Adversarial Examples for Neural Network

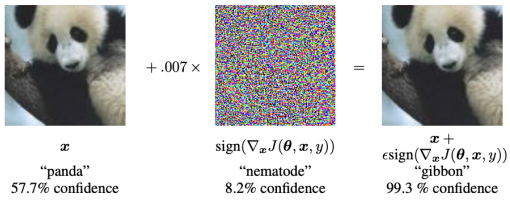
Doheon Kim
dok015@ucsd.edu

Abstract

Neural Networks have achieved remarkable performance across a wide range of applications, from image recognition to natural language processing. However, their susceptibility to adversarial examples—inputs that are designed with subtle perturbations to mislead the network into making incorrect predictions—poses a significant challenge to their security. These adversarial perturbations, often imperceptible to humans, can lead to erroneous outputs.
In this paper, we aim to explore the construction and the evaluation of several Adversarial Examples. We study both Targeted and Non-Targeted Adversary Examples, report the accuracy of each examples for different epsilon values, and critically evaluate whether our newer approach is better compared to the Fast Gradient Sign Method with Non-Targeted Examples.

We are initally given a 3-hidden-layer MLP trained on the MNIST dataset, each row of this file contains a vector representation of a 28 × 28 image, followed by its label (between 0 and 9).

1 Fast Gradient Sign Method



Fast Gradient Sign method is finding a small perturbation of Δx so that neural network f assigns a label different from y to x + Δx. To find the gradient, we want to increase cross-entropy loss of network f at (x,y). We want to take a small step of δ along where loss increases, causing misclassification. The formula is as follows $\tilde{x} = x + \epsilon \cdot \text{sign}(\nabla L(f, x, y))$.

We would first have to compute the gradient of the cross-entropy loss.

$$z^i = h^{i-1}W^i + b^i \quad \text{for } i = 1, 2, 3, 4 \tag{1}$$

$$h^i = \text{ReLU}(z^i) \quad \text{for } i = 1, 2, 3 \tag{2}$$

$$p = \text{Softmax}(z^4) \tag{3}$$

This will be the initialization of the Multi-layer perceptron, where RELU activates the input function, and softmax is used to predict the probability of the class. Always used to initialize forward function, which finds the output or the probability of the class using forward propagation.

$$\begin{aligned}\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h^0} &= \frac{\partial L}{\partial z^4}\frac{\partial z^4}{\partial h^3}\prod_{i=1}^{3}\left(\frac{\partial h^i}{\partial z^i}\frac{\partial z^i}{\partial h^{i-1}}\right) \\ &= \frac{\partial L}{\partial z^4}\frac{\partial z^4}{\partial h^3}\prod_{i=1}^{3}\left(\frac{\partial h^i}{\partial z^i}\frac{\partial z^i}{\partial h^{i-1}}\right)\end{aligned} \tag{4}$$

The intermediate terms can be computed as follows.

$$\frac{\partial L}{\partial z^4} = p - y$$

$$\frac{\partial z^i}{\partial h^{i-1}} = (W^i)^T$$

$$\frac{\partial h^i}{\partial z^i} = \text{diag}(1(h^i > 0))$$

In doing this we can print out the images with different epsilon values.

```
self.forward(x)

batch_size = x.shape[0]
one_hot = np.zeros_like(self.p)
one_hot[y] = 1
dl_dz4 = (self.p - one_hot)/batch_size
dz_4_dh3 = self.W4.T
dz_3_dh2 = self.W3.T
dz_2_dh1 = self.W2.T
dz_1_dh0 = self.W1.T

dh3_dz3 = np.diag(self.h3 > 0).astype(int)
dh2_dz2 = np.diag(self.h2 > 0).astype(int)
dh1_dz1 = np.diag(self.h1 > 0).astype(int)

grad = np.matmul(dl_dz4,dz_4_dh3)
#grad = (grad * dh3_dz3)
grad = np.matmul(grad,dh3_dz3)
grad = np.matmul(grad, dz_3_dh2)
#grad = (grad * dh2_dz2)
grad = np.matmul(grad,dh2_dz2)
grad = np.matmul(grad,dz_2_dh1)
#grad = (grad * dh1_dz1)
grad = np.matmul(grad,dh1_dz1)
gradient = np.matmul(grad,dz_1_dh0)


return gradient
```

Figure 1: Gradient code

```
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.
    '''

    ######################################
    #
    # TODO
    #
    ######################################


    #prediction = self.predict(grad_x)
    #prediction = self.predict(x)
    grad_x = self.gradient(x,y)

    x_att = x+ self.eps * np.sign(grad_x)
    #x_att = np.clip(x_att,0,1)

    return x_att
```

Figure 2: FSGM Attack code

```
def display_adversarial_images(n, sab):
    clf.set_attack_budget(sab)
    plt.figure(figsize=(n * 2, 2))
    for i in range(n):
        x, y = X_test[i], Y_test[i]
        x_tatt = clf.attack(x, y,sab)  # Generate adversarial example
        x_tatt = np.clip(x_tatt,0,1)
        pixels = x_tatt.reshape((28, 28))
        plt.subplot(1, n, i + 1)  # 1 row, n columns, ith subplot
        plt.title(f"ε={sab}")
        plt.imshow(pixels, cmap="gray")
        plt.axis('off')  # Hide the axes

    plt.show()

# Display n adversarial images
for sab in [0.1, 0.12, 0.14, 0.15, 0.21]:
    display_target_adversarial_images(5, sab)
```

Figure 3: FSGM images

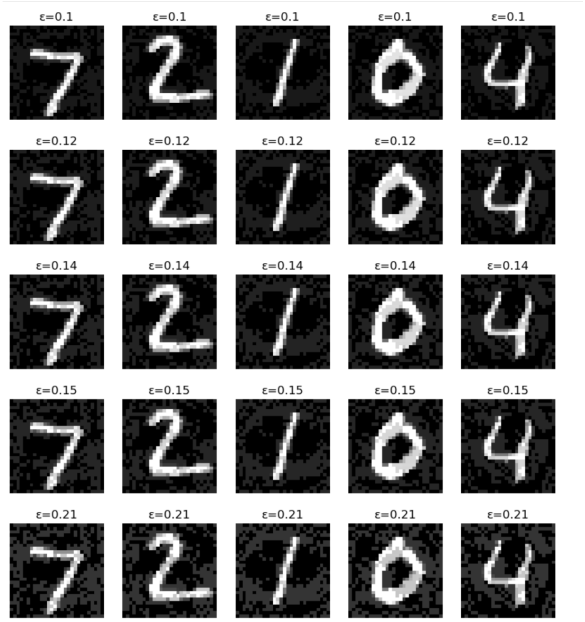


Figure 4: FSGM Images

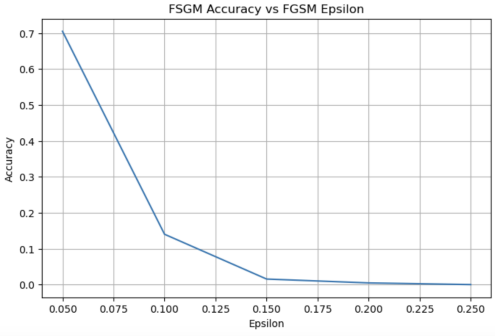


Figure 5: FSGM Accuracy vs Epsilon

2 Fast Gradient Method with Targeted Examples

Fast Gradient method is quite similar to the FSGM Method described above, however it differs in that Fast Gradient considers the Magnitude of the gradient, whereas FSGM only cares about the sign of the gradient, like a binary classifier.

$\tilde{x} = x - \epsilon \cdot (\nabla L(f, x, y))$.

This is the formula for Fast Gradient Method. The attack method also aims to have a specific target class as a method, and to modify the original input in a way that model misclassifies it as a particular chosen class. Therefore, in a FSGM Method, an objective is to maximize the loss function so that it converges faster, but FG with targeted examples have specific $target_y$ that wants to minimize the loss function.

```python
def fg_target_attack(self, x,y, epsilon):
    # Calculate gradient approximation
    fg_grads = self.fg_target_gradient(x, y)

    # Create adversarial example by subtracting epsilon times the sign of the gradient
    x_tatt = x - epsilon * np.sign(fg_grads)

    # Clip the adversarial example to ensure pixel values remain valid
    x_tatt = np.clip(x_tatt, 0, 1)

    return x_tatt
```

Figure 6: Fast Gradient Target

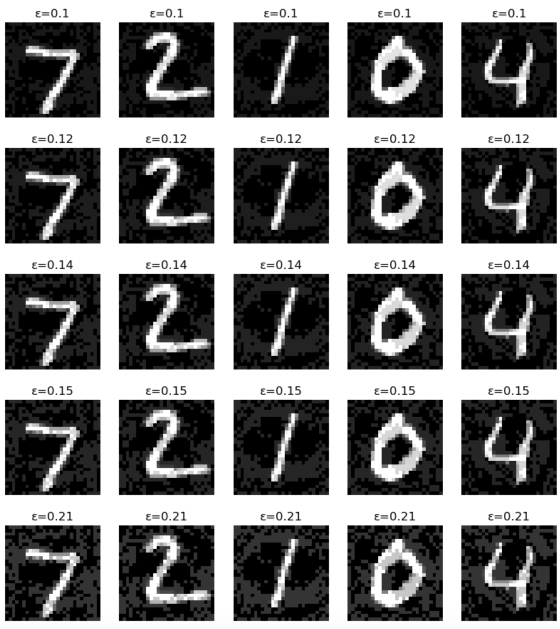


Figure 7: Fast Gradient Images

We could see that the Fast Gradient Images and the code is almost similar to the FSGM, except for the fact that it tries to minimize the loss function, and not so much accuracy drops with increasing epsilons.

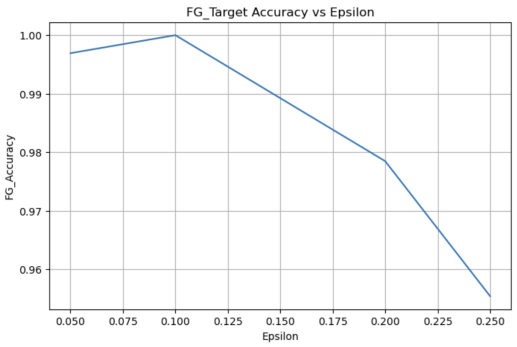


Figure 8: FG Accuracy vs Epsilon

3 Projected Gradient Descent Method

Projected Gradient Descent Method is an iterative approach to the FSGM Method.

1. Starts with an input of images
2. Apply small perturbation like FSGM or FG with targeted Examples
3. At each Iteration, take a step in direction of gradient of loss function respect to input.
4. Project the input changed δx to be constrained space satisfying constraints.
5. Repeat Iteration and Projection until convergence.

$$x^{(t+1)} = \Pi_{x+\epsilon}\left(x^{(t)} + \alpha \cdot \text{sign}(\nabla_x J(\theta, x^{(t)}, y))\right)$$

This is the formula for Projected Gradient Descent.

```python
pgd_attack(self,x, y_target, epsilon, alpha, num_iterations):

itialize perturbed image as a copy of the original
x_adv = np.copy(x)

for _ in range(num_iterations):

    grad = self.fg_target_gradient(x_adv, y_target)

    x_adv -= alpha * np.sign(grad)

    # Project the perturbed image back into the epsilon-ball of the original image
    x_adv = np.clip(x_adv, x- epsilon, x + epsilon)

    # Ensure the perturbed image's pixel values are still valid
    x_adv = np.clip(x_adv, 0, 1)

return x_adv
```

Figure 9: PGD code

We could see higher drops for PGD as it increases in epsilon values compared to the FG method that we implemented Earlier. This is likely due to the fact that our PGD is better, has more number of iterations, and clips or projects the perturbation into the original function.

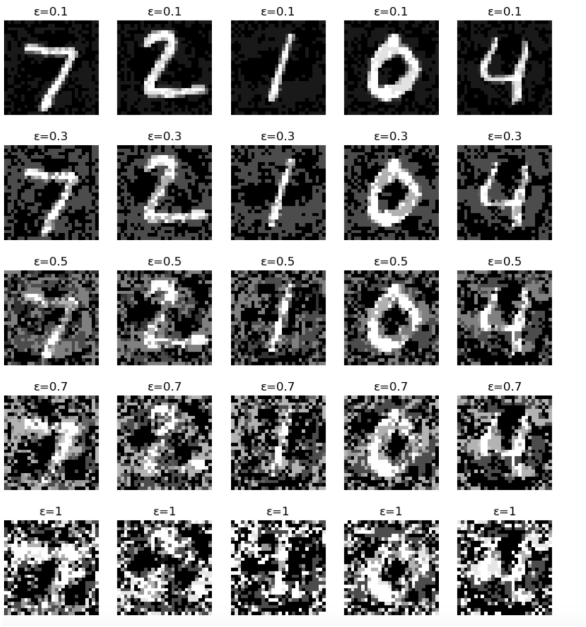


Figure 10: PGD image

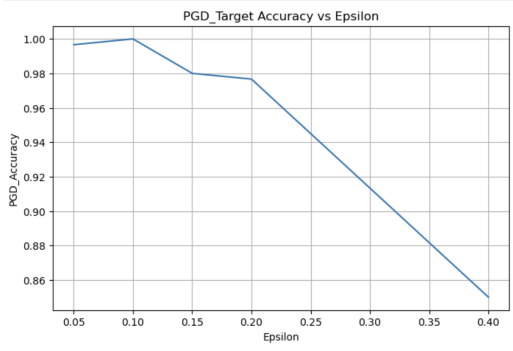


Figure 11: PGD Accuracy vs Epsilon

4 Critical Evaluation and Conclusion

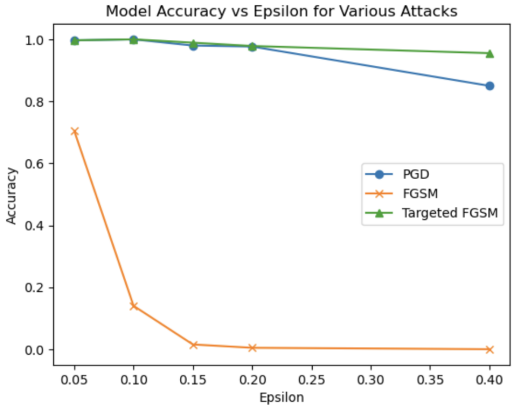


Figure 12: Comparing Three Examples

Comparing the 3 graphs and getting the results in a graph of Accuracy vs Epsilon, we could see that the Fast Gradient Sign method would have the highest drop in accuracy as epsilon increases. Between the other two attack methods, Projected Gradient Descent and Targeted Fast Gradient Method, it seems that Projected gradient descent method has lower accuracy with higher epsilon values. Experimentally, it proves that my method in Fast Gradient with Targeted examples and Projected Gradient Descent is in contrast to my hypothesis, such that these newer method should be having higher accuracy drop when epsilon increases compared to Fast Sign Gradient Method leading to better attack effectiveness.

This could be due to several reasons, such as the hyperparameters we fine-tuned in our FG and PGD method. We had to implement targeted y and minimize the loss function and change the step size alpha with more iterations. The Multilayer Perceptron model that we were given could be showing stronger resilience to our attack method compared to the FSGM Method. The model could be less sensitive to the attack methods that I have described. In addition to this, the distribution of perturbations might have been insignificant such that it affected the accuracies of our FG and Projected Gradient Descent Method compared to the simple FSGM Method with gradient calculation. It could further be concluded that better epsilon values, step sizes and experimenting with different iteration values, and using error bars to represent the change in accuracy drop could be meaningful for this paper.

5 Citations

[1] Medium Article on tricking-neural-networks-create-your-own-adversarial-examples-a61eb7620fd8, Daniel Geng and Rishi Veerapaneni
[2] DELVING INTO TRANSFERABLE ADVERSARIAL EXAMPLES AND BLACK-BOX ATTACKS, Yanpei Liu, Xinyun Chen, Chang Liu, Dawn Song
[3] Wikipedia.org/wiki/Adversarial Examples
[4] EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES, Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy
[5] Maximal Jacobian-based Saliency Map Attack, Rey Wiyatno and Anqi Xu