samples_per_class) == num_classes:  # Stop once we have one sample per clas
        break
def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

```
similarity_scores = np.zeros((num_classes, num_classes))

for i, (label_i, sample_i) in enumerate(samples_per_class.items()):
    for j, (label_j, sample_j) in enumerate(samples_per_class.items()):
        if i <= j:
            similarity = cosine_similarity(sample_i, sample_j)
            similarity_scores[i, j] = similarity
            similarity_scores[j, i] = similarity
most_similar_classes = {}
for i in range(num_classes):
    similarity_scores[i, i] = -1
    most_similar_to_i = np.argmax(similarity_scores[i, :])
    similarity_scores[i, i] = 1
    most_similar_classes[i] = most_similar_to_i
for class_label, most_similar in most_similar_classes.items():
    print(f"Class {class_label} is most similar to class {most_similar}.")
```

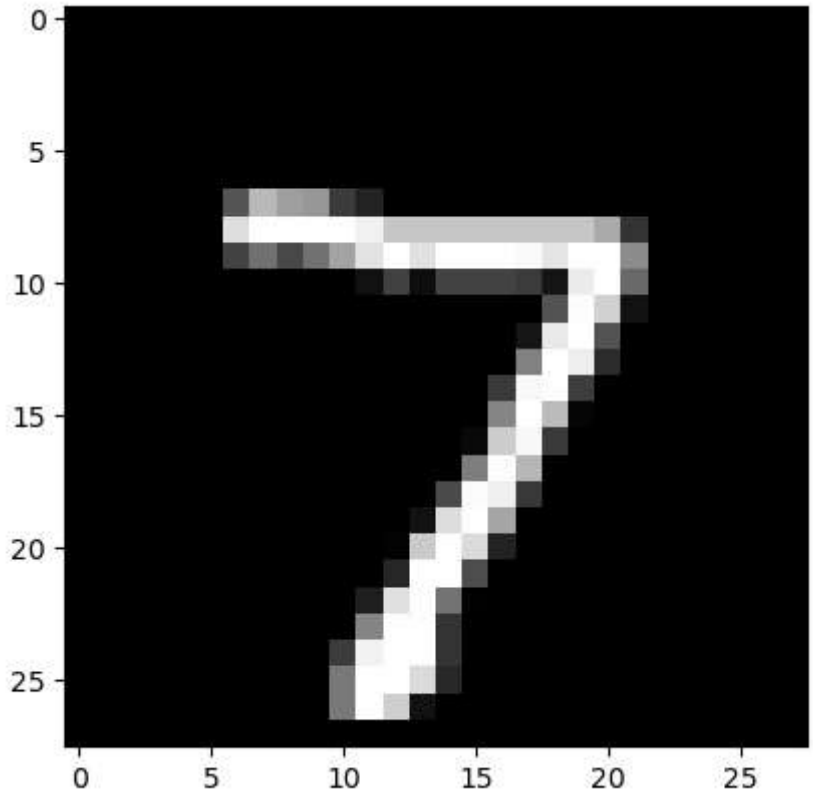
```
Class 0 is most similar to class 3.
Class 1 is most similar to class 8.
Class 2 is most similar to class 6.
Class 3 is most similar to class 7.
Class 4 is most similar to class 6.
Class 5 is most similar to class 8.
Class 6 is most similar to class 4.
Class 7 is most similar to class 3.
Class 8 is most similar to class 1.
Class 9 is most similar to class 3.
```

Check if the image data are loaded correctly. Let's visualize the first image in the data set.

In [6]:
```
x, y = X_test[0], Y_test[0]
print ("This is an image of Number", y)
pixels = x.reshape((28,28))
plt.imshow(pixels,cmap="gray")
```

```
This is an image of Number 7
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x1cab181b190>
```



Check if the model is loaded correctly. The test accuracy should be 97.6%

In [116…

```
nTest = 1000
Y_pred = np.zeros(nTest)
for i in range(nTest):
    x, y = X_test[i], Y_test[i]
    Y_pred[i] = clf.predict(x)
acc = np.sum(Y_pred == Y_test[:nTest])*1.0/nTest
print ("Test accuracy is", acc)
```

```
Test accuracy is 0.976
```

Have fun!

In [53]:

```
def test_with_adversarial_attacks(clf, X_test, Y_test, eps):
    clf.set_attack_budget(eps)
    correct = 0
    Y_pred = np.zeros(len(X_test))
    for i in range(len(X_test)):
        x, y = X_test[i], Y_test[i]
        Y_pred[i] = np.argmax(clf.forward(x))
        newX = clf.attack(x, y)
        newY = clf.predict(newX)
        if newY == y:
            correct += 1
    accuracy = correct / len(X_test)
    return accuracy
```

In [54]:

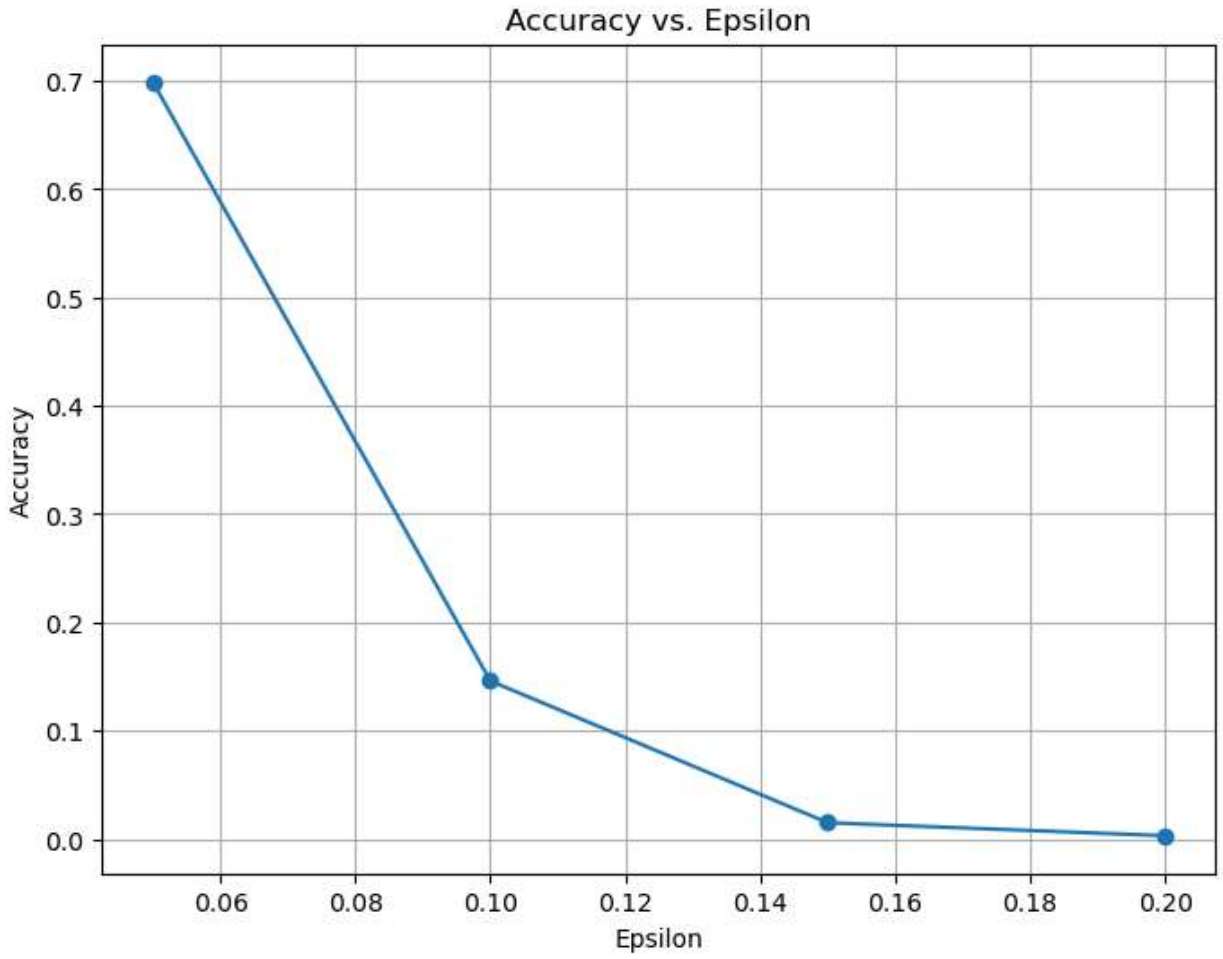
```
eps_values = [0.05, 0.1, 0.15, 0.2]
accuracies = []

for eps in eps_values:
```

```
    total_time = 0
    start_time = time.time()
    accuracy = test_with_adversarial_attacks(clf, X_test, Y_test, eps)
    total_time = time.time() - start_time
    accuracies.append(accuracy)
    print(f"Epsilon: {eps}, Accuracy: {accuracy}, Run Time: {total_time}")

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(eps_values, accuracies, marker='o', linestyle='-')
plt.title("Accuracy vs. Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
```

```
Epsilon: 0.05, Accuracy: 0.699, Run Time: 150.98073387145996
Epsilon: 0.1, Accuracy: 0.146, Run Time: 152.62874341011047
Epsilon: 0.15, Accuracy: 0.015, Run Time: 151.52640509605408
Epsilon: 0.2, Accuracy: 0.003, Run Time: 156.81603360176086
```



New Method

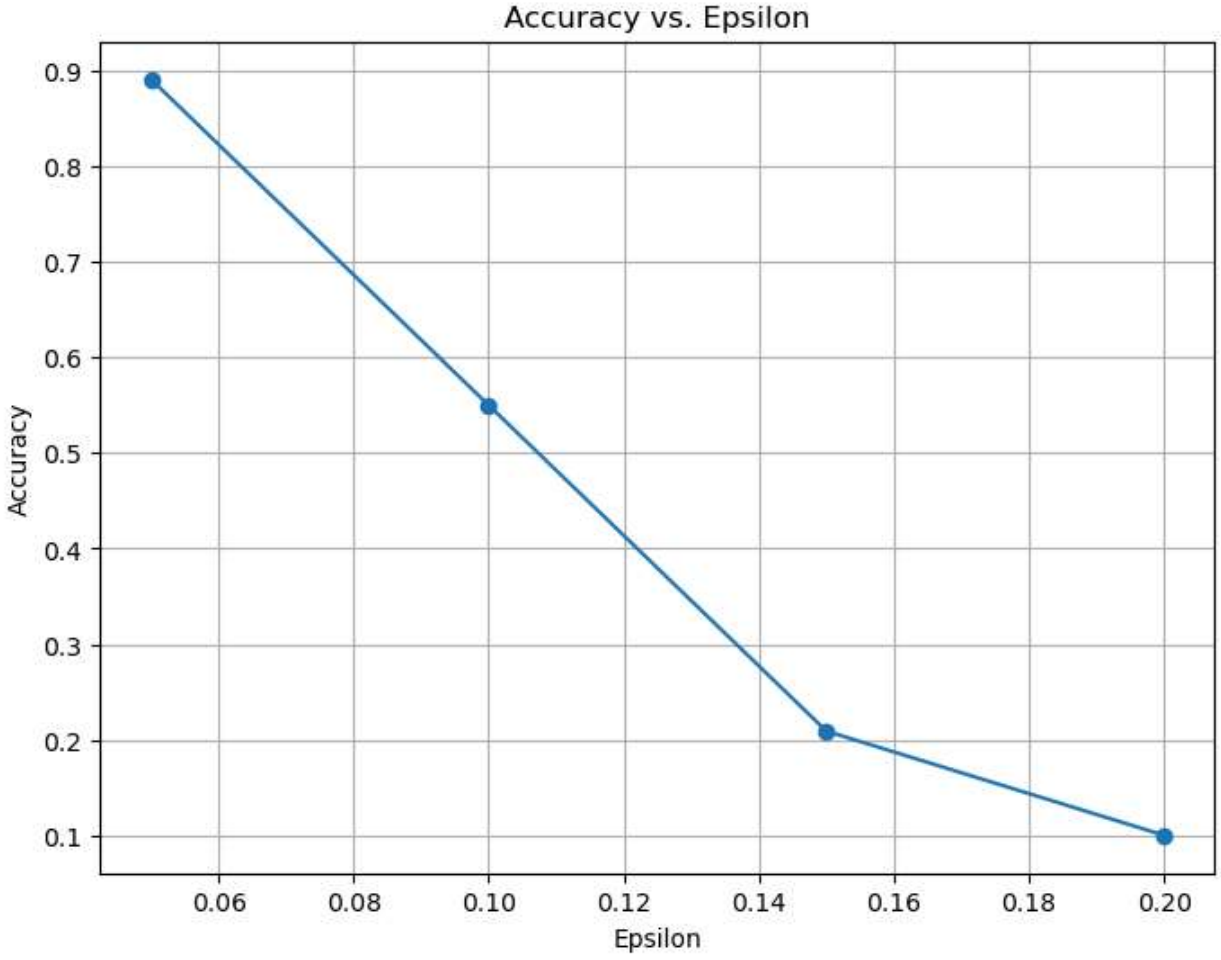
In [41]:
```
def test_with_adversarial_attacks_one(clf, X_test, Y_test, eps):
    clf.set_attack_budget(eps)
    correct = 0
    for i in range(len(X_test)):
        x, y = X_test[i], Y_test[i]
        newX = clf.one_fgsm(x, y, 0)
```

```
        newY = clf.predict(newX)
        if newY == y:
            correct += 1
    accuracy = correct / len(X_test)
    return accuracy
eps_values = [0.05, 0.1, 0.15, 0.2]
accuracies_one = []


for eps in eps_values:
    total_time = 0
    start_time = time.time()
    accuracy_one = test_with_adversarial_attacks_one(clf, X_test, Y_test, eps)
    total_time = time.time() - start_time
    accuracies_one.append(accuracy_one)
    print(f"Epsilon: {eps}, Accuracy: {accuracy_one}, Run Time: {total_time}")

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(eps_values, accuracies_one, marker='o', linestyle='-')
plt.title("Accuracy vs. Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
```

```
Epsilon: 0.05, Accuracy: 0.891, Run Time: 138.97497010231018
Epsilon: 0.1, Accuracy: 0.55, Run Time: 138.34783816337585
Epsilon: 0.15, Accuracy: 0.209, Run Time: 137.88552689552307
Epsilon: 0.2, Accuracy: 0.1, Run Time: 138.53603339195251
```

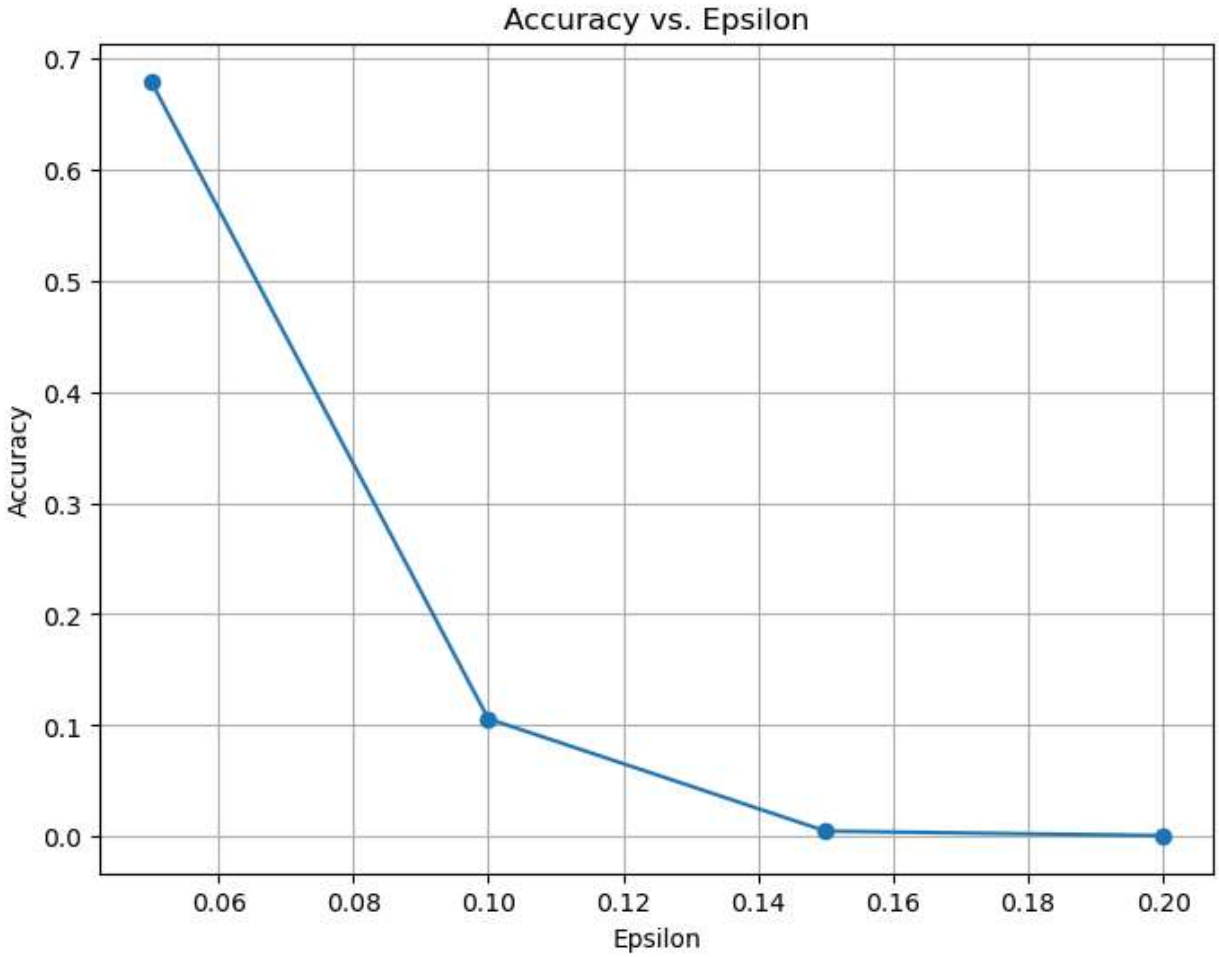


In [69]:
```
def test_with_adversarial_attacks_it(clf, X_test, Y_test, eps):
    clf.set_attack_budget(eps)
    correct = 0
    for i in range(len(X_test)):
        x, y = X_test[i], Y_test[i]
        newX = clf.it_fgsm(x, y)
        newY = clf.predict(newX)
        if newY == y:
            correct += 1
    accuracy = correct / len(X_test)
    return accuracy
eps_values = [0.05, 0.1, 0.15, 0.2]
accuracies_it = []


for eps in eps_values:
    total_time = 0
    start_time = time.time()
    accuracy_it = test_with_adversarial_attacks_it(clf, X_test, Y_test, eps)
    total_time = time.time() - start_time
    accuracies_it.append(accuracy_it)
    print(f"Epsilon: {eps}, Accuracy: {accuracy_it}, Run Time: {total_time}")

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(eps_values, accuracies_it, marker='o', linestyle='-')
plt.title("Accuracy vs. Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
```

```
Epsilon: 0.05, Accuracy: 0.679, Run Time: 640.8126909732819
Epsilon: 0.1, Accuracy: 0.106, Run Time: 639.2908184528351
Epsilon: 0.15, Accuracy: 0.005, Run Time: 636.8892302513123
Epsilon: 0.2, Accuracy: 0.001, Run Time: 642.2958681583405
```



In [63]:

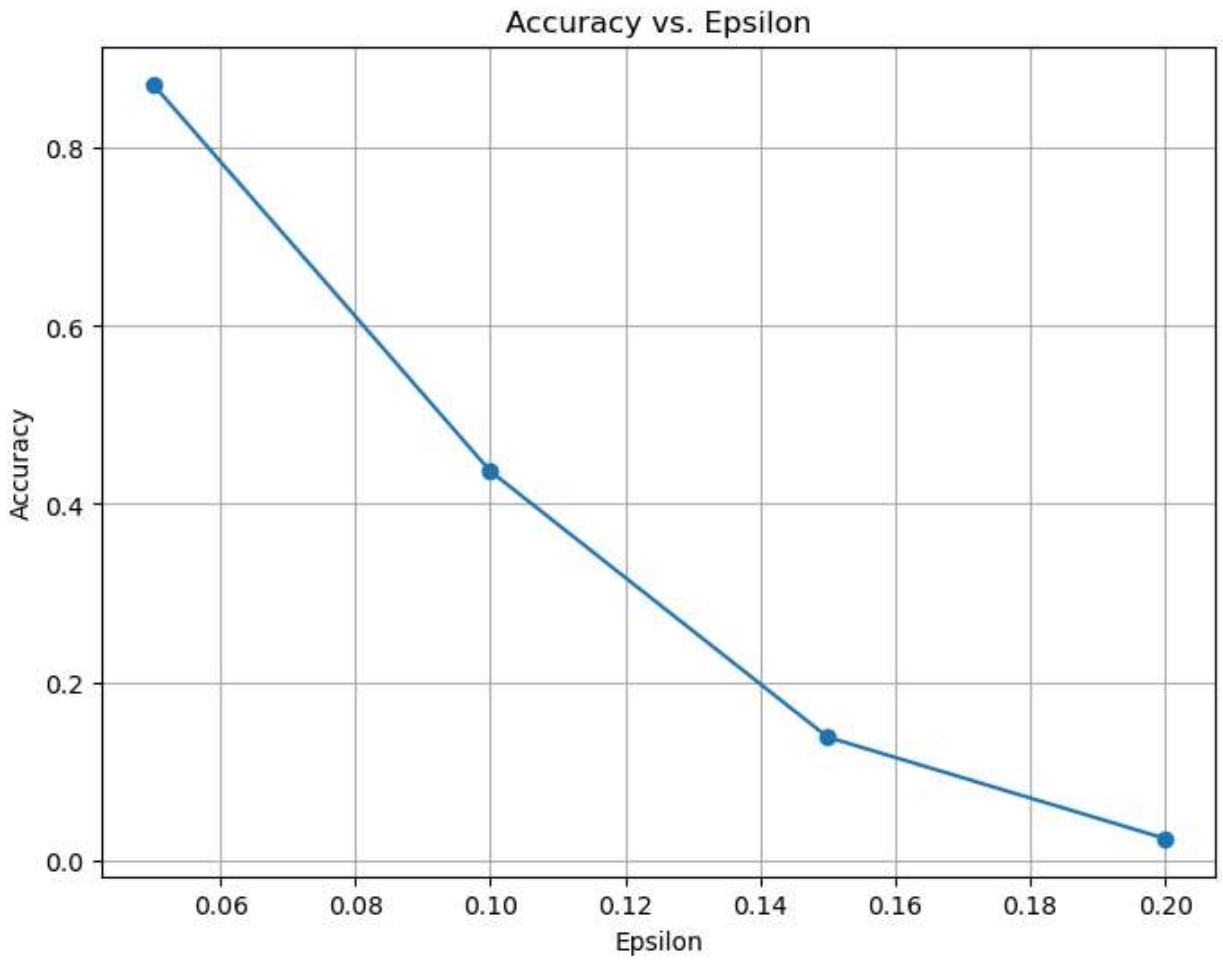
```python
def test_with_adversarial_attacks_cust(clf, X_test, Y_test, eps):
    clf.set_attack_budget(eps)
    correct = 0
    for i in range(len(X_test)):
        x, y = X_test[i], Y_test[i]
        newX = clf.cust_fgsm(x, y)
        newY = clf.predict(newX)
        if newY == y:
            correct += 1
    accuracy = correct / len(X_test)
    return accuracy
eps_values = [0.05, 0.1, 0.15, 0.2]
accuracies_cust = []


for eps in eps_values:
    total_time = 0
    start_time = time.time()
    accuracy_cust = test_with_adversarial_attacks_cust(clf, X_test, Y_test, eps)
    total_time = time.time() - start_time
    accuracies_cust.append(accuracy_cust)
    print(f"Epsilon: {eps}, Accuracy: {accuracy_cust}, Run Time: {total_time}")

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(eps_values, accuracies_cust, marker='o', linestyle='-')
plt.title("Accuracy vs. Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
```

```
plt.grid(True)
plt.show()
```

```
Epsilon: 0.05, Accuracy: 0.872, Run Time: 138.68426489830017
Epsilon: 0.1, Accuracy: 0.437, Run Time: 137.99535965919495
Epsilon: 0.15, Accuracy: 0.138, Run Time: 137.41251683235168
Epsilon: 0.2, Accuracy: 0.024, Run Time: 137.988050699234
```



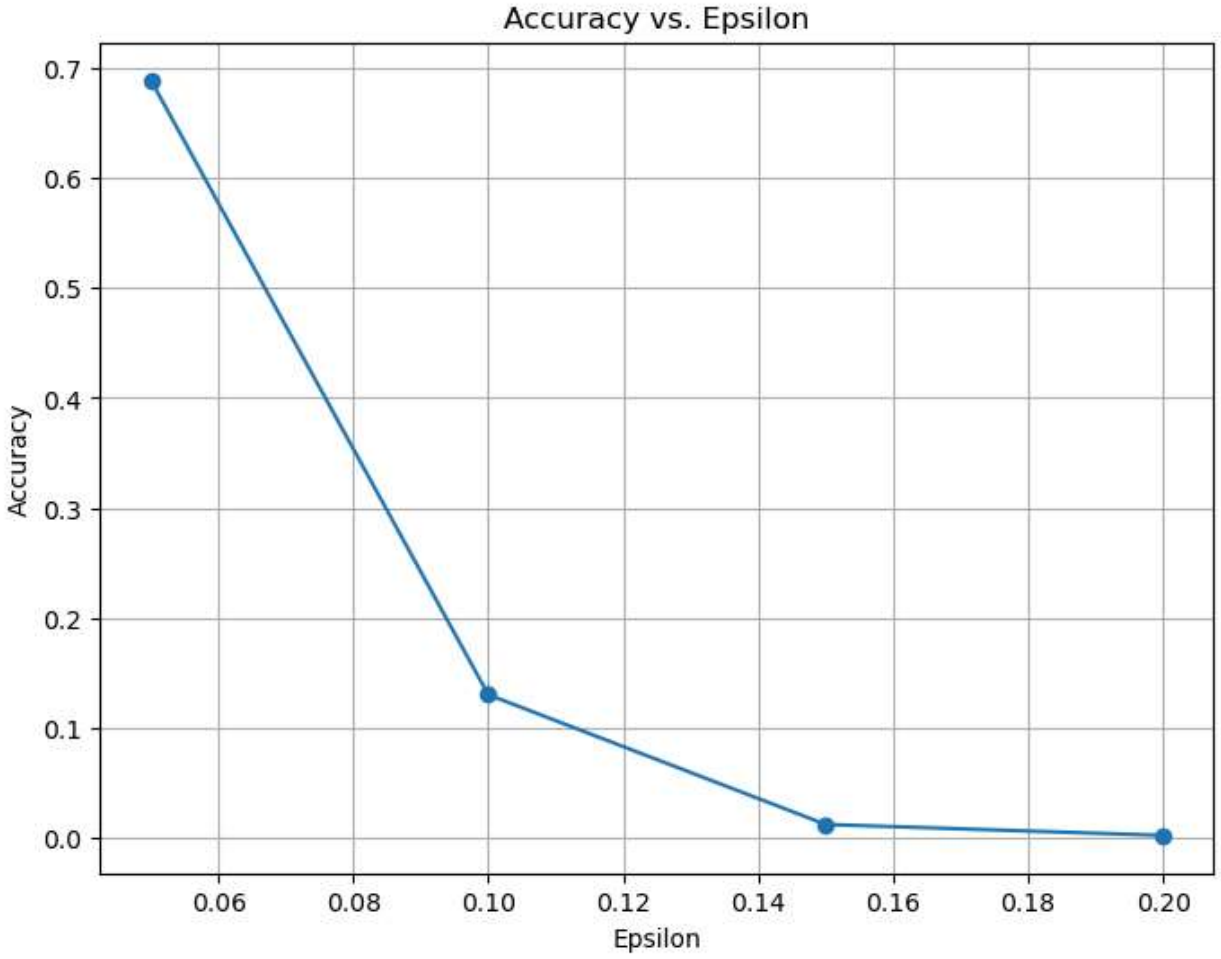
In [76]:

```
def test_with_adversarial_attacks_cust2(clf, X_test, Y_test, eps):
    clf.set_attack_budget(eps)
    correct = 0
    for i in range(len(X_test)):
        x, y = X_test[i], Y_test[i]
        newX = clf.cust_fgsm2(x, y)
        newY = clf.predict(newX)
        if newY == y:
            correct += 1
    accuracy = correct / len(X_test)
    return accuracy
eps_values = [0.05, 0.1, 0.15, 0.2]
accuracies_cust2 = []


for eps in eps_values:
    total_time = 0
    start_time = time.time()
    accuracy_cust2 = test_with_adversarial_attacks_cust2(clf, X_test, Y_test, eps)
    total_time = time.time() - start_time
    accuracies_cust2.append(accuracy_cust2)
    print(f"Epsilon: {eps}, Accuracy: {accuracy_cust2}, Run Time: {total_time}")
```

```
# Plotting
plt.figure(figsize=(8, 6))
plt.plot(eps_values, accuracies_cust2, marker='o', linestyle='-')
plt.title("Accuracy vs. Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
```

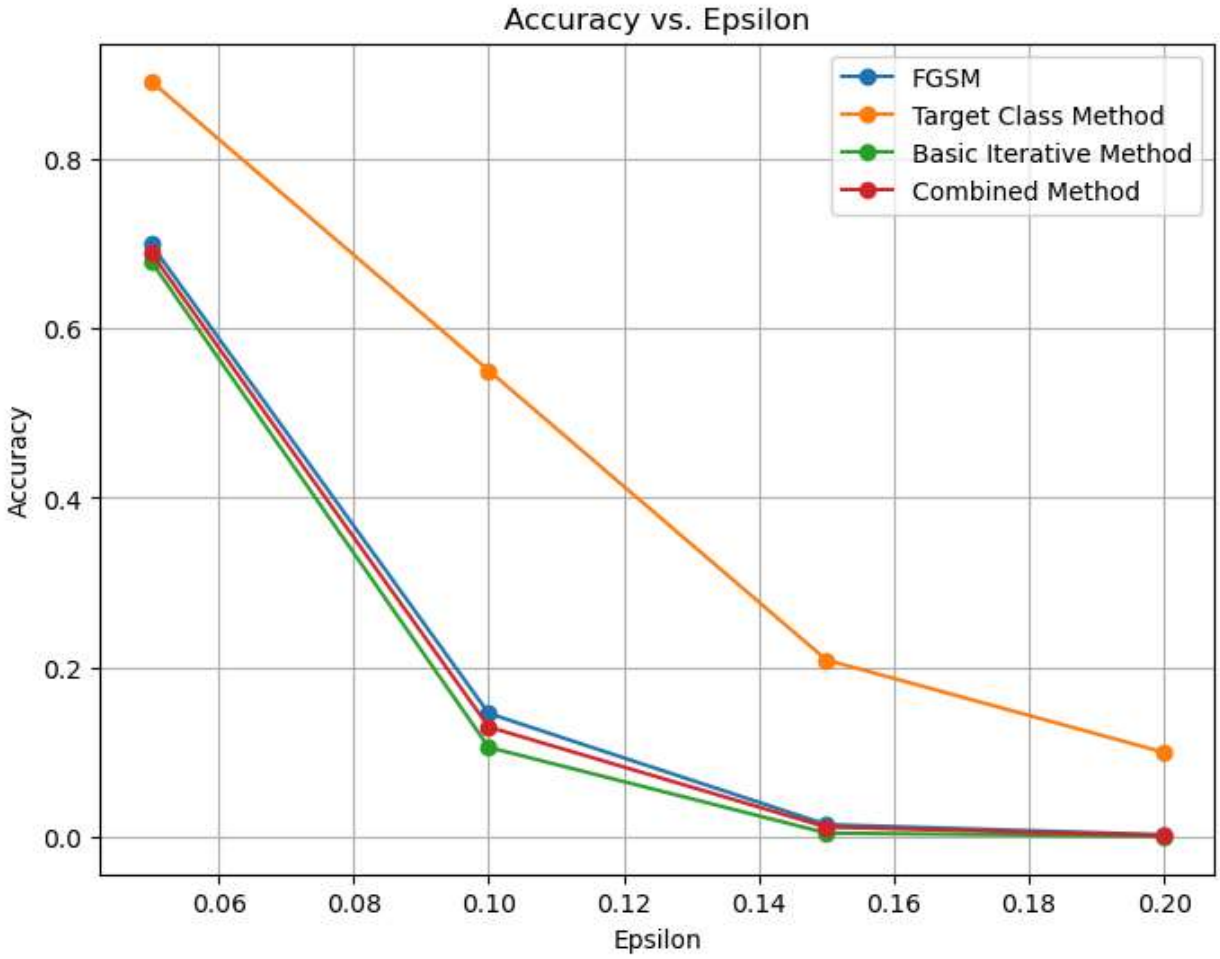
```
Epsilon: 0.05, Accuracy: 0.689, Run Time: 285.512437582016
Epsilon: 0.1, Accuracy: 0.13, Run Time: 286.3854601383209
Epsilon: 0.15, Accuracy: 0.012, Run Time: 284.5097234249115
Epsilon: 0.2, Accuracy: 0.002, Run Time: 285.65510630607605
```



In [78]:
```
plt.figure(figsize=(8, 6))

plt.plot(eps_values, accuracies, marker='o', linestyle='-', label='FGSM')
plt.plot(eps_values, accuracies_one, marker='o', linestyle='-', label='Target Class Me
plt.plot(eps_values, accuracies_it, marker='o', linestyle='-', label='Basic Iterative
plt.plot(eps_values, accuracies_cust2, marker='o', linestyle='-', label='Combined Meth

plt.title("Accuracy vs. Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()
```


Accuracy vs. Epsilon
FGSM
Target Class Method
Basic Iterative Method
Combined Method
Accuracy
0.0
0.2
0.4
0.6
0.8
0.06
0.08
0.10
0.12
0.14
0.16
0.18
0.20
Epsilon

In [ ]: