50.039 Theory and Practice of Deep Learning
W8-S1 Transition

Matthieu De Mari, Berrak Sisman

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About me

- **Dr. Matt** (Matthieu) **DE MARI**
- Lecturer at SUTD (Python, Deep Learning, AI, and more)
- Information Systems Technology and Design (ISTD) pillar/faculty
- PhD from CentraleSupelec (France)

- Email: matthieu_demari@sutd.edu.sg
- Office @ SUTD: 1.702.27

Recap: Weeks 1-6

- **Week 1:** General introduction, linear model for regression and classification, recap on gradients for machine learning.
- **Week 2:** Introduction to PyTorch, classification using logistic regression. Basic concepts of Neural Networks.
- **Week 3:** Backpropagation, initialization and data augmentation. Convolutional Neural Networks. PyTorch practice.
- **Week 4:** Advanced optimizers and better ways to apply gradients. Transfer learning, brief state-of-the-art on computer vision architectures (ResNets, etc.). More PyTorch practice.
- **Week 5:** Recurrent neural networks (RNNs) architectures.
- **Week 6:** Conference. Midterm exam.

What is coming next?

At this point, you have learnt the basics of Deep Learning, i.e., what most online courses teach.

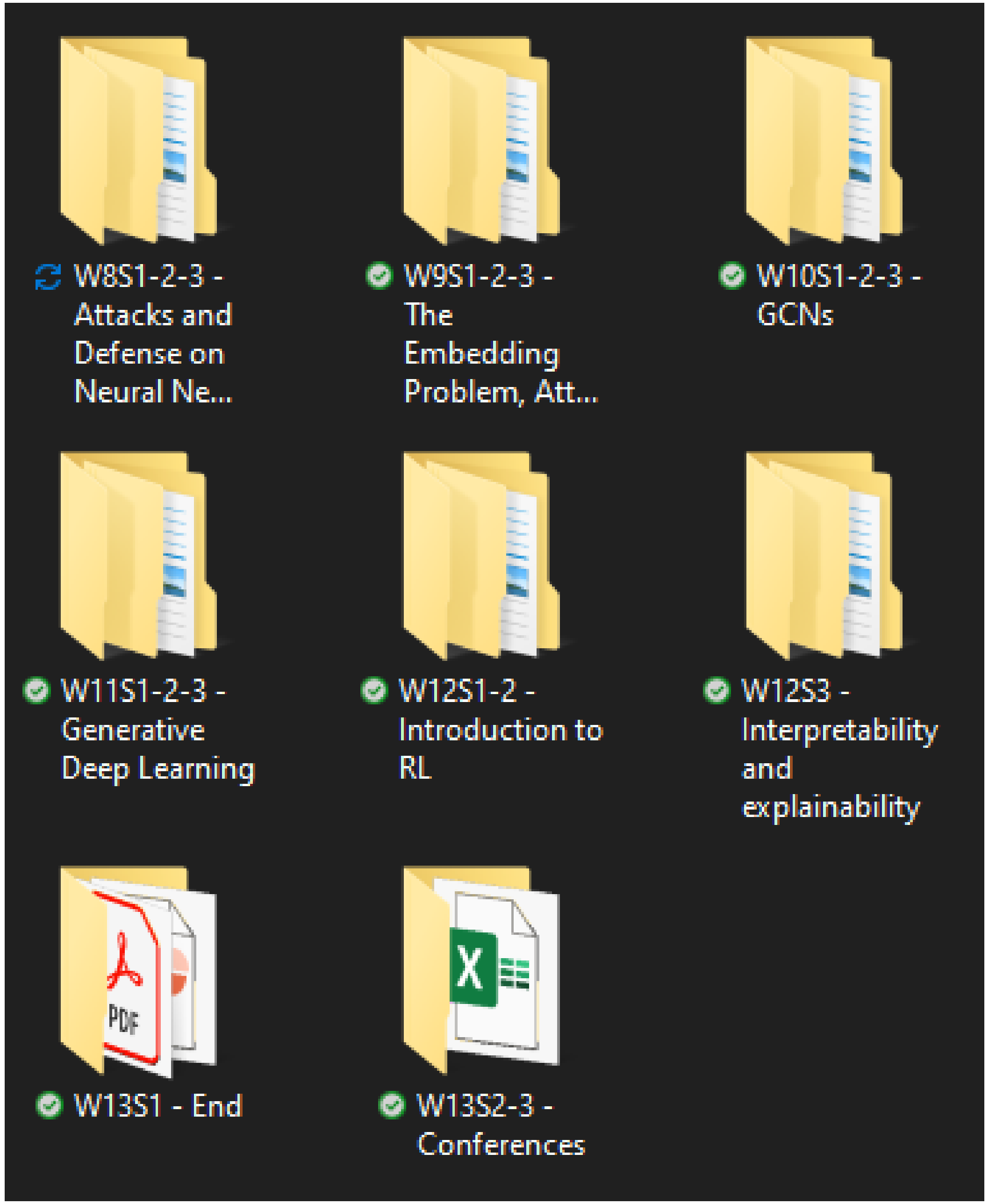
Advanced concepts are often missing online, as they are

- open questions in research,
- too niche for an online course
- or simply not yet prepared into an online course.

In this second half of the 50.039 DL course, we propose to discuss some of these advanced concepts and research directions.

The way I teach things

- Each week has a specific topic.
- Notebooks demos that you can reuse for practice (**autonomy is key**).
- Discussions about more advanced concepts (often out-of-scope).
- Extra readings for those who are curious.
- If time allows, homeworks/projects/exam feedbacks.
- Materials uploaded each morning before class (or the day before class).
- Homeworks given on Fridays.
- Project announced on W5.



50.039 Theory and Practice of Deep Learning

W8-S1 Introduction to Attacks and Defense on Neural Networks

Matthieu De Mari, Berrak Sisman

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 8)

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 attacks and defense, **open questions** in research.

About this week (Week 8)

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 attacks and defense, **open questions** in research.

Attacks: definition

Definition (Attacks on Neural Networks):

Adversarial machine learning, or **attacks on Neural Networks,** refers to machine learning techniques that attempt to fool models by supplying deceptive input.

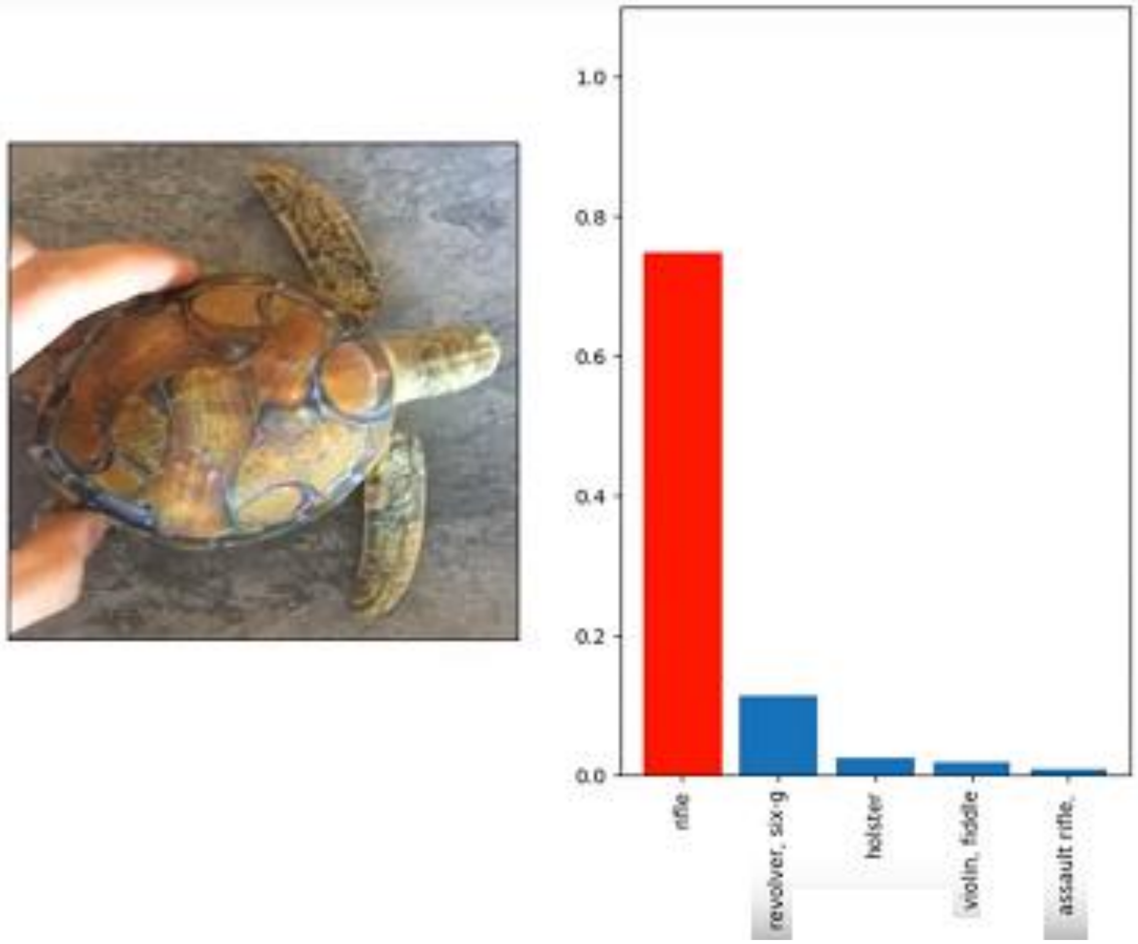
The most common reason is to cause a malfunction in a machine learning model.

Attack/Adversarial samples: definition

Definition (Attack samples, adversarial samples):

An **input sample** is considered an **attack sample** (or **adversarial sample**) for a given trained model, if and only if, it makes this model malfunction on purpose.

Example: this picture of a turtle has been altered on some of its pixels to be misclassified as a weapon (rifle, revolver, etc.).

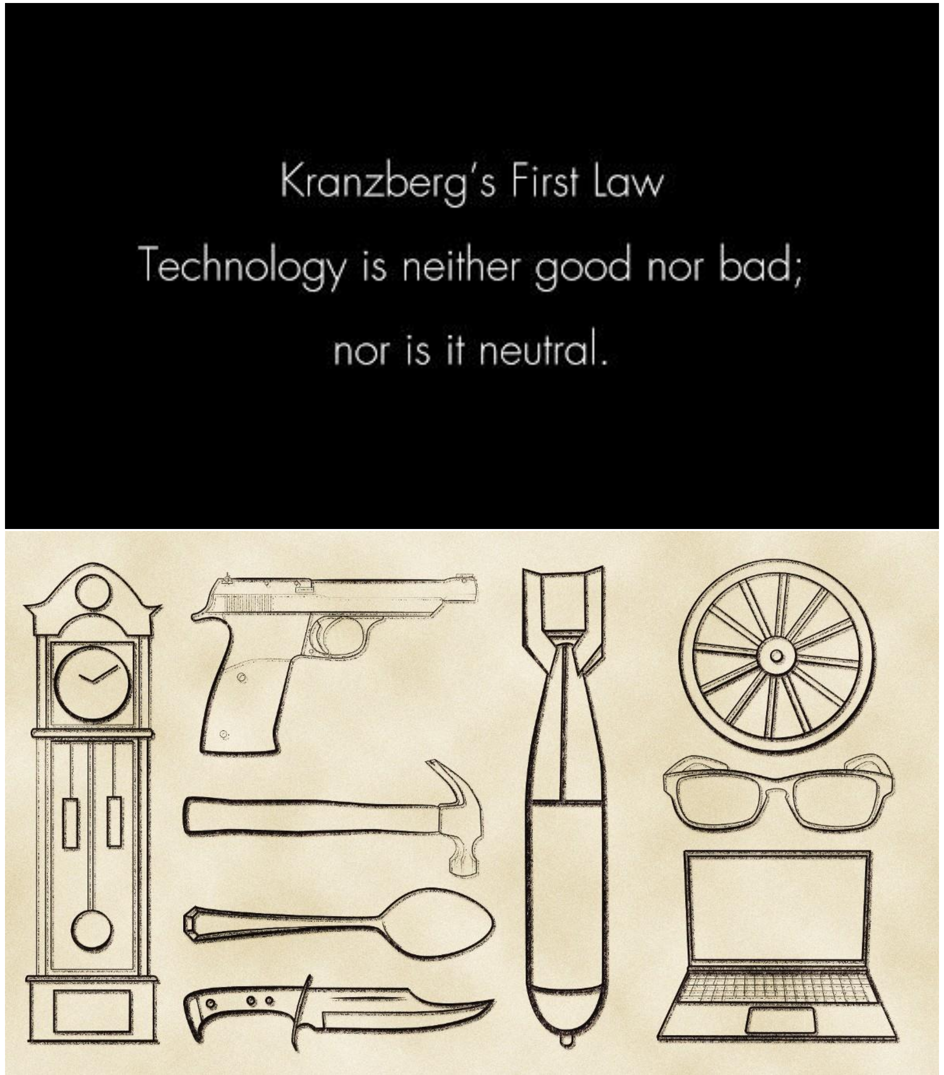


Source: Google's AI thinks this turtle looks like a gun, which is a problem [Verge1].

On the ethics of **attacking** Neural Networks

This week's lectures and notebooks will introduce techniques, whose objective is to **make a trained Neural Network malfunction on purpose**.

- These techniques are NOT, so to speak, illegal.
- But let us keep in mind what the consequences of using these attacks could be…



On the ethics of **attacking** Neural Networks

- **Example #1:** This stop sign has stickers put in specific locations.
- It is an **attack sample** as **it can no longer be detected as a stop sign, and is instead misclassified as several bottles**.

Think: What would be the effect/consequence of such an attack sample on a self-driving car using computer vision?

Source: Slight Street Sign Modifications Can Completely Fool Machine Learning Algorithms [Spectrum1].

On the ethics of **attacking** Neural Networks

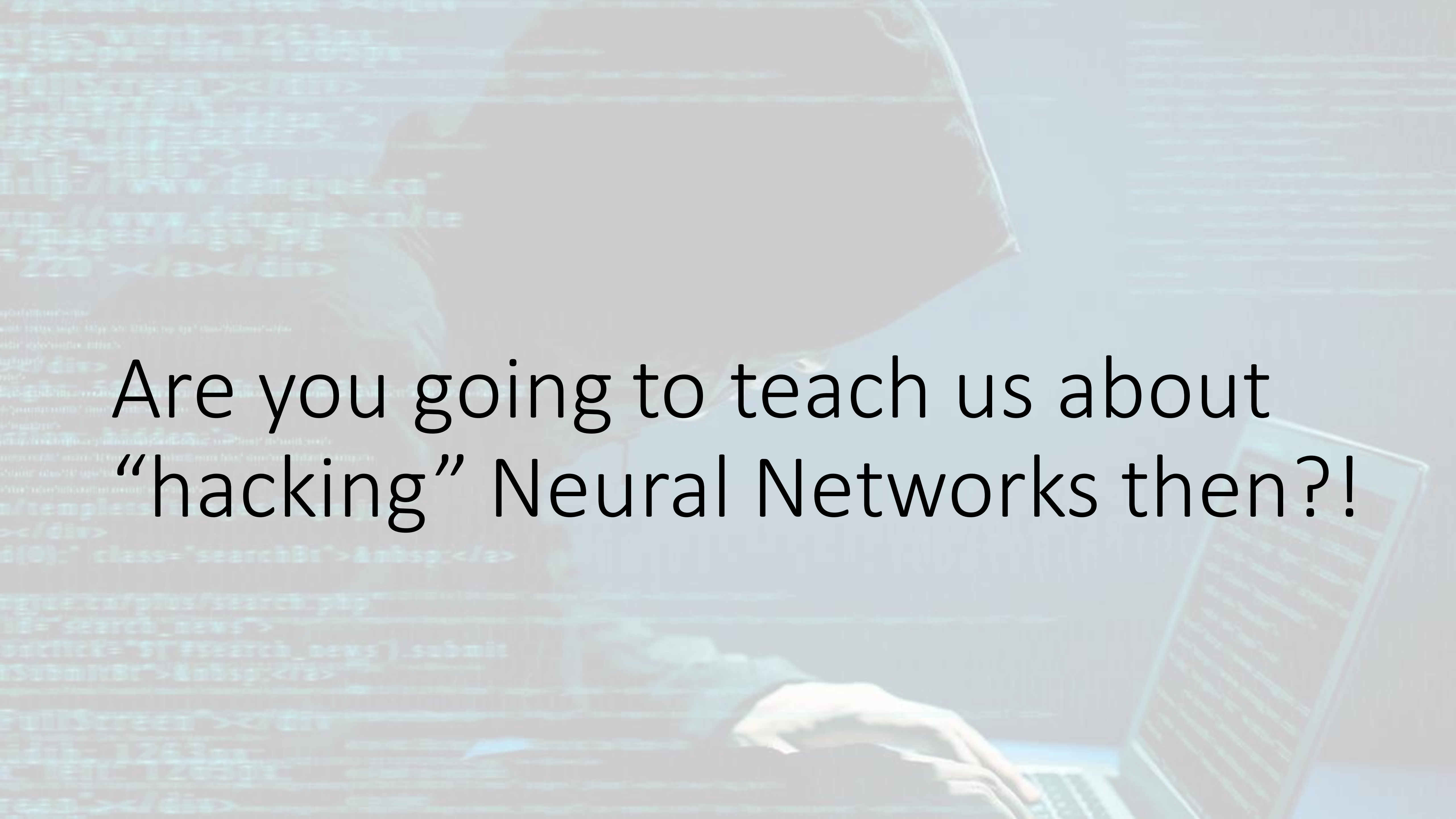
- **Example #2:** Covering some areas of your face with paint or glasses with specific patterns can fool facial recognition algorithms.
- These facial recognition AIs are no longer able to detect a face, let alone recognize the identity of the person.

(b) (c) (d)

Source: These glasses trick facial recognition software into thinking you're someone else [Verge2].

Source: Defeating Facial Recognition [YTB1].

<u>Think:</u> Is that a good or a bad discovery for computer vision?



Are you going to teach us about “hacking” Neural Networks then?!

Are you going to teach us about “hacking” Neural Networks then?!

No…?

Are you going to teach us about “hacking” Neural Networks then?!

Okay, yes, fine.

Are you going to teach us about “hacking” Neural Networks then?!

Okay, yes, fine.

But only for **two reasons**.

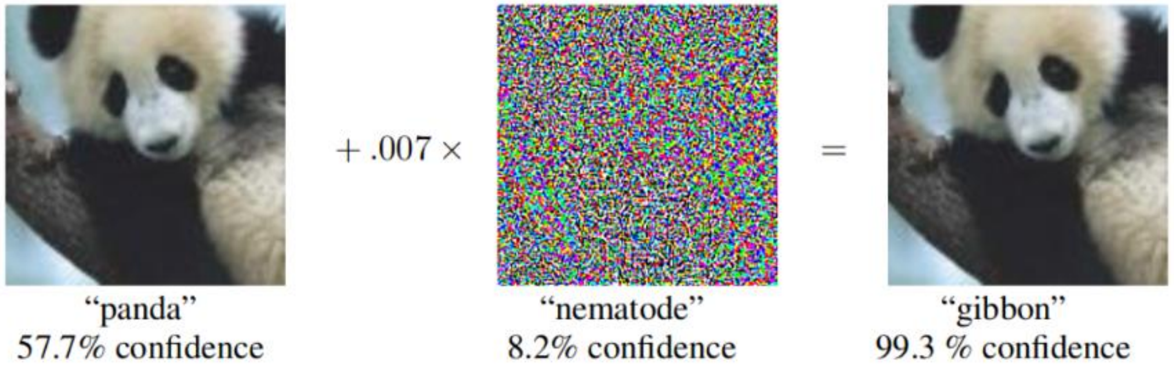
To teach you about limits/vulnerabilities of Neural Networks
and **how to defend them against such attacks**.

Reason #1: Limits

- **Reason #1:** Neural networks are **limited and vulnerable**, by design.
- **They will always be at risk of attacks making them malfunction, no matter how many safeguards you decide to put in place.**

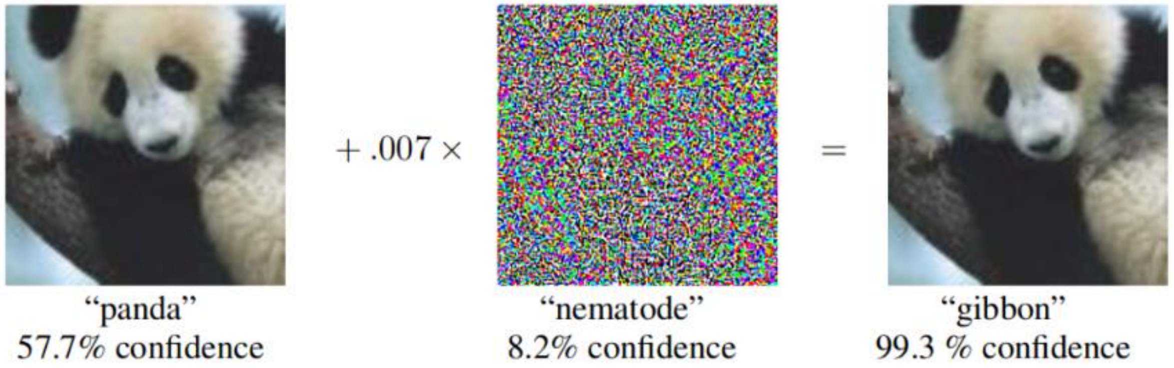
Reason #1: Limits

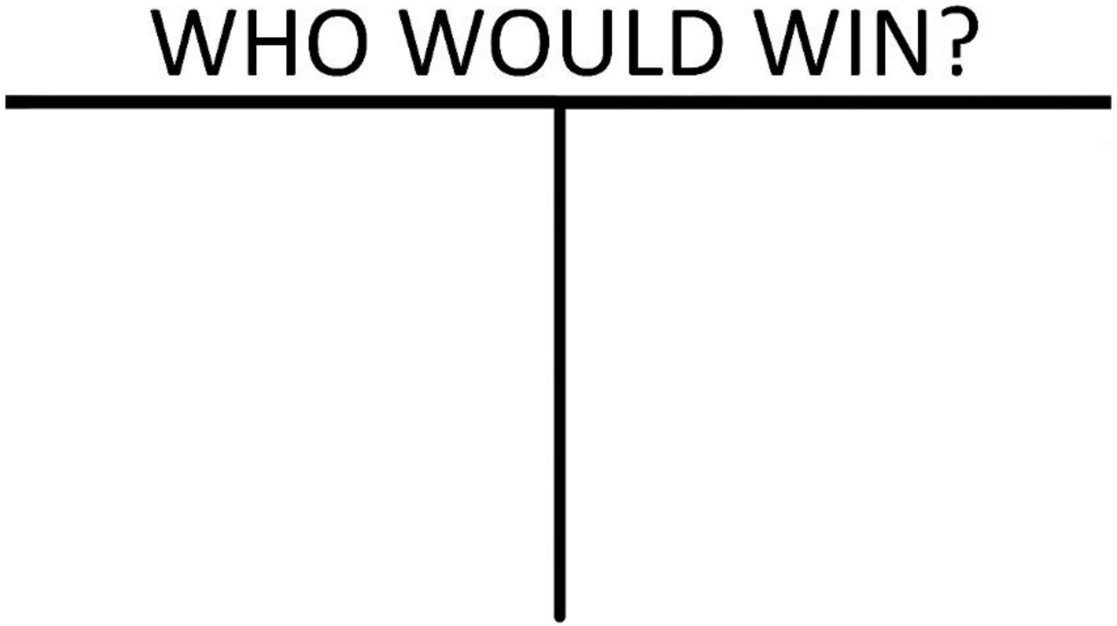
- <u>**Reason #1:**</u> Neural networks are **limited and vulnerable**, by design.
- **They will always be at risk of attacks making them malfunction, <u>no matter how many safeguards</u> you decide to put in place.**
- For instance, adding noise to an image is often enough to fool any image recognition algorithm.



Reason #1: Limits

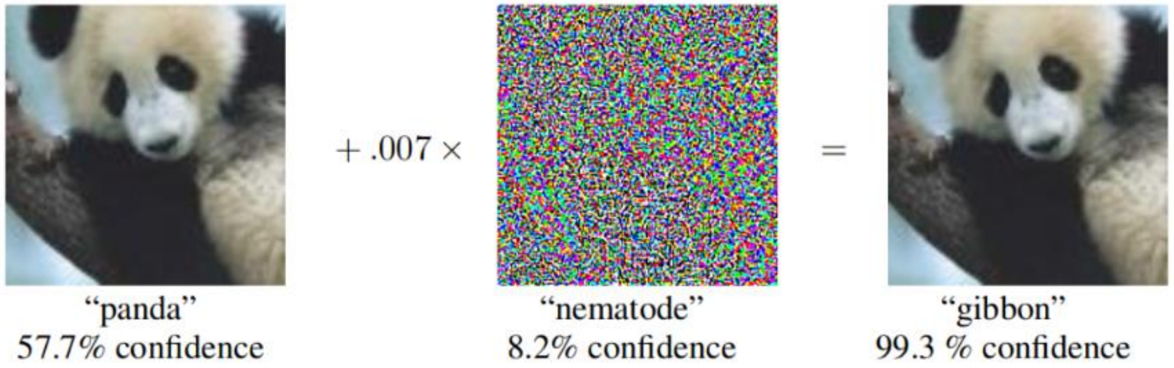
- **<u>Reason #1:</u>** Neural networks are **limited and vulnerable**, by design.
- **They will always be at risk of attacks making them malfunction, <u>no matter how many safeguards</u> you decide to put in place.**
- For instance, adding noise to an image is often enough to fool any image recognition algorithm.

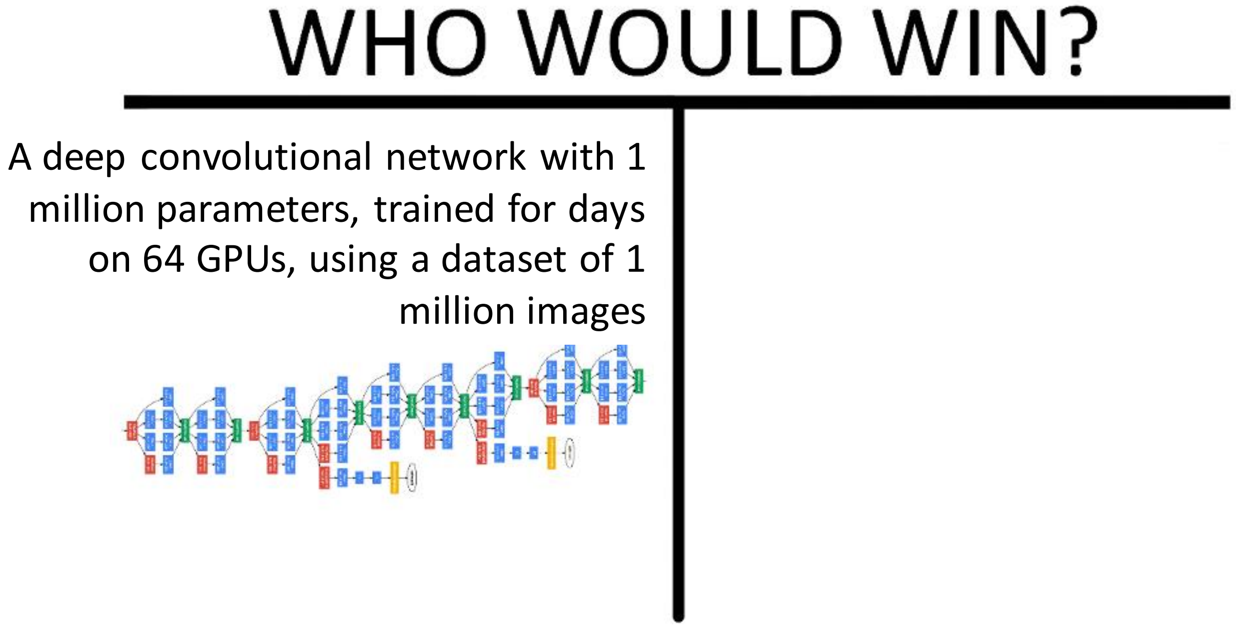




Reason #1: Limits

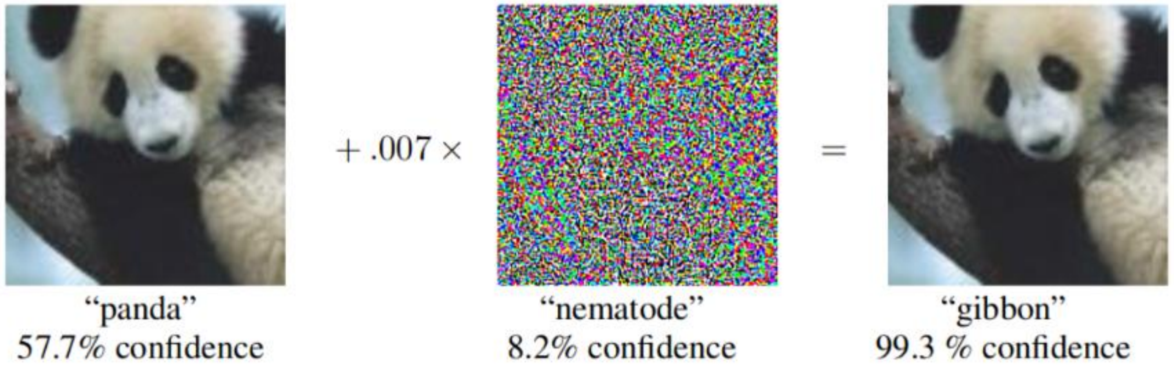
- **<u>Reason #1:</u>** Neural networks are **limited and vulnerable**, by design.
- **They will always be at risk of attacks making them malfunction, <u>no matter how many safeguards</u> you decide to put in place.**
- For instance, adding noise to an image is often enough to fool any image recognition algorithm.

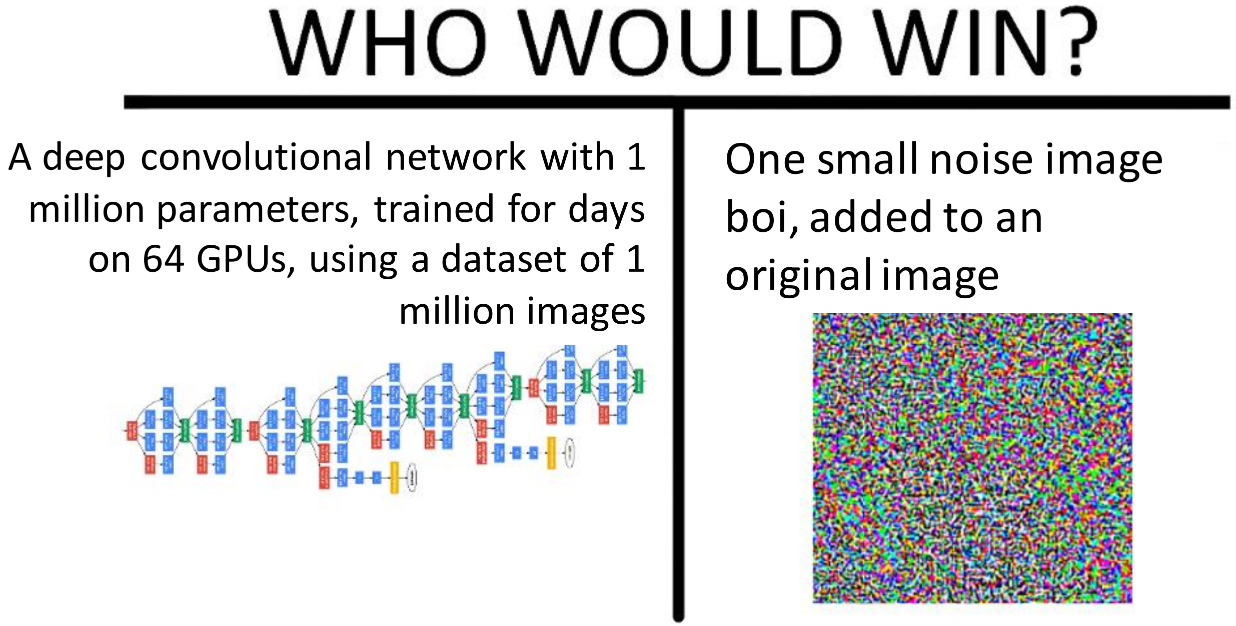




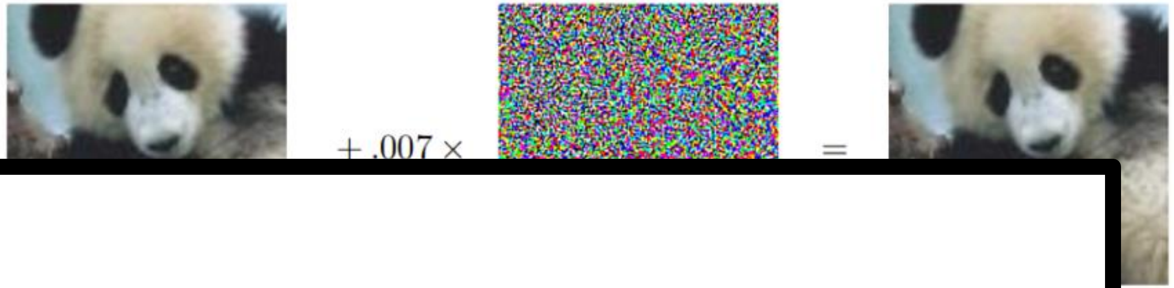
Reason #1: Limits

- **<u>Reason #1:</u>** Neural networks are **limited and vulnerable**, by design.
- **They will always be at risk of attacks making them malfunction, <u>no matter how many safeguards</u> you decide to put in place.**
- For instance, adding noise to an image is often enough to fool any image recognition algorithm.





Reason #1: Limits



- Neural networks are **limited and vu**
- **Th**
  at
  m
  <u>m</u>
  pu

This raises two questions.

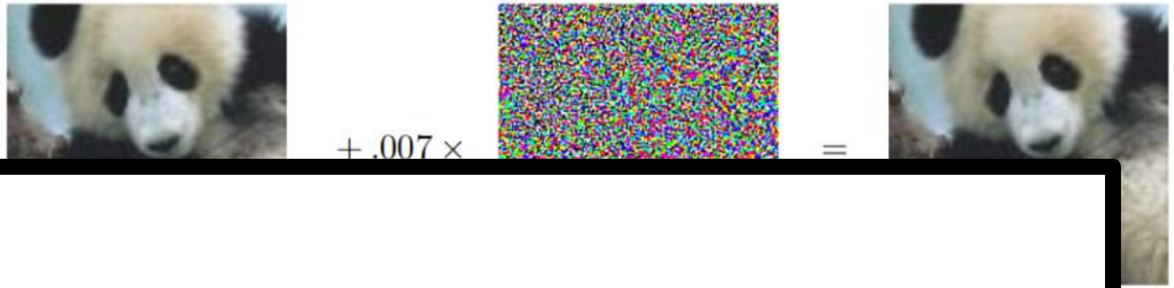
1. Shall we give up on neural networks then?

2. But, wait, how does that even work?!

- Fo
  im
  any image recognition algorithm.

Reason #1: Limits

- Neural networks are **limited and vu**
- **Th at m** <u>**m**</u> **pu**
- Fo im any image recognition algorithm.



This raises two questions.

1. Shall we give up on neural networks then?
   No, because of reason #2, defense (more on this later).

2. But, wait, how does that even work?!

Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - **Dataset and Dataloader**

Dataset and Dataloader

```
# Transform definition
# (Basic: only convert image to torch tensor)
tf = transforms.Compose([transforms.ToTensor()])
```

```
# MNIST dataset and dataloader
# (For testing only, we will use a pre-trained model)
ds = datasets.MNIST('./data', train = False, \
                    download = True, transform = tf)
test_loader = torch.utils.data.DataLoader(ds, batch_size = 1, \
                                          shuffle = True)
```

Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - **Model**

```python
# Model definition
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # Conv. 1
        self.conv1 = nn.Conv2d(1, 10, kernel_size = 5)
        # Conv. 2
        self.conv2 = nn.Conv2d(10, 20, kernel_size = 5)
        # Dropout for Conv. layers
        self.conv2_drop = nn.Dropout2d()
        # FC 1
        self.fc1 = nn.Linear(320, 50)
        # FC 2
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        # Conv. 1 + ReLU + Dropout
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        # Conv. 2 + ReLU + Dropout
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        # Flatten
        x = x.view(-1, 320)
        # FC 1 + ReLU + Droupout
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training = self.training)
        # FC 2 + Log-Softmax
        x = self.fc2(x)
        return F.log_softmax(x, dim = 1)
```

Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - **(Pre-trained) Model**

```
# Model definition
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # Conv. 1
        self.conv1 = nn.Conv2d(1, 10, kernel_size = 5)
        # Conv. 2
        self.conv2 = nn.Conv2d(10, 20, kernel_size = 5)
        # Dropout for Conv. layers
        self.conv2_drop = nn.Dropout2d()
        # FC 1
        self.fc1 = nn.Linear(320, 50)
        # FC 2
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        # Conv. 1 + ReLU + Dropout
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        # Conv. 2 + ReLU + Dropout
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        # Flatten
        x = x.view(-1, 320)
        # FC 1 + ReLU + Droupout
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training = self.training)
        # FC 2 + Log-Softmax
        x = self.fc2(x)
        return F.log_softmax(x, dim = 1)
```

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

```
<All keys matched successfully>
```

Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - **Attack function**

```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray([[2*(np.random.random() - 0.5)*epsilon
                                for i in range(img_rows)]
                                for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```

Our first attack: Epsilon Noising

Definition (Epsilon Noising Method):

The Epsilon Noising Method (ENM) is the simplest type of attack. It consists of generating an image $\tilde{x}$, by **adding a random noise to each pixel of an image x, with amplitude $[-\epsilon, \epsilon]$.**

```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray(([[2*(np.random.random() - 0.5)*epsilon
                               for i in range(img_rows)]
                               for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```

Our first attack: Epsilon Noising

Definition (Epsilon Noising Method):

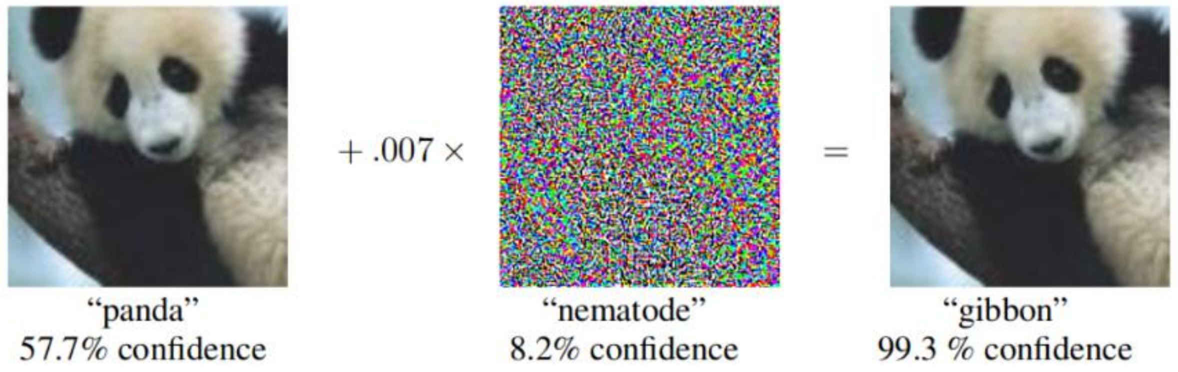
The **Epsilon Noising Method (ENM)** is the simplest type of attack. It consists of generating an image $\tilde{x}$, by **adding a random noise to each pixel of an image x, with amplitude $[-\epsilon, \epsilon]$.**

```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray(([[2*(np.random.random() - 0.5)*epsilon
                                for i in range(img_rows)]
                                for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```

$$\forall\, i,j \in Pixel\_Indexes_x$$

$$\boldsymbol{\tilde{x}_{i,j} = x_{i,j} + \omega_{i,j}}$$

$$\begin{cases} \omega_{i,j} \to U([-\epsilon, \epsilon]) & (Unif.Dist.) \\ \omega_{i,j} \to N([-\epsilon, \epsilon]) & (Normal\ Dist.) \end{cases}$$



Our first attack: Epsilon Noising

Definition (Epsilon Noising Method):

The **Epsilon Noising Method (ENM)** is the simplest type of attack. It consists of generating an image $\tilde{x}$, by **adding a random noise to each pixel of an image x, with amplitude $[-\epsilon, \epsilon]$.**

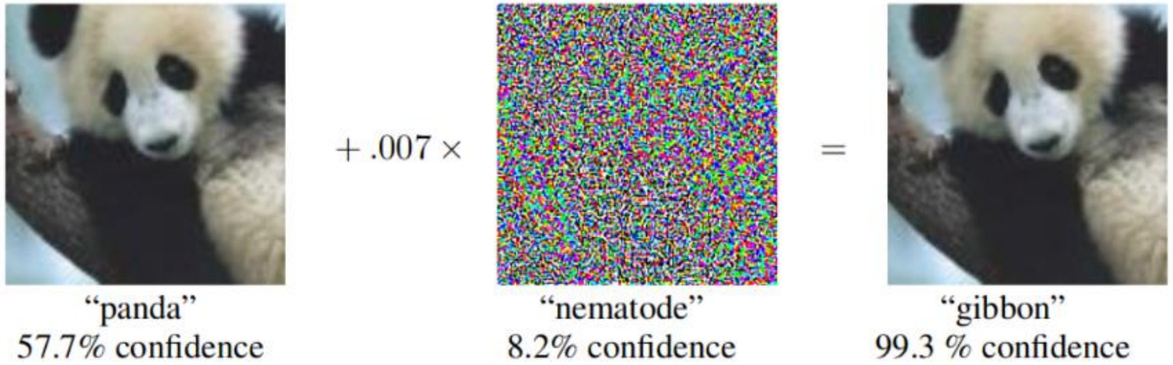
$$\forall\, i,j \in Pixel\_Indexes_x$$

$$\tilde{x}_{i,j} = x_{i,j} + \omega_{i,j}$$

$$\begin{cases} \omega_{i,j} \rightarrow U([-\epsilon, \epsilon]) & (Unif.\,Dist.) \\ \omega_{i,j} \rightarrow N([-\epsilon, \epsilon]) & (Normal\ Dist.) \end{cases}$$

Generate noise vector with same size as image, and amplitude in $[-\epsilon, \epsilon]$.

```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray(([[2*(np.random.random() - 0.5)*epsilon
                                for i in range(img_rows)]
                                for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```



Our first attack: Epsilon Noising

Add noise to original image.

Definition (Epsilon Noising Method):

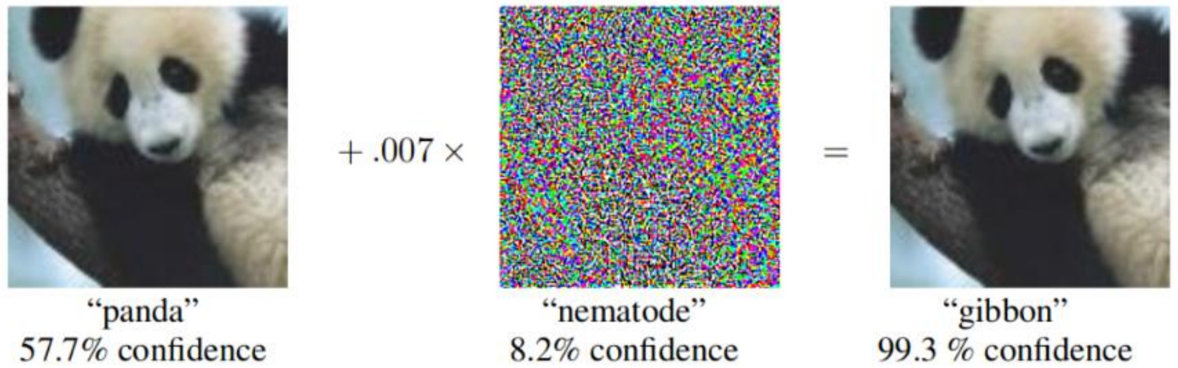
The **Epsilon Noising Method (ENM)** is the simplest type of attack. It consists of generating an image $\tilde{x}$, by **adding a random noise to each pixel of an image x, with amplitude $[-\epsilon, \epsilon]$.**

$$\forall\, i,j \in Pixel\_Indexes_x$$

$$\tilde{x}_{i,j} = x_{i,j} + \omega_{i,j}$$

$$\begin{cases} \omega_{i,j} \rightarrow U([-\epsilon, \epsilon]) & (Unif.Dist.) \\ \omega_{i,j} \rightarrow N([-\epsilon, \epsilon]) & (Normal\ Dist.) \end{cases}$$

```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray([[2*(np.random.random() - 0.5)*epsilon
                               for i in range(img_rows)]
                               for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```



Clipping to prevent unwanted pixel values.

Our first attack: Epsilon Noising

Reminder (Clipping a value):

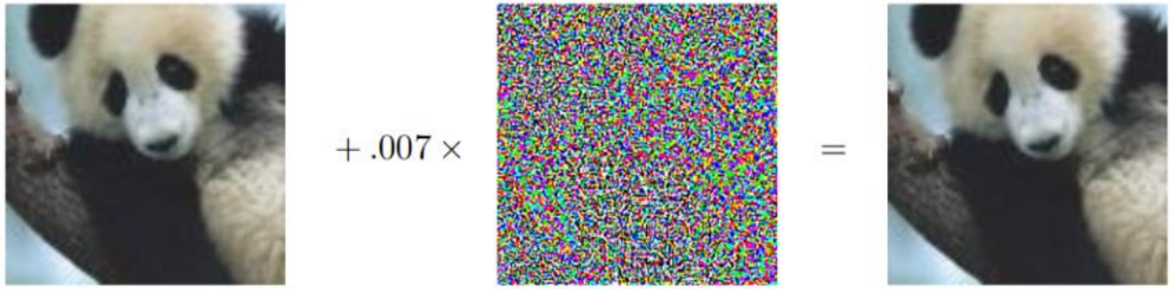
Clipping a value x forces it to **remain in an interval** $[\boldsymbol{a}, \boldsymbol{b}]$, with $a \leq b$.

We define the **clipping function** $\gamma_{a,b}(x)$, as follows.

$$\gamma_{a,b}(x) = \max(a, \min(x, b))$$

$$\gamma_{a,b}(x) = \begin{cases} a & if\ x \leq a \\ b & if\ x \geq b \\ x & otherwise \end{cases}$$

```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray([[2*(np.random.random() - 0.5)*epsilon
                               for i in range(img_rows)]
                               for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```



"panda" 57.7% confidence | "nematode" 8.2% confidence | "gibbon" 99.3 % confidence

Our first attack: Epsilon Noising

Clipping to prevent pixel values to go out of [0, 1].
(Normalization taken into account)

Definition (Epsilon Noising Method):

The Epsilon Noising Method (ENM) is the simplest type of attack. It consists of generating an image $\tilde{x}$, by **adding a random noise to each pixel of an image x, with amplitude $[-\epsilon, \epsilon]$.**

```
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray(([[2*(np.random.random() - 0.5)*epsilon
                                for i in range(img_rows)]
                                for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```

$$\forall\, i,j \in Pixel\_Indexes_x$$

$$\tilde{x}_{i,j} = \gamma_{0,1}\left(x_{i,j} + \omega_{i,j}\right)$$

$$\begin{cases} \omega_{i,j} \rightarrow U([-\epsilon,\epsilon]) & (Unif.\,Dist.) \\ \omega_{i,j} \rightarrow N([-\epsilon,\epsilon]) & (Normal\,Dist.) \end{cases}$$

"panda"
57.7% confidence

"nematode"
8.2% confidence

"gibbon"
99.3 % confidence

Our first attack: Epsilon Noising

Add noise to original image.

Definition (Epsilon Noising Method):

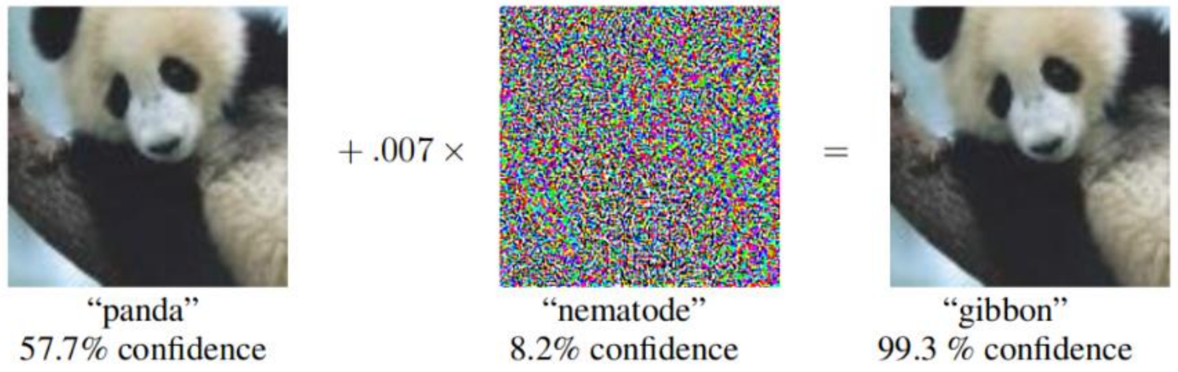
The **Epsilon Noising Method (ENM)** is the simplest type of attack. It consists of generating an image $\tilde{x}$, by **adding a random noise to each pixel of an image x, with amplitude $[-\epsilon, \epsilon]$.**

$$\forall\, i,j \in Pixel\_Indexes_x$$

$$\tilde{x}_{i,j} = \gamma_{0,1}\left(x_{i,j} + \omega_{i,j}\right)$$

$$\begin{cases} \omega_{i,j} \rightarrow U([-\epsilon, \epsilon]) & (Unif.\,Dist.) \\ \omega_{i,j} \rightarrow N([-\epsilon, \epsilon]) & (Normal\ Dist.) \end{cases}$$

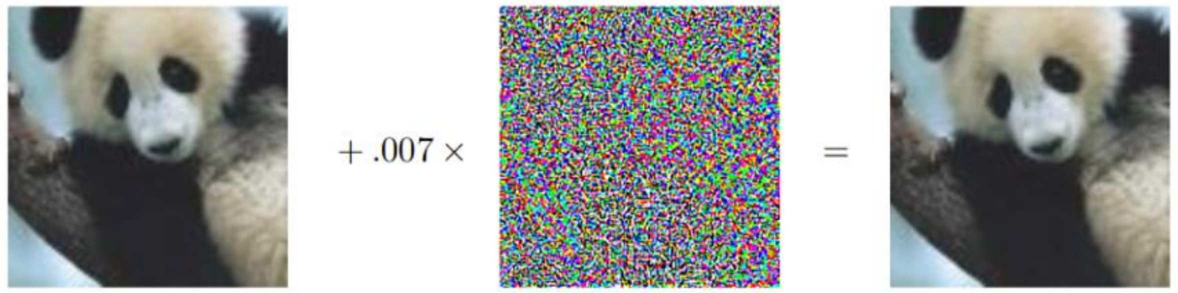
```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray([[2*(np.random.random() - 0.5)*epsilon
                              for i in range(img_rows)]
                              for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```



Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - **Attack function**

```python
def enm_attack(image, epsilon):

    # Generate noise matrix, with same shape as image,
    # and random values in [- epsilon, epsilon]
    img_rows = image.shape[-2]
    img_cols = image.shape[-1]
    epsilon_mat = np.asarray(([[2*(np.random.random() - 0.5)*epsilon
                                for i in range(img_rows)]
                                for j in range(img_cols)]))

    # Create the attack image by adjusting each pixel of the input image
    eps_image = image.detach().numpy() + epsilon_mat

    # Clipping eps_image to maintain pixel values into the [0, 1] range
    eps_image = torch.from_numpy(eps_image).float()
    eps_image = torch.clamp(eps_image, 0, 1)

    # Return
    return eps_image
```



"panda" 57.7% confidence | "nematode" 8.2% confidence | "gibbon" 99.3 % confidence

Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - Attack function
  - **Testing effect of attack on model**

Testing Function

This function attempts to 1) track how an attack with amplitude epsilon, used on the samples in the test_loader will affect our model;

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

This function attempts to 1) track how an attack with amplitude epsilon, used on the samples in the test_loader will affect our model; and 2) return attack samples that worked for later visualization.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                             correct_counter, \
                                                             len(test_loader), \
                                                             final_acc))

    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

This will track how many samples were correctly classified despite the attack being performed on the test samples.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

This will store up to 5 attack samples that made the model malfunction (used for visualization later).

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                         correct_counter, \
                                                         len(test_loader), \
                                                         final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

This is very typical for our test functions so far, just browsing through (normal) test samples and trying those on our model.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

If the model already misclassifies the sample, <u>do not bother attacking</u> (Attack could make the model right!).

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

This is again very typical. Right now, it does not appear necessary, but more advanced attacks will rely on the gradients of the model, and that would be the way to compute them (more on this later!).

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

Generate an attack sample, using our ENM attack function.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                       correct_counter, \
                                                       len(test_loader), \
                                                       final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

Try attack sample on our model.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()
        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                         correct_counter, \
                                                         len(test_loader), \
                                                         final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

If attack sample is correctly classified, the attack is a failure. Increase correct score by one.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()
        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

Add sample to adversarial samples list if epsilon = 0 and list not full (attacking with epsilon = 0 will always fail as it will not modify the image). This gives the baseline accuracy of the model before attacks.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()
        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                             correct_counter, \
                                                             len(test_loader), \
                                                             final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

If attack sample makes the model misclassify, it is a successful attack. Do not increase correct_counter, and store sample in adversarial samples list if not already full.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

After the for loop, compute accuracy of model after attack.

```
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()

        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                            correct_counter, \
                                                            len(test_loader), \
                                                            final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Testing Function

Display accuracy for given epsilon value.
Return accuracy score and the list of five adversarial samples.

```python
def test(model, device, test_loader, epsilon):

    # Counter for correct values (used for accuracy)
    correct_counter = 0

    # List of successful adversarial samples
    adv_examples_list = []

    # Loop over all examples in test set
    for image, label in test_loader:

        # Send the data and label to the device
        image, label = image.to(device), label.to(device)

        # Pass the image through the model
        output = model(image)
        # Get the index of the max log-probability
        init_pred = output.max(1, keepdim = True)[1]

        # If the initial prediction is wrong, do not
        # bother attacking, skip current image
        if init_pred.item() != label.item():
            continue

        # Calculate the loss
        loss = F.nll_loss(output, label)

        # Zero all existing gradients
        model.zero_grad()

        # Backpropagate
        loss.backward()
```

```python
        # Call ENM Attack
        eps_image = enm_attack(image, epsilon)

        # 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]

        # Check for successful attack
        # (Successful meaning eps_pred label different from init_pred)
        if eps_pred.item() == label.item():
            correct_counter += 1
            # Special case for saving 0 epsilon examples
            # (Maximal number of saved samples is set to 5)
            if (epsilon == 0) and (len(adv_examples_list) < 5):
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))
        else:
            # Save some adv examples for visualization later
            # (Maximal number of saved samples is set to 5)
            if len(adv_examples_list) < 5:
                adv_ex = eps_image.squeeze().detach().cpu().numpy()
                adv_examples_list.append((init_pred.item(), eps_pred.item(), adv_ex))

    # Calculate final accuracy for this epsilon value
    final_acc = correct_counter/float(len(test_loader))

    # Display for progress
    print("Epsilon: {} - Test Accuracy = {}/{} = {}".format(epsilon, \
                                                             correct_counter, \
                                                             len(test_loader), \
                                                             final_acc))


    # Return the accuracy and an adversarial example
    return final_acc, adv_examples_list
```

Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - Attack function
  - **Testing effect of attack on model**

```
epsilons = [0, .1, .2, .5, 1, 2, 5, 10]
accuracies = []
examples = []

# Run test() function for each epsilon
for eps in epsilons:
    acc, ex = test(model, device, test_loader, eps)
    accuracies.append(acc)
    examples.append(ex)
```

```
Epsilon: 0 - Test Accuracy = 9810/10000 = 0.981
Epsilon: 0.1 - Test Accuracy = 9792/10000 = 0.9792
Epsilon: 0.2 - Test Accuracy = 9775/10000 = 0.9775
Epsilon: 0.5 - Test Accuracy = 9578/10000 = 0.9578
Epsilon: 1 - Test Accuracy = 6367/10000 = 0.6367
Epsilon: 2 - Test Accuracy = 2203/10000 = 0.2203
Epsilon: 5 - Test Accuracy = 1162/10000 = 0.1162
Epsilon: 10 - Test Accuracy = 1074/10000 = 0.1074
```

Before we start

Display a simple plot of accuracy vs. epsilon value for our given attack and model.

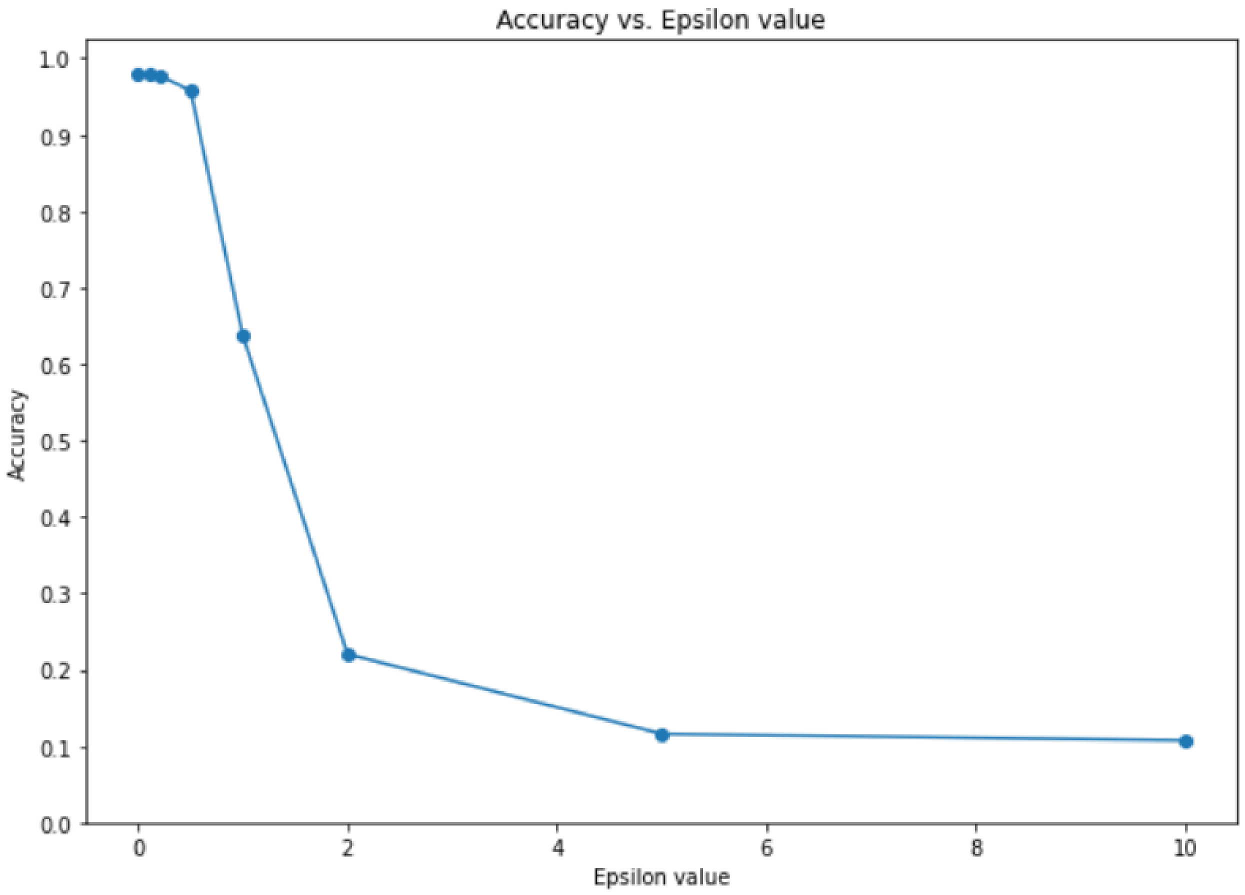
- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - Attack function
  - Testing effect of attack on model
  - **Accuracy drop and attack samples visualization**

```python
# Initialize figure
plt.figure(figsize = (7, 10))

# Display accuracy vs. Epsilon values plot
plt.plot(epsilons, accuracies, "o-")

# Adjust x-axis and y-axis labels and ticks
plt.yticks(np.arange(0, 1.1, step = 0.1))
#plt.xticks(np.arange(0, .35, step = 0.05))
plt.title("Accuracy vs. Epsilon value")
plt.xlabel("Epsilon value")
plt.ylabel("Accuracy")

# Display
plt.show()
```

Before we start

Display a simple plot of accuracy vs. epsilon value for our given attack and model.

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - Attack function
  - Testing effect of attack on model
  - Accuracy drop and attack samples visualization



Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - Attack function
  - Testing effect of attack on model
  - Accuracy drop and attack samples visualization

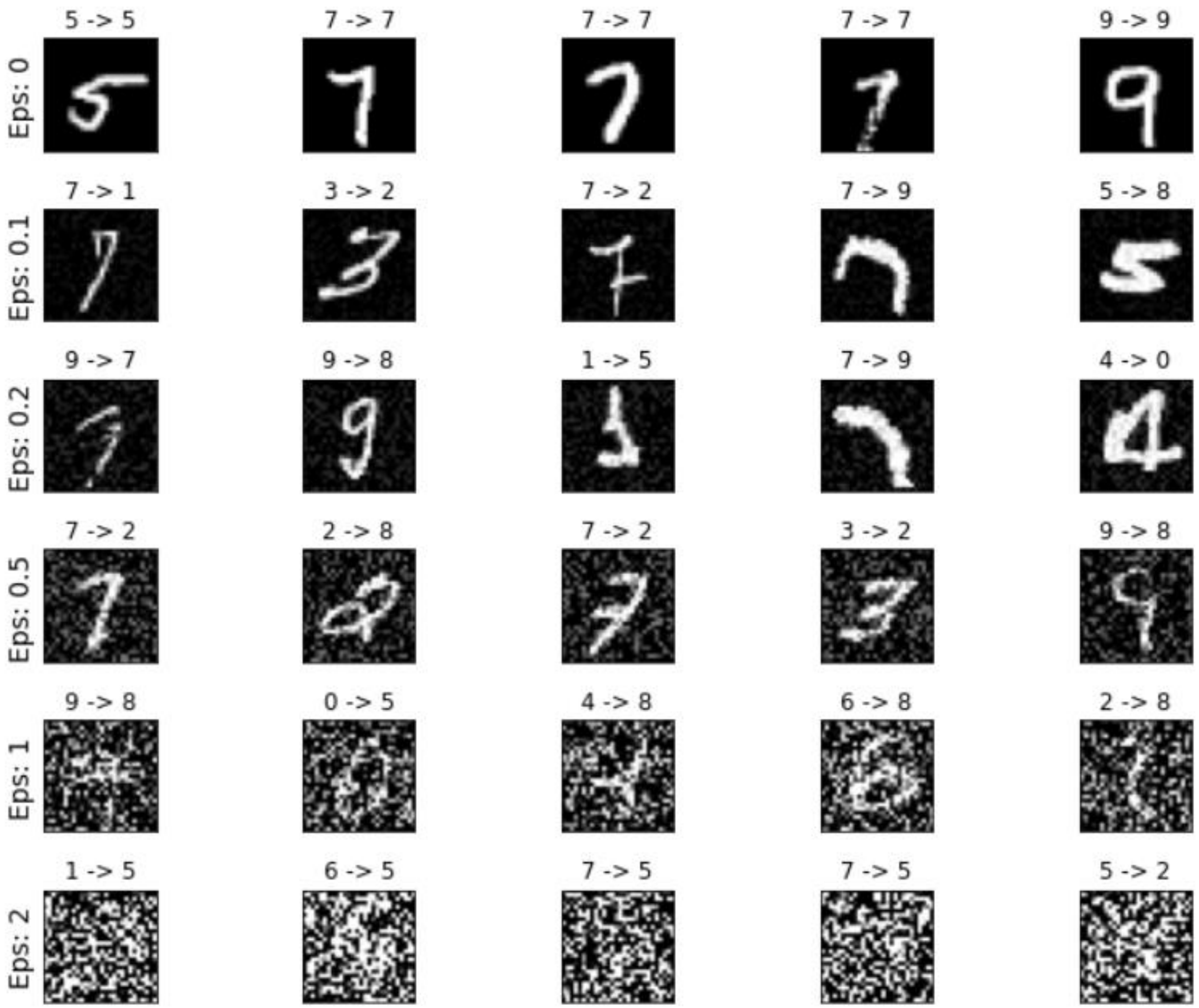
Display some adversarial samples for each value of epsilon.

```python
# Plot several examples of adversarial samples at each epsilon
cnt = 0

# Initialize figure
plt.figure(figsize = (10, 10))

# Browse through epsilon values and adversarial examples
for i in range(len(epsilons)):
    for j in range(len(examples[i])):
        cnt += 1
        plt.subplot(len(epsilons), len(examples[0]), cnt)

        # Remove x-axis and y-axis ticks from plot
        plt.xticks([], [])
        plt.yticks([], [])

        # Labels for y axis
        if j == 0:
            plt.ylabel("Eps: {}".format(epsilons[i]), fontsize = 14)

        # Labels for each image subplot
        orig, adv, ex = examples[i][j]
        plt.title("{} -> {}".format(orig, adv))

        # Display image
        plt.imshow(ex, cmap = "gray")

# Display full plot
plt.tight_layout()
plt.show()
```

Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - Attack function
  - Testing effect of attack on model
  - Accuracy drop and attack samples visualization

Display some adversarial samples for each value of epsilon.



Before we start

- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
  - Dataset and Dataloader
  - (Pre-trained) Model
  - Attack function
  - Testing effect of attack on model
  - Accuracy drop and attack samples visualization
  - **Defense against such an attack**

Before we start

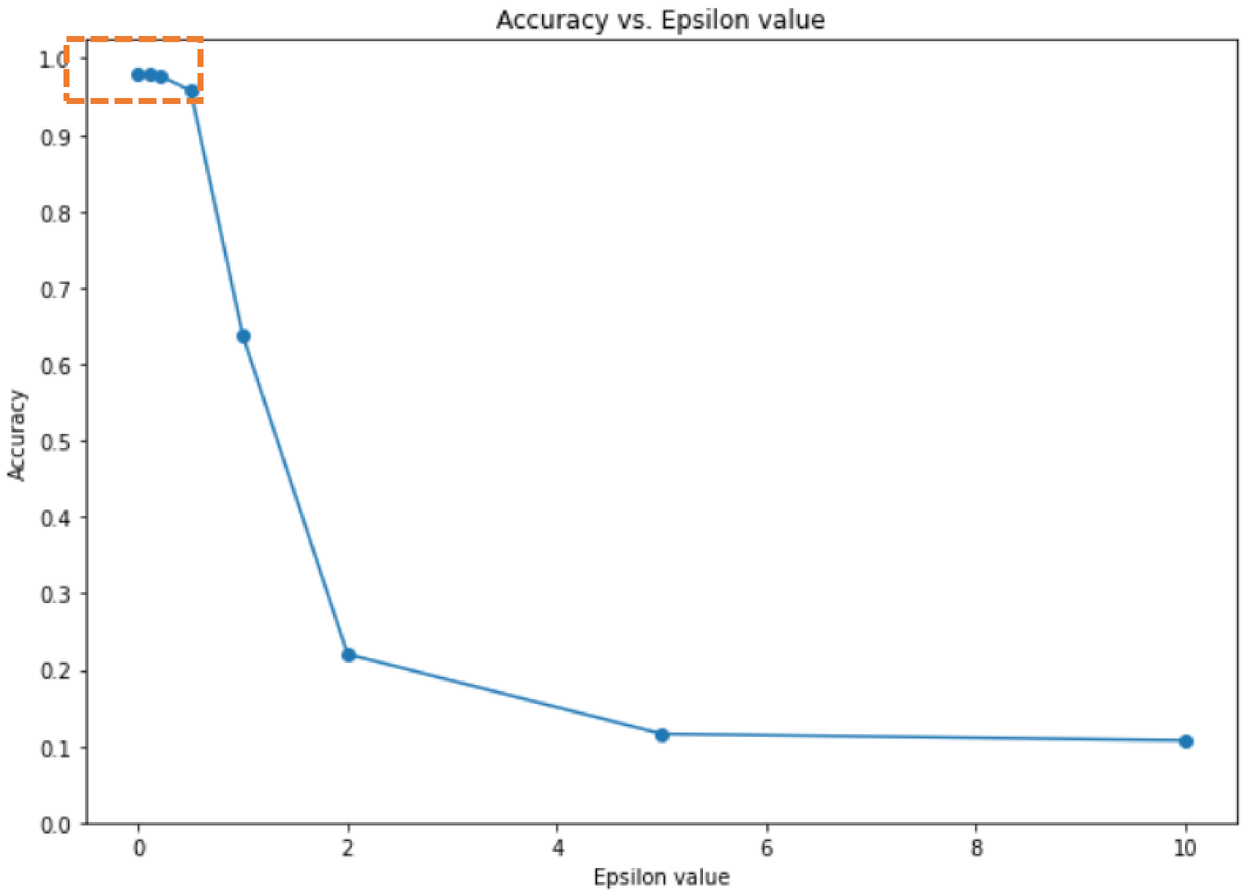
- Please refer to Notebook 1. Using Epsilon Noising Attack to Generate Attack Samples.
- All notebooks this week follow the same structure
    - Dataset and Dataloader
    - (Pre-trained) Model
    - Attack function
    - Testing effect of attack on model
    - Accuracy drop and attack samples visualization
    - ~~Defense against such an attack~~

SOMETHING FOR LATER...

Effect of ENM on accuracy

Adding noise to an image tends to make the model malfunction.

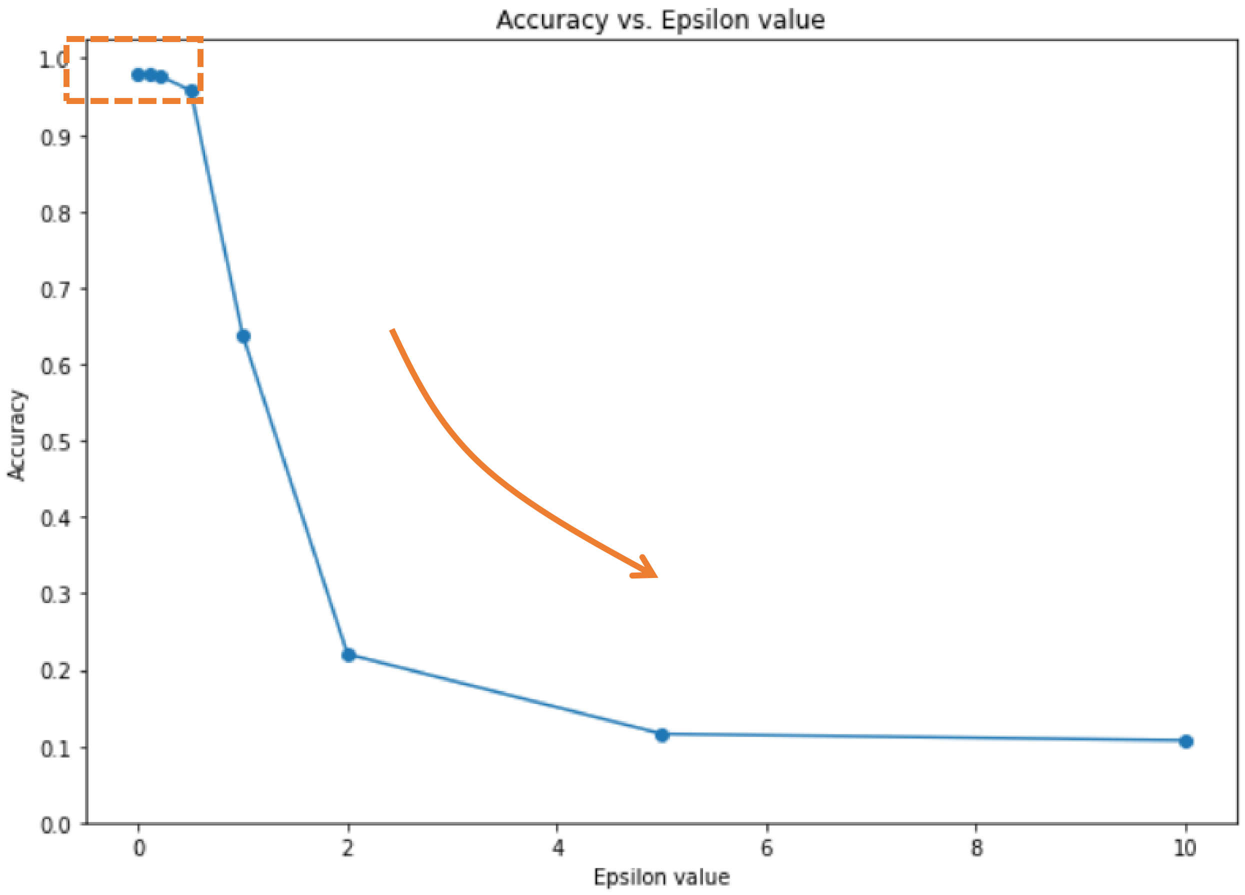
- When no noise (epsilon = 0), 98.1% accuracy.
  - This is our **baseline accuracy**.



Effect of ENM on accuracy

Adding noise to an image tends to make the model malfunction.

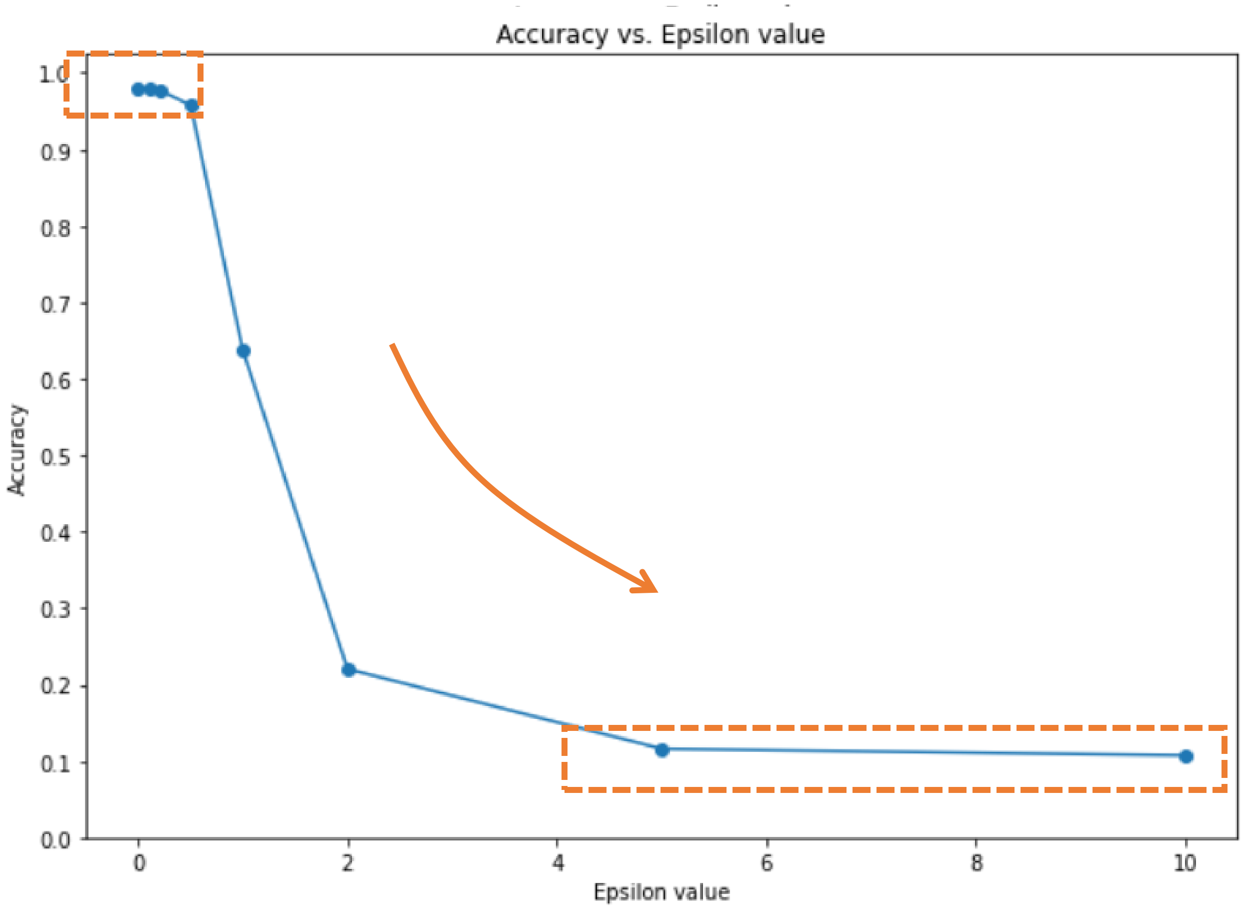
- When no noise (epsilon = 0), 98.1% accuracy.
  - This is our **baseline accuracy**.
- Accuracy decreases, the further we increment the noise amplitude epsilon.



Effect of ENM on accuracy

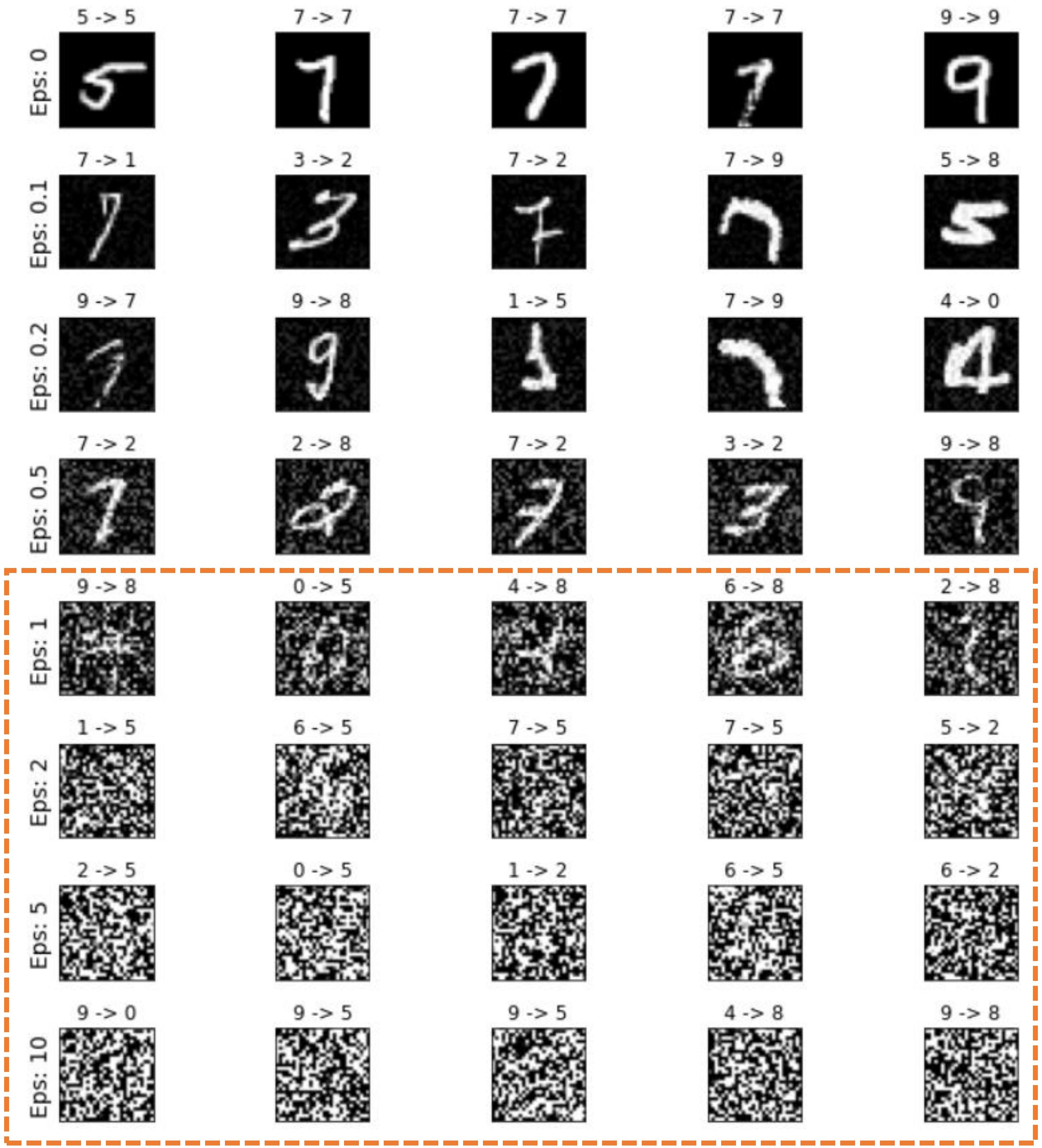
Adding noise to an image tends to make the model malfunction.

- When no noise (epsilon = 0), 98.1% accuracy.
  - This is our **baseline accuracy**.
- Accuracy decreases, the further we increment the noise amplitude epsilon.
- Eventually, with full noise (large epsilon), the image will be **randomly classified** (Accuracy $\sim 10\%$).



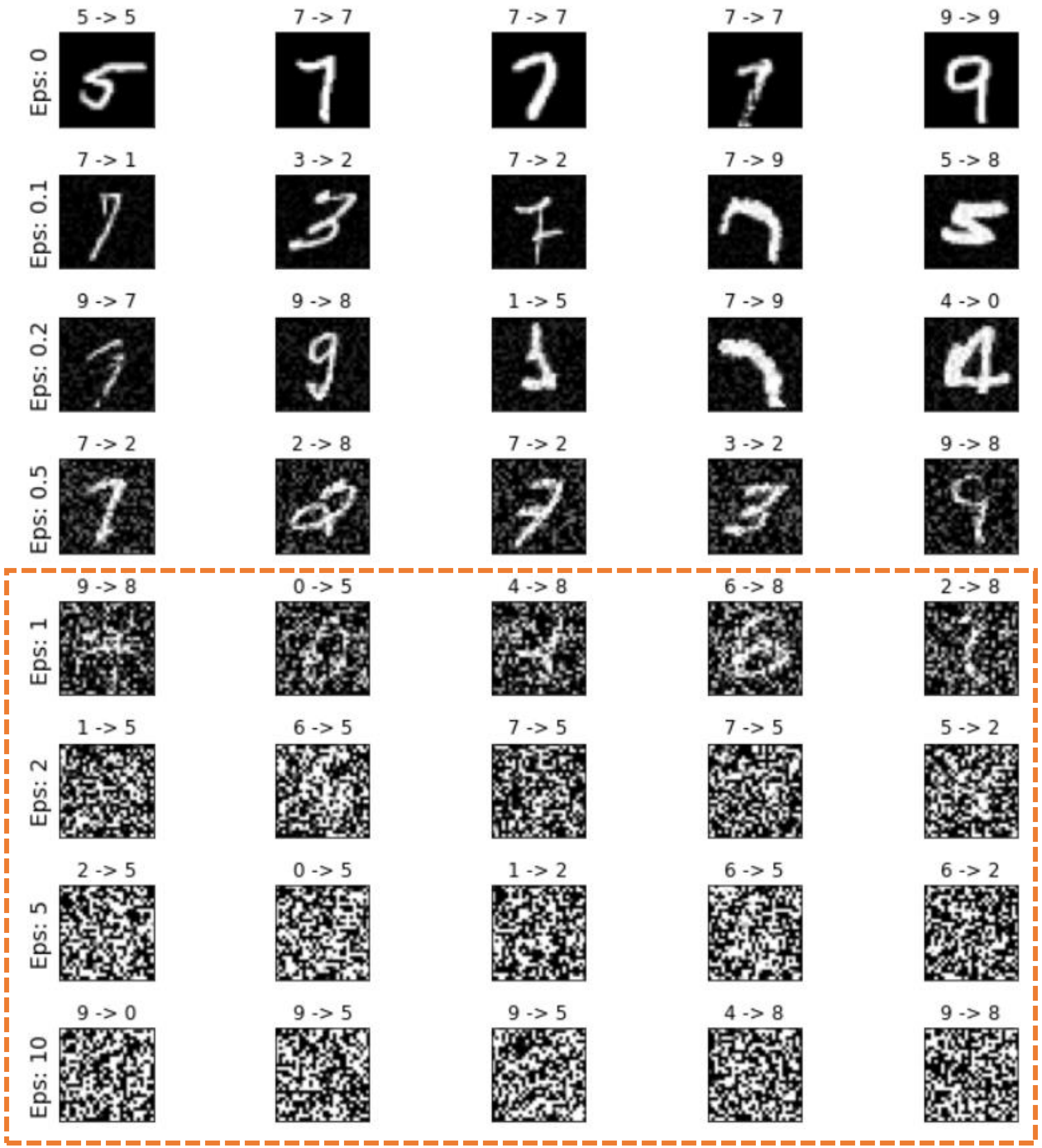
What makes a good attack sample?

- However, for large values of epsilon, the attack samples **simply become random noise**.



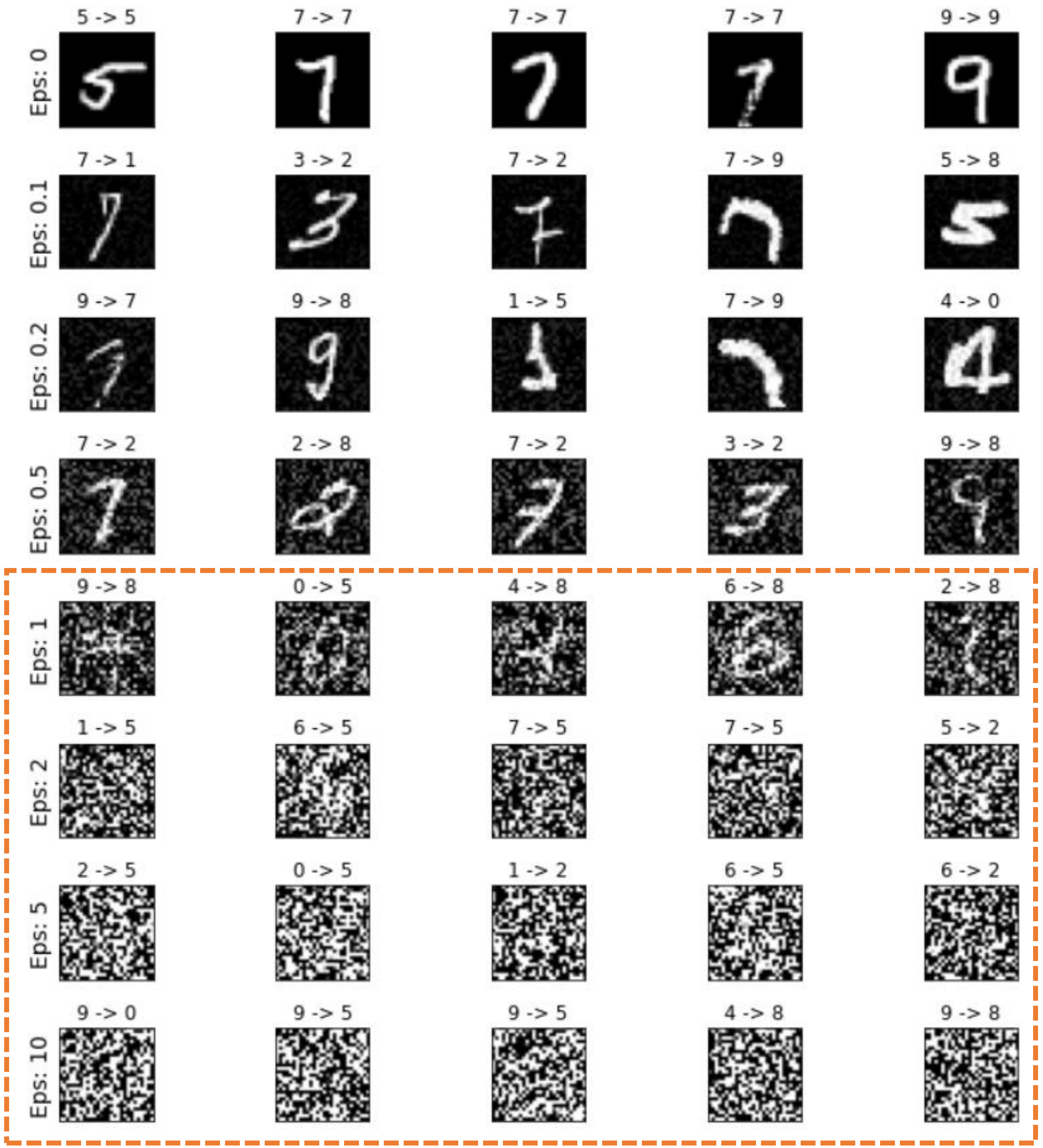
What makes a good attack sample?

- However, for large values of epsilon, the attack samples **simply become random noise**.
- This is mostly why the classifier ends up struggling.



What makes a good attack sample?

- However, for large values of epsilon, the attack samples **simply become random noise**.
- This is mostly why the classifier ends up struggling.
- These are **NOT considered good attack samples!**
- (<u>**Think:**</u> humans would struggle to classify those as well!)

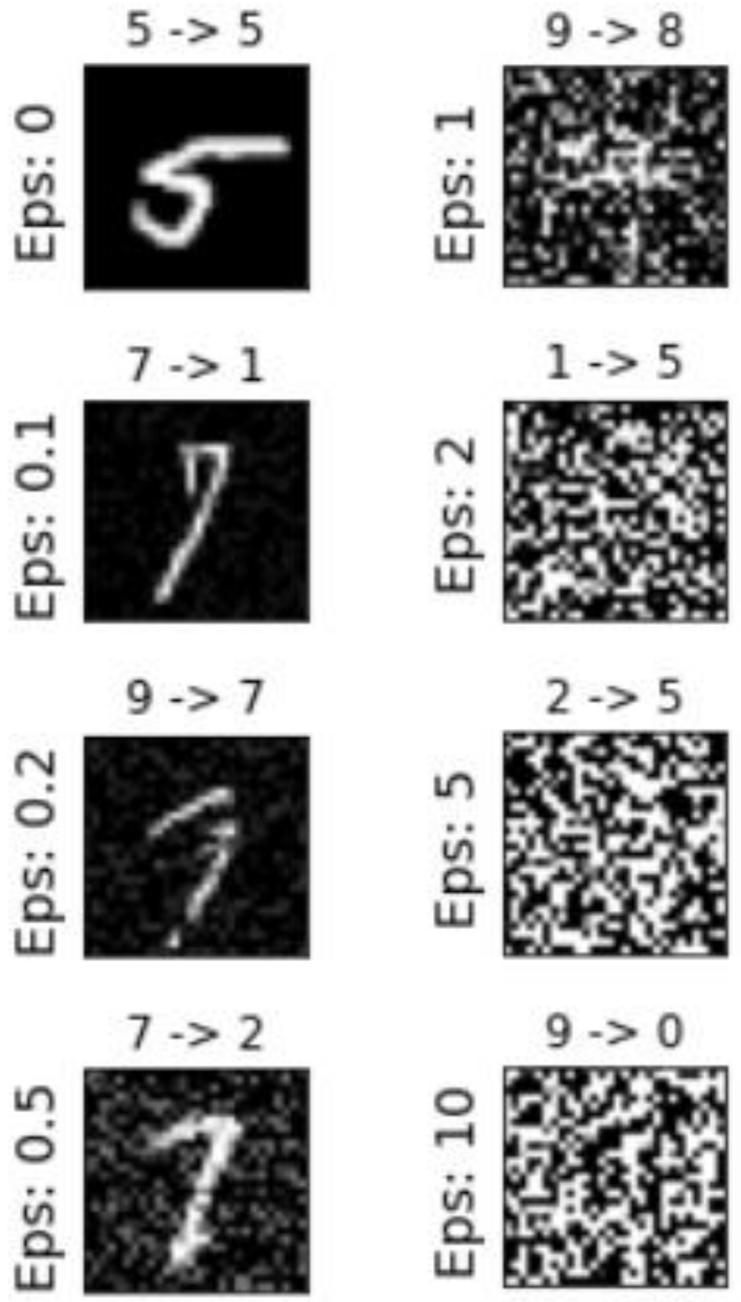


What makes a good attack sample?

Definition (what makes a "good" attack sample):

A **"good" attack sample** satisfies two properties:

1. **Model failure:** it makes the model malfunction.
2. **Plausibility:** It looks "plausible" or "normal" to a human. It looks like it could have been a sample from the dataset.

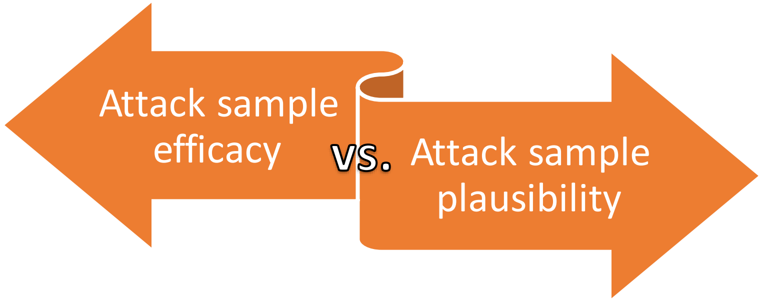


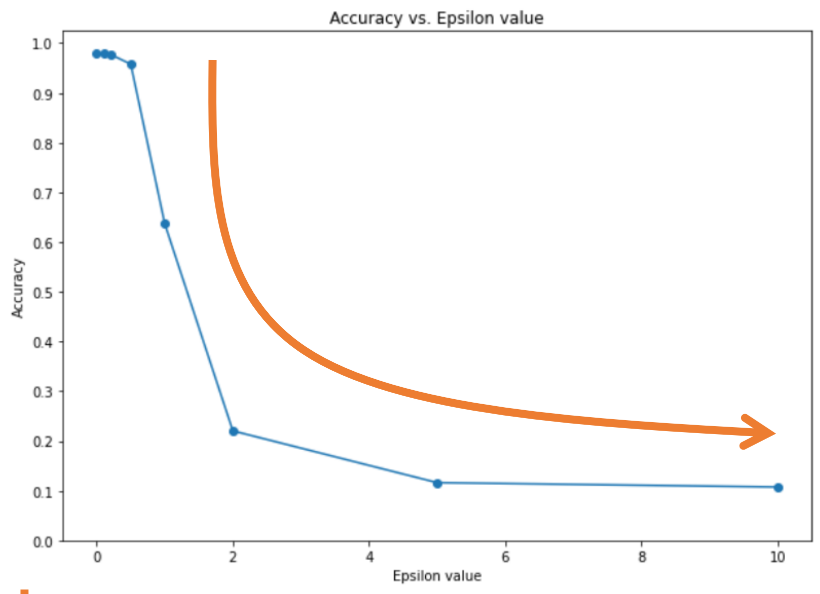
Attack samples tradeoff

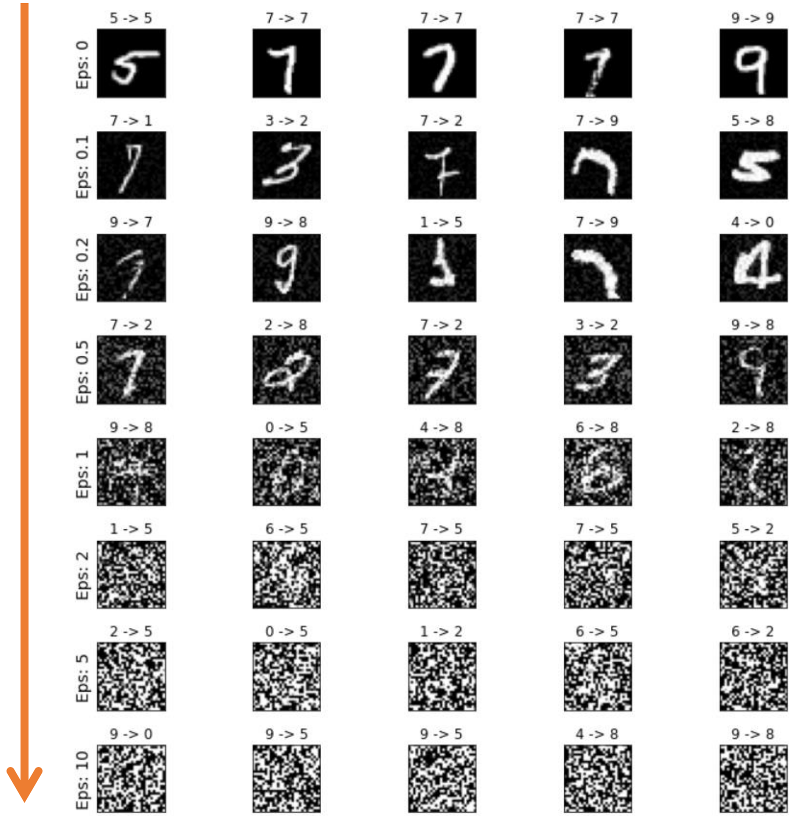
Definition (attack samples tradeoff in adversarial ML):

Attack samples are subject to a **tradeoff**. In general,

- **the higher the odds of the attack sample to make the model malfunction,**
- **the less plausible it will look.**







Plausibility

- In general, we would like to have the generated attack sample $\tilde{x}$, to be **"close enough"** to the original sample x.

$$\forall\, i, j \in Pixel\_Indexes_x$$
$$\tilde{x}_{i,j} = x_{i,j} + \omega_{i,j}$$
$$\begin{cases} \omega_{i,j} \rightarrow U([-\epsilon, \epsilon]) \quad (Unif.Dist.) \\ \omega_{i,j} \rightarrow N([-\epsilon, \epsilon]) \quad (Normal\ Dist.) \end{cases}$$

Plausibility

- In general, we would like to have the generated attack sample $\tilde{x}$, to be **"close enough"** to the original sample x.
- This is a simple way to ensure **plausibility** for the attack sample.

$$\forall\, i,j \in Pixel\_Indexes_x$$
$$\tilde{x}_{i,j} = x_{i,j} + \omega_{i,j}$$
$$\begin{cases} \omega_{i,j} \to U([-\epsilon,\epsilon]) \quad (Unif.Dist.) \\ \omega_{i,j} \to N([-\epsilon,\epsilon]) \quad (Normal\ Dist.) \end{cases}$$

Plausibility

- In general, we would like to have the generated attack sample $\tilde{x}$, to be **"close enough"** to the original sample x.
- This is a simple way to ensure **plausibility** for the attack sample.
- In practice, we often enforce a constraint on a **distance metric (or norm) between both the original image** x and **attack sample** $\tilde{x}$.

$$\forall\, i,j \in Pixel\_Indexes_x$$
$$\tilde{x}_{i,j} = x_{i,j} + \omega_{i,j}$$
$$\begin{cases} \omega_{i,j} \rightarrow U([-\epsilon, \epsilon]) \quad (Unif.Dist.) \\ \omega_{i,j} \rightarrow N([-\epsilon, \epsilon]) \quad (Normal\ Dist.) \end{cases}$$

$$\|\tilde{\boldsymbol{x}} - \boldsymbol{x}\| \leq \boldsymbol{\alpha},$$
$$\textit{\textbf{with}}\ \boldsymbol{\alpha}\ \textbf{chosen arbitrarily}$$

A reminder about norms

- $\boldsymbol{L^0}$**norm:** bounds the total number of pixels in $\tilde{x}$ that can be modified with respect to x (though they can be modified by any amount).

$$\|\tilde{x} - x\|_0 = card\big(\{(i,j)\ s.t.\ x_{i,j} \neq \tilde{x}_{i,j}\}\big)$$

A reminder about norms

- $\boldsymbol{L^0}$**norm:** bounds the total number of pixels in $\tilde{x}$ that can be modified with respect to x (though they can be modified by any amount).
- $\boldsymbol{L^1}$ **norm:** bounds the average absolute distance between the values of pixels in $\tilde{x}$ and the corresponding pixels in x.

$$\|\tilde{x} - x\|_1 = \frac{1}{N}\sum_{i,j}\left|\tilde{x}_{i,j} - x_{i,j}\right|, \qquad with\ N\ the\ number\ of\ pixels$$

A reminder about norms

- $\boldsymbol{L^0}$ **norm:** bounds the total number of pixels in $\tilde{x}$ that can be modified with respect to x (though they can be modified by any amount).
- $\boldsymbol{L^1}$ **norm:** bounds the average absolute distance between the values of pixels in $\tilde{x}$ and the corresponding pixels in x.
- $\boldsymbol{L^2}$ **norm:** bounds the total squared distance between the values of pixels in $\tilde{x}$ and the corresponding pixels in x. Most commonly referred to as Euclidean distance.

$$\|\tilde{x} - x\|_2 = \frac{1}{N}\sqrt{\sum_{i,j}\left(\tilde{x}_{i,j} - x_{i,j}\right)^2}, \qquad with\ N\ the\ number\ of\ pixels$$

A reminder about norms

- $\boldsymbol{L^0}$ **norm:** bounds the total number of pixels in $\tilde{x}$ that can be modified with respect to x (though they can be modified by any amount).
- $\boldsymbol{L^1}$ **norm:** bounds the average absolute distance between the values of pixels in $\tilde{x}$ and the corresponding pixels in x.
- $\boldsymbol{L^2}$ **norm:** bounds the total squared distance between the values of pixels in $\tilde{x}$ and the corresponding pixels in x. Often referred to as the Euclidean distance.
- $\boldsymbol{L^\infty}$ **norm:** bounds the maximum difference between any pixel in $\tilde{x}$ and the corresponding pixel in x. Often referred to as max norm.

$$\|\tilde{x} - x\|_\infty = \max_{i,j}\left(\left|\tilde{x}_{i,j} - x_{i,j}\right|\right)$$

A reminder about norms

- $\boldsymbol{L^0}$ **norm:** bounds the total number of pixels in $\tilde{x}$ that can be modified with respect to x (though they can be modified by any amount).
- $\boldsymbol{L^1}$ **norm:** bounds the average absolute distance between the values of pixels in $\tilde{x}$ and the corresponding pixels in x.
- $\boldsymbol{L^2}$ **norm:** bounds the total squared distance between the values of pixels in $\tilde{x}$ and the corresponding pixels in x. Often referred to as the Euclidean distance.
- $\boldsymbol{L^\infty}$ **norm:** bounds the maximum difference between any pixel in $\tilde{x}$ and the corresponding pixel in x. Often referred to as max norm.

$$\|\tilde{x} - x\|_\infty = \max_{i,j}\left(\left|\tilde{x}_{i,j} - x_{i,j}\right|\right)$$

Preferred one!

Impact of noise on classifiers

- **Expectation for Neural Networks:** Just like with human perception, small changes on inputs should not yield a different label!

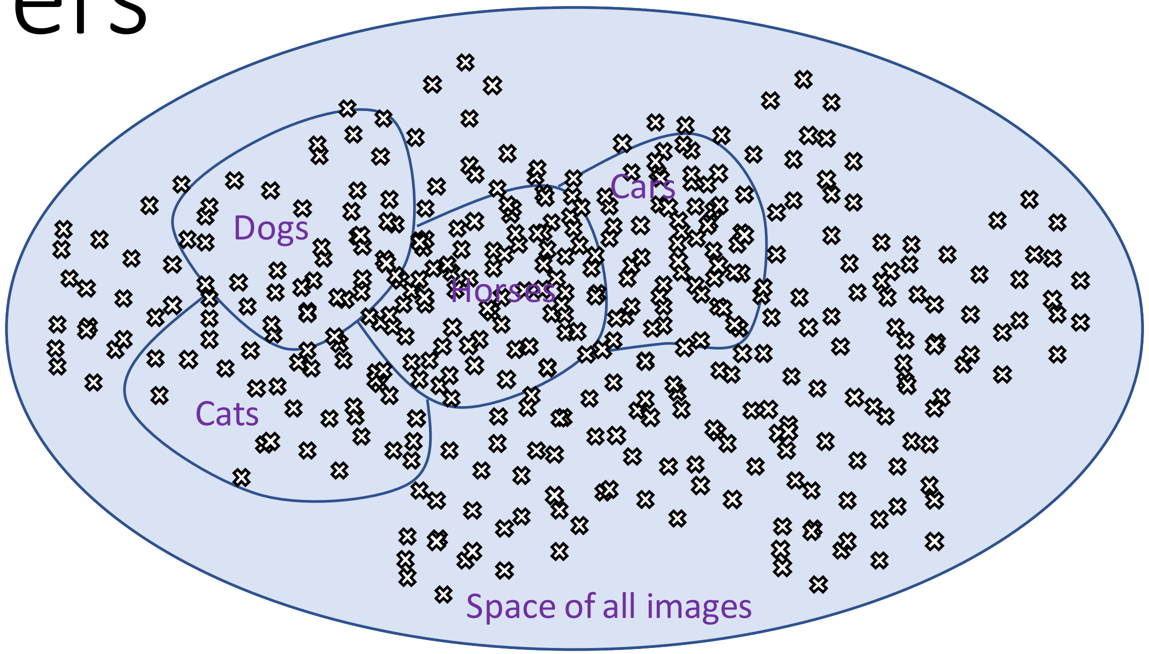
- **Unfortunately, deep learning predictions are different:** Deep learning algorithms process data differently from humans, with strong discontinuities in the change of prediction as a function the inputs.
- **And that is the reason for their vulnerability to attacks.**

- **Following: A tentative of explaining why that is the case (not the absolute truth, but my intuition as to why this happens!)**

Impact of noise on classifiers

Consider the arbitrary representation, with regions and boundaries between them.

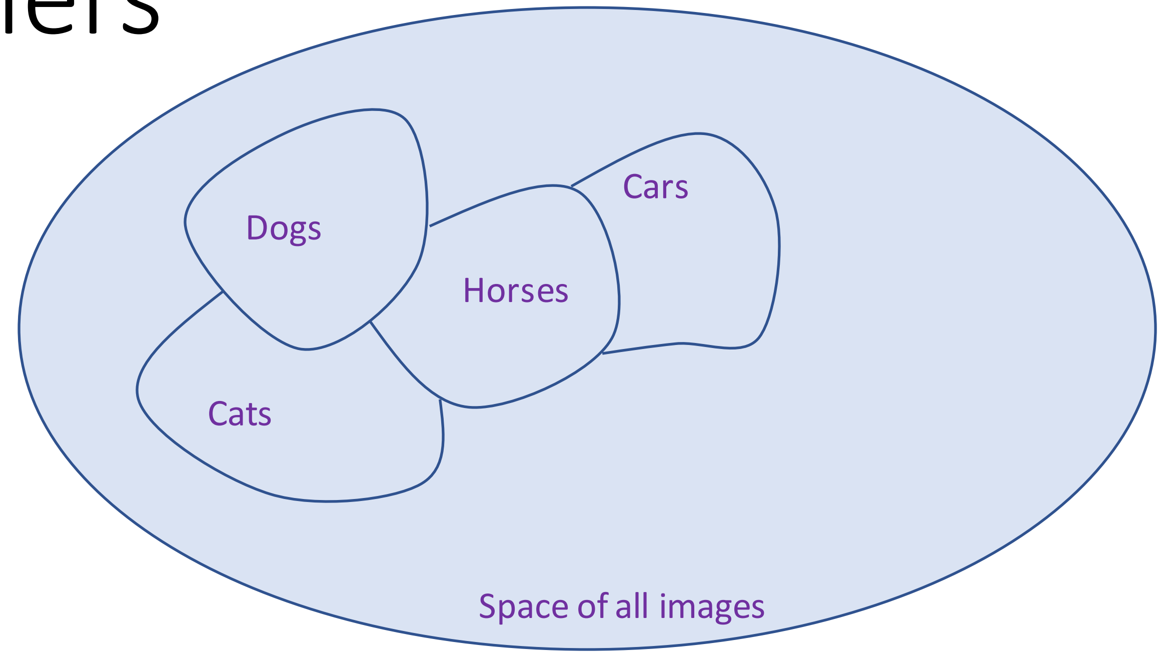
- **Misconception #1:** the whole space of possible inputs was densely filled with training examples during training.



Impact of noise on classifiers

Consider the arbitrary representation, with regions and boundaries between them.

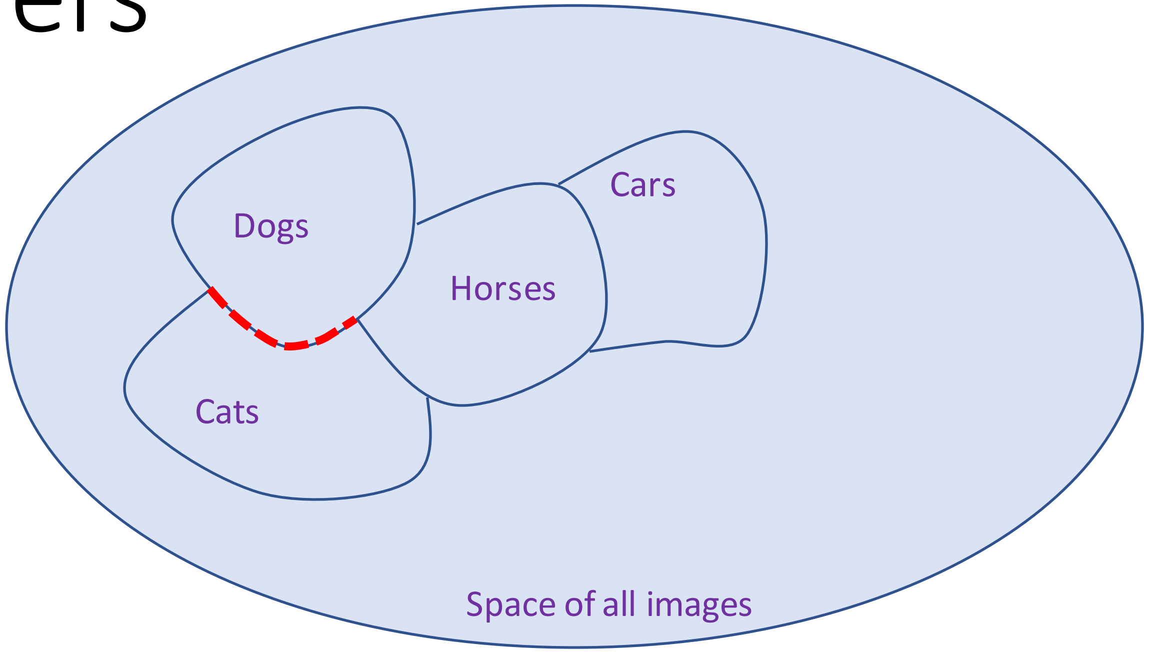
- **Misconception #1:** the whole space of possible inputs was densely filled with training examples during training.
- **Misconception #2:** regions are contiguous and filled with samples.

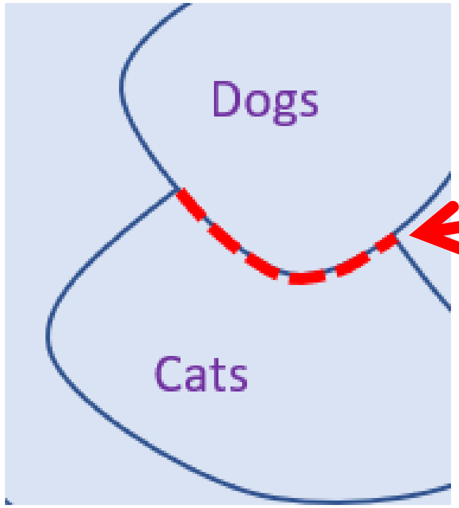


Impact of noise on classifiers

Consider the arbitrary representation, with regions and boundaries between them.

- **Misconception #1:** the whole space of possible inputs was densely filled with training examples during training.
- **Misconception #2:** regions are contiguous and filled with samples.
- **Misconception #3:** the decision boundaries between classes are smooth and make perfect sense.

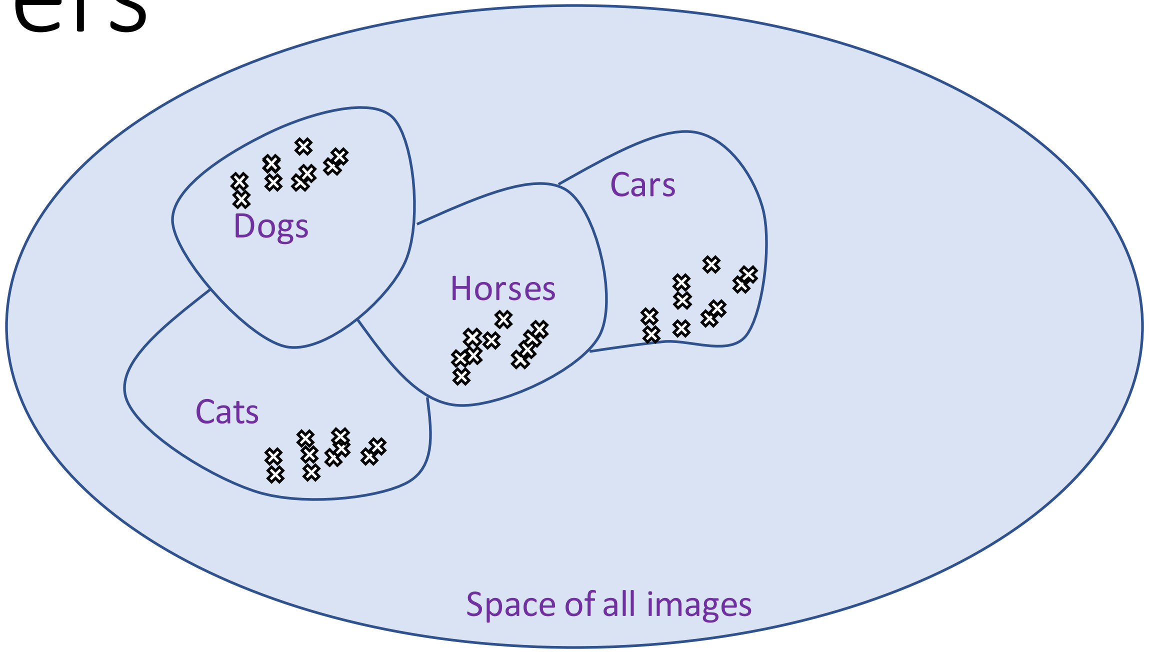




Has a clear logic, explaining the difference between dogs and cats

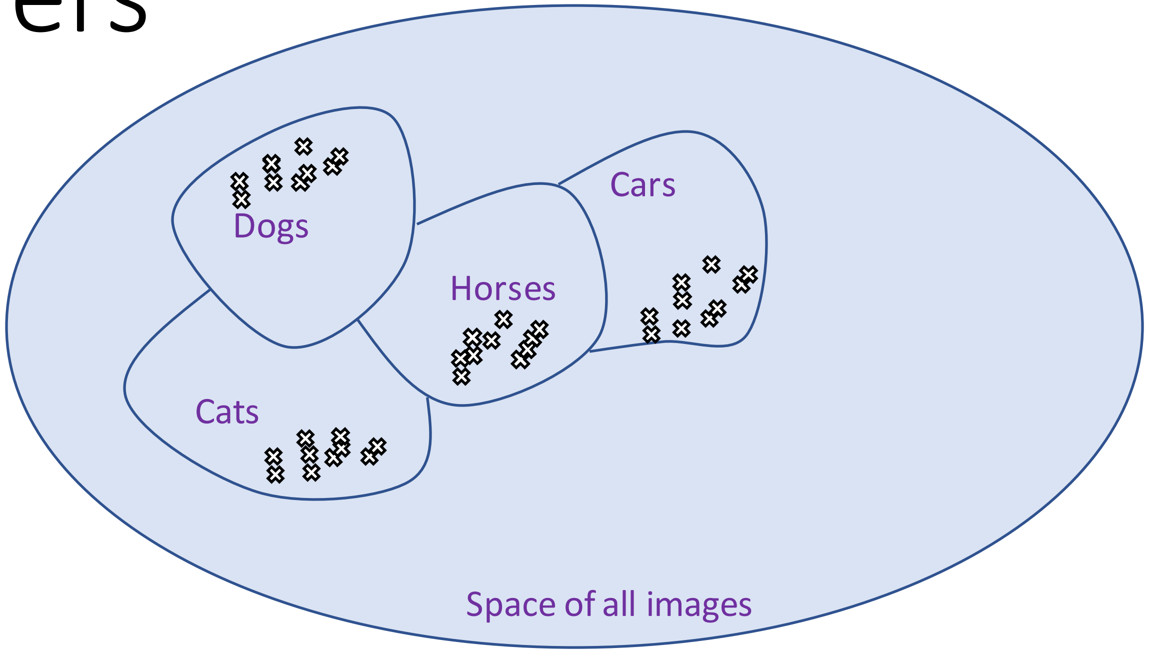
Impact of noise on classifiers

- **Misconception #1:** the whole space of possible inputs was densely filled with training examples during training.
- **Correction #1:** the space is mostly noise images and sparsely filled with relevant training examples.
- Also, the training samples do not cover for all possible relevant images.



Impact of noise on classifiers

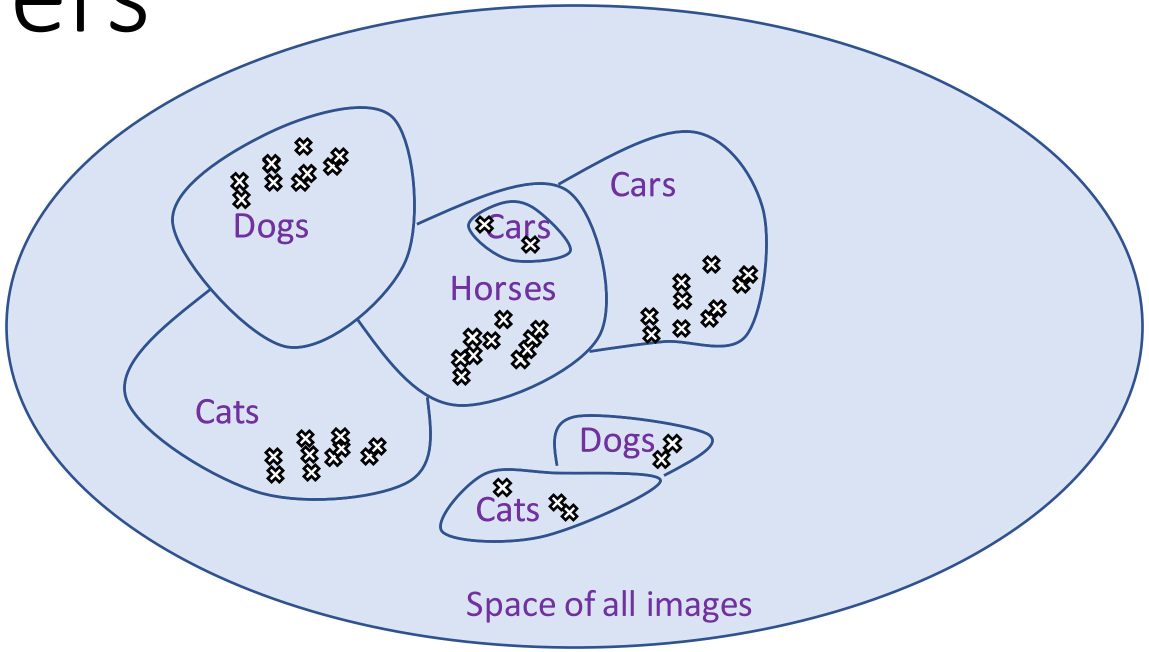
- **Misconception #1:** the whole space of possible inputs was densely filled with training examples during training.
- **Correction #1:** the space is mostly noise images and sparsely filled with relevant training examples.
- Also, the training samples do not cover for all possible relevant images.



E.g. MNIST contains 28 by 28 pixels images, with pixel values in $[0, 255]$. The whole space is roughly $(256)^{28\times 28} \approx 10^{1888}$ images, and most of them are noise. The MNIST dataset contains 60000 images only.

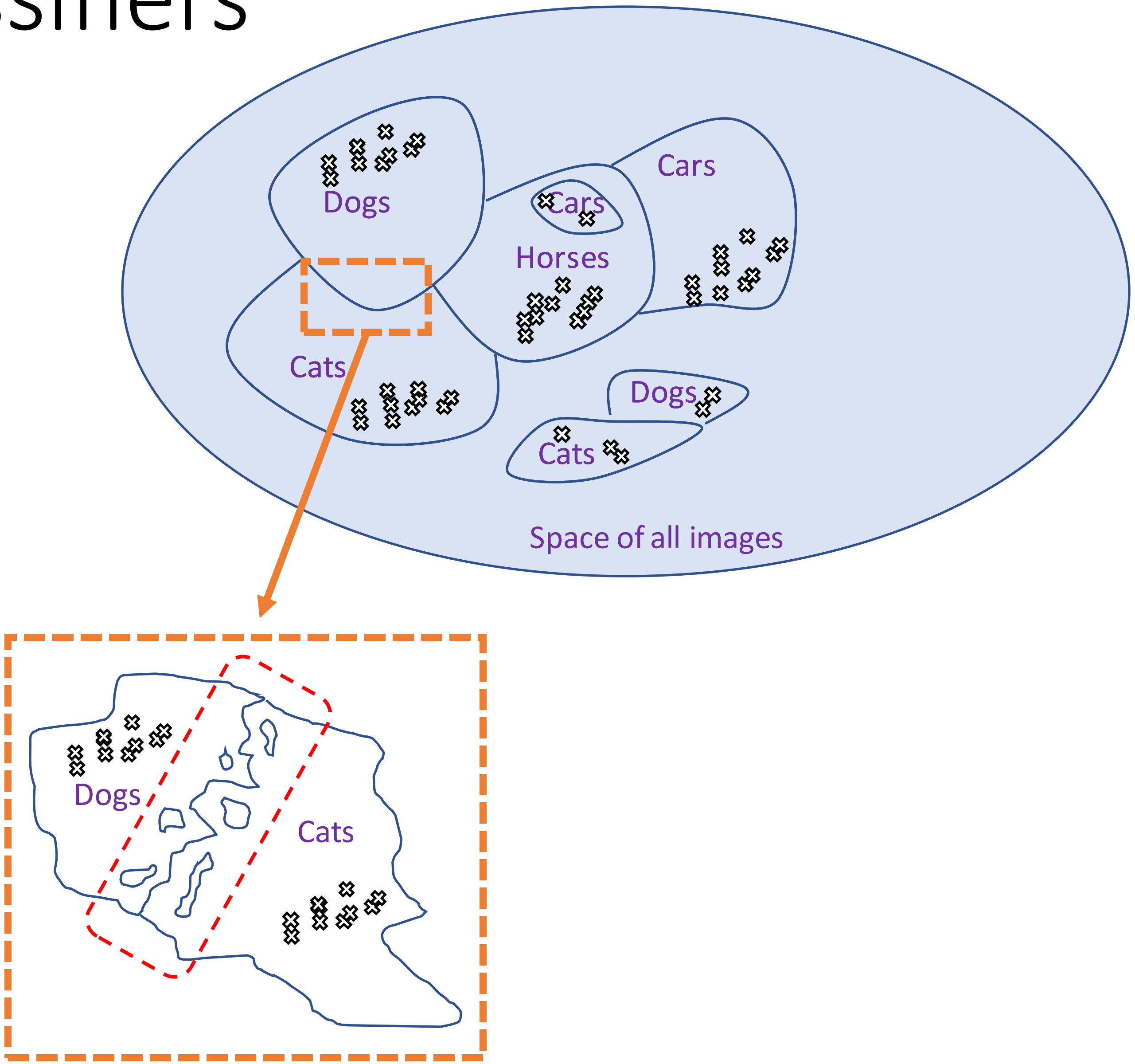
Impact of noise on classifiers

- **Misconception #2:** regions are contiguous and filled with samples.
- **Correction #2:** regions will not necessarily be contiguous.



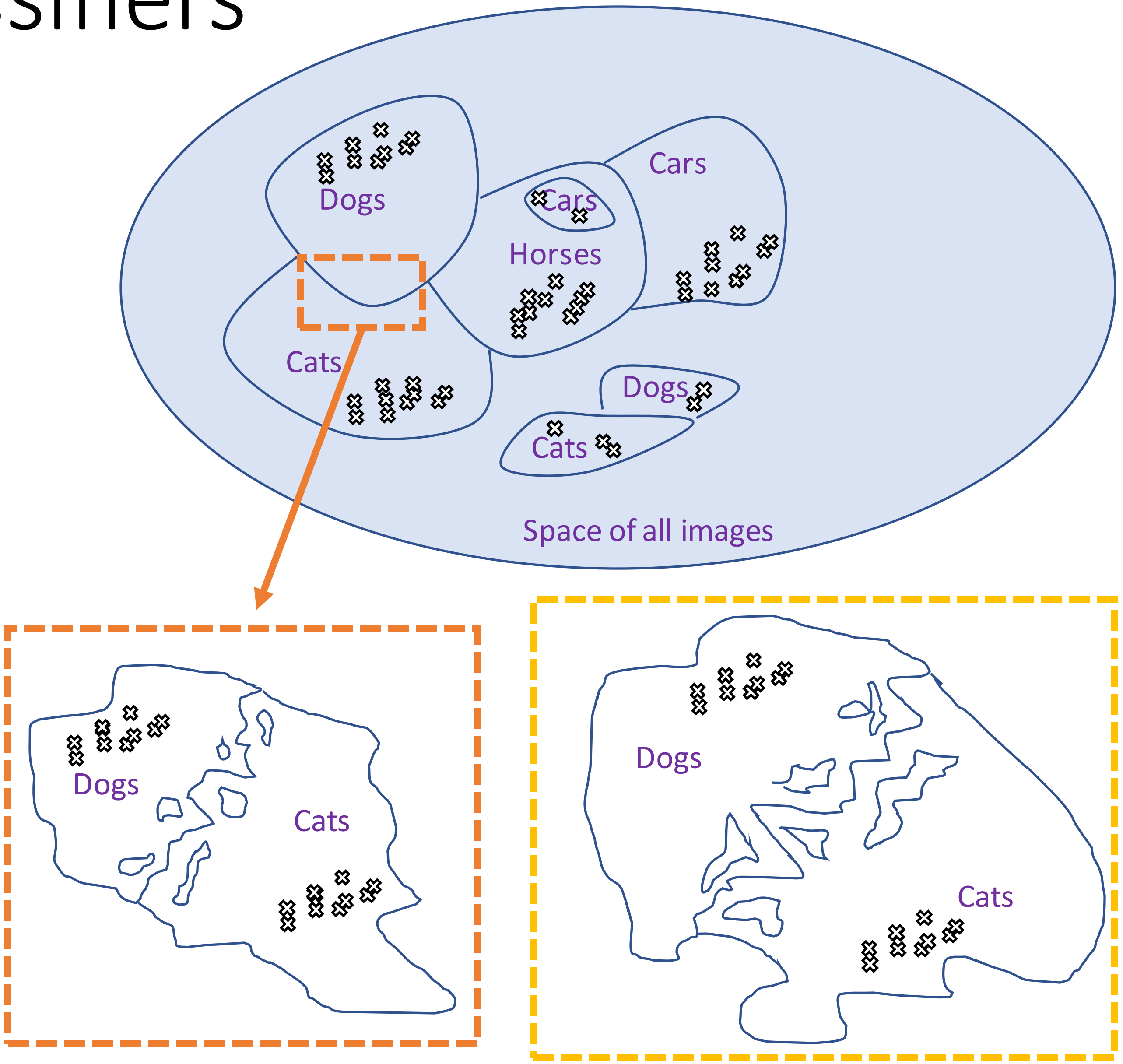
Impact of noise on classifiers

- **Misconception #2:** regions are contiguous and filled with samples.
- **Correction #2:** regions will not necessarily be contiguous. Boundaries between classes might even be very erratic!



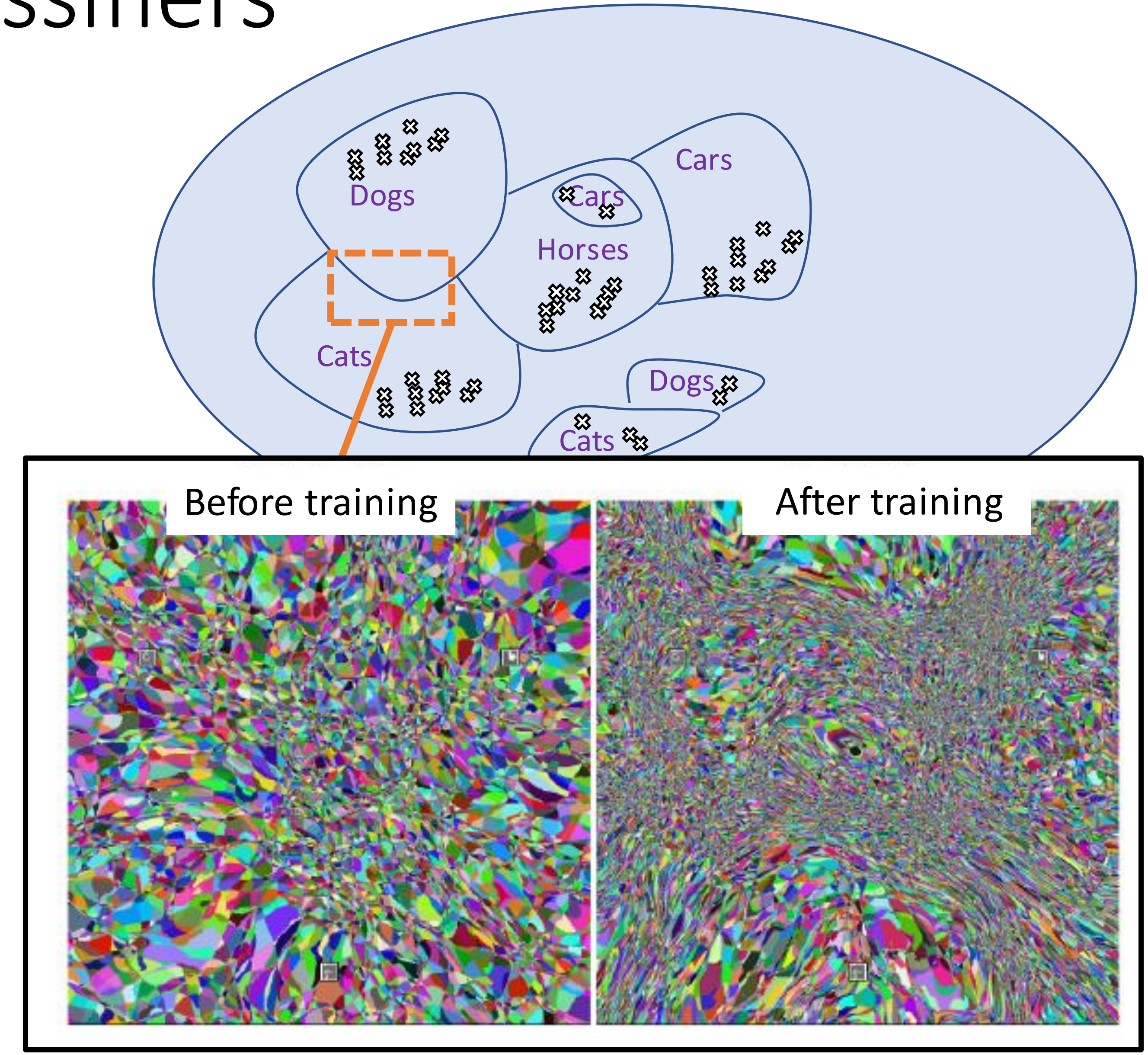
Impact of noise on classifiers

- **Misconception #3:** the decision boundaries between classes are smooth and make perfect sense.
- **Correction #3:** In fact, the boundaries between samples are often “randomly” decided.
- On different epochs, the boundaries might change randomly.



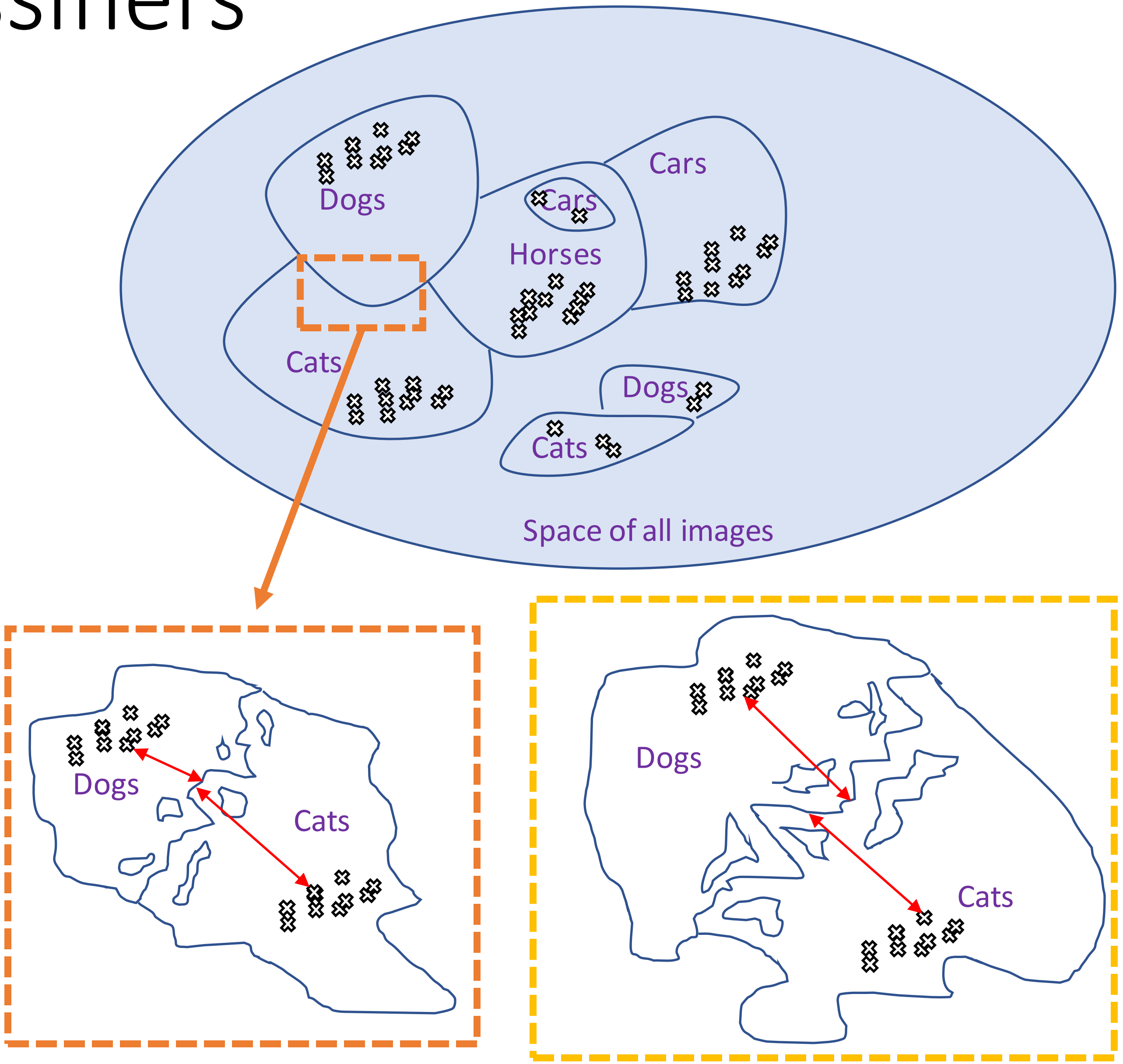
Impact of noise on classifiers

- **Misconception #3:** the decision boundaries between classes are smooth and make perfect sense.
- **Correction #3:** In fact, the boundaries between samples are often “randomly” decided.
- On different epochs, the boundaries might change randomly.



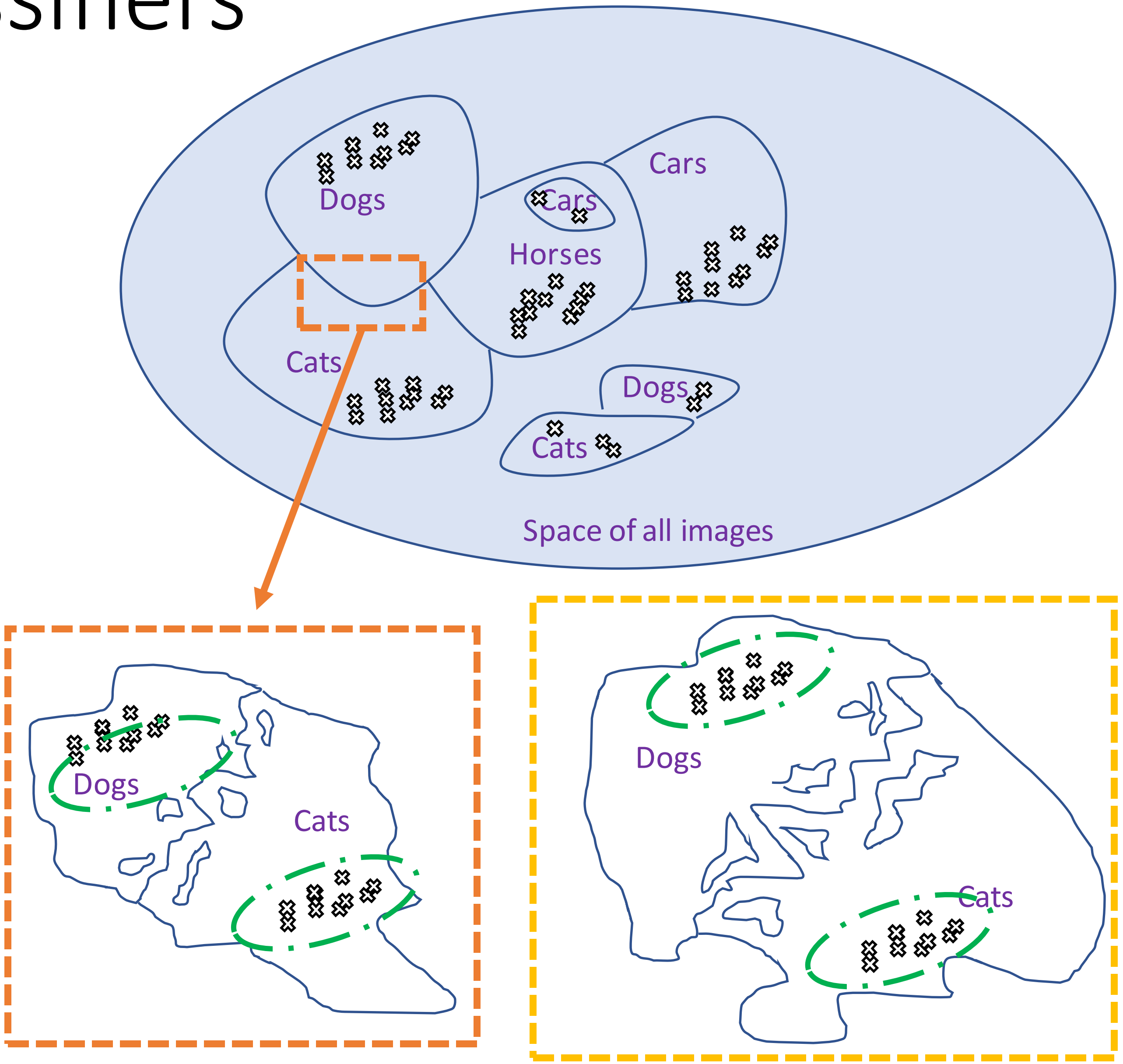
Impact of noise on classifiers

- **Misconception #3:** the decision boundaries between classes are smooth and make perfect sense.
- **Correction #3-bis:** Training samples are often condensed, far away from the boundaries.



Impact of noise on classifiers

Correction #3-bis: Training samples are often condensed, far away from the boundaries.



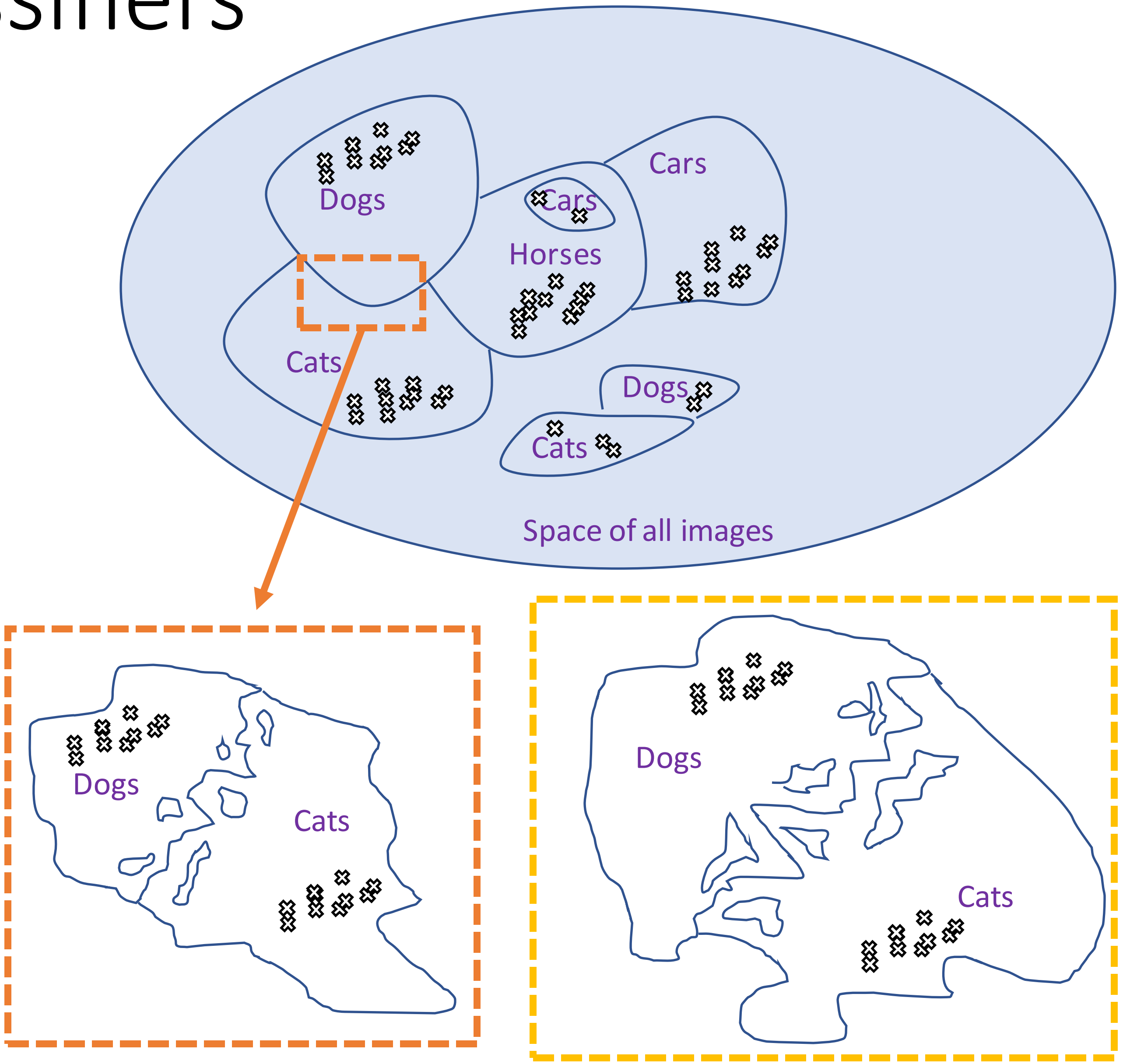
Definition (manifolds):

These **small regions containing a large number of training examples** are mathematically called **manifolds**.

See [TDS1], if curious.

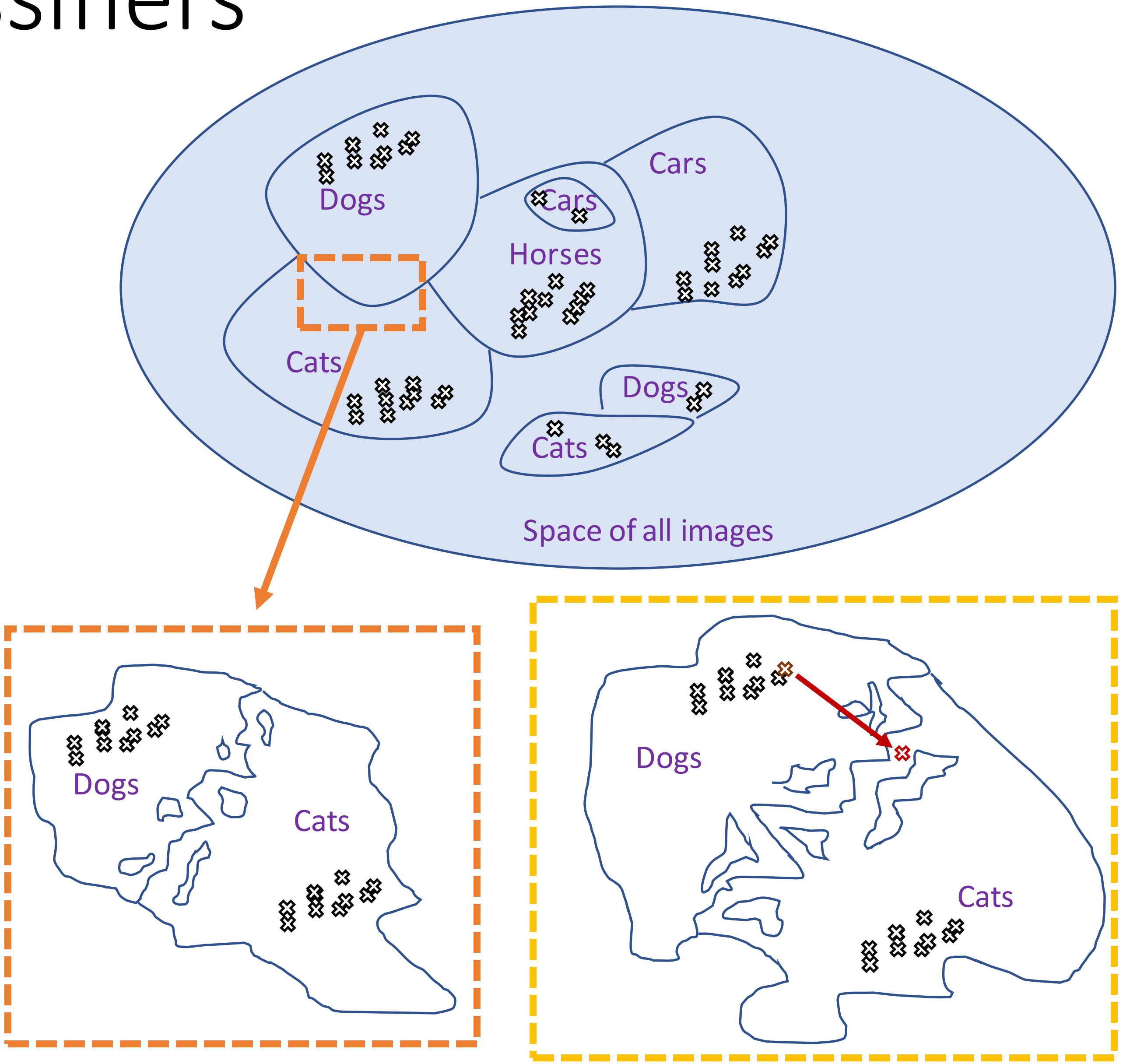
Impact of noise on classifiers

- **Misconception #3:** the decision boundaries between classes are smooth and make perfect sense.
- **Correction #3-bis:** Training samples are often condensed, far away from the boundaries.
- Boundaries decided by the Deep Learning models often exhibit the same behavior as the **Support Vector Machines** boundaries. But in a more random manner.



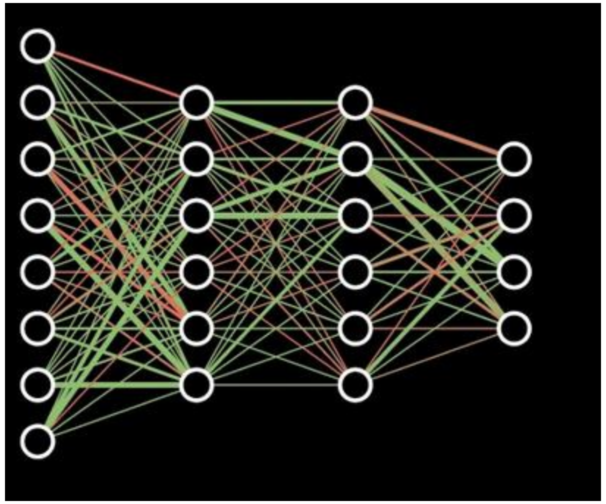
Impact of noise on classifiers

- **ENM prdocure, explained:** When randomly noising an **original sample** to make an **attack sample**, we move randomly in the feature map.
- We may even move in the boundary region, where the sample might become misclassified.
- The **attack sample** will therefore look similar to a dog picture, but will be misclassified as a cat.



Lessons from ENM attacks

Cat or Dog Picture (high dimensional input)



Neural Network, processing input image

Is the **presence of fur** relevant for discerning cats from dogs? **No, discard info**.

Is the **shape of eyes** relevant for discerning cats from dogs? **Yes, process info.**

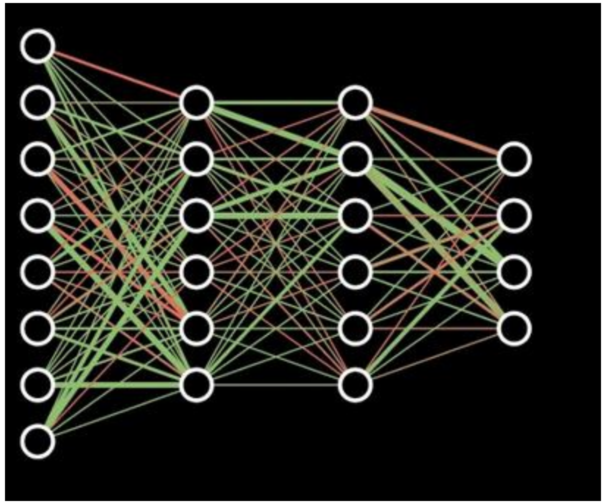
Here is my intuition about attacks. (Again, not claiming it is the universal truth, free from counter examples.)

- In Deep Learning, to learn means to understand that some objects are similar on certain aspects and use that for classification.
- It means to discard irrelevant information which does not contribute to understand those similarities.

Lessons from ENM attacks

- Discarding information is achieved by progressively compressing input data into a space of lower dimensionality using several Neural Networks layers.
- A linear neuron would achieve this, by mapping inputs onto a lower-dimension hyperplane, and learns projections of the training data.

Cat or Dog Picture (high dimensional input)



Neural Network, processing input image

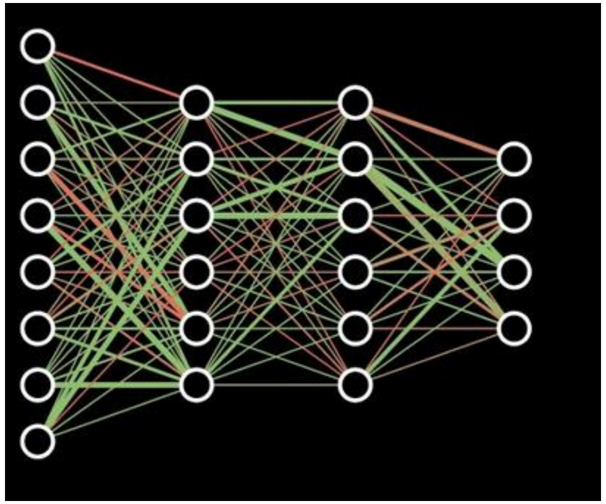
$$\boldsymbol{a} = [a_1 \quad a_2 \quad \dots \quad a_n].$$

Lower dimensional vector produced by final hidden layer of Neural Network.

Lessons from ENM attacks

- Neural Network then aims to map similar objects close together in this low dimensionality space.
- I.e., all pictures of cats will produce vectors a, with roughly similar values. Same for dogs.
- Cats vectors and dogs vectors will, however, be very different!
- Final layer implements binary decision on vector a.


Cat or Dog Picture (high dimensional input)



Neural Network, processing input image

$$\boldsymbol{a} = [\, a_1 \quad a_2 \quad \dots \quad a_n \,].$$

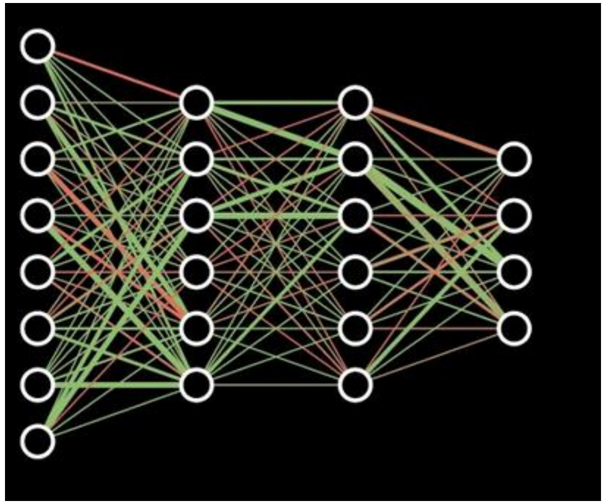
Lower dimensional vector produced by final hidden layer of Neural Network.

Lessons from ENM attacks

- Due to compression, there are many directions in a high dimension feature space along which a small step might lead to big changes in predictions.
- In zones with low training data densities, the decision boundaries can lie very close together, because they were never properly learned from training samples!


Cat or Dog Picture (high dimensional input)

Small change here…



Neural Network, processing input image

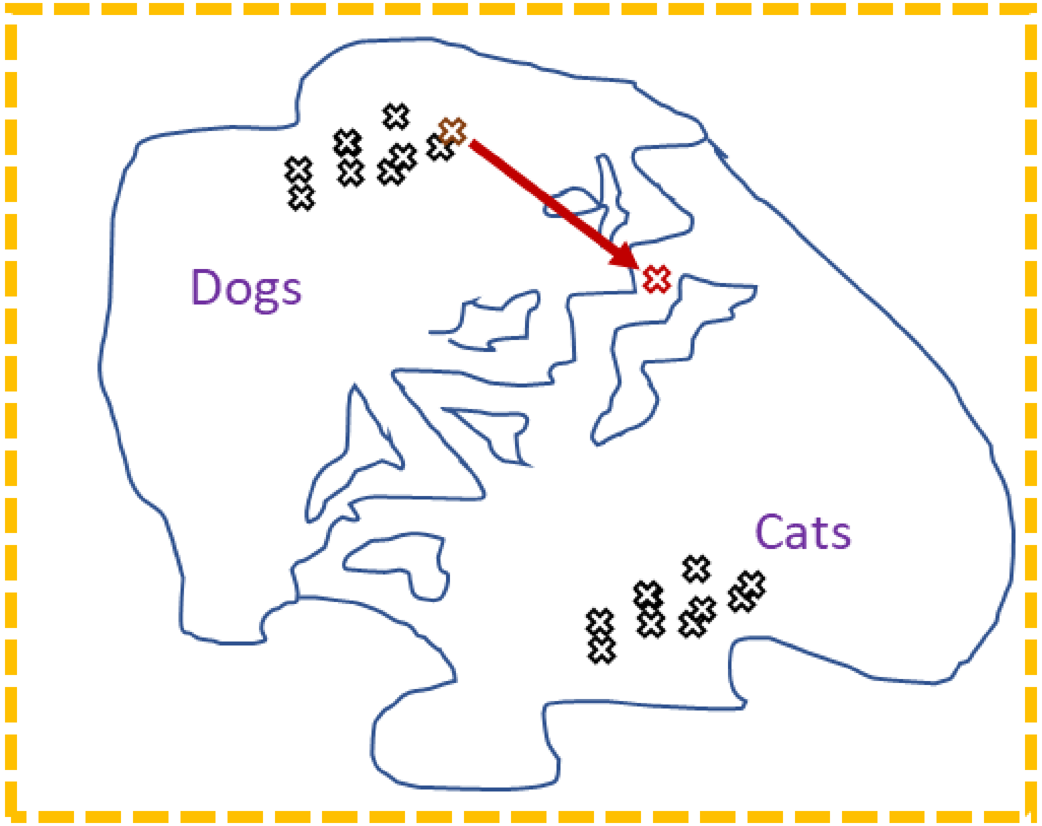
… Huge change here!

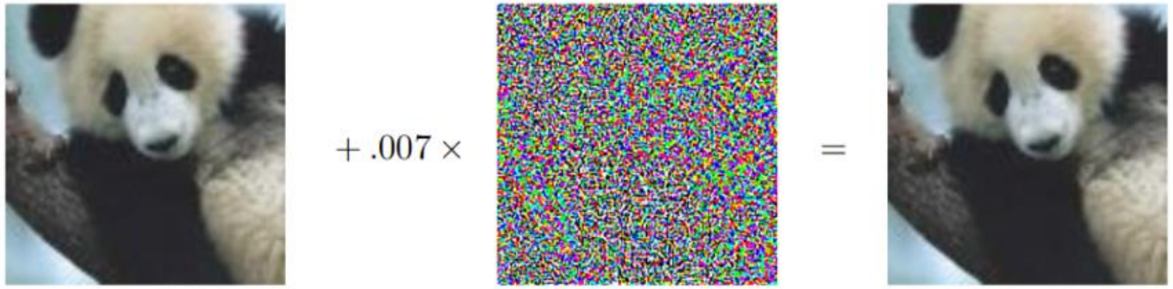
$$\boldsymbol{a} = [a_1 \quad a_2 \quad \ldots \quad a_n].$$

Lower dimensional vector produced by final hidden layer of Neural Network.

Lessons from ENM attacks

- In zones with low training data densities, some small changes on an input can then lead to big changes in predictions produced by the trained Neural Network.
- And that is what we exploit to generate **attack samples!**





"panda"
57.7% confidence

"nematode"
8.2% confidence

"gibbon"
99.3 % confidence

Lessons from ENM attacks

→ Unfortunately, this means that all deep learning models will always be susceptible to attacks.

(And that is something we have to accept.)

Lessons from ENM attacks

→ Unfortunately, this means that all deep learning models will always be susceptible to attacks.

(And that is something we have to accept.)

So what? Is that it then?

Are Neural Networks flawed beyond repair?

Are we giving up on Neural Networks then?

Reason #2: Defense

Definition (Defense on Neural Networks):

In adversarial machine learning, **defense** refers to machine learning techniques that attempt to **protect models from being attacked** by malicious attempts.

Reason #2: Defense

Definition (Defense on Neural Networks):

In adversarial machine learning, **defense** refers to machine learning techniques that attempt to **protect models from being attacked** by malicious attempts.

Important: defense mechanisms often rely on an understanding of how attacks work.

SOMETHING FOR LATER...

Conclusion (W8S1)

- Attacks on Neural Networks, a definition.
- Our first attack, simply by noising images.
- Attacks exploits intrinsic limitations that deep learning models will always have.
- We can defend against attacks.
- The random noise attacks are very random with unpredictable efficacy.
- These random noise attacks will not necessarily move the samples towards the boundary region.
- Advanced attacks techniques will improve on this matter.
  More on this later!
- Defense mechanics will attempt to prevent these attacks from working.

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [Xie2017] Xie at al., “Adversarial Examples for Semantic Segmentation and Object Detection”, 2017. https://arxiv.org/abs/1703.08603
- [Szegedy2013] Szegedy et al., “Intriguing properties of neural networks”, 2013. https://arxiv.org/abs/1312.6199

Learn more about these topics

- [Moosavi2017] Moosavi-Dezfooli et al., “Universal adversarial perturbations”, 2017. https://arxiv.org/abs/1610.08401

- [Hayes2017] Hayes et al., “Learning Universal Adversarial Perturbations with Generative Models”, 2017. https://arxiv.org/abs/1708.05207

- [Goodfellow2018] Goodfellow et al., “Making machine learning robust against adversarial inputs”, 2018. https://dl.acm.org/doi/10.1145/3134599

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [Verge1] Google's AI thinks this turtle looks like a gun, which is a problem: https://www.theverge.com/2017/11/2/16597276/google-ai-image-attacks-adversarial-turtle-rifle-3d-printed

- [Spectrum1] Slight Street Sign Modifications Can Completely Fool Machine Learning Algorithms: https://spectrum.ieee.org/cars-that-think/transportation/sensors/slight-street-sign-modifications-can-fool-machine-learning-algorithms

Learn more about these topics

- [Verge2] These glasses trick facial recognition software into thinking you're someone else: https://www.theverge.com/2016/11/3/13507542/facial-recognition-glasses-trick-impersonate-fool
- [YTB1] Defeating Facial Recognition: https://www.youtube.com/watch?v=tbdcL5Ux-9Y
- [TDS1] Manifolds in Data Science — A Brief Overview: https://towardsdatascience.com/manifolds-in-data-science-a-brief-overview-2e9dde9437e5