

50.039 Theory and Practice of Deep Learning

W5-S3 Introduction to Attacks and Defense on Neural Networks

Matthieu De Mari



About this week (Week 5)

1. What are **attacks** on Neural Networks (NNs)?
2. Why are attacks an **important concept** when studying NNs?
3. What are the different **types of attacks** and what is the intuition behind basic attacks?
4. How to **defend** against such attacks?
5. **State-of-the-art** of (more advanced) attacks and defense, **open questions** in research.

About this week (Week 5)

1. What are **attacks** on Neural Networks (NNs)?
2. Why are attacks an **important concept** when studying NNs?
3. What are the different **types of attacks** and what is the intuition behind basic attacks?
4. How to **defend** against such attacks?
5. **State-of-the-art** of (more advanced) attacks and defense, **open questions** in research.

In the last episode

Exploiting the intrinsic properties/limits of deep learning models allows for powerful attacks with high efficacy and plausibility.

In some cases, we might be able to attack a model with 100% efficacy and very good plausibility.

- Can we implement **more advanced attacks**?
- Can we **defend** against these attacks?
- Can attackers respond to such **defenses**?

About attacks

Basic attacks (discussed in previous lecture):

0. (Epsilon Noising Attack)

- 1. Untargeted, one-shot, white-box** gradient attack
- 2. Untargeted, one-shot, white-box** fast gradient sign attack
- 3. Untargeted, iterated, white-box** fast gradient sign attack
- 4. Targeted, one-shot, white-box** fast gradient sign attack
- 5. Targeted, iterated, white-box** fast gradient sign attack

About attacks (and defense!)

Defense strategies against basic attacks:

- 1. Arms-race defense**
- 2. Defensive distillation**
- 3. Black-boxing your model**
- 4. More?**

About attacks

Advanced attacks (only discussing the concepts briefly):

1. **Carlini-Wagner attack** (**targeted**, **iterated**, **white-box**)
2. **Surrogate attack** (**targeted/untargeted**, **one-shot/iterated**, **black-box**)
3. **Boundary attack** (**targeted**, **iterated**, **black-box**)
4. **(And, if time allows, a brief state-of-the-art of other remarkable attacks, for curious students!)**

Untargeted Gradient Attack (option #1)

Definition (untargeted gradient attack):

The **untargeted gradient attack** takes a single sample x , of original class $c \in \mathcal{C}$ and attempts to produce a sample \tilde{x} of class $\tilde{c} \in \mathcal{C}$, with $\tilde{c} \neq c$.

$$c = \operatorname{argmax}_{i \in \mathcal{C}} (f_i(x))$$

- **Option #1:** look for the most probable class $c \in \mathcal{C}$ and use gradient **ascent** to move the sample **away from its original class**, with step ϵ .

$$\tilde{x} \leftarrow x + \epsilon \nabla_x L(x, \theta, c)$$

The attack is successful if

$$\tilde{c} = \operatorname{argmax}_{i \in \mathcal{C}} (f_i(\tilde{x})) \neq c$$

And then two options...

Untargeted Gradient Attack (option #2)

Definition (untargeted gradient attack):

The **untargeted gradient attack** takes a single sample x , of original class $c \in \mathcal{C}$ and attempts to produce a sample \tilde{x} of class $\tilde{c} \in \mathcal{C}$, with $\tilde{c} \neq c$.

$$c = \operatorname{argmax}_{i \in \mathcal{C}} (f_i(x))$$

And then two options...

- **Option #2:** look for the least probable class $c^* \in \mathcal{C}$ and use gradient **descent** to move the sample **in the direction of the least probable class**, with step ϵ .

$$\begin{aligned} c^* &= \operatorname{argmin}_{i \in \mathcal{C}} (f_i(x)) \\ \tilde{x} &\leftarrow x - \epsilon \nabla_x L(x, \theta, c^*) \end{aligned}$$

The attack is successful if

$$\tilde{c} = \operatorname{argmax}_{i \in \mathcal{C}} (f_i(\tilde{x})) \neq c$$

Fast Gradient Sign Method (FGSM)

Definition (**Fast Gradient Sign Method attack**):

The **Fast Gradient Sign Method attack** only uses the sign of the gradient to create an attack sample.

$$\tilde{x} \leftarrow x + \epsilon \nabla_x L(x, \theta, c)$$

(Gradient attack)

$$\tilde{x} \leftarrow x + \epsilon \mathbf{sign}(\nabla_x L(x, \theta, c))$$

(FGSM attack)

- **Important property:** this also helps to make more plausible samples, as it will, by design, verify $\|\tilde{x} - x\|_\infty \leq \epsilon$.
- (Plausibility constraint, we did not have in the previous attacks!)

```

1 def fgsm_attack(image, epsilon, data_grad):
2     # Get element-wise signs of each element of the data gradient
3     data_grad_sign = data_grad.sign()
4
5     # Create the attack image by adjusting each pixel of the input image
6     eps_image = image + epsilon*data_grad_sign
7
8     # Clipping eps_image to maintain pixel values into the [0, 1] range
9     eps_image = torch.clamp(eps_image, 0, 1)
10
11     # Return
12     return eps_image

```

Iterative and Targeted FGSM attack

Definition (iterated targeted Fast Gradient Sign Method attack):

The **iterated targeted Fast Gradient Sign Method attack** will use the FGSM attack but will use the gradients of a targeted class \tilde{c} .

This follows the same logic as moving towards the least probable class as in Gradient attack option #2, but you can use it with any class of your choice. This attack uses gradient descent to move the sample towards the targeted class \tilde{c} .

This is repeated until it reaches a maximal number of iterations or makes the model malfunction with targeted class \tilde{c} .

$$\begin{aligned} x_0 &= x \\ x_{n+1} &\leftarrow x_n - \epsilon \text{sign}(\nabla_x L(x_n, \theta, \tilde{c})) \end{aligned}$$

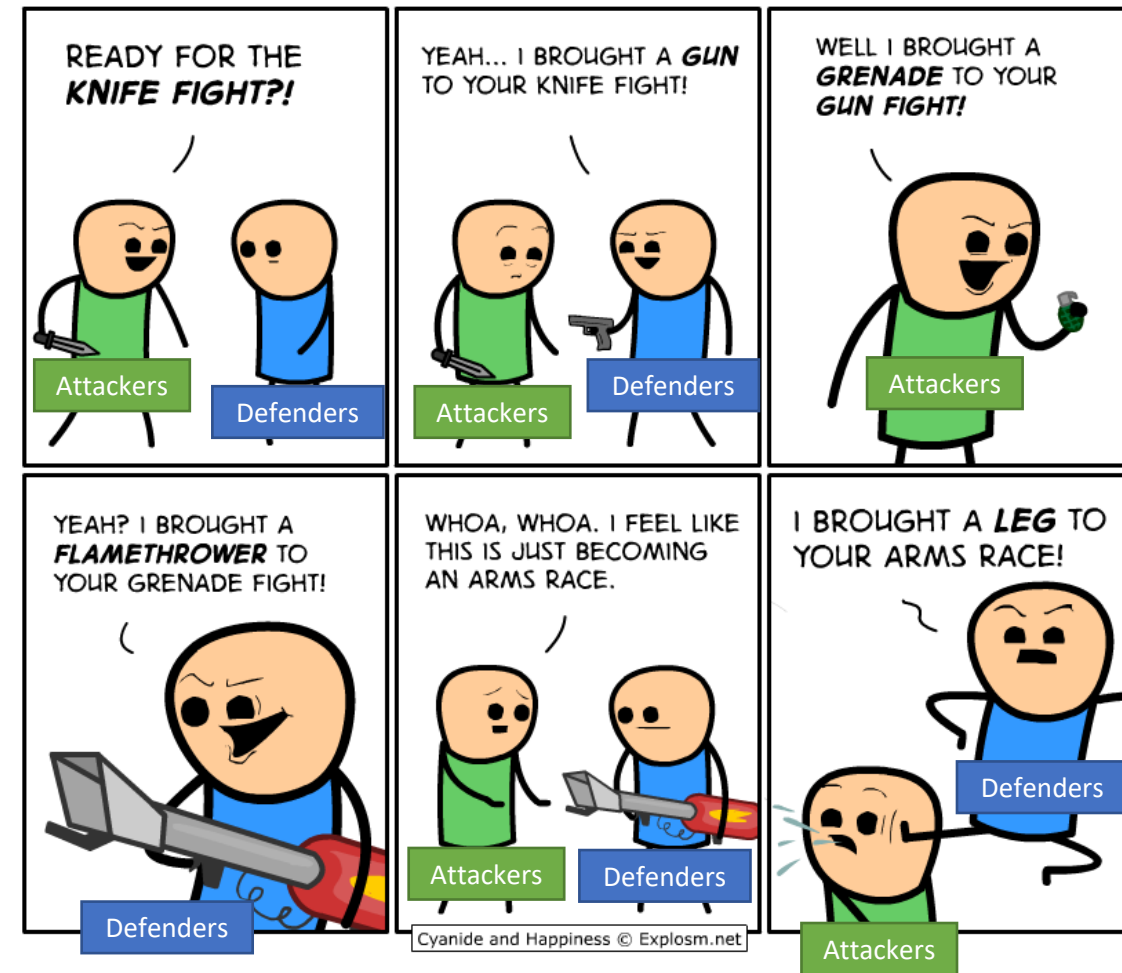
The Madry's (or arms race) defense

Definition (the Madry's or arms race defense):

The **arms race defense strategy**, is the most basic defense strategy.

If we know the type of attack that is coming, we can generate our own attack samples and train our model to correctly classify some of these attack samples.

Doing so, we therefore anticipate for future attacks of this type.



Source: <https://explosm.net/comics/3939/>

Note: this is also known as the **Madry's defense** [Madry2017].

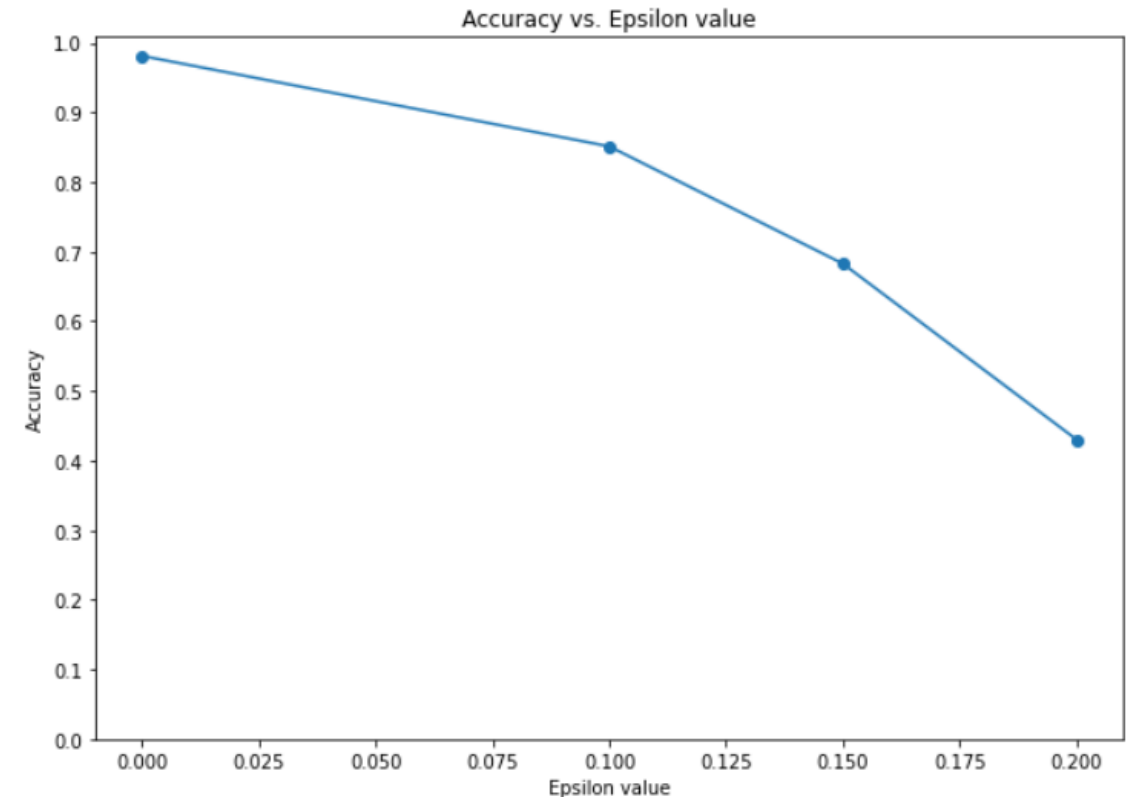
The arms race defense

Let us start with our pre-trained model, and the basic FGSM attack (one-shot, untargeted version).

- Our model is not prepared to face this type of attacks and will suffer badly from it.

```
1 epsilons = [0, .1, .15, .2]
2 accuracies = []
3 examples = []
4
5 # Run test() function for each epsilon
6 for eps in epsilons:
7     acc, ex = test(model, device, test_loader, eps)
8     accuracies.append(acc)
9     examples.append(ex)
```

Epsilon: 0 - Test Accuracy = 9810/10000 = 0.981
Epsilon: 0.1 - Test Accuracy = 8510/10000 = 0.851
Epsilon: 0.15 - Test Accuracy = 6826/10000 = 0.6826
Epsilon: 0.2 - Test Accuracy = 4301/10000 = 0.4301



The arms race defense

Our next step would then be to continue the training of our model, integrating samples that have been generated using said FGSM attack.

- **Intuition:** in a sense, the arms race defense is a sort of **data augmentation technique**, that helps make the model more robust to attacks of the FGSM type.

The arms race defense

- For demonstration, we will create a second model, using the same pre-trained weights.
- We will also retrieve the training dataset and make its dataloader (train_loader), with mini-batches.
- We will then continue the training of this model, using CrossEntropy as our loss and a basic SGD as our optimizer.

```
1 # Load the pretrained model
2 model2 = Net().to(device)
3 pretrained_model = "./mnist_model.data"
4 model2.load_state_dict(torch.load(pretrained_model, map_location = 'cpu'))
```

<All keys matched successfully>

```
1 # MNIST dataset and dataloader
2 # (For testing only, we will use a pre-trained model)
3 ds2 = datasets.MNIST('./data', train = True, download = True, transform = tf)
4 train_loader = torch.utils.data.DataLoader(ds2, batch_size = 64, shuffle = True)
```

```
1 print(len(train_loader))
```

938

```
1 # Define a loss function and an optimizer for training
2 criterion = nn.CrossEntropyLoss()
3 optimizer = optim.SGD(model2.parameters(), lr = 0.001, momentum = 0.9)
```

From Notebook 5.!

Retraining our model

```

1 def retrain(model, train_loader, optimizer, criterion, n_iter = 5):
2
3     # This will make prints happen every 50 mini-batches
4     mod_val = 50
5
6     # Train over n_iter epochs
7     for epoch in range(n_iter):
8
9         # Keep track of the running losses over batches
10        running_loss_normal = 0.0
11        running_loss_attack = 0.0
12
13        for i, data in enumerate(train_loader):
14            """
15            1. Mini-batches on normal samples
16            """
17            # Retrieve input images and labels
18            inputs, labels = data
19            inputs.requires_grad = True
20
21            # Zeroing gradients
22            optimizer.zero_grad()
23
24            # Forward, Loss, Backprop
25            outputs = model(inputs)
26            loss = criterion(outputs, labels)
27            loss.backward()
28
29            # Keep track of running loss (normal samples)
30            running_loss_normal += loss.item()
31

```

```

32        """
33        2. Mini-batches on generated attack samples
34        """
35
36        # Collect gradients of image
37        data_grad = inputs.grad.data
38
39        # Call FGSM Attack with the 0.15 epsilon value
40        epsilon = .15
41        eps_image = fgsm_attack(inputs, epsilon, data_grad)
42
43        # Re-classify the epsilon image
44        output2 = model(eps_image)
45        # Get the index of the max log-probability
46        eps_pred = output2.max(1, keepdim = True)[1]
47
48        # Loss, Backprop, Optimize
49        loss2 = criterion(output2, labels)
50        loss2.backward()
51        optimizer.step()
52
53        # Keep track of running loss (attack samples)
54        running_loss_attack += loss2.item()

```


Retraining our model

```

1 def retrain(model, train_loader, optimizer, criterion, n_iter = 5):
2
3     # This will make prints happen every 50 mini-batches
4     mod_val = 50
5
6     # Train over n_iter epochs
7     for epoch in range(n_iter):
8
9         # Keep track of the running losses over batches
10        running_loss_normal = 0.0
11        running_loss_attack = 0.0
12
13        -----
14        for i, data in enumerate(train_loader):
15            """
16            | 1. Mini-batches on normal samples
17            | """
18            | # Retrieve input images and labels
19            | inputs, labels = data
20            | inputs.requires_grad = True
21            |
22            | # Zeroing gradients
23            | optimizer.zero_grad()
24            |
25            | # Forward, Loss, Backprop
26            | outputs = model(inputs)
27            | loss = criterion(outputs, labels)
28            | loss.backward()
29            |
30            | # Keep track of running loss (normal samples)
31            | running_loss_normal += loss.item()
32            |
33            -----

```

Just like in “normal” training we are going to train on the training samples (this uses our train_loader, not the test one!)

```

32        """
33        2. Mini-batches on generated attack samples
34        """
35
36        # Collect gradients of image
37        data_grad = inputs.grad.data
38
39        # Call FGSM Attack with the 0.15 epsilon value
40        epsilon = .15
41        eps_image = fgsm_attack(inputs, epsilon, data_grad)
42
43        # Re-classify the epsilon image
44        output2 = model(eps_image)
45        # Get the index of the max log-probability
46        eps_pred = output2.max(1, keepdim = True)[1]
47
48        # Loss, Backprop, Optimize
49        loss2 = criterion(output2, labels)
50        loss2.backward()
51        optimizer.step()
52
53        # Keep track of running loss (attack samples)
54        running_loss_attack += loss2.item()

```

Retraining our model

```

1 def retrain(model, train_loader, optimizer, criterion, n_iter = 5):
2
3     # This will make prints happen every 50 mini-batches
4     mod_val = 50
5
6     # Train over n_iter epochs
7     for epoch in range(n_iter):
8
9         # Keep track of the running losses over batches
10        running_loss_normal = 0.0
11        running_loss_attack = 0.0
12
13        for i, data in enumerate(train_loader):
14            """
15            1. Mini-batches on normal samples
16            """
17            # Retrieve input images and labels
18            inputs, labels = data
19            inputs.requires_grad = True
20
21            # Zeroing gradients
22            optimizer.zero_grad()
23
24            # Forward, Loss, Backprop
25            outputs = model(inputs)
26            loss = criterion(outputs, labels)
27            loss.backward()
28
29            # Keep track of running loss (normal samples)
30            running_loss_normal += loss.item()
31

```

And in the second part, we will do the same, but will transform the samples using our attack function and train on this sample.

```

32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
"""
2. Mini-batches on generated attack samples
"""

# Collect gradients of image
data_grad = inputs.grad.data

# Call FGSM Attack with the 0.15 epsilon value
epsilon = .15
eps_image = fgsm_attack(inputs, epsilon, data_grad)

# Re-classify the epsilon image
output2 = model(eps_image)
# Get the index of the max log-probability
eps_pred = output2.max(1, keepdim = True)[1]

# Loss, Backprop, Optimize
loss2 = criterion(output2, labels)
loss2.backward()
optimizer.step()

# Keep track of running loss (attack samples)
running_loss_attack += loss2.item()

```

Retraining our model

```

1 def retrain(model, train_loader, optimizer, criterion, n_iter = 5):
2
3     # This will make prints happen every 50 mini-batches
4     mod_val = 50
5
6     # Train over n_iter epochs
7     for epoch in range(n_iter):
8
9         # Keep track of the running losses over batches
10        running_loss_normal = 0.0
11        running_loss_attack = 0.0
12
13        for i, data in enumerate(train_loader):
14            """
15            1. Mini-batches on normal samples
16            """
17            # Retrieve input images and labels
18            inputs, labels = data
19            inputs.requires_grad = True
20
21            # Zeroing gradients
22            optimizer.zero_grad()
23
24            # Forward, Loss, Backprop
25            outputs = model(inputs)
26            loss = criterion(outputs, labels)
27            loss.backward()
28
29            # Keep track of running loss (normal samples)
30            running_loss_normal += loss.item()
31

```

In addition (not shown here), we will display the running losses for both the normal and attack samples.

```

32        """
33        2. Mini-batches on generated attack samples
34        """
35
36        # Collect gradients of image
37        data_grad = inputs.grad.data
38
39        # Call FGSM Attack with the 0.15 epsilon value
40        epsilon = .15
41        eps_image = fgsm_attack(inputs, epsilon, data_grad)
42
43        # Re-classify the epsilon image
44        output2 = model(eps_image)
45        # Get the index of the max log-probability
46        eps_pred = output2.max(1, keepdim = True)[1]
47
48        # Loss, Backprop, Optimize
49        loss2 = criterion(output2, labels)
50        loss2.backward()
51        optimizer.step()
52
53        # Keep track of running loss (attack samples)
54        running_loss_attack += loss2.item()

```

The arms race defense

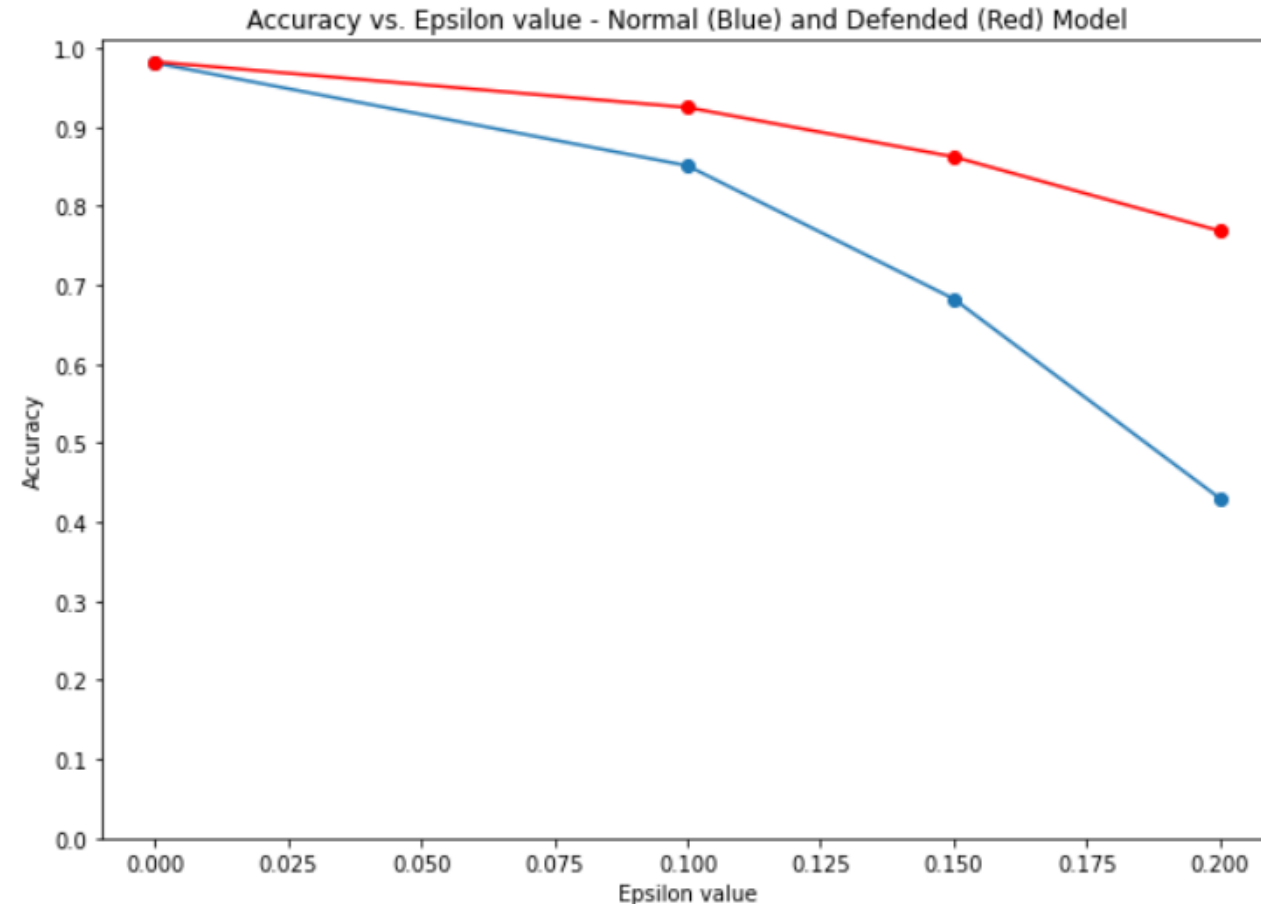
Retraining will not necessarily affect the loss on normal samples (it will keep on decreasing, possibly overfitting, but this should not change the accuracy performance of the model on normal samples).

- **What is important:** We are able to train to recognize attack samples and our model progressively becomes better at classifying those correctly.

1	retrain(model2, train_loader, optimizer, criterion)			
[Epoch 1, Batch 51]	Normal Loss: 0.296	-	Attack Loss: 0.941	
[Epoch 1, Batch 101]	Normal Loss: 0.257	-	Attack Loss: 0.877	
[Epoch 1, Batch 151]	Normal Loss: 0.244	-	Attack Loss: 0.857	
[Epoch 1, Batch 201]	Normal Loss: 0.234	-	Attack Loss: 0.831	
[Epoch 1, Batch 251]	Normal Loss: 0.226	-	Attack Loss: 0.817	
[Epoch 1, Batch 301]	Normal Loss: 0.220	-	Attack Loss: 0.798	
[Epoch 1, Batch 351]	Normal Loss: 0.219	-	Attack Loss: 0.789	
[Epoch 1, Batch 401]	Normal Loss: 0.216	-	Attack Loss: 0.776	
[Epoch 1, Batch 451]	Normal Loss: 0.216	-	Attack Loss: 0.769	
[Epoch 1, Batch 501]	Normal Loss: 0.214	-	Attack Loss: 0.761	
[Epoch 1, Batch 551]	Normal Loss: 0.213	-	Attack Loss: 0.755	
[Epoch 1, Batch 601]	Normal Loss: 0.212	-	Attack Loss: 0.753	
[Epoch 1, Batch 651]	Normal Loss: 0.211	-	Attack Loss: 0.749	
[Epoch 1, Batch 701]	Normal Loss: 0.209	-	Attack Loss: 0.744	
[Epoch 1, Batch 751]	Normal Loss: 0.209	-	Attack Loss: 0.739	
[Epoch 1, Batch 801]	Normal Loss: 0.209	-	Attack Loss: 0.733	
[Epoch 1, Batch 851]	Normal Loss: 0.208	-	Attack Loss: 0.729	
[Epoch 1, Batch 901]	Normal Loss: 0.207	-	Attack Loss: 0.725	
[Epoch 2, Batch 51]	Normal Loss: 0.187	-	Attack Loss: 0.626	
[Epoch 2, Batch 101]	Normal Loss: 0.184	-	Attack Loss: 0.634	
[Epoch 2, Batch 151]	Normal Loss: 0.191	-	Attack Loss: 0.625	
[Epoch 2, Batch 201]	Normal Loss: 0.191	-	Attack Loss: 0.628	
[Epoch 2, Batch 251]	Normal Loss: 0.189	-	Attack Loss: 0.628	
[Epoch 2, Batch 301]	Normal Loss: 0.188	-	Attack Loss: 0.628	
[Epoch 2, Batch 351]	Normal Loss: 0.189	-	Attack Loss: 0.629	
[Epoch 2, Batch 401]	Normal Loss: 0.187	-	Attack Loss: 0.629	
[Epoch 2, Batch 451]	Normal Loss: 0.187	-	Attack Loss: 0.626	
[Epoch 2, Batch 501]	Normal Loss: 0.187	-	Attack Loss: 0.625	
[Epoch 2, Batch 551]	Normal Loss: 0.187	-	Attack Loss: 0.621	
[Epoch 2, Batch 601]	Normal Loss: 0.186	-	Attack Loss: 0.618	
[Epoch 2, Batch 651]	Normal Loss: 0.185	-	Attack Loss: 0.616	
[Epoch 2, Batch 701]	Normal Loss: 0.186	-	Attack Loss: 0.614	
[Epoch 2, Batch 751]	Normal Loss: 0.186	-	Attack Loss: 0.614	
[Epoch 2, Batch 801]	Normal Loss: 0.186	-	Attack Loss: 0.614	
[Epoch 2, Batch 851]	Normal Loss: 0.186	-	Attack Loss: 0.611	
[Epoch 2, Batch 901]	Normal Loss: 0.186	-	Attack Loss: 0.610	
[Epoch 3, Batch 51]	Normal Loss: 0.176	-	Attack Loss: 0.584	
[Epoch 3, Batch 101]	Normal Loss: 0.178	-	Attack Loss: 0.594	
[Epoch 3, Batch 151]	Normal Loss: 0.181	-	Attack Loss: 0.591	
[Epoch 3, Batch 201]	Normal Loss: 0.181	-	Attack Loss: 0.590	
[Epoch 3, Batch 251]	Normal Loss: 0.180	-	Attack Loss: 0.585	
[Epoch 3, Batch 301]	Normal Loss: 0.179	-	Attack Loss: 0.582	
[Epoch 3, Batch 351]	Normal Loss: 0.179	-	Attack Loss: 0.581	
[Epoch 3, Batch 401]	Normal Loss: 0.179	-	Attack Loss: 0.577	
[Epoch 3, Batch 451]	Normal Loss: 0.180	-	Attack Loss: 0.576	

The arms race defense

- Training on the attack samples then makes the model a bit more robust to this type of attacks.
- In **blue**, you have the **original undefended model**. In **red**, the **defended model**.
- **Conclusion: Both of them have the same baseline accuracy, but the second model seems to resist more to the FGSM attacks (Can be improved with more training than just 5 iterations!)**



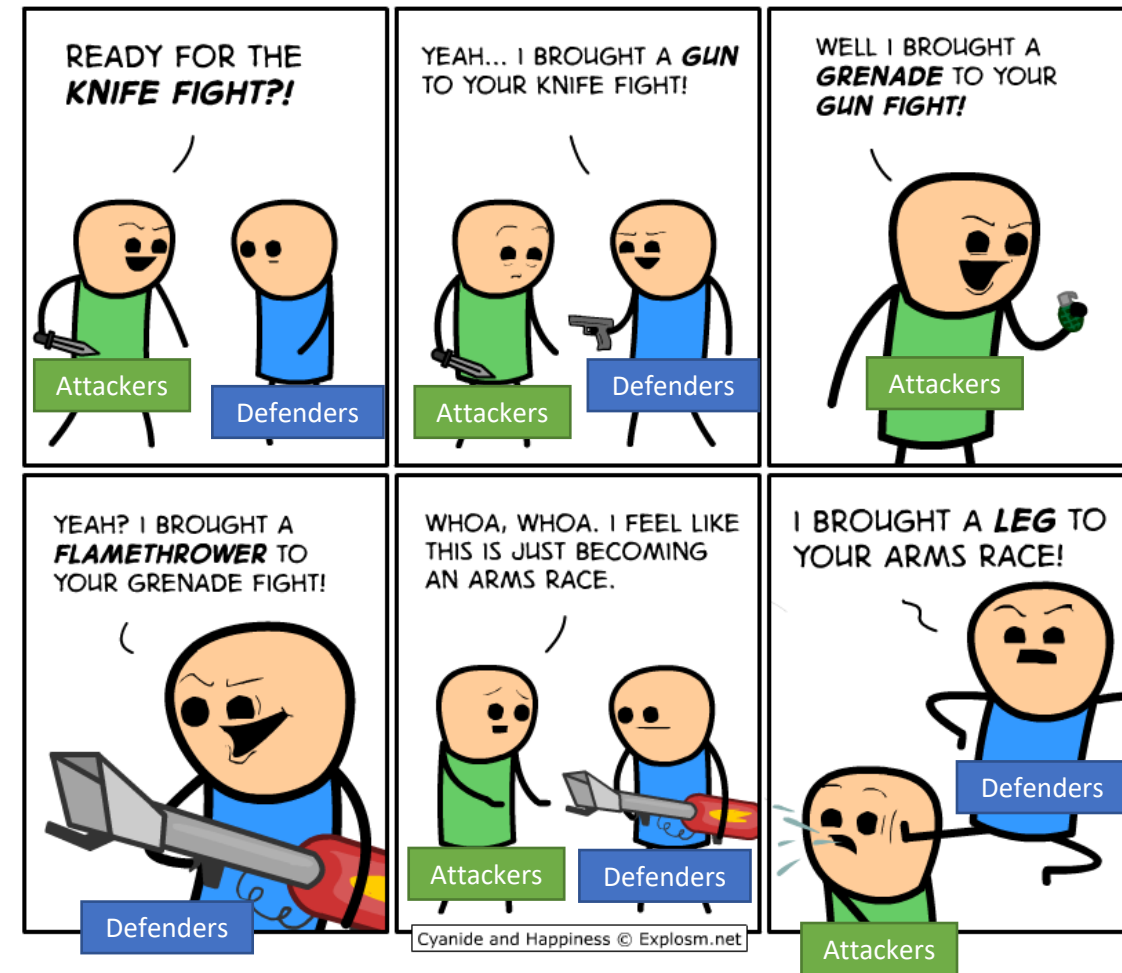
The Madry's (or arms race) defense

Definition (the Madry's or arms race defense):

The **arms race defense strategy**, is the most basic defense strategy.

If we know the type of attack that is coming, we can generate our own attack samples and train our model to correctly classify some of these attack samples.

Doing so, we therefore anticipate for future attacks of this type.



Source: <https://explosm.net/comics/3939/>

Note: this is also known as the **Madry's defense** [Madry2017].

The arms race defense

While this is the most intuitive approach and might work on the type of attacks you train your model for...

- This defense strategy will ultimately be defeated by attackers, who just have to implement a new type of attack...
- And your model simply will not know how to handle it.
- For instance, here, we defended against FGSM, but the Gradient attack remains undefended.

The arms race defense

While this is the most intuitive approach and the work on the type of attacks you train your model for...

- This defense strategy will ultimately be defeated by attackers, who just have to implement a new type of attack.

And attackers these days are getting pretty creative with new types of attacks!

- For instance, here, we defended against GSM, but the Gradient attack remains undefended.

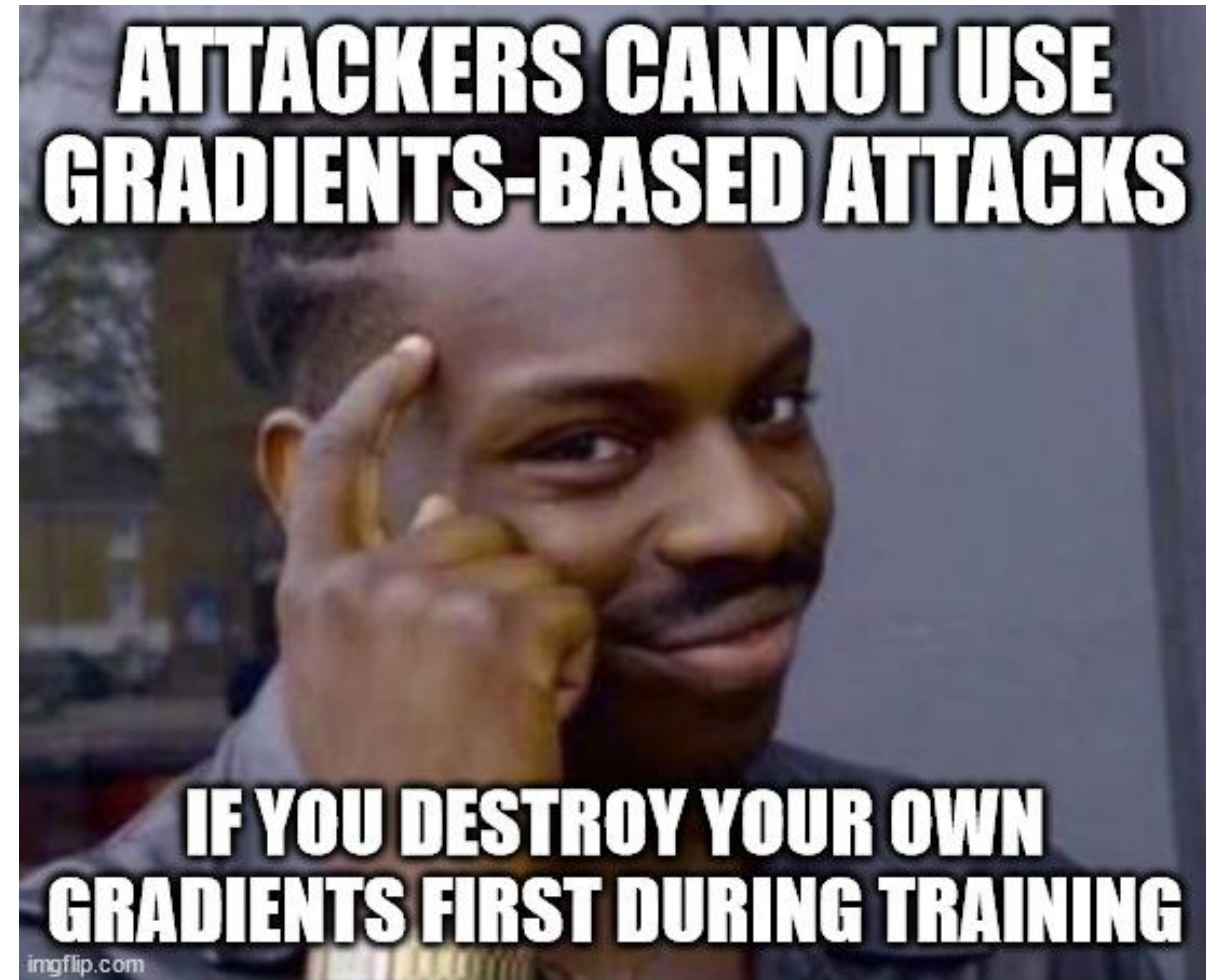
There are even challenges/competitions for coming up with new attacks against “defended” systems!

See [Medium1] and [Dong2017].

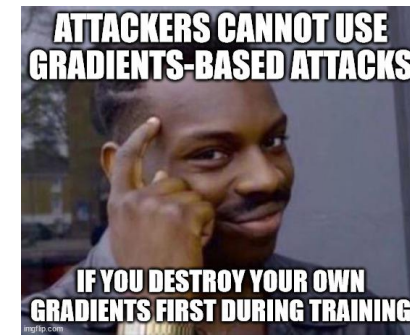
Defensive distillation

Core idea behind gradient-based defensive distillation.

- Most gradient-based attacks rely on the gradients to compute an attack sample.
- If we were to annihilate these gradients, the attackers would not be able to produce attack samples.



Defensive distillation



Core idea behind gradient-based defensive distillation.

- Most gradient-based attacks rely on the gradients to compute an attack sample.
- If we were to annihilate these gradients, the attackers would not be able to produce attack samples.

- **Important note:** This is a pretty risky strategy, as you need those gradients during the training phase of your model!
- Defensive distillation will however do so, by using some of the softmax function properties.
- Idea comes from [Papernot2015].

Reminder: the softmax function

Definition (**softmax** function):

The **softmax** function receives a list of values $x = (x_1, \dots, x_N)$ and produces the following outputs $y = (y_1, \dots, y_N)$.

$$\forall i \in [1, N], \quad y_i = \frac{\exp(x_i)}{\sum_{j=1}^N \exp(x_j)}$$

Reminder: the softmax function

Definition (**softmax** function):

The **softmax** function receives a list of values $x = (x_1, \dots, x_N)$ and produces the following outputs $y = (y_1, \dots, y_N)$.

$$\forall i \in [1, N], \quad y_i = \frac{\exp(x_i)}{\sum_{j=1}^N \exp(x_j)}$$

Definition (**softmax** function with **temperature T**):

The **softmax** function with **temperature T**, receives a list of values $x = (x_1, \dots, x_N)$ and produces the following outputs $y = (y_1, \dots, y_N)$.

$$\forall i \in [1, N], \quad y_i = \frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^N \exp\left(\frac{x_j}{T}\right)}$$

The softmax function (with temperature T)

Definition (softmax function with temperature T):

The **softmax** function with **temperature T** , receives a list of values $x = (x_1, \dots, x_N)$ and produces the following outputs $y = (y_1, \dots, y_N)$.

$$\forall i \in [1, N], \quad y_i = \frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^N \exp\left(\frac{x_j}{T}\right)}$$

Three interesting properties for the softmax with temperature.

1. When $T \rightarrow 0$, we have
 - $\forall i \in [1, N], y_i \rightarrow 0$,
 - Except for one value k , where $y_k \rightarrow 1$.
 - The index value k , corresponds to the largest one among the $(x_i)_{i \in [1, N]}$, that is $k = \operatorname{argmax}_{i \in [1, N]}(x_i)$.

The softmax function (with temperature T)

Definition (softmax function with temperature T):

The **softmax function with temperature T** , receives a list of values $x = (x_1, \dots, x_N)$ and produces the following outputs $y = (y_1, \dots, y_N)$.

2. When $T \rightarrow \infty$, we have
- $\forall i \in [1, N], y_i \rightarrow \frac{1}{N}$.

$$\forall i \in [1, N], \quad y_i = \frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^N \exp\left(\frac{x_j}{T}\right)}$$

The softmax function (with temperature T)

Definition (softmax function with temperature T):

The **softmax function with temperature T** , receives a list of values $x = (x_1, \dots, x_N)$ and produces the following outputs $y = (y_1, \dots, y_N)$.

3. When $T \rightarrow \infty$, the gradients will vanish.

$$\forall i \in [1, N], \quad y_i = \frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^N \exp\left(\frac{x_j}{T}\right)}$$

The softmax function (with temperature T)

Definition (softmax function with temperature T):

The **softmax function with temperature T** , receives a list of values $x = (x_1, \dots, x_N)$ and produces the following outputs $y = (y_1, \dots, y_N)$.

$$\forall i \in [1, N], \quad y_i = \frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^N \exp\left(\frac{x_j}{T}\right)}$$

3. When $T \rightarrow \infty$, the gradients will vanish.

- Indeed, if f is the softmax function, then its derivative f' will produce the following outputs $z = (z_1, \dots, z_N)$.

$$z_i = \frac{1}{T} \cdot \left(\frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^N \exp\left(\frac{x_j}{T}\right)} \right) \cdot \left(1 - \frac{\exp\left(\frac{x_i}{T}\right)}{\sum_{j=1}^N \exp\left(\frac{x_j}{T}\right)} \right)$$

- Using Property 2., this gives

$$\forall i \in [1, N], \quad \lim_{T \rightarrow \infty} z_i = \lim_{T \rightarrow \infty} \frac{1}{T} \cdot \frac{1}{N} \cdot \left(1 - \frac{1}{N} \right) = 0$$

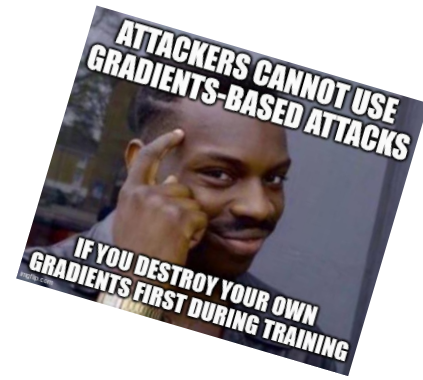
The defensive distillation strategy

Definition (the defensive distillation strategy):

The **defensive distillation strategy** makes use of the last property of the softmax function and its tendency to vanish gradients.

We can make the gradients of our model vanish by using a softmax with a large temperature T , instead of the standard $T = 1$ value.

The defensive distillation strategy



Definition (the **defensive distillation** strategy):

The **defensive distillation** strategy makes use of the last property of the softmax function and its tendency to vanish gradients.

We can make the gradients of our model vanish by using a softmax with a large temperature T , instead of the standard $T = 1$ value.

This will make the gradients vanish, and they might no longer be usable by the attackers for the gradient and fast gradient sign attacks.

Important: It might however complicate the training of the model! (Remember that we need these gradients to train and backpropagate!)

Model changes

- We will first change the constructor of our model, and add a temperature T .

```
Re 1 # Model definition
2 class Net(nn.Module):
3     """
4     This is a basic Neural Net for MNIST
5     - Two convolutions, into ReLU activations and dropouts after ReLU,
6     - Flattening,
7     - Fully connected, into ReLU activation and dropout after ReLU,
8     - Fully connected, into Log-Softmax.
9     """
10
11 def __init__(self):
12     super(Net, self).__init__()
13     # Conv. 1
14     self.conv1 = nn.Conv2d(1, 10, kernel_size = 5)
15     # Conv. 2
16     self.conv2 = nn.Conv2d(10, 20, kernel_size = 5)
17     # Dropout for Conv. layers
18     self.conv2_drop = nn.Dropout2d()
19     # FC 1
20     self.fc1 = nn.Linear(320, 50)
21     # FC 2
22     self.fc2 = nn.Linear(50, 10)
23     # Temperature (set to 1 by default)
24     self.T = 1
25
26 def forward(self, x):
27     # Conv. 1 + ReLU + Dropout
28     x = F.relu(F.max_pool2d(self.conv1(x), 2))
29     # Conv. 2 + ReLU + Dropout
30     x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
31     # Flatten
32     x = x.view(-1, 320)
33     # FC 1 + ReLU + Dropout
34     x = F.relu(self.fc1(x))
35     x = F.dropout(x, training = self.training)
36     # FC 2 + Log-Softmax
37     x = self.fc2(x)
38     return F.log_softmax(x/self.T, dim = 1)
```

Model changes

- We will first change the constructor of our model, and add a temperature T.
- This temperature is then used in the softmax function of the forward() method.

```
Re 1 # Model definition
2 class Net(nn.Module):
3     """
4     This is a basic Neural Net for MNIST
5     - Two convolutions, into ReLU activations and dropouts after ReLU,
6     - Flattening,
7     - Fully connected, into ReLU activation and dropout after ReLU,
8     - Fully connected, into Log-Softmax.
9     """
10
11     def __init__(self):
12         super(Net, self).__init__()
13         # Conv. 1
14         self.conv1 = nn.Conv2d(1, 10, kernel_size = 5)
15         # Conv. 2
16         self.conv2 = nn.Conv2d(10, 20, kernel_size = 5)
17         # Dropout for Conv. layers
18         self.conv2_drop = nn.Dropout2d()
19         # FC 1
20         self.fc1 = nn.Linear(320, 50)
21         # FC 2
22         self.fc2 = nn.Linear(50, 10)
23         # Temperature (set to 1 by default)
24         self.T = 1
25
26     def forward(self, x):
27         # Conv. 1 + ReLU + Dropout
28         x = F.relu(F.max_pool2d(self.conv1(x), 2))
29         # Conv. 2 + ReLU + Dropout
30         x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
31         # Flatten
32         x = x.view(-1, 320)
33         # FC 1 + ReLU + Dropout
34         x = F.relu(self.fc1(x))
35         x = F.dropout(x, training = self.training)
36         # FC 2 + Log-Softmax
37         x = self.fc2(x)
38         return F.log_softmax(x/self.T, dim = 1)
```

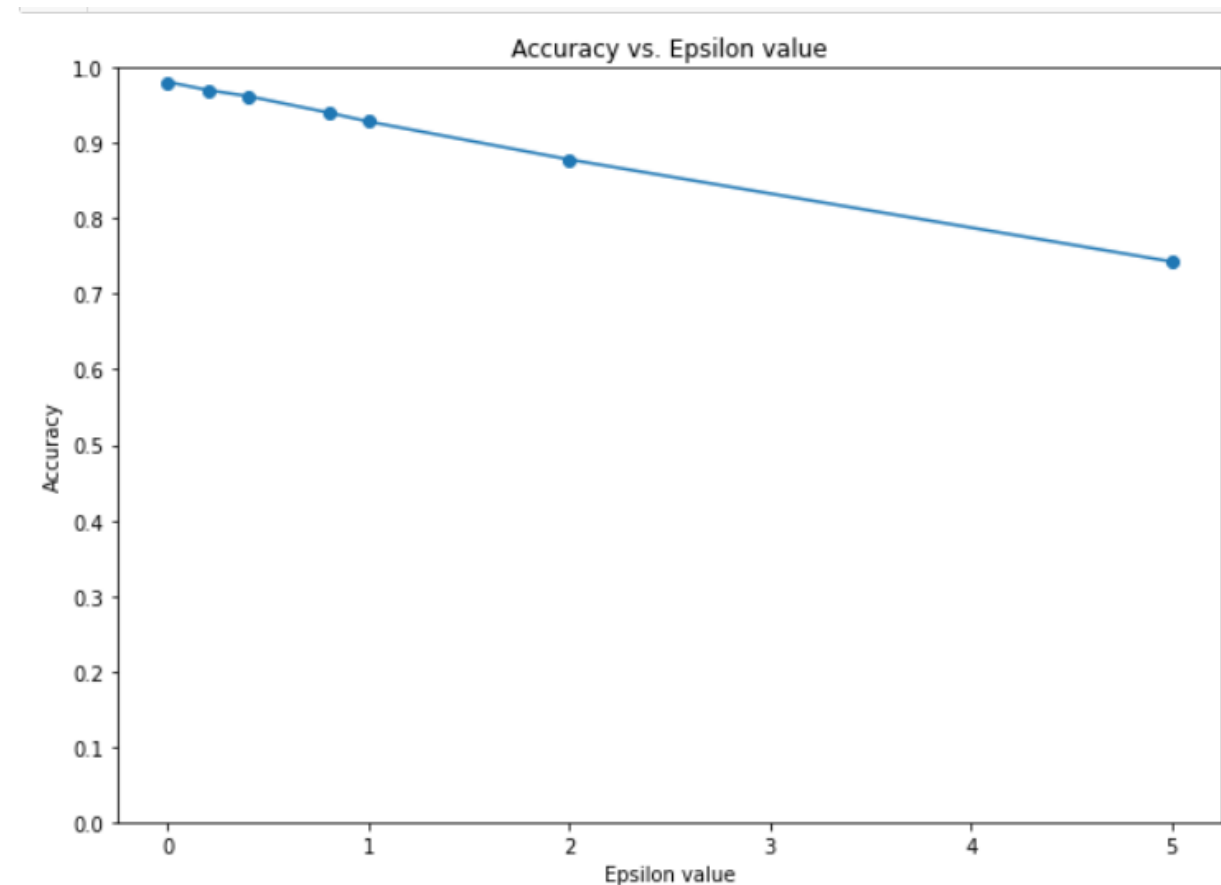
Model changes

- We will first change the constructor of our model, and add a temperature T.
- This temperature is then used in the softmax function of the forward() method.
- It can later be changed using `model.T = new_value`

```
Re 1 # Model definition
2 class Net(nn.Module):
3     """
4     This is a basic Neural Net for MNIST
5     - Two convolutions, into ReLU activations and dropouts after ReLU,
6     - Flattening,
7     - Fully connected, into ReLU activation and dropout after ReLU,
8     - Fully connected, into Log-Softmax.
9     """
10
11     def __init__(self):
12         super(Net, self).__init__()
13         # Conv. 1
14         self.conv1 = nn.Conv2d(1, 10, kernel_size = 5)
15         # Conv. 2
16         self.conv2 = nn.Conv2d(10, 20, kernel_size = 5)
17         # Dropout for Conv. layers
18         self.conv2_drop = nn.Dropout2d()
19         # FC 1
20         self.fc1 = nn.Linear(320, 50)
21         # FC 2
22         self.fc2 = nn.Linear(50, 10)
23         # Temperature (set to 1 by default)
24         self.T = 1
25
26     def forward(self, x):
27         # Conv. 1 + ReLU + Dropout
28         x = F.relu(F.max_pool2d(self.conv1(x), 2))
29         # Conv. 2 + ReLU + Dropout
30         x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
31         # Flatten
32         x = x.view(-1, 320)
33         # FC 1 + ReLU + Dropout
34         x = F.relu(self.fc1(x))
35         x = F.dropout(x, training = self.training)
36         # FC 2 + Log-Softmax
37         x = self.fc2(x)
38         return F.log_softmax(x/self.T, dim = 1)
```

Original model for baseline

- In Notebook 6., we will be using our original model and the Gradient-based attack for reference.
- This is our efficacy vs. epsilon graph for the undefended model.



Defended model

- The defended model, used for comparison will be a copy of the undefended model.

```
1 # Load the pretrained model
2 model2 = Net().to(device)
3 pretrained_model = "./mnist_model.data"
4 model2.load_state_dict(torch.load(pretrained_model, map_location = 'cpu'))
```

<All keys matched successfully>

```
1 # MNIST dataset and dataloader
2 # (For testing only, we will use a pre-trained model)
3 ds2 = datasets.MNIST('./data', train = True, download = True, transform = tf)
4 train_loader = torch.utils.data.DataLoader(ds2, batch_size = 64, shuffle = True)
```

```
1 print(len(train_loader))
```

938

```
1 # Define a loss function and an optimizer for training
2 criterion = nn.CrossEntropyLoss()
3 optimizer = optim.SGD(model2.parameters(), lr = 0.001, momentum = 0.9)
```

```
1 # Setting temperature to 100 for training
2 model2.T = 100
```

Defended model

- The defended model, used for comparison will be a copy of the undefended model.
- We will then change the temperature to $T = 100$ (arbitrarily chosen).

```
1 # Load the pretrained model
2 model2 = Net().to(device)
3 pretrained_model = "./mnist_model.data"
4 model2.load_state_dict(torch.load(pretrained_model, map_location = 'cpu'))
```

<All keys matched successfully>

```
1 # MNIST dataset and dataloader
2 # (For testing only, we will use a pre-trained model)
3 ds2 = datasets.MNIST('./data', train = True, download = True, transform = tf)
4 train_loader = torch.utils.data.DataLoader(ds2, batch_size = 64, shuffle = True)
```

```
1 print(len(train_loader))
```

938

```
1 # Define a loss function and an optimizer for training
2 criterion = nn.CrossEntropyLoss()
3 optimizer = optim.SGD(model2.parameters(), lr = 0.001, momentum = 0.9)
```

```
1 # Setting temperature to 100 for training
2 model2.T = 100
```


Defended model

- The defended model, used for comparison will be a copy of the undefended model.
- We will then change the temperature to $T = 100$ (arbitrarily chosen).
- And will continue the training of this model on the training set, with this new “high” temperature.

```
1 # Load the pretrained model
2 model2 = Net().to(device)
3 pretrained_model = "./mnist_model.data"
4 model2.load_state_dict(torch.load(pretrained_model, map_location = 'cpu'))
```

<All keys matched successfully>

```
1 # MNIST dataset and dataloader
2 # (For testing only, we will use a pre-trained model)
3 ds2 = datasets.MNIST('./data', train = True, download = True, transform = tf)
4 train_loader = torch.utils.data.DataLoader(ds2, batch_size = 64, shuffle = True)
```

```
1 print(len(train_loader))
```

938

```
1 # Define a loss function and an optimizer for training
2 criterion = nn.CrossEntropyLoss()
3 optimizer = optim.SGD(model2.parameters(), lr = 0.001, momentum = 0.9)
```

```
1 # Setting temperature to 100 for training
2 model2.T = 100
```

Defended model

- The defended model, used for comparison will be a copy of the undefended model.
- We will then change the temperature to $T = 100$ (arbitrarily chosen).
- If we want, we can continue the training of this model, with this new “high” temperature, to check it leads to very little changes in model parameters (gradients are close to zero).

```
1 def retrain(model, train_loader, optimizer, criterion, n_iter = 10):
2
3     # This will make prints happen every 50 mini-batches
4     mod_val = 50
5
6     # Train over n_iter epochs
7     for epoch in range(n_iter):
8
9         # Keep track of the running losses over batches
10        running_loss_normal = 0.0
11
12        for i, data in enumerate(train_loader):
13
14            # Retrieve input images and labels
15            inputs, labels = data
16            inputs.requires_grad = True
17
18            # Zeroing gradients
19            optimizer.zero_grad()
20
21            # Forward, Loss, Backprop, Optimize
22            outputs = model(inputs)
23            loss = criterion(outputs, labels)
24            loss.backward()
25            optimizer.step()
```

Defended model

- The defended model, used for comparison will be a copy of the undefended model.
- We will then change the temperature to $T = 100$ (arbitrarily chosen).
- If we want, we can continue the training of this model, with this new “high” temperature, to check it leads to very little changes in model parameters (gradients are close to zero).

```
1 retrain(model2, train_loader, optimizer, criterion, n_iter = 50)
```

```
[Epoch 1, Batch 51] Loss: 2.258
[Epoch 1, Batch 101] Loss: 2.256
[Epoch 1, Batch 151] Loss: 2.254
[Epoch 1, Batch 201] Loss: 2.252
[Epoch 1, Batch 251] Loss: 2.250
[Epoch 1, Batch 301] Loss: 2.248
[Epoch 1, Batch 351] Loss: 2.245
[Epoch 1, Batch 401] Loss: 2.243
[Epoch 1, Batch 451] Loss: 2.241
[Epoch 1, Batch 501] Loss: 2.238
[Epoch 1, Batch 551] Loss: 2.236
[Epoch 1, Batch 601] Loss: 2.233
[Epoch 1, Batch 651] Loss: 2.230
[Epoch 1, Batch 701] Loss: 2.227
[Epoch 1, Batch 751] Loss: 2.223
[Epoch 1, Batch 801] Loss: 2.220
[Epoch 1, Batch 851] Loss: 2.216
[Epoch 1, Batch 901] Loss: 2.212
[Epoch 2, Batch 51] Loss: 2.122
```

```
1 # After retraining, set the model in evaluation mode
2 # (Important, because we have dropout layers!)
3 model2.eval()
```

Defended model performance

- The high temperature makes the gradients vanish.
- This means that the gradient based attack will now be unable to operate correctly on our defended model!

$$\tilde{x} = x + \epsilon \nabla_x L(x, \theta, c)$$

$$\text{But } \nabla_x L(x, \theta, c) \approx 0$$

$$\text{Implies } \tilde{x} \approx x$$

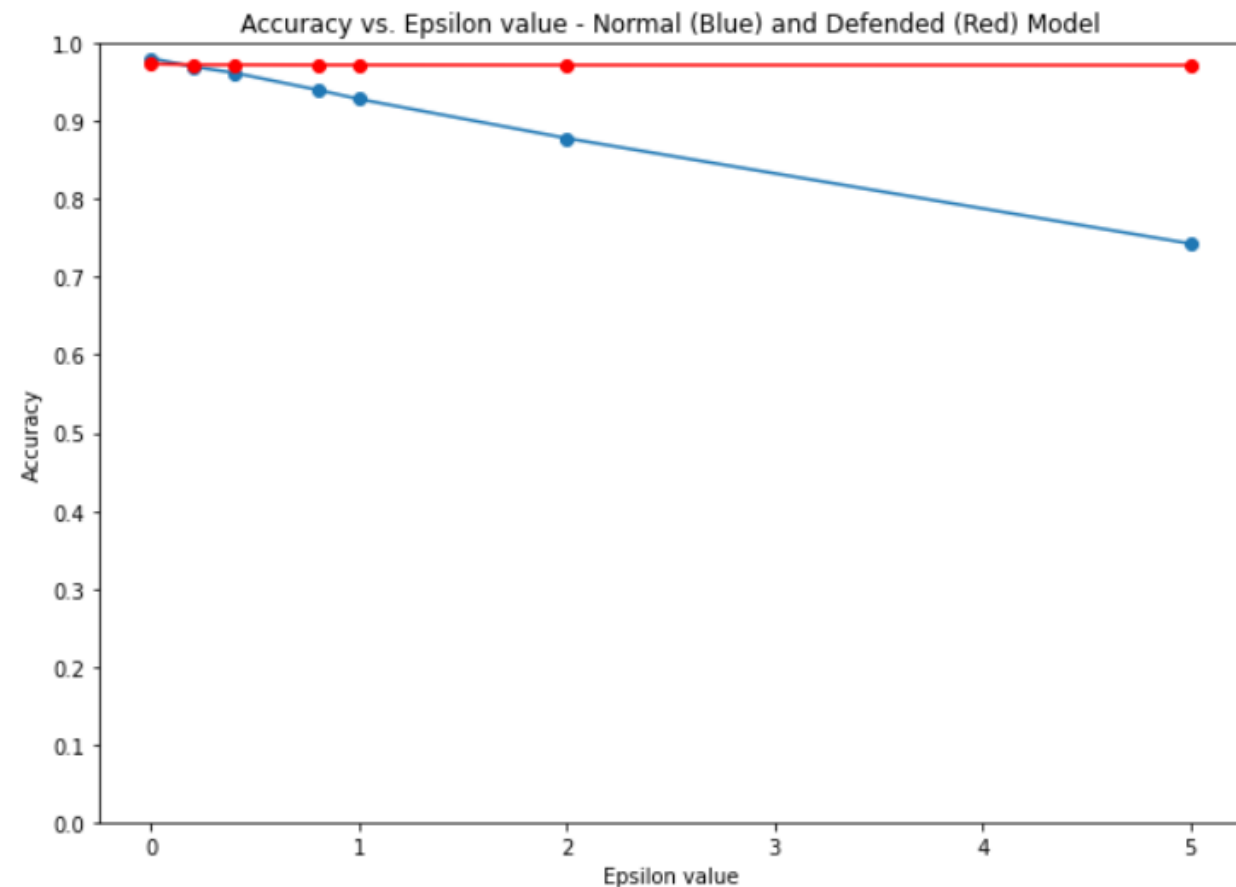
Defended model performance

- The high temperature makes the gradients vanish.
- This means that the gradient based attack will now be unable to operate correctly on our defended model!

$$\tilde{x} = x + \epsilon \nabla_x L(x, \theta, c)$$

$$\text{But } \nabla_x L(x, \theta, c) \approx 0$$

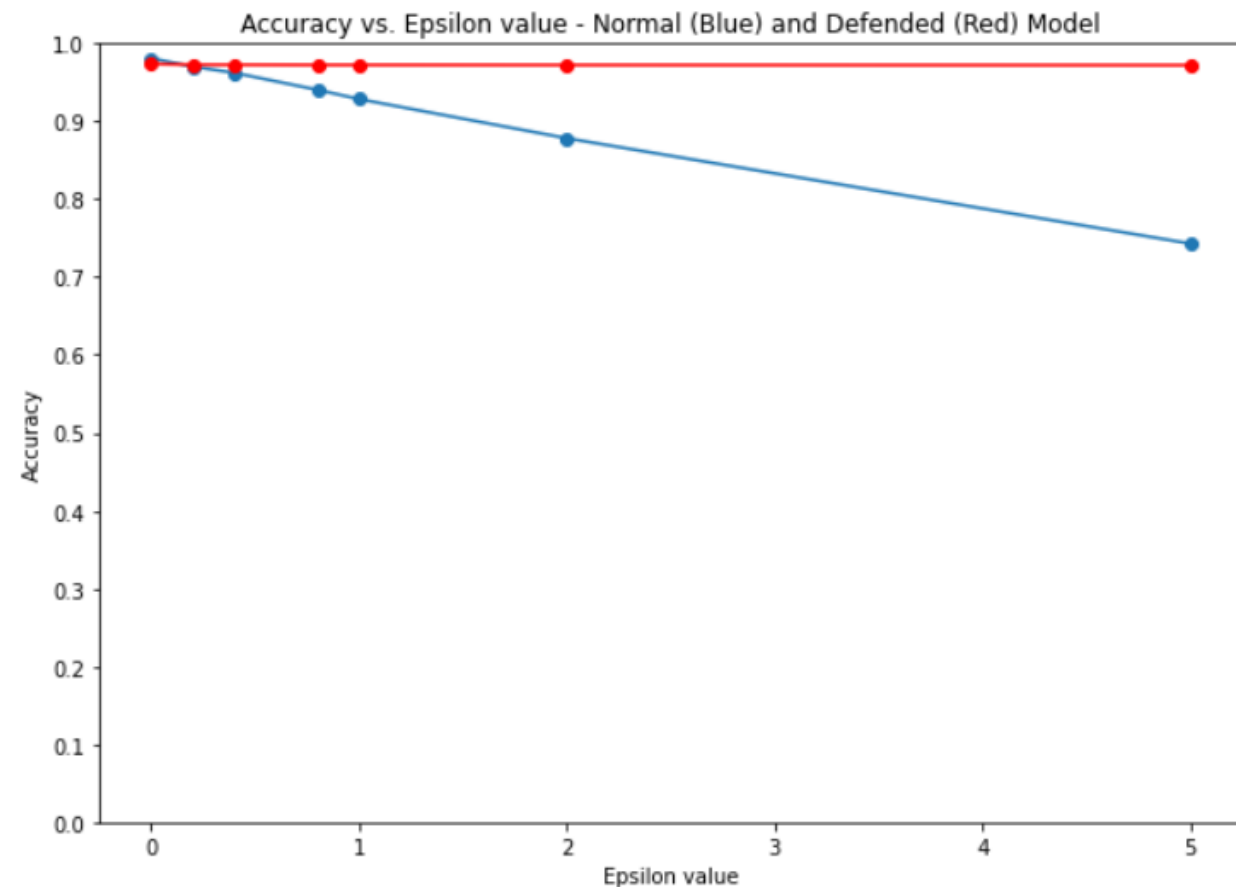
$$\text{Implies } \tilde{x} \approx x$$



Defended model performance

- The high temperature makes the gradients vanish

Open question...
Will defensive distillation work on the FGSM attack, which only uses the SIGN of the gradient?



The current state of our defenses against white-box models

So far, defenders have only two options to defend against gradient-based attacks.

1. Either train specifically in anticipation for these attacks (and we have seen it works quite well on FGSM attacks).
2. Or make the gradients vanish to mess with the attackers (and we have seen it works quite well on gradient-based attacks).

The current state of our defenses against white-box models

So far, defenders have only two options to defend against gradient-based attacks.

1. Either train specifically in anticipation for these attacks (and we have seen it works quite well on FGSM attacks).
2. Or make the gradients vanish to mess with the attackers (and we have seen it works quite well on gradient-based attacks).

That should be good enough, right?

The current state of our defenses against white-box models

So far, defenders have only two options to defend against gradient-based attacks.

1. Either train specifically in anticipation for these attacks (and we have seen it works quite well on FGSM attacks).
2. Or make the gradients vanish to mess with the attackers (and we have seen it works quite well on gradient-based attacks).

That should be good enough, right?

I mean, there is now way for attackers to come up with something else, to attack my model, right?

Carlini-Wagner attack (targeted, iterated, white-box)

- The Carlini-Wagner attack is a targeted, (one-shot or) iterated, white-box attack [Carlini2017].

Carlini-Wagner attack (targeted, iterated, white-box)

- The Carlini-Wagner attack is a **targeted**, (**one-shot** or) **iterated**, **white-box** attack [Carlini2017].
- It uses some of the model properties to perform create adversarial samples.
 - It does not use the gradients...
 - It uses the logits of the model instead.
 - This means that the defensive distillation or the arms race defense **will not work** against this attack.

Carlini-Wagner attack (targeted, iterated, white-box)

- The Carlini-Wagner attack is a **targeted**, (**one-shot** or) **iterated**, **white-box** attack [Carlini2017].
- It uses some of the model properties to perform create adversarial samples.
 - It does not use the gradients...
 - It uses the **logits** of the model instead.
 - This means that the defensive distillation or the arms race defense **will not work** against this attack.
- This attack generates samples by solving a “**simple**” **optimization problem**.
- It also ensures **plausibility** by controlling a norm $\|\tilde{x} - x\|$ between the generated sample and the original image.

Carlini-Wagner optimization

Objective: generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the unconstrained multi-objective optimization problem below.

$$\min_u \left(\|u - x\|_2 + \alpha \max(d(u), 0) \right)$$

with $d(u) = \max_{t \neq \tilde{c}} (f_t(u) - f_{\tilde{c}}(u))$

Carlini-Wagner optimization

Objective: generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the unconstrained multi-objective optimization problem below.

$$\min_u \left(\|u - x\|_2 + \alpha \max(d(u), 0) \right) \\ \text{with } d(u) = \max_{t \neq \tilde{c}} (f_t(u) - f_{\tilde{c}}(u))$$

- The objective to generate the best image u , which **minimizes** our **objective function**.
- The **first part** penalizes images u , which are far from the original image x .
(this encourages **plausibility**).
- The **second part** is a positive non-zero value if the image u does not have the target label, and zero otherwise.

Carlini-Wagner optimization

Objective: generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the unconstrained multi-objective optimization problem below.

$$\min_u \left(\|u - x\|_2 + \alpha \max(d(u), 0) \right)$$

with $d(u) = \max_{t \neq \tilde{c}} (f_t(u) - f_{\tilde{c}}(u))$

- Indeed, we will have $d(u) > 0$ if and only if
- there exists a class $t \neq \tilde{c}$, such that $f_t(u) - f_{\tilde{c}}(u) > 0$
- Equivalently, this means that image u is not of target class \tilde{c} .

Carlini-Wagner optimization

Objective: generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the unconstrained multi-objective optimization problem below.

$$\min_u \left(\|u - x\|_2 + \alpha \max(d(u), 0) \right)$$

with $d(u) = \max_{t \neq \tilde{c}} (f_t(u) - f_{\tilde{c}}(u))$

- The **first part** penalizes images u , which are far from the original image x .
(this encourages **plausibility**).
- The **second part** is non-zero if the image u does not have the target label.
(this encourages **efficacy**).
- Both important aspects are covered, and α can be arbitrarily set to decide on the importance of each aspect.

Carlini-Wagner optimization

Roughly equivalent formulation #1:

generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the **constrained** optimization problem below.

$$\min_u \left(\max(d(u), 0) \right)$$

$$\text{with } d(u) = \max_{t \neq \tilde{c}} (f_t(u) - f_{\tilde{c}}(u))$$

$$\text{and } \|u - x\|_2 \leq \alpha$$

Roughly equivalent formulation #2:

generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the **constrained** optimization problem below.

$$\min_u \left(\max(d(u), 0) \right)$$

$$\text{with } d(u) = \max_{t \neq \tilde{c}} (f_t(u) - f_{\tilde{c}}(u))$$

$$\text{and } \|u - x\|_\infty \leq \alpha$$

Carlini-Wagner optimization

Roughly equivalent formulation #1:

generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the **constrained** optimization problem below.

$$\min_u \left(\max(d(u), 0) \right)$$

Roughly equivalent formulation #2:

generate a sample \tilde{x} with class \tilde{c} , from a sample x of class c . Do so by solving the **constrained** optimization problem below.

$$\min_u \left(\max(d(u), 0) \right)$$

Note: Solving these optimization problems can prove to be challenging. Implementation is not necessarily difficult, but cumbersome. We leave the implementation of these methods out of the scope of this class (Have a look at the papers with code website!)

Interested in implementing advanced algos? Look them up on PapersWithCode first!

Towards Evaluating the Robustness of Neural Networks

[Edit social preview](#)

16 Aug 2016 • [Nicholas Carlini](#) • [David Wagner](#)

Neural networks provide state-of-the-art results for most machine learning tasks. Unfortunately, neural networks are vulnerable to adversarial examples: given an input x and any target classification t , it is possible to find a new input x' that is similar to x but classified as t ... [\(read more\)](#)

[PDF](#)[Abstract](#)

Code

[Edit](#)[carlini/nn_robust_attacks](#)

★ 560

TensorFlow

[MadryLab/cifar10_challenge](#)

★ 315

TensorFlow

[LeMinhThong/blackbox-attack](#)

★ 51

PyTorch

[kkew3/pytorch-cw2](#)

★ 38

PyTorch

[inspire-group/advml-traffic-sign](#)

★ 16

[See all 22 implementations](#)

<https://paperswithcode.com/paper/towards-evaluating-the-robustness-of-neural>

Black-boxing your model for defense

Definition (**Black-boxing** defense):

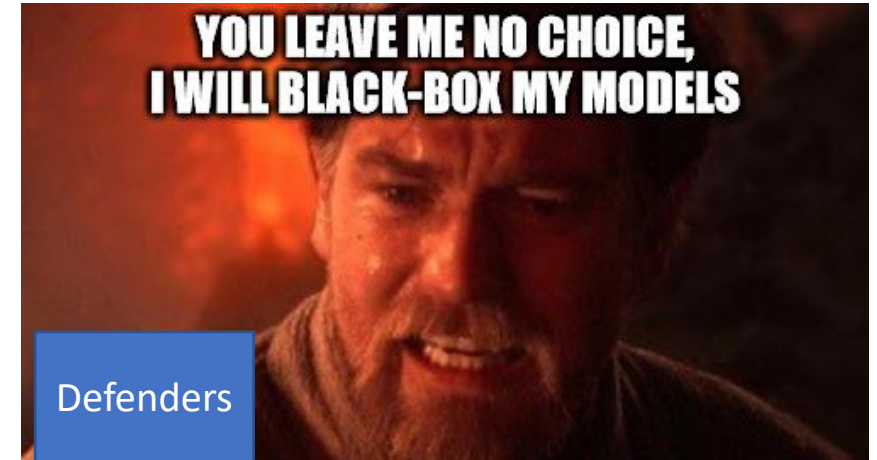
The **black-boxing** defense refers to the concept of hiding your model parameters to prevent attackers from accessing and using some of these parameters (neither weights, gradients nor logits) in their attacks.

Black-boxing your model for defense

Definition (**Black-boxing** defense):

The **black-boxing** defense refers to the concept of hiding your model parameters to prevent attackers from accessing and using some of these parameters (neither weights, gradients nor logits) in their attacks.

The only thing attackers can do is submit inputs to and get outputs from the model.



Black-boxing your model for defense

Definition (**Black-boxing** defense):

The **black-boxing** defense refers to the concept of hiding your model parameters to prevent attackers from accessing and using some of these parameters (neither weights, gradients nor logits) in their attacks.

The only thing attackers can do is submit inputs to and get outputs from the model.



The surrogate attack

Definition (the **surrogate** attack):

The **surrogate** attack (from [Papernot2017]) is a **black-box** attack.

Its objective is to recreate a **copy** g of the **model** f being attacked.

Then, produce attack samples using your **white-box** attacks we mentioned earlier on your recreated model g . **If they work on g , use them on the model f .**

The surrogate attack

Definition (the **surrogate** attack):

The **surrogate** attack (from [Papernot2017]) is a **black-box** attack.

Its objective is to recreate a **copy** g of the **model** f being attacked.

Then, produce attack samples using your **white-box** attacks we mentioned earlier on your recreated model g . **If they work on g , use them on the model f .**

Step 1: training a **surrogate model** g

- Use a typical dataset, but do not use the ground truth labels. Instead use the labels produced by the predictions of model f .
- Train the **surrogate model** g to match the predictions of the original model f .
- A good loss function would be:

$$L(g(x), f) = \sum_c -f_c(x) \log(g_c(x))$$

The surrogate attack

Definition (the **surrogate** attack):

The **surrogate** attack (from [Papernot2017]) is a **black-box** attack.

Its objective is to recreate a **copy g** of the **model f being attacked**.

Then, produce attack samples using your **white-box** attacks we mentioned earlier on your recreated model g . **If they work on g , use them on the model f .**

Step 2: use **white-box** attacks on your **surrogate model g** .

- For instance, attack your **surrogate model g** with FGSM.
If the attack sample \tilde{x} manages to fool the **surrogate model g** , then odds are it will probably work on the original model f as well.
- Ultimately, this depends on how close you managed to make your **surrogate model g** similar to the original model f .

The surrogate attack

Definition (the surrogate attack):

The **surrogate attack** (from [Papernot2017]) is a **black-box** attack

Step 2: use **white-box** attacks on your model g .

- For instance, attack your model g with EGSIM. If the attack sample \tilde{x}

Note: Implementation is not necessarily difficult, but has to be done cautiously. Its guided implementation might be part of the next homework! More on this later.

If you are curious, training a surrogate model and attacking it was a Homework in 2023. See attached folder for Homework Notebook (and solutions, maybe)!

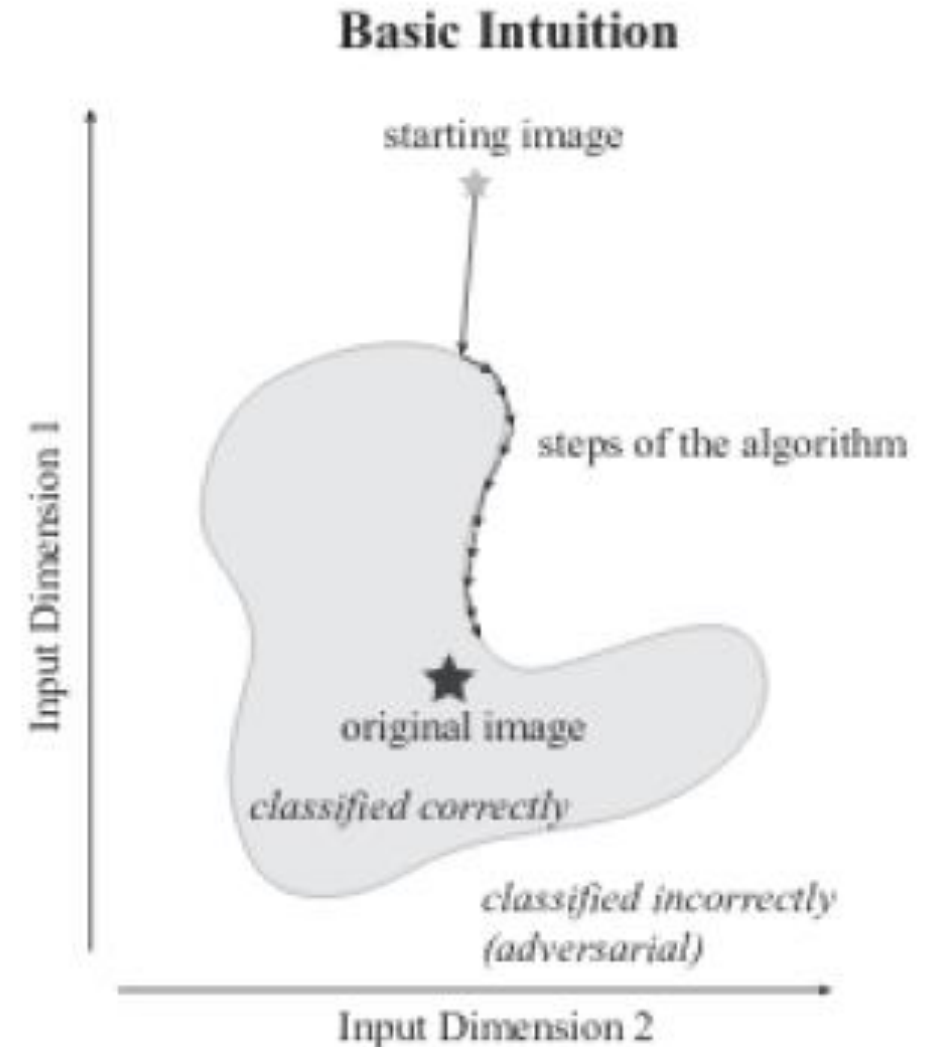
not always essential but helps.

The boundary attack

Definition (the boundary attack):

The **boundary attack** (from [Brendel2018]) is a **black-box** type of attack, which only requires access to the inputs and outputs of the model under attack f .

It attempts to create an image, which will look plausible, but will be misclassified. It does so by walking on the boundary of the correct prediction region, and attempts to minimize $\|\tilde{x} - x\|$.



The boundary attack

Step 1: Start with a non-adversarial image x (does not have to belong in the original dataset), which is correctly predicted as c .

Make another image x_0 , by (heavily) noising your original image x .

This image x_0 should be misclassified.

The boundary attack

Step 2: modify your image x_n for instance by using random noising, with low amplitude.

If the new image x_{n+1} is incorrectly classified and verifies

$\|x_{n+1} - x\| \leq \|x_n - x\|$, then keep x_{n+1} .

The boundary attack

Step 2: modify your image x_n for instance by using random noising, with low amplitude.

If the new image x_{n+1} is incorrectly classified and verifies

$\|x_{n+1} - x\| \leq \|x_n - x\|$, then keep x_{n+1} .

Otherwise reuse the previous image, i.e. $x_{n+1} = x_n$.

The boundary attack

Step 2: modify your image x_n for instance by using random noising, with low amplitude.

If the new image x_{n+1} is incorrectly classified and verifies

$\|x_{n+1} - x\| \leq \|x_n - x\|$, then keep x_{n+1} .

Otherwise reuse the previous image, i.e. $x_{n+1} = x_n$.

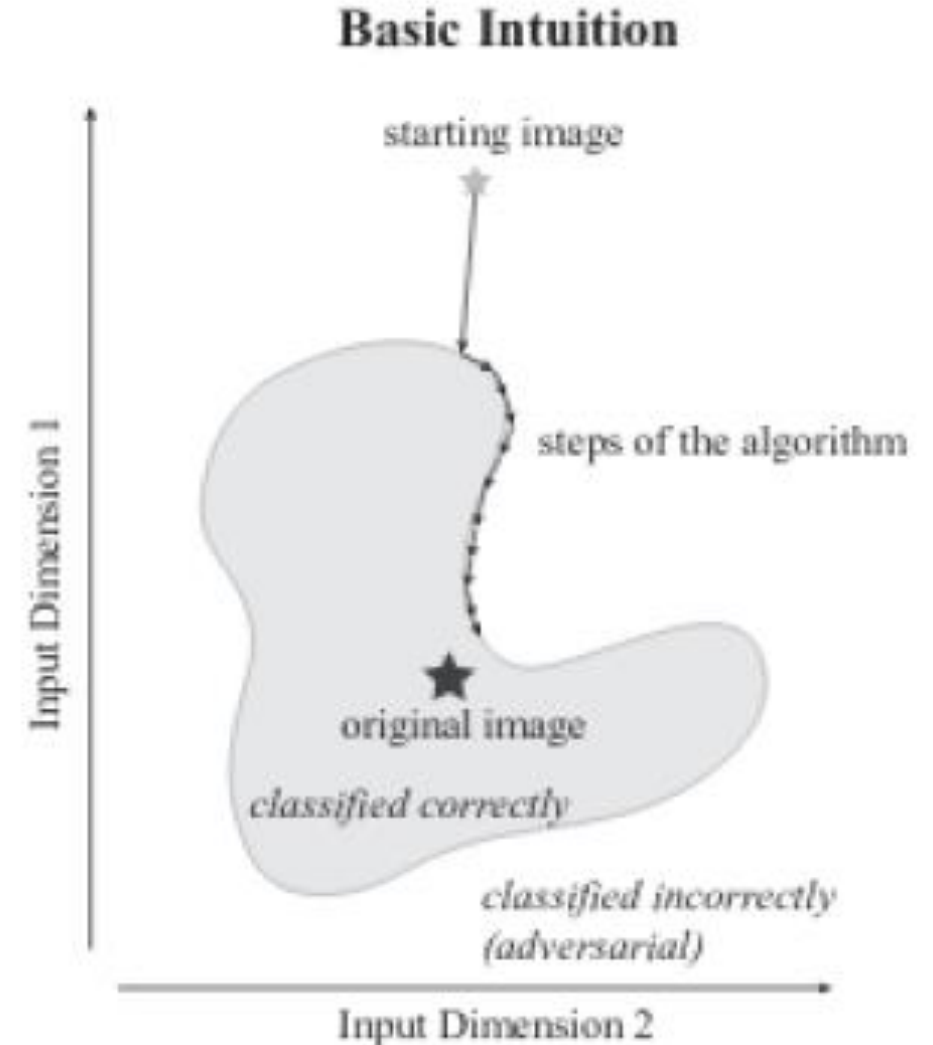
Step 3: repeat step 2, until a maximal number of iterations N is reached.

The boundary attack

Definition (the boundary attack):

The **boundary attack** (from [Brendel2018]) is a **black-box** type of attack, which only requires access to the inputs and outputs of the model under attack f .

It attempts to create an image, which will look plausible, but will be misclassified. It does so by walking on the boundary of the correct prediction region, and attempts to minimize $\|\tilde{x} - x\|$



The boundary attack

Definition (the boundary attack):

The **boundary attack** (from [Brendel2018]) is a **black-box** type of attack, which only requires access to the inputs and outputs of the model under attack f .

It attempts to create an image, which will look plausible, but will be misclassified. It does so by walking on the boundary of the correct prediction region, and attempts to minimize $\|\tilde{x} - x\|$

Step 3: repeat step 2, until a maximal number of iterations N is reached.

The boundary attack

Definition (the boundary attack):

The **boundary attack** (from [Brendel2018]) is a **black-box** type of attack, which only requires access to the inputs and outputs of the model under attack f .

It attempts to create an image, which will look plausible, but will be misclassified. It does so by walking on the boundary of the correct prediction region, and attempts to minimize $\|\tilde{x} - x\|$

Additional notes

- You can progressively decrease the noise amplitude when generating x_{n+1} from x_n (in a similar fashion as with the learning rate scheduling).
- Add an extra condition, being that your x_n must be of a certain target class \tilde{c} to make it a targeted attack.

The boundary attack

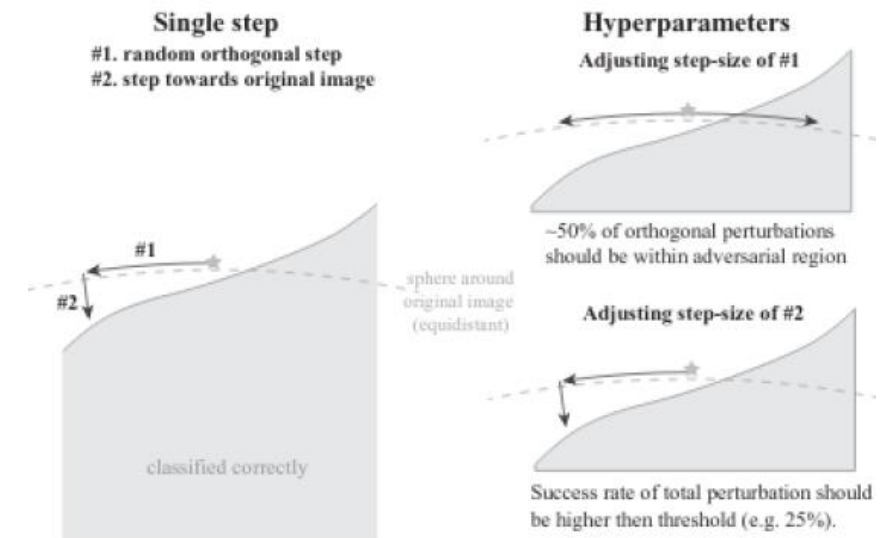
Definition (the boundary attack):

The **boundary attack** (from [Brendel2018]) is a **black-box** type of attack, which only requires access to the inputs and outputs of the model under attack f .

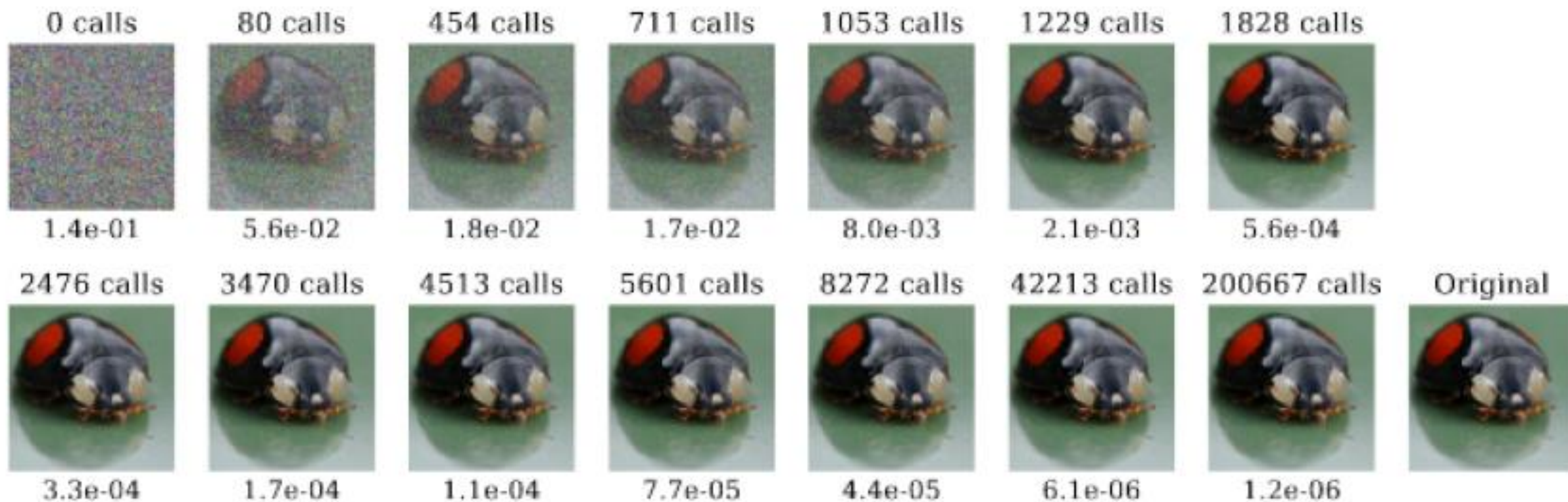
It attempts to create an image, which will look plausible, but will be misclassified. It does so by walking on the boundary of the correct prediction region, and attempts to minimize $\|\tilde{x} - x\|$

Additional notes

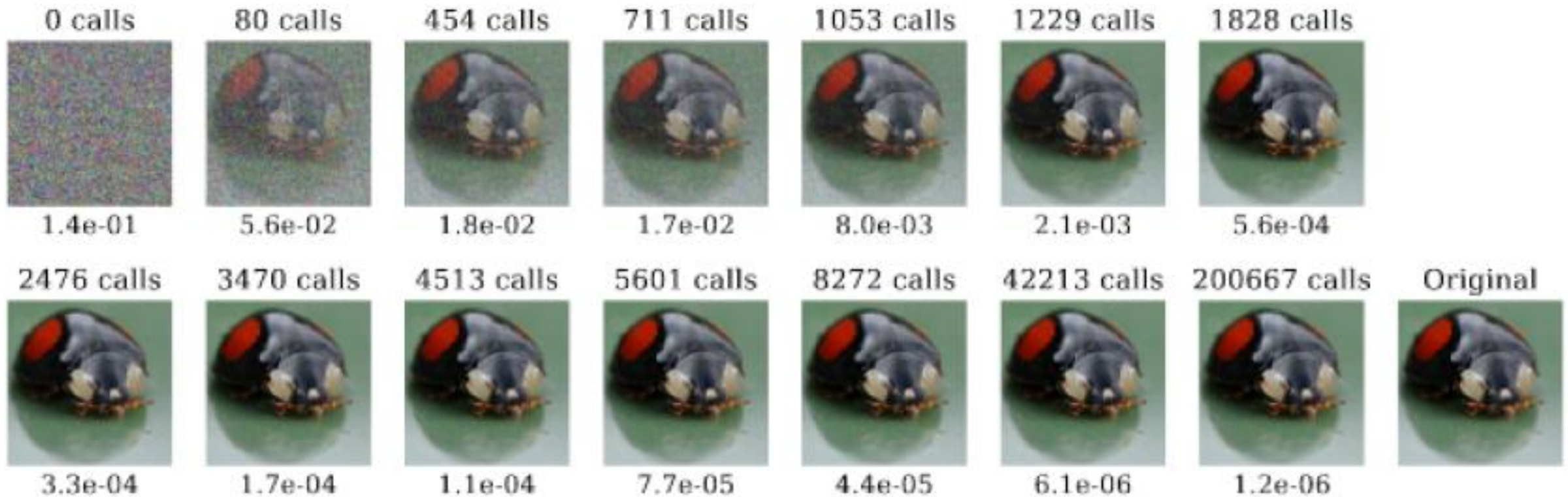
- More advanced methods may even try and guess the local geometry of the boundary to make better iterations on the x_n images.



The boundary attack in action



The boundary attack in action



Note: Implementation is not necessarily difficult, but cumbersome.
We leave the implementation of these methods out of the scope of this class (Have a look at the papers with code website!)

Black-boxing your model for defense (v2.0!).

Definition (Black-boxing** defense, version 2.0!):**

The **black-boxing** defense refers to the concept of hiding your model parameters to prevent attackers from accessing and using some of these parameters (neither weights, gradients nor logits) in their attacks.

~~The only thing attackers can do is submit inputs and get outputs from the model.~~

To make the model as black as possible, we can forbid the attacker from even **ACCESSING THE MODEL!**

Defending by ensemble-ing your model

Definition (The **ensemble** defense):

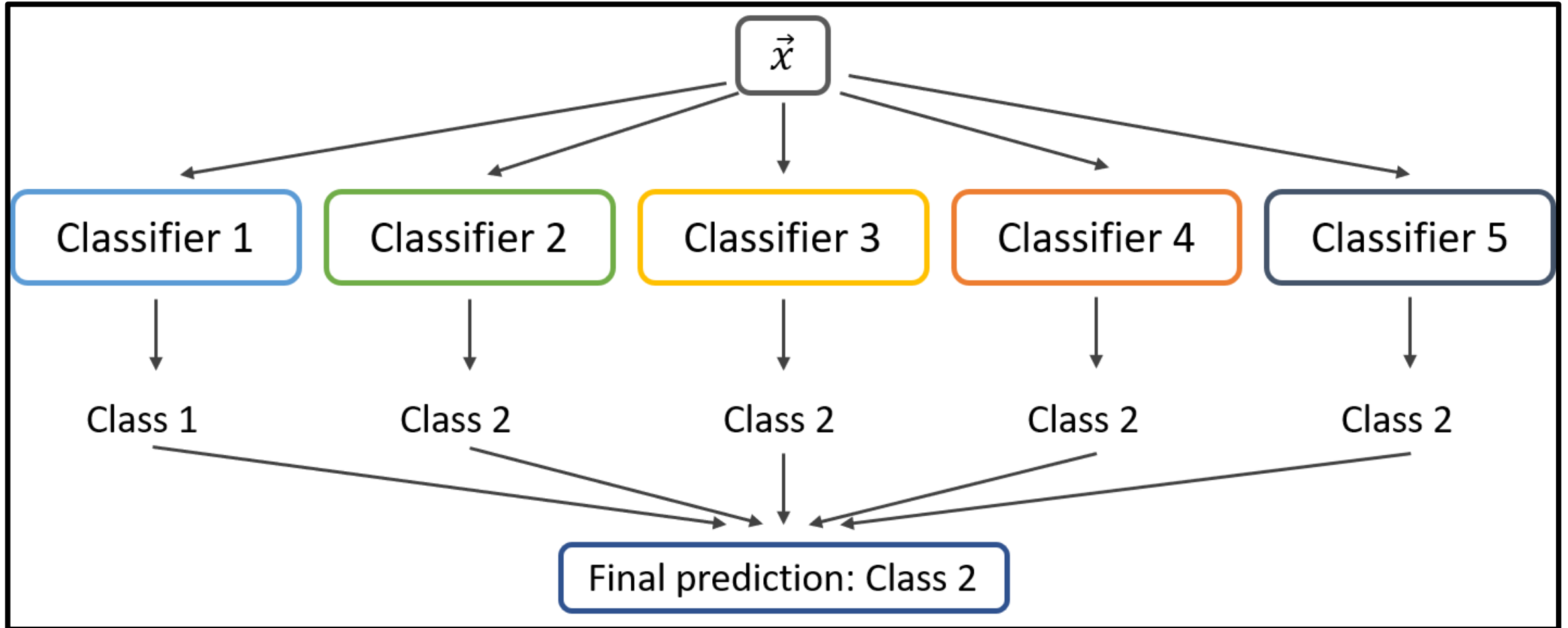
The **ensemble** defense refers to the concept of using multiple models to prevent attackers from making your classifier malfunction.

Each model is trained with a different initialization, hence leading to different weights, gradients, etc.

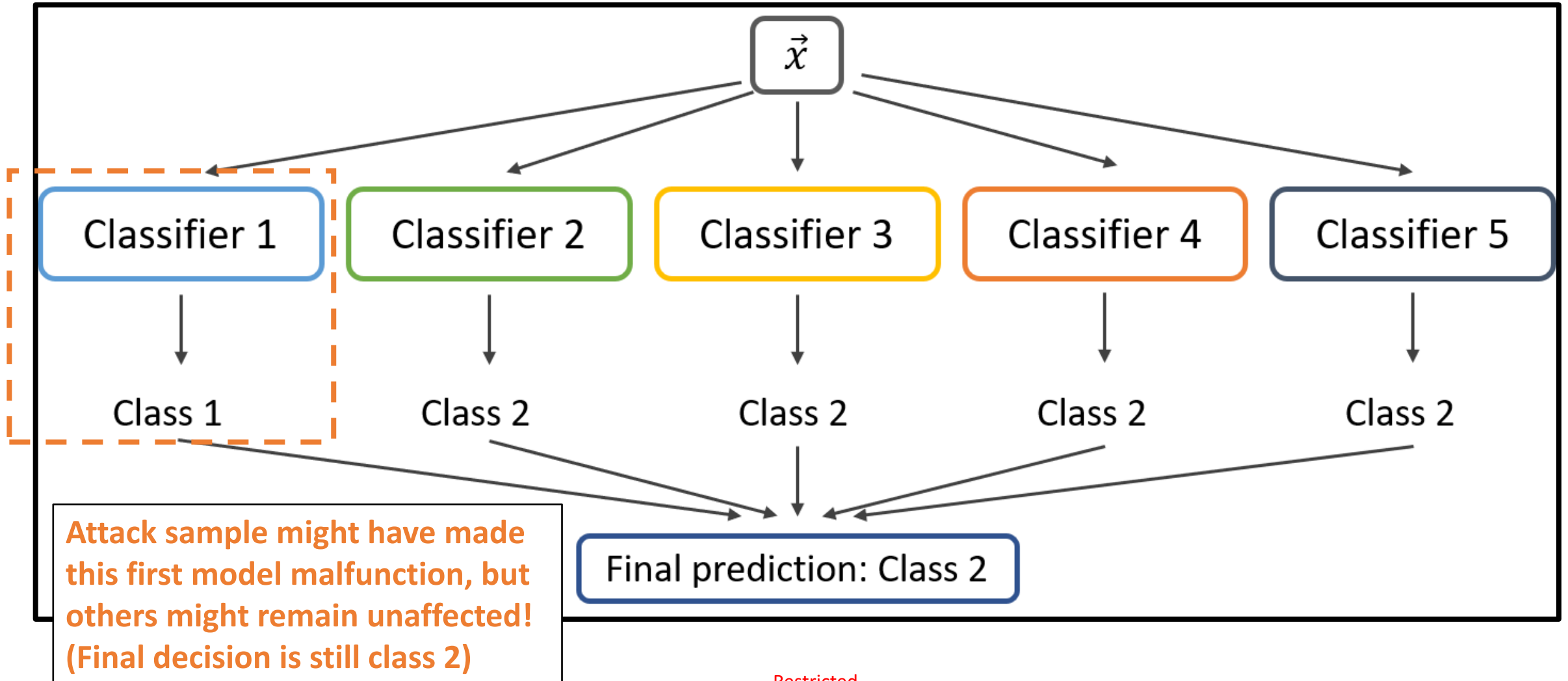
During testing, each image is passed to each different model, and a list of predictions is obtained (one for each model). The label, which appears the most often is the one returned by our ensemble classifier.

This makes the life of attackers extra difficult, as they have to successfully attack multiple models, with the same sample to make the prediction wrong.

Defending by ensemble-ing your model



Defending by ensemble-ing your model



Maximal defense

- Nothing more defenders can really do at this point. They have kept their model secret and “ensembled” it.
- The attackers only know that the defender has a model, which is trying to accomplish a certain task, but have no access to it.
- Defenders have even used multiple instances of the model to get an ensemble classifier.
- This covers for attacks that would make one model malfunction.

Attackers obviously lost, right?

On the transferability of attacks

Attackers obviously lost, right?

Well... Yes and no.

- While it may seem very difficult to make an **ensemble, hidden** model malfunction...
- Recent papers have shown that attacks might be **transferable** [Liu2017].
- **Note:** this is what we exploited for our **surrogate attack**, in the first place.
- Unfortunately, this means an attack, which makes a certain facial recognition algorithm fail, has great chances to make another model, even different, fail as well.
- And even more true for models within a same ensemble.

Conclusion (W5S3)

- Training and attacking a model seem to be the two faces of a same coin.
- Being able to do one, unfortunately exposes you to the second one.
- Both attackers and defenders have many options up their sleeve. But attackers seem to have a clear advantage.
- Attackers are able to exploit
 - Intrinsic information from models.
 - But more importantly, fundamental limits of neural networks.
- It is a very active research field, with many more advanced concepts, for obvious reasons, as any “sensitive” application will require some kind of defense.

Learn more about these topics

- [Madry2017] **Madry** et al., “Towards Deep Learning Models Resistant to Adversarial Attacks”, 2017
<https://arxiv.org/abs/1706.06083>
- [Papernot2015] **Papernot** et al., “Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks”, 2015
<https://arxiv.org/abs/1511.04508>
- [Dong2017] Dong et al., “Boosting Adversarial Attacks with Momentum”, 2017.
<https://arxiv.org/abs/1710.06081>
- [Carlini2017] **Carlini** and **Wagner**, “Towards Evaluating the Robustness of Neural Networks”, 2017.
<https://arxiv.org/abs/1608.04644>

Learn more about these topics

- [Papernot2017] **Papernot** et al., “Practical Black-Box Attacks against Machine Learning”, 2017.
<https://arxiv.org/abs/1602.02697>
- [Brendel2018] Brendel et al., “Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models”, 2018.
<https://arxiv.org/abs/1712.04248>
- [Liu2017] Liu et al., “Delving into Transferable Adversarial Examples and Black-box Attacks”, 2017.
<https://arxiv.org/abs/1611.02770>

Interested in implementing advanced algos? Look them up on PapersWithCode first!

Towards Evaluating the Robustness of Neural Networks

[Edit social preview](#)

16 Aug 2016 • [Nicholas Carlini](#) • [David Wagner](#)

Neural networks provide state-of-the-art results for most machine learning tasks. Unfortunately, neural networks are vulnerable to adversarial examples: given an input x and any target classification t , it is possible to find a new input x' that is similar to x but classified as t ... [\(read more\)](#)

[PDF](#)[Abstract](#)

Code

[Edit](#)

carlini/nn_robust_attacks	★ 560	TensorFlow
MadryLab/cifar10_challenge	★ 315	TensorFlow
LeMinhThong/blackbox-attack	★ 51	PyTorch
kkew3/pytorch-cw2	★ 38	PyTorch
inspire-group/advml-traffic-sign	★ 16	

[See all 22 implementations](#)

Tasks

[Edit](#)

ADVERSARIAL ATTACK

<https://paperswithcode.com/paper/towards-evaluating-the-robustness-of-neural>

Learn more about these topics

Out of class, for those of you who are curious

- [Medium1] On the ongoing race of coming up with new attacks.
<https://medium.com/mlreview/the-intuition-behind-adversarial-attacks-on-neural-networks-71fdd427a33b>
- [Medium2] On why security systems using deep learning are potentially under threat.
<https://medium.com/@chami.soufiane/security-threats-against-machine-learning-based-systems-a-real-concern-2515115c88e4>

Learn more about these topics

Out of class, for those of you who are curious

- [YTB1] Generating adversarial patches against YOLOv2, a.k.a. printing patterns to fool AIs and become invisible.
<https://www.youtube.com/watch?v=MlbFvK2S9g8>
- [Stanford] Stanford lecture, by Ian **Goodfellow** @ Stanford on attacks and defense mechanisms for Neural Networks (long!)
https://www.youtube.com/watch?v=ClfsB_EYsVI

Learn more about these topics

More fun stuff

- [Eykholt2018] Eykholt et al. “Robust Physical-World Attacks on Deep Learning Models”, 2018.
Printing stickers to put on your tee-shirts to fool facial recognition systems.
<https://arxiv.org/abs/1707.08945>
- [Moosavi-Dezfooli2016] **Moosavi-Dezfooli** et al., “DeepFool: a simple and accurate method to fool deep neural networks”, 2016.
A toolbox with previously mentioned attacks and more advanced attacks.
<https://arxiv.org/abs/1511.04599>
- [Athalye2018] Athalye et al., “Synthesizing Robust Adversarial Examples”, 2018.
3D printing misclassified stuff.
<https://arxiv.org/abs/1707.07397>

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Aleksander Madry: Professor at MIT**, Director of the MIT Center for Deployable Machine Learning and a Faculty Co-Lead of the MIT AI Policy Forum.

<http://madry-lab.ml/>

<https://people.csail.mit.edu/madry/>

<https://scholar.google.ch/citations?user=SupjsEUAAAJ&hl=en>

- **Nicolas Papernot: Assistant Professor at University of Toronto.**

<https://www.papernot.fr>

<https://scholar.google.com/citations?user=cGxq0cMAAAAJ&hl=en>

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Nicholas Carlini**: Research Scientist at Google Brain.

<https://nicholas.carlini.com>

<https://scholar.google.com/citations?user=q4qDvAoAAAAJ&hl=en>

- **David Wagner**: Professor at UC Berkeley.

<https://people.eecs.berkeley.edu/~daw/papers>

<https://scholar.google.com/citations?user=67kghxAAAAAJ&hl=en>