Adversarial Attack Using DI-FGSM MI-FGSM

KIM SEONGMIN
2021105575
Department of Computer Science
KyungHee University

Abstract

Several methods were attempted to reduce the accuracy of the model by increasing the acuity_score (attack success rate) by using the base code, and as a result, DI-FGSM and MI-FGSM were combined. After selecting the method, hyperparameter tuning was performed, and the success rate was up to 0.5475 (about 54%) when epsilon 32/255 was performed through about 1500 trials (about 200 times excluding duplicates). However, considering that epsilon 32/255 is quite large, and epsilon of the base code is 16/255, 0.36 (36%) was finally submitted as the optimal number when epsilon was 16/255. The reason why epsilon was judged important and decided will be described in detail later.

1 Introduction to Adversarial Attacks

Adversarial attacks exploit vulnerabilities in neural networks by applying small, carefully crafted perturbations to input data, which result in significant misclassifications. Despite being imperceptible to humans, these perturbations effectively deceive models, highlighting inherent blind spots and limitations in their training processes.

Key Findings

- Adversarial examples are universal and can transfer across different models and datasets, indicating systemic vulnerabilities rather than isolated flaws [3].
- These attacks can be generated efficiently using methods such as the Fast Gradient Sign Method (FGSM), which computes perturbations aligned with the gradient of the loss function [2].
- Training models with adversarial examples can improve robustness but often requires significant computational resources and careful parameter tuning [3].

Adversarial training and model design improvements are promising approaches to mitigate these vulnerabilities. However, the existence of such examples underscores the need for further research into robust and interpretable machine learning models.

2 Select a method

2.1 FGSM

Fast Gradient Sign Method (FGSM)

The Fast Gradient Sign Method (FGSM), introduced by **(author?)** [2], is a simple and efficient approach for generating adversarial examples. FGSM works by leveraging the gradient of the loss function with respect to the input data to construct a perturbation that maximizes the model's error.

Given a neural network with parameters θ, input x, target label y, and loss function $J(\theta, x, y)$, the adversarial example x_{adv} is generated as follows:

$$x_{\text{adv}} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y)),$$

where ϵ is a small scalar controlling the magnitude of the perturbation, and $\text{sign}(\cdot)$ denotes the element-wise sign function.

By exploiting the direction of the gradient, FGSM serves as a foundational method for adversarial attack research and inspired more advanced techniques to assess and improve the robustness of machine learning models.

Since the base code uses FGSM, I tried to increase the *acuity_score* by changing its main hyperparameters. The main hyperparameters used are as follows:

- `batch_size` = 20
- `max_iterations` = 100
- `lr` = $\frac{1}{255}$
- ϵ = 16 (actual value: $\frac{16}{255}$)

Results

$$\texttt{accuracy\_score} = 0.0025$$

2.2 MI-FGSM

Momentum Adversarial Attack(MI-FGSM)

The next method applied based on the base code was MI-fgsm.

The momentum iterative fast gradient sign method (MI-FGSM), proposed by Dong, Yinpeng, and Liao *et al.* [1], improves the performance of iterative gradient-based attacks by incorporating momentum. The method stabilizes the update direction during optimization and helps escape poor local maxima, resulting in adversarial examples that are both effective and highly transferable.

Key Idea: Given a model with parameters θ, input x, target label y, and a loss function $J(\theta, x, y)$, MI-FGSM updates the adversarial example iteratively as follows:

$$g_{t+1} = \mu \cdot g_t + \frac{\nabla_x J(\theta, x_t, y)}{\|\nabla_x J(\theta, x_t, y)\|_1},$$
$$x_{t+1} = x_t + \alpha \cdot \text{sign}(g_{t+1}),$$

where:

- g_t is the accumulated gradient (momentum term) at iteration t,
- μ is the decay factor controlling the contribution of previous gradients,
- α is the step size, and
- $\text{sign}(\cdot)$ represents the element-wise sign function.

Advantages:

- **Improved Transferability**: The momentum term stabilizes the update direction, preventing overfitting to the source model and increasing success rates for black-box attacks.
- **Escaping Local Maxima**: Accumulating gradients helps the attack escape poor local maxima, which iterative methods often get stuck in.
- **Strong White-Box and Black-Box Attacks**: MI-FGSM balances attack strength and transferability better than FGSM and standard iterative FGSM (I-FGSM).

By leveraging momentum, MI-FGSM produces adversarial examples that can deceive both white-box and black-box models with high success rates, even against adversarially trained models.

Key Changes in MI-FGSM Implementation

1. Added part

```
momentum = torch.zeros_like(delta)
decay_factor = 0.8
```

2. Modified update loop

```
grad = delta.grad.clone()
grad_norm = torch.norm(grad, p=1)
momentum = decay_factor * momentum + grad / grad_norm
delta.grad.zero_()
delta.data = delta.data - lr * torch.sign(momentum)
```

The hyperparameters were the same as FGSM except for decay_factor.

- `batch_size` = 20
- `max_iterations` = 100
- `lr` = $\frac{1}{255}$
- ϵ = 16 (actual value: $\frac{16}{255}$)
- `decay_factor` = 0.8

Results

$$\texttt{accuracy\_score} = 0.015$$

The attack accuracy increased compared to using only FGSM, but I decided to look for other methods.

2.3 DI-FGSM

Diverse Input Iterative Fast Gradient Sign Method(DI-FGSM)

The Diverse Input Iterative Fast Gradient Sign Method (DI-FGSM), proposed by Xie et al. (2019)[4], is an adversarial attack method designed to improve the **transferability** of adversarial examples by applying **input diversity** during the generation process. This approach extends the traditional Iterative Fast Gradient Sign Method (I-FGSM) by introducing random transformations to the input images.

Key Concepts:

- **Input Transformations:** Random transformations, such as resizing and padding, are applied to the input images with a probability p at each iteration. This helps mitigate overfitting to specific model parameters and enhances transferability.
- **Objective:** To generate adversarial examples that are effective in both white-box and black-box settings by introducing diverse input patterns during the attack process.

Update Rule: The adversarial examples are generated iteratively, where the standard I-FGSM update rule is modified as follows:

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

where $T(X; p)$ is a stochastic transformation function that applies a random transformation to X with probability p.

Key Advantages:

- **Improved Black-Box Success Rates:** Adversarial examples generated with diverse inputs demonstrate higher transferability to other models.
- **Maintained White-Box Success Rates:** DI-FGSM retains high success rates in white-box settings.
- **Comparison with Existing Methods:** DI-FGSM outperforms I-FGSM in black-box scenarios while being more effective than FGSM in generating adversarial examples.

Key Changes in DI-FGSM Implementation

1. Added Input Diversity Function

A new function, `input_diversity`, has been added to apply random resizing and padding to the input images with a specified probability. This transformation increases the diversity of the inputs, enhancing the transferability of the generated adversarial examples.

```
def input_diversity(image, prob=0.8, size_range=(299, 330)):
    """Applies random resizing and padding with a given probability."""
    if np.random.rand() < prob:
        rnd = np.random.randint(*size_range)
        rescaled = F.interpolate(image, size=(rnd, rnd), mode='bilinear',
        ↪  align_corners=False)
        pad_top = np.random.randint(0, size_range[1] - rnd)
        pad_bottom = size_range[1] - rnd - pad_top
        pad_left = np.random.randint(0, size_range[1] - rnd)
        pad_right = size_range[1] - rnd - pad_left
        padded = F.pad(rescaled, [pad_left, pad_right, pad_top, pad_bottom],
        ↪  mode='constant', value=0)
        return padded
    return image
```

2. Modified Update Loop

In the update loop, the input to the model is now transformed using the `input_diversity` function before calculating the loss and gradients. The new process is as follows:

New Code (DI-FGSM):

```
transformed_input = input_diversity(X_ori + delta, prob=p, size_range=size_range)
logits = resnet(norm(transformed_input))
loss = nn.CrossEntropyLoss(reduction='sum')(logits, labels)
```

New Hyper Parameter:

```
p=0.8 #prob
```

The hyperparameters were the same as FGSM except for p.

- `batch_size` = 20
- `max_iterations` = 100
- `lr` = $\frac{1}{255}$
- $\epsilon = 16$ (actual value: $\frac{16}{255}$)
- p = 0.8

Results

$$\texttt{accuracy\_score} = 0.2525$$

Although the accuracy is much better than the previous methods, I wanted to combine them.

2.4 DI-FGSM MI-FGSM

Final Code

The following is the final implementation that combines the concepts of DI-FGSM and MI-FGSM to generate adversarial examples:

```python
preds_ls = []
labels_ls =[]
origin_ls = []

def input_diversity(image, prob=0.5, size_range=(299, 330)):
    """Applies random resizing and padding with a given probability."""
    if np.random.rand() < prob:
        rnd = np.random.randint(*size_range)
        rescaled = F.interpolate(image, size=(rnd, rnd), mode='bilinear',
        ↪  align_corners=False)
        pad_top = np.random.randint(0, size_range[1] - rnd)
        pad_bottom = size_range[1] - rnd - pad_top
        pad_left = np.random.randint(0, size_range[1] - rnd)
        pad_right = size_range[1] - rnd - pad_left
        padded = F.pad(rescaled, [pad_left, pad_right, pad_top, pad_bottom],
        ↪  mode='constant', value=0)
        return padded
    return image

# Parameters for DI-FGSM
p = 0.8  # Probability to apply transformation
size_range = (299, 330)

torch.cuda.empty_cache()
for k in tqdm(range(epochs), total=epochs):
    batch_size_cur = min(batch_size, len(ids) - k * batch_size)
    X_ori = torch.zeros(batch_size_cur, 3, img_size, img_size).to(device)
    delta = torch.zeros_like(X_ori, requires_grad=True).to(device)
    momentum = torch.zeros_like(delta)
    decay_factor = 0.8
    for i in range(batch_size_cur):
        X_ori[i] = trn(Image.open(input_path + ids[k * batch_size + i] + '.png'))
    ori_idx = origins[k * batch_size:k * batch_size + batch_size_cur]
    labels = torch.tensor(targets[k * batch_size:k * batch_size +
    ↪  batch_size_cur]).to(device)

    for t in range(max_iterations):
        transformed_input = input_diversity(X_ori + delta, prob=p,
        ↪  size_range=size_range)
        # Calculate logits and loss
        logits = resnet(norm(transformed_input))
        loss = nn.CrossEntropyLoss(reduction='sum')(logits, labels)
        loss.backward()

        # Update delta with gradient information
        grad = delta.grad.clone()
        grad_norm = torch.norm(grad, p=1)  # L1 normalization
        momentum = decay_factor * momentum + grad / grad_norm  # Momentum update

        delta.data = delta.data - lr * torch.sign(momentum)
        delta.data = delta.data.clamp(-epsilon / 255, epsilon / 255)
        delta.data = ((X_ori + delta.data).clamp(0, 1)) - X_ori
        delta.grad.zero_()

    X_pur = norm(X_ori + delta)
    preds = torch.argmax(vgg(X_pur), dim=1)

    preds_ls.append(preds.cpu().numpy())
    labels_ls.append(labels.cpu().numpy())
    origin_ls.append(ori_idx)
```

Key Changes and Rationale

1. Input Diversity Function

The function `input_diversity` was added to introduce random resizing and padding to the input images. This transformation is applied with a probability $p = 0.8$ to increase the diversity of inputs and reduce overfitting to specific model parameters.

```python
def input_diversity(image, prob=0.5, size_range=(299, 330)):
    """Applies random resizing and padding with a given probability."""
    if np.random.rand() < prob:
        rnd = np.random.randint(*size_range)
        rescaled = F.interpolate(image, size=(rnd, rnd), mode='bilinear',
        ↪  align_corners=False)
        pad_top = np.random.randint(0, size_range[1] - rnd)
        pad_bottom = size_range[1] - rnd - pad_top
        pad_left = np.random.randint(0, size_range[1] - rnd)
        pad_right = size_range[1] - rnd - pad_left
        padded = F.pad(rescaled, [pad_left, pad_right, pad_top, pad_bottom],
        ↪  mode='constant', value=0)
        return padded
    return image
```

Reason: This function implements the core idea of DI-FGSM, where input diversity increases the transferability of adversarial examples by making the attack less dependent on specific model parameters.

2. Momentum Integration

The use of momentum was added to stabilize gradient updates and prevent the attack from getting stuck in poor local maxima.

```python
momentum = torch.zeros_like(delta)
decay_factor = 0.8
grad = delta.grad.clone()
grad_norm = torch.norm(grad, p=1)  # L1 normalization
momentum = decay_factor * momentum + grad / grad_norm  # Momentum update
```

Reason: This reflects the concept of MI-FGSM, where momentum smooths the update direction and improves attack performance, especially in transfer-based settings.

3. Combined Input Diversity and Momentum

The final attack combines the strengths of DI-FGSM and MI-FGSM. Input diversity reduces overfitting, while momentum improves the efficiency and robustness of gradient updates. The combined approach generates adversarial examples with higher success rates in both white-box and black-box settings.

```python
transformed_input = input_diversity(X_ori + delta, prob=p, size_range=size_range)
logits = resnet(norm(transformed_input))
loss = nn.CrossEntropyLoss(reduction='sum')(logits, labels)
```

Reason: This ensures that the attack benefits from both input diversity and stable gradient updates, combining the advantages of DI-FGSM and MI-FGSM.

The hyperparameters were the same as the previous methods.

- $\texttt{batch\_size} = 20$
- $\texttt{max\_iterations} = 100$
- $\texttt{lr} = \frac{1}{255}$
- $\epsilon = 16$ (actual value: $\frac{16}{255}$)
- $\texttt{decay\_factor} = 0.8$
- $\texttt{p} = 0.8$

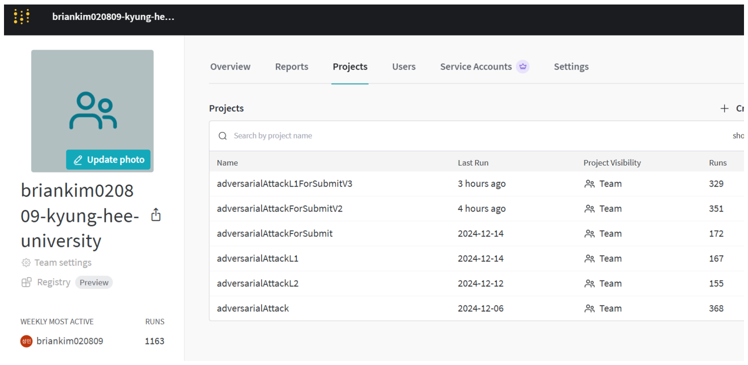
Results

$$\texttt{accuracy\_score} = 0.2925$$

I think the method is good enough, so I moved on to hyperparameter tuning.

3 Hyperparameter Tuning

I tried to find a hyperparameter that performs best using a hyperparameter search tool called WanDB. There were many overlapping trials, but we searched more than 1000 times.



briankim020809-kyung-hee-university

Projects

Name	Last Run	Project Visibility	Runs
adversarialAttackL1ForSubmitV3	3 hours ago	Team	329
adversarialAttackForSubmitV2	4 hours ago	Team	351
adversarialAttackForSubmit	2024-12-14	Team	172
adversarialAttackL1	2024-12-14	Team	167
adversarialAttackL2	2024-12-12	Team	155
adversarialAttack	2024-12-06	Team	368

WEEKLY MOST ACTIVE	RUNS
briankim020809	1163

Figure 1: Hyper Parameter Searching

3.1 Hyperparameter Descriptions

- `batch_size`: The number of samples processed in one iteration of the training or evaluation loop.
- `max_iterations`: The maximum number of gradient update steps performed during the adversarial attack process.
- `lr`: The learning rate used to control the step size of the perturbation update.
- ϵ: The maximum allowable perturbation magnitude for each pixel, constrained to ensure imperceptibility.
- `weight_decay`: Regularization parameter used to penalize large weight updates. This helps prevent overfitting by applying a decay factor to the model weights.

- `p`: The probability of applying the *input diversity* transformation at each iteration, which introduces randomness for improved transferability.

- `norm`: Specifies the normalization type for the gradient update. Here, L_1 norm is used to normalize the gradient values.

Hyper Parameter Context in the Code

- **Input Diversity** (`p`, `size_range`): The `input_diversity` function introduces randomness by resizing and padding the input with a probability of p. This increases the transferability of the adversarial examples across different models and datasets.

- **Momentum** (`momentum`, `decay_factor`): Momentum is accumulated over iterations to stabilize the direction of the gradient update. The `decay_factor` determines the contribution of previous gradients to the current update.

- **Perturbation Constraints** (`lr`, ϵ): The learning rate (`lr`) controls the step size of each perturbation, while ϵ limits the maximum magnitude to ensure imperceptibility.

- **Gradient Normalization** (`norm`): Gradients are normalized using the L_1 or L_2 norm before applying the momentum update. This ensures the update direction is consistent.

3.2 Hyperparameter Tuning and the Impact of Epsilon

During hyperparameter tuning, it was observed that **epsilon** (ϵ) has a significantly greater impact on accuracy compared to other parameters. Therefore, the selection of epsilon was made with careful consideration.

As shown in **Figure 2** and **Figure 3**, the accuracy graph clearly demonstrates the effect of epsilon on model performance. In contrast, when comparing with **Figure 4**, which shows accuracy as a function of the learning rate (`lr`), the difference becomes evident.

Although the relationship between epsilon and accuracy is not perfectly linear due to the influence of other parameters, the average trend (depicted as the light blue line) appears to exhibit a near-linear increase.

The accuracy reached its peak when epsilon was set to a maximum value of 32 ($\frac{32}{255}$). However, for the final version, epsilon was set to 16 ($\frac{16}{255}$), consistent with the base code configuration, to avoid introducing excessive noise.

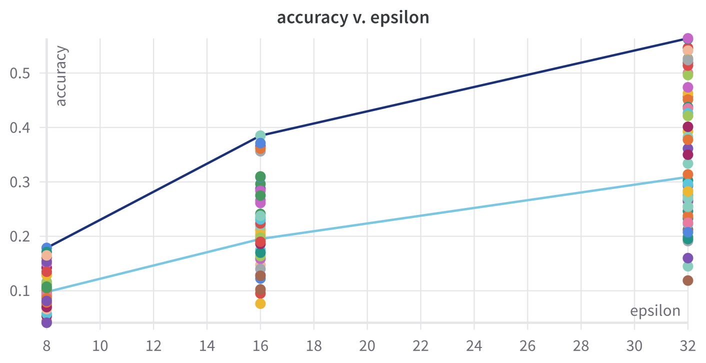


Figure 2: Accuracy as a function of epsilon (Graph 1).

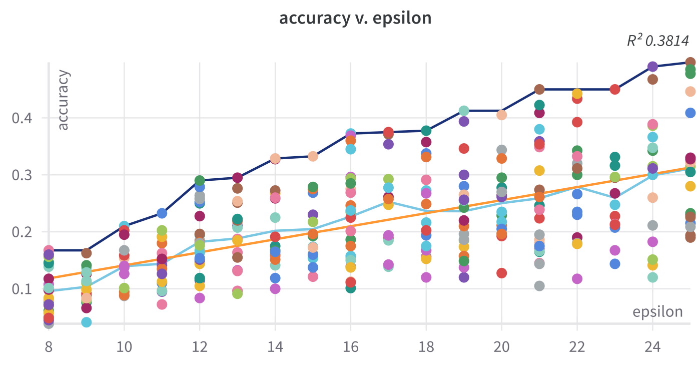


Figure 3: Accuracy as a function of epsilon (Graph 2).

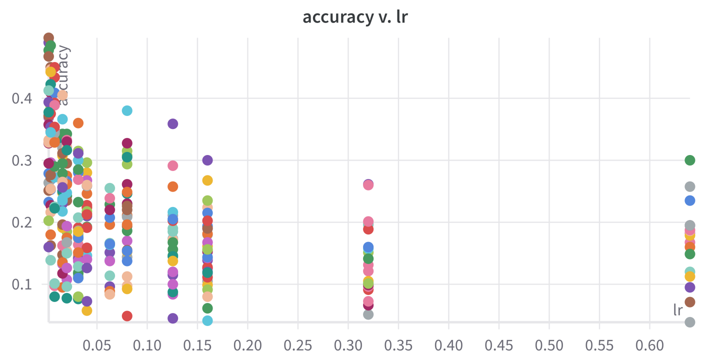


Figure 4: Accuracy as a function of the learning rate (lr).

3.2.1 Hyperparameter Combinations

The following table presents the hyperparameter combinations explored during the search:

Parameter	**Type**	**Values / Range**
`batch_size`	Discrete Values	10, 20, 30, 40
`max_iterations`	Integer (Uniform)	Range: 50 to 300
`lr`	Discrete Values	$\frac{0.125}{255}, \frac{0.25}{255}, \frac{0.5}{255}, \frac{1}{255}, \frac{2}{255}, \frac{4}{255}, \frac{8}{255}, \frac{16}{255}, \frac{32}{255}, \frac{1}{50}, \frac{2}{50}, \frac{4}{50}, \frac{8}{50}, \frac{16}{50}, \frac{32}{50}$
`epsilon`	Discrete Values	8, 16, 32
`p`	Continuous (Uniform)	Range: 0.5 to 1.0
`size_min`	Fixed Value	299
`size_max`	Fixed Value	350
`decay_factor`	Continuous (Uniform)	Range: 0.5 to 1.0

Table 1: Hyperparameter combinations explored during the search.

3.3 Hyperparameter Searching Results

The following table presents the hyperparameter configurations for the maximum accuracy version and the final optimized version:

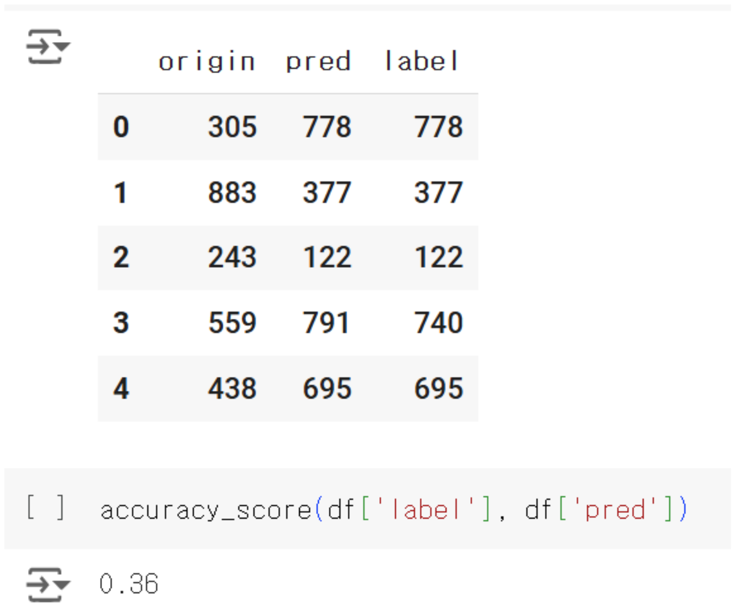
Hyperparameter	Maximum Accuracy Version	Final (Optimized) Version
`batch_size`	20	10
`max_iterations`	242	271
`img_size`	–	299
`lr (step size)`	$\frac{0.5}{255}$	$\frac{0.5}{255}$
ϵ	32	16
`p`	0.8816	0.81735
`decay_factor`	0.7426	0.75978
`norm`	L1 norm	L2 norm
Accuracy	0.5475	0.36

Table 2: Comparison of hyperparameter configurations for maximum accuracy and final optimized versions.

4 Conclusion and Future Work

4.1 Conclusion

The final optimized version of the model achieved an accuracy of 0.36, demonstrating a balanced trade-off between attack strength and noise. This was achieved using hyperparameters such as `batch_size` $= 10$, `max_iterations` $= 271$, and $\epsilon = \frac{16}{255}$. Despite the lower success rate compared to the maximum configuration, this choice ensured reduced noise and better alignment with practical constraints.



```
   origin  pred  label
0     305   778    778
1     883   377    377
2     243   122    122
3     559   791    740
4     438   695    695
```

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

```
0.36
```

Figure 5: Accuracy achieved by the final version of the model.

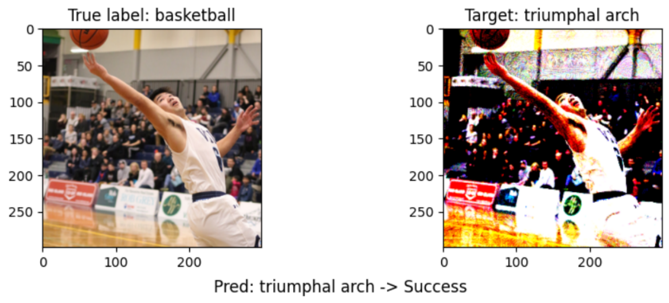


Figure 6: It's some of the success stories of the attack.

4.2 Future Work

Future work will focus on exploring a wider range of hyperparameter combinations to identify configurations with higher accuracy and better generalization. We would like to use different methods to generate adversarial examples and apply them to different models.

References

[1] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9185–9193, 2018.

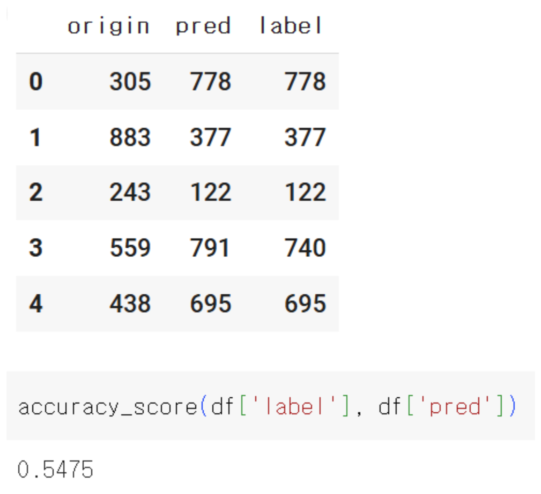
[2] Ian Goodfellow et al. Explaining and harnessing adversarial examples. 2015.

[3] Christian Szegedy et al. Intriguing properties of neural networks. 2014.

[4] Cihang Xie, Zhishuai Zhang, Yuyin Zhou, Song Bai, Jianyu Wang, Zhou Ren, and Alan Yuille. Improving transferability of adversarial examples with input diversity. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2730–2739, 2019.

Appendix: Results and Submission Data

1. Accuracy Results Image



	origin	pred	label
0	305	778	778
1	883	377	377
2	243	122	122
3	559	791	740
4	438	695	695

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

```
0.5475
```

Figure 7: Highest Accuracy version(0.5475).