Various FGSM Methods for Adversarial Attack

김원진(Kim Wonjin)
Department of Computer Science & Engineering
wjkim9653@khu.ac.kr

한지훈(Han Jihoon)
Department of Software Convergence
jhhan0208@khu.ac.kr

0. Index

1. Introduction

1.1 Project overview

The task is to make the prediction of an adversarial example different from its original label but the same as a certain pre-defined label. What is noticeable is that this is requiring more than normal adversarial attacks, in which you only have to make the prediction different from the original label.

For the sake of convenience and ease of understanding, in this report we choose to refer to the process of (various) FGSM attacks and their respective delta training process as a whole, as 'MODEL'.

1.2 Code understanding

The code is introducing the Iterative-Fast Gradient Sign Method(I-FGSM). This method applies FGSM many times with small perturbation size instead of applying adversarial noise with one large perturbation size.

$$X_0^{adv} = X,$$
$$X_{N+1}^{adv} = Clip_{X,\epsilon}\{X_N^{adv} - \alpha \text{sign}(\nabla_X J(X_N^{adv}, y_t))\}$$

The accuracy of the code, which indicates how many predicted labels are the same as the pre-defined labels, was 0.00375(0.375%). The accuracy was much lower than we expected. We planned various ways to increase the accuracy.

```
accuracy_score(df['label'], df['pred'])
```

```
0.00375
```

2. Improving existing model(I-FGSM)

2.1 Manually tuning parameters

We checked the accuracy of the model over many iterations, changing the values of some parameters, including the values of “epsilon”, “alpha”, and “max_iterations”. Decreasing the value of “max_iterations” 100 to 7, 14, and such helped us get many accuracy results in the early stage.

2.2 Adam optimizer

Instead of using a fixed learning rate, we adjusted the learning rate during training using the Adam optimizer. (We used it in the initial stage, but we decided not to use the later stages because we thought it was unnecessary.)

```
delta_optimizer = optim.Adam([delta], lr=adam_lr)
```

2.3 Define Non-target accuracy

In addition to the accuracy based on whether 'pred' == 'label', we wanted to calculate another type of accuracy metric which will tell us whether the adversarial attack was successful or not. In other words, the metric is based on whether 'pred' != 'origin' because the attack is successful if the predicted label is different from the ground-truth label.

The default I-FGSM model that was provided showed a 42% success rate in adversarial attack.

```
Target Score: 0.375%
No Target Score: 41.875%
```

3. Applying new model concepts

3.1 Momentum Iterative FGSM(MI-FGSM)

3.1.1 Basics

Momentum Iterative FGSM is basically I-FGSM with momentum. The momentum helps to prevent poor local minima or maxima.

$$g_{t+1} = \mu \cdot g_t + \frac{J(x_t^*, y^*)}{\|\nabla_x J(x_t^*, y^*)\|_1}.$$

The gradient is calculated in the way above, with the μ as weight decay.

3.1.2 Code

```
momentum = torch.zeros_like(X_ori).to(device)
decay = 0.9
momentum = decay * momentum + grad_c / torch.norm(grad_c, p=1)
delta.data = delta.data - alpha * torch.sign(momentum)
```

To implement the momentum, we defined a zero tensor 'momentum' and calculated the momentum value with weight decay = 0.9. Then we used the momentum gradient to decrease the loss.

3.1.3 Results

```
Target Score: 1.375%
No Target Score: 78.875%
```

Compared to the defalut I-FGSM model, both accuracy metrics showed an increase. Especially, the 'No Target Score" increased from 42% to 79%, significantly increasing the success rate of an adverserial attack.

3.2 Diverse input patterns FGSM(DI-FGSM)

3.2.1 Basics

Diverse input patterns FGSM transforms input images with probability p. Transformations include rotation, flipping, random resizing, and more.

$$X_{n+1}^{\text{adv}} = \text{Clip}_X^{\epsilon}\{X_n^{adv} + \alpha \cdot \text{sign}\left(\nabla_X L(T(X_n^{adv}; p), y^{\text{true}}; \theta)\right)\}$$

The transformations help to increase the transferability of adversarial examples and the robustness of a model.

3.2.2 Code

```
def add_input_diversity(X_ori, diversity_factor=0.02):
  noise = torch.randn_like(X_ori).to(device) * diversity_factor
  X_ori = X_ori + noise
  X_ori = torch.clamp(X_ori, 0, 1)
  return X_ori

X_ori = add_input_diversity(X_ori)
```

To apply the transformation, we defined the function 'add_input_diversity" which adds random noise to the image, adding diversity to it.

3.2.3 Results

```
Target Score: 1.375%
No Target Score: 80.625%
```

Compared to MI-FGSM, there was a slight increase in the "No Target Score", but the "Target Score" was the same as MI-FGSM.

4. Hyperparameter tuning

This time, instead of focusing on the model itself, we tried to find the optimal value of the hyperparameters. Among the hyperparameters, we focused on "max_iterations", "learning rate", and "epsilon". We tried 5*6*7=210 different combinations with the values bellow.

```
max_iterations = [5, 25, 50, 100, 500]
lrs = [2/255, 5/255, 10/255, 25/255, 50/255, 100/255]
epsilons = [16, 32, 48, 64, 80, 96, 112]
```

The sota performance was Target Score: 1.13%

and No-Target Score: 85.19%, with the values of the three parameters being 50, 10/255, and 80, respectively.

```
##################################################
 Iter: 50 | LR: 0.03922 | Eps: 80
++++++++++++++++++++++++++++++++++++++++!SOTA PERFORMANCE!
Target Score: 1.1347087378640777
No-Target Score: 85.19417475728154
```

5. Applying fundamental changes to the code

5.1 No normalization

From the professor we heard that in deep learning, unlike machine learning, normalization is not essential. Rather, it may be over confusing and even lead to undesired results. So we tried to adjust the code so that we can change the images itself, with no normalization.

5.2 Not using DI-FGSM

We went back to using MI-FGSM instead of DI-MI-FGSM. This is because using it only decreased the accuracy in this case.

```
| Iter: 20 | LR: 0.01569 | Eps: 0.18823529411764706 |
++++++++++++++++++++++++++++++++++++++++VGG16 TEST SOTA
PERFORMANCE++++++++++++++++++++++++++++++++++++++++++++
With-Target Score (VGG16): 3.875
No-Target Score (VGG16): 96.0
With-Target Score (InceptionV3): 1.0
No-Target Score (InceptionV3): 79.125
With-Target Score (ResNet): 100.0
No-Target Score (ResNet): 100.0
```

For the VGG16 model, The target score increased about three times from 1.375% to 3.875%, and the no-target score is 96%, close to 100%. We think that getting rid of the normalization played a big role in this performance.

5.3 Not using learning rate independently

In another implementation of FGSM and I-FGSM, we noticed that instead of using alpha and learning rate independently, alpha is defined as a smaller step, in which applying alpha 'iteration' times equals to epsilon. So we defined alpha as "alpha = epsilon/iterations".

5.4 Increase number of iterations

After running the iterations, this time we were able to reach almost 100% for the Non target Acc. But we noticed that 100% itself isn't that significant because when epsilon becomes too big, the augmented images become noticeable to the human eye. For example if epsilon = 64/255 then pixels can change between −64 ~ +64. So we decided to make epsilon small, but increase the number of iterations instead to make a more delicate augmentation.

```
==================================================
| Iter: 50 | Eps: 0.18823529411764706 |
Targeted Attack Score (VGG16): 6.125 % <----------
Untargeted Attack Score (VGG16): 91.375 %
Targeted Attack Score (InceptionV3): 1.5 %
Untargeted Attack Score (InceptionV3): 65.875 %
Targeted Attack Score (ResNet): 100.0 %
Untargeted Attack Score (ResNet): 100.0 %
```

The non-target score was similar, but the target score increased again from 3.875% to 6.125%. It is quite improving, but there is something more important that we found.

No matter how many iterations we tried(100, 200), the sota performance was at 50 iterations. This means that there is a sweet spot for the number of iterations. This is because when the number of iterations gets too large, the step size alpha gets too small, which becomes meaningless because pixels are integer type.

6. Focusing on DI-FGSM

6.1 Reimplementing “add_input_diversity()”

We found that there was a flaw in the function “add_input_diversity()” that we defined in 3.2.2, and there wasn’t a significant increase in performance. So we redefined the function as:

```python
def add_input_diversity(X_ori, resize_rate=0.9, diversity_factor=0.5):
    if torch.rand(1) < diversity_factor:
        return X_ori

    img_size = X_ori.shape[-1]
    img_resize = int(img_size * resize_rate)

    if resize_rate < 1:
        img_size = img_resize
        img_resize = X_ori.shape[-1]

    rnd = torch.randint(low=img_size, high=img_resize, size=(1,), dtype=torch.int32)
    rescaled = F.interpolate(X_ori, size=[rnd, rnd], mode='nearest')

    h_rem = img_resize - rnd
    w_rem = img_resize - rnd

    pad_top = torch.randint(low=0, high=h_rem.item(), size=(1,), dtype=torch.int32)
    pad_bottom = h_rem - pad_top
    pad_left = torch.randint(low=0, high=w_rem.item(), size=(1,), dtype=torch.int32)
    pad_right = w_rem - pad_left

    pad = [pad_left.item(), pad_right.item(), pad_top.item(), pad_bottom.item()]
    padded = F.pad(rescaled, pad=pad, mode='constant', value=0)

    return padded
```

6.2 Compare DI, TI, NI

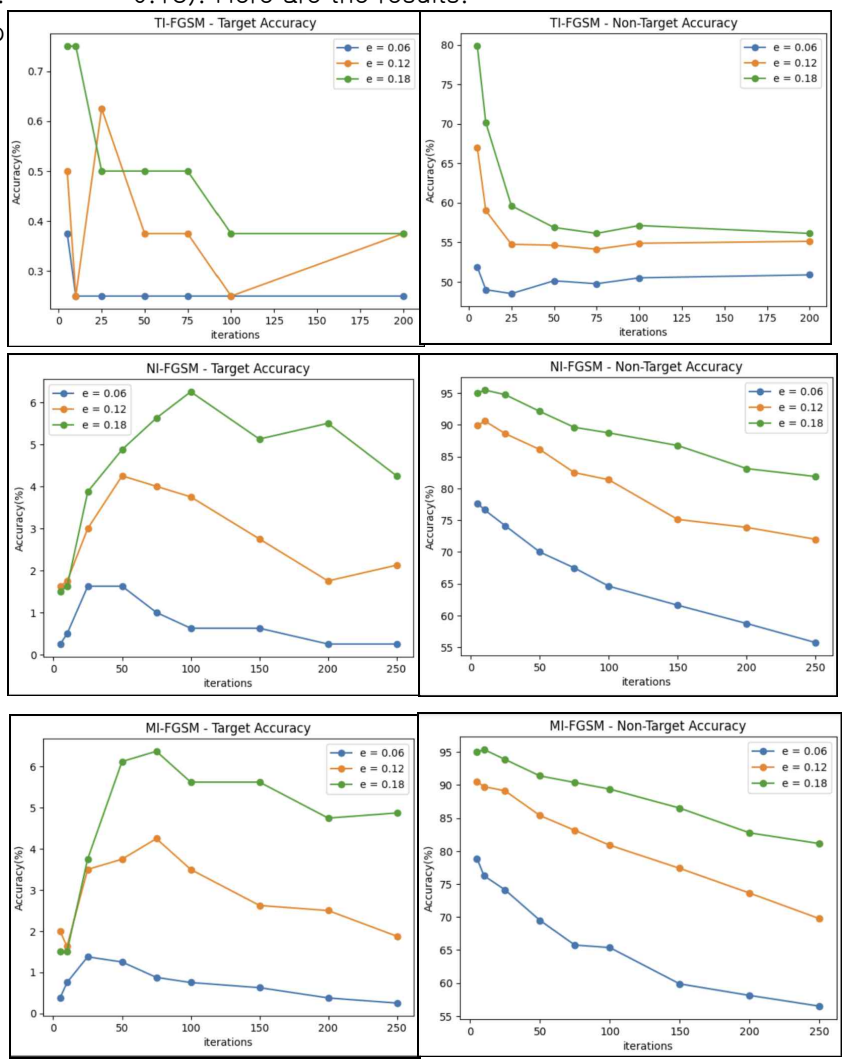
With the baseline method defined in "**5. Applying fundamental changes to the code**", we decided to add three FGSM methods: DI-FGSM, TI-FGSM, and the new NI-FGSM (Nesterov Iterative FGSM). We tried various 'max_iterations' ranging from 5 to 800. The results of the three models are:

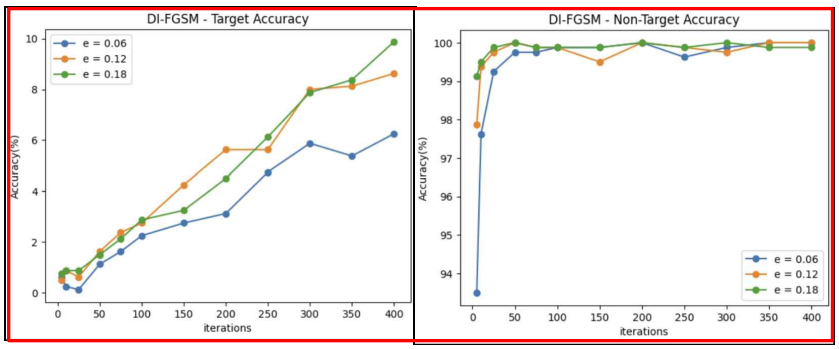
```
<Baseline + DI>
================================================
| Iter: 200 | Eps: 0.12549019607843137 |
Targeted Attack Score (VGG16): 5.375 % <---------
Untargeted Attack Score (VGG16): 99.75 %
Targeted Attack Score (InceptionV3): 0.625 %
Untargeted Attack Score (InceptionV3): 100.0 %
Targeted Attack Score (ResNet): 82.75 %
Untargeted Attack Score (ResNet): 100.0 %


<Baseline + NI>
================================================
| Iter: 100 | Eps: 0.18823529411764706 |
Targeted Attack Score (VGG16): 6.25 % <---------
Untargeted Attack Score (VGG16): 88.75 %
Targeted Attack Score (InceptionV3): 0.875000000
Untargeted Attack Score (InceptionV3): 54.125 %
Targeted Attack Score (ResNet): 100.0 %
Untargeted Attack Score (ResNet): 100.0 %


<Baseline + TI>
================================================
| Iter: 5 | Eps: 0.18823529411764706 |
Targeted Attack Score (VGG16): 0.75 % <---------
Untargeted Attack Score (VGG16): 79.875 %
Targeted Attack Score (InceptionV3): 0.125 %
Untargeted Attack Score (InceptionV3): 60.124999
Targeted Attack Score (ResNet): 91.875 %
Untargeted Attack Score (ResNet): 99.25 %
```

Fundamentally, the three models (DI, NI, TI) are all similar in the sense that it tries to improve the poor transferability of the input image. Therefore, we assumed that we made a mistake in implementing TI-FGSM because its performance is a lot lower compared to the other two. Comparing DI-FGSM and NI-FGSM, even though the highest value of 'max_iterations' was 800, NI-FGSM reached its highest performance at 100 iterations and started to decrease. On the other hand, DI-FGSM showed a linear increase in accuracy. We thought that it would be meaningful to see how the accuracy of DI-FGSM changes as we increase the number of iterations, and we did just that.

6.3 Plotting change of acc according to iters

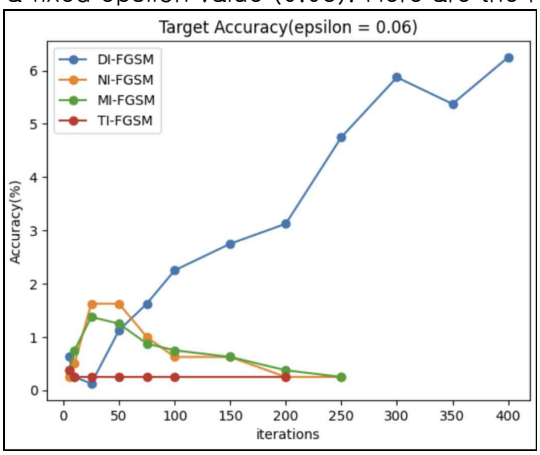
First, we plotted the target accuracy and the non-target accuracy for the four models(DI-FGSM, TI-FGSM, NI-FGSM, MI-FGSM) respectively according to the three epsilon values(0.06, 0.12, 0.18). Here are the results.

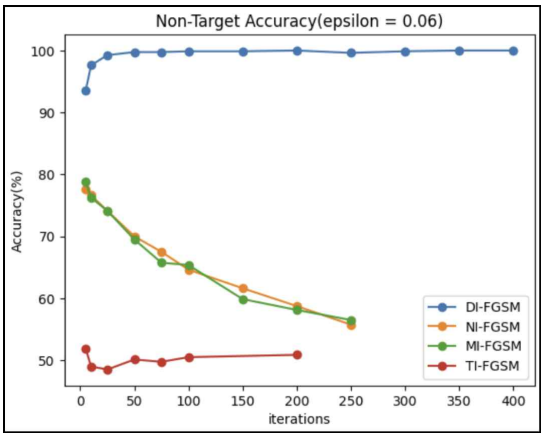




Compared to the other three models, DI-FGSM showed the best performance on target accuracy (sota: 9.875%) while preserving the non-target accuracy close to 100%.

Next, we plotted the target accuracy and the non-target accuracy for the four models combined with a fixed epsilon value (0.06). Here are the results.

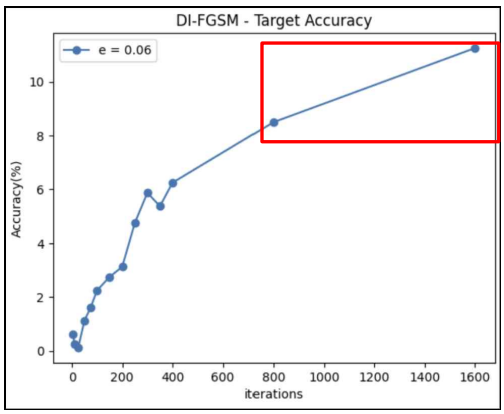


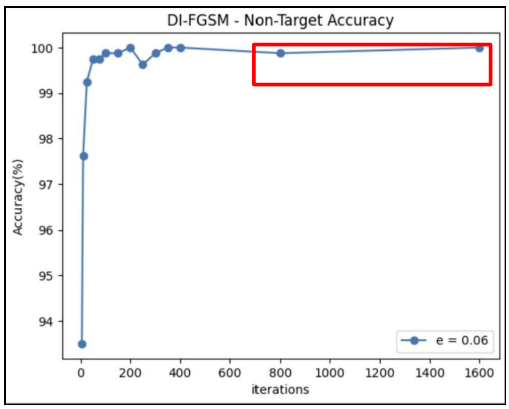


Again, DI-FGSM shows a significant advantage when trained with the same number of iterations, building substantial deficits over other models in terms of accuracy and retaining its linear accuracy increase. With this trend(and time constraints) in mind, we decided to train only the DI-FGSM model over 250 iterations.

7. Final model, conclusion of results

First, we saw that in DI-FGSM, the accuracy continuously grows as the number of iterations increases. So we increased the number of iterations in DI-FGSM greatly to 800 and 1600. (Due to the lack of time, we only tested using the smallest epsilon value of 0.06). Here are the results.



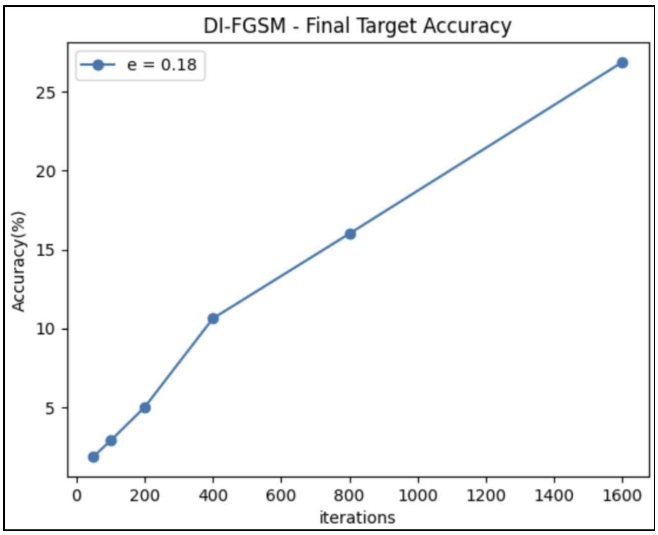


According to the graph, the target accuracy increases even more as the number of iterations increases (sota: 11.25%), and the non-target accuracy stays close to 100% as well. Thus, we concluded that it is plausible to derive our final model in this way. Based on what we did before, we trained delta with a high epsilon value and many iterations because these two factors played a big role in increasing the target accuracy. We wanted to do even more than 1600 iterations, but we couldn't do it due to the lack of time.

Also, we are aware of the fact that increasing the value of epsilon too much causes the noise to be noticeable by the naked eye. After trying various epsilon values from 8~48, we concluded that epsilons within this range can make the noise unnoticeable enough. We could get higher scores by using even higher epsilon values, but the noise would be noticeable, which isn't intended in this project. Therefore, we decided to only use epsilon values no bigger than 48/255.

After some final H.P tuning, we finally chose our final model by fixing the parameters in our DI-FGSM model.

[Final model]
Method: DI-FGSM
Num_of_iterations: 1600
Epsilon: 0.18



The targeted attack accuracy for the final model was 26.875%. Also, the non-target accuracy hovered between 99~100% during all iterations.

The submission file 'submission.csv' was filled based on this model.

Also We figured that it would be wotrh mentioning that we migrated the code bases for this project into several python files so that we could deploy and carry out the training process on Docker-based GPU Server.
Since we are only submitting the final DI-FGSM Attack related code, we decided to create a GITHUB repository storing all relevant files and codes(including the log files and models that we ended up not using) used in this project.
https://github.com/wjcldply/FGSM-Attacks-with-PyTorch

References

[1] FGSM
https://arxiv.org/abs/1412.6572

[2] I-FGSM
https://arxiv.org/abs/1607.02533

[3] MI-FGSM
https://arxiv.org/pdf/1710.06081.pdf

[4] DI-FGSM
https://arxiv.org/pdf/1803.06978.pdf

[5] NI-FGSM
https://arxiv.org/abs/1908.06281

[6] TI-FGSM
https://arxiv.org/abs/1904.02884