

# Chapter 1 - Introduction

## Chapter 1

### Diving right in

ResNet50 모델을 이용해 돼지 이미지를 분류



일반적으로 PyTorch에서 image classification을 하기 위해서는 ( 대략적으로 zero-mean, unit variance ) torch.vision.transforms 를 이용해 이미지 변환을 먼저 하는게 대부분이다.

하지만 우리는 원본(unnormalized) 이미지에다가 perturbation을 하는 것이 목적이기 때문에 다른 방식으로 접근해야하고 PyTorch layers 에다가 transformation을 해줘야 한다.

먼저 이미지를 224\*224 크기로 resize 해준다. (대부분의 ImageNet 이미지 및 pretrained 된 classifier가 받는 기본 크기)

```
from PIL import Image from torchvision import transforms # read the
image, resize to 224 and convert to PyTorch Tensor pig_img =
Image.open("pig.jpg") preprocess = transforms.Compose([
transforms.Resize(224), transforms.ToTensor(), ]) pig_tensor =
preprocess(pig_img)[None,:,:,:] # plot image (note that numpy using HWC
whereas Pytorch user CHW, so we need to convert)
plt.imshow(pig_tensor[0].numpy().transpose(1,2,0))
```

필요한 transform을 거친 후 pre-trained 된 ResNet50 모델을 가져와서 이미지에 적용해봅시다.

(한 가지 주의할 점은 PyTorch standards를 준수하기 위해 모듈에 대한 모든 input들은 batch\_size\*num\_channels\*height\*width 형태로 들어가야 한다는 점)

```
import torch import torch.nn as nn from torchvision.models import
resnet50 # simple Module to normalize an image class
Normalize(nn.Module): def __init__(self, mean, std): super(Normalize,
self).__init__() self.mean = torch.Tensor(mean) self.std =
torch.Tensor(std) def forward(self, x): return (x - self.mean.type_as(x))
[None,:,:None,None]) / self.std.type_as(x)[None,:,:None,None] # values are
standard normalization for ImageNet images, # from
https://github.com/pytorch/examples/blob/master/imagenet/main.py norm =
Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # load
pre-trained ResNet50, and put into evaluation mode (necessary to e.g.
turn off batchnorm) model = resnet50(pretrained=True) model.eval();
```

```
# form predictions pred = model(norm(pig_tensor))
```

예측값은 1000개의 imagenet class에 대한 class logit 정보가 담긴 1000 dimension의 vector를 포함하고 있다. (이를 probability vector로 변환하고 싶다면 이 벡터값에다가 softmax를 적용하면 된다.) 가장 높은 likelihood class를 찾기 위해 단순히 최대값의 인덱스를 가져오고 imagenet class에서 해당하는 label을 찾아보기로 했다.

```
import json with open("imagenet_class_index.json") as f: imagenet_classes  
= {int(i):x[1] for i,x in json.load(f).items()}  
print(imagenet_classes[pred.argmax(dim=1)[1].item()])
```

hog

ImageNet에서는 pig==hog이다. 결과가 잘 나왔음을 알 수 있다.

## Some introductory notation

# Chapter 1

Some introductory notation

## Model

$h_\theta: X \rightarrow \mathbb{R}^K$  Hypothesis function [Input space  $\rightarrow$  Output space]  
 $\downarrow$   
3-dimensional tensor

output  $\Rightarrow$  logit space  
real-valued numbers (positive or negative)

$K$ : number of classes

$\theta$ : model parameters (to be optimized)  
(all convolutional filters,  
weight matrices, biases)

$h_\theta$ : model

## Loss function

$l: \mathbb{R}^K \times \mathcal{Z}_+ \rightarrow \mathbb{R}_+$  mapping model predictions and true labels  
to non-negative number

$\mathbb{R}^K$ : model output (logits); positive or negative

$\mathcal{Z}_+$ : index of true class ( $1 \sim K$ )

$l(h_\theta(x), y)$  loss that the classifier achieves in its predictions on  $x$ , assuming the true class  $y$ .

$x \in X$  : input

$y \in \mathcal{Z}$  : true class

### Cross Entropy Loss (= Softmax loss)

$$l(h_\theta(x), y) = \log \left( \sum_{j=1}^k \exp(h_\theta(x)_j) \right) - h_\theta(x)_y$$

$h_\theta(x)_j$  : jth elements of the vector  $h_\theta(x)$   
 (x의 예측값에 대한 j번째 값)

### Softmax activation

$$\sigma: \mathbb{R}^k \rightarrow \mathbb{R}^k$$

$$\sigma(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^k \exp(z_j)}$$

mapping from the class logits returned by  $h_\theta$  to a probability distribution.

True class label의 probability를 maximize 하는 것인 특성.

probability 자체는 자기 vanishing 하므로 일반적으로 true class label의 probability의 log값을 최대화 해준다.

$$\begin{aligned} \log \sigma(h_\theta(x))_y &= \log \left( \frac{\exp(h_\theta(x)_y)}{\sum_{j=1}^k \exp(h_\theta(x)_j)} \right) \\ &= h_\theta(x)_y - \log \left( \sum_{j=1}^k \exp(h_\theta(x)_j) \right) \end{aligned}$$

↳ 음의 log값을 이용해 probability가 낮은 경우 penalty를 더 크게 부여 

## Creating an adversarial example

$$\underset{\theta}{\text{minimize}} \frac{1}{m} \sum_{i=1}^m l(h_{\theta}(x_i), y_i)$$

training a classifier to optimize parameter  $\theta$ , so as to minimize  
 the average loss over some training set  $\{x_i \in X, y_i \in Z\}$   
 $i = 1, \dots, m$

$$\theta := \theta - \frac{\alpha}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\theta} l(h_{\theta}(x_i), y_i)$$

compute gradient of our loss with respect to parameters  $\theta$

$\mathcal{B} \subseteq \{1, \dots, m\}$  : mini batch.       $\alpha$  : Step size.

repeat this process for different minibatches covering the entire training set, until the parameters convergence

$$\nabla_{\theta} l(h_{\theta}(x_i), y_i)$$

각 parameter의 변화량이 loss function에 미치는 영향력을 계산

$$\underset{\hat{x}}{\text{maximize}} l(h_{\theta}(\hat{x}), y)$$

$\theta$ 에 대한 gradient를 구하는게 아니라  $x$ 에 대한 gradient를 구하게 되면

image에서의 변화가 loss function에 얼마나 영향을 주는지 계산 가능

→ AE를 형성하기 위해 이용됨.

+ loss 값을 maximize 해야 함.

$\hat{x}$  : AE  $\sim x$ 와 최대한 비슷해야 된다.

$$\underset{\delta \in \Delta}{\text{maximize}} \ l(h_{\theta}(x + \delta), y)$$

$\delta$  : perturbation to  $x$

$\Delta$  : allowable set of perturbations

→  $x$ 에 적용한 이후에도 사람이 보기에는  $x$ 와 같아야 함.

수학적으로 허용되어야 하는 모든 perturbation에 대한 rigorous definition을 줄 수 없기 때문에 AE를 만들 때는 perturbation이 가능한 공간의 subset을 고려하는 것이다. reasonable한 정의 하에 어떠한 image의 실제 semantic content는 이 perturbation으로 변경될 수 없다.

$$\Delta = \{ \delta : \|\delta\|_{\infty} < \epsilon \}$$

$\|\cdot\|_{\infty}$  norm a vector  $\vec{z}$

$$\|\vec{z}\|_{\infty} = \max_i |z_i|$$

$$\|\delta\|_{\infty} = \max_i |\delta_i| \Rightarrow \delta \text{ 성분 중 가장 큰 값이}$$

은보다 작은 벡터가 allowable

set에 해당함.

과연 L-infinity norm에 기반한 방법이 reasonable한가는 생각해볼 필요가 있다. L-infinity ball의 장점은 아주 작은  $\epsilon$ 에 대해서는 이미지의 각 pixel에 대해 small component를 더해줌으로써 perturbation을 생성해준다는 것이다. 그리고 perturbation이 들어간다하더라도 original image와 indistinguishable하다는 것이다.

다음 예시는 PyTorch의 SGD optimizer를 통해 perturbation을 input 값에다가 적용하여 loss값을 maximize하는 코드이다.

아래 코드에서는 각 step에서 L-infinity ball에 대해 projection back을 수행해줬기 때문에 이는 사실 PGD를 적용한 것이다. (simply clipping the values that exceed  $\epsilon$  magnitude to  $\pm\epsilon$ .)

```
import torch.optim as optim epsilon = 2./255 delta =
torch.zeros_like(pig_tensor, requires_grad=True) opt = optim.SGD([delta],
lr=1e-1) for t in range(30): pred = model(norm(pig_tensor + delta)) loss
= -nn.CrossEntropyLoss()(pred, torch.LongTensor([341])) if t % 5 == 0:
print(t, loss.item()) opt.zero_grad() loss.backward() opt.step()
delta.data.clamp_(-epsilon, epsilon) print("True class probability:",
nn.Softmax(dim=1)(pred)[0,341].item())
```

```
0 -0.0038814544677734375 5 -0.00693511962890625 10 -0.015821456909179688
15 -0.08086681365966797 20 -12.229072570800781 25 -14.300384521484375 True
class probability: 1.4027455108589493e-06
```

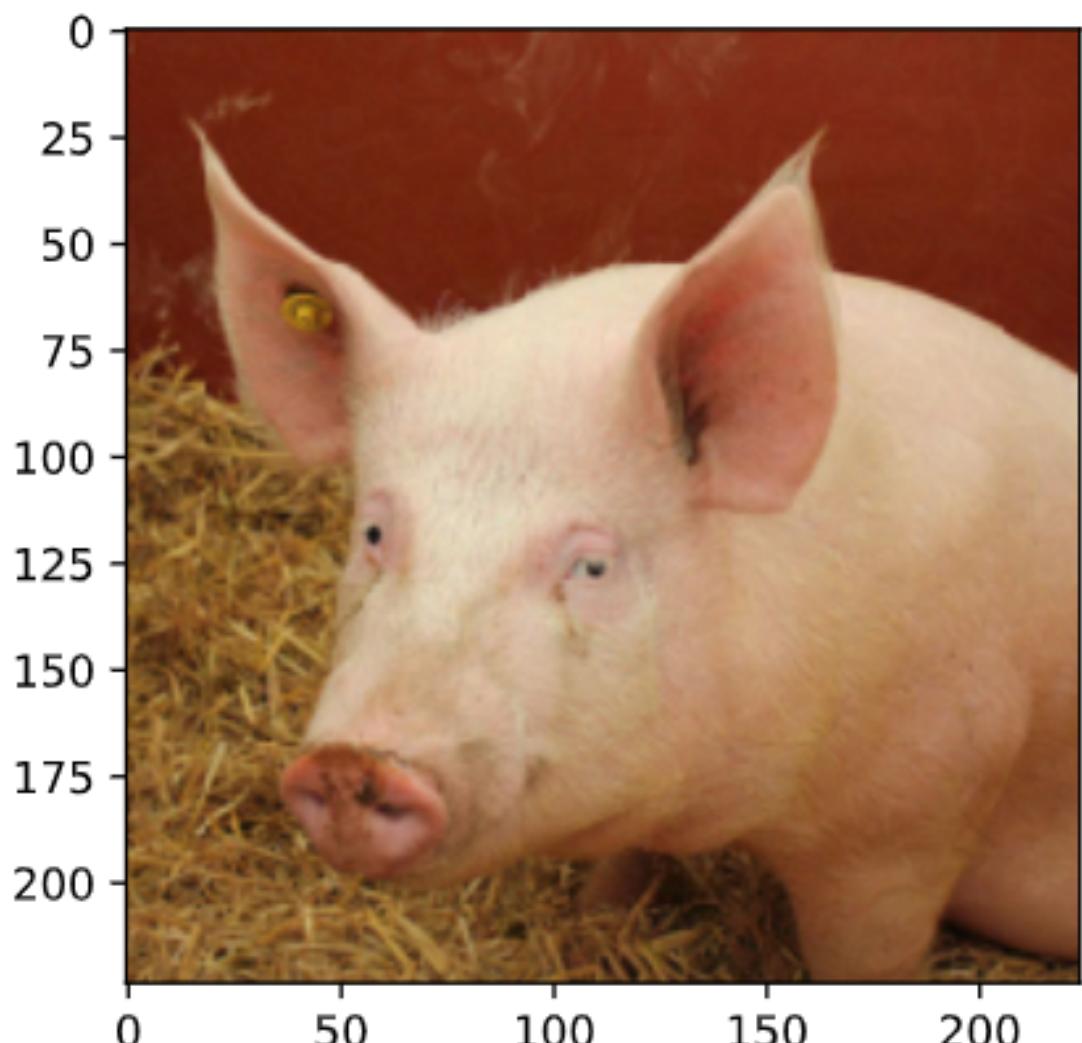
30번의 step 이후에는 ResNet50에서 이 이미지를 0.00001의 확률 아래로 pig라고 판단하게 된다. 그리고 꽤 확실하게 wombat으로 분류하게 된다.

```
max_class = pred.max(dim=1)[1].item() print("Predicted class: ",
imagenet_classes[max_class]) print("Predicted probability:",
nn.Softmax(dim=1)(pred)[0,max_class].item())
```

```
Predicted class: wombat Predicted probability: 0.9997960925102234
```

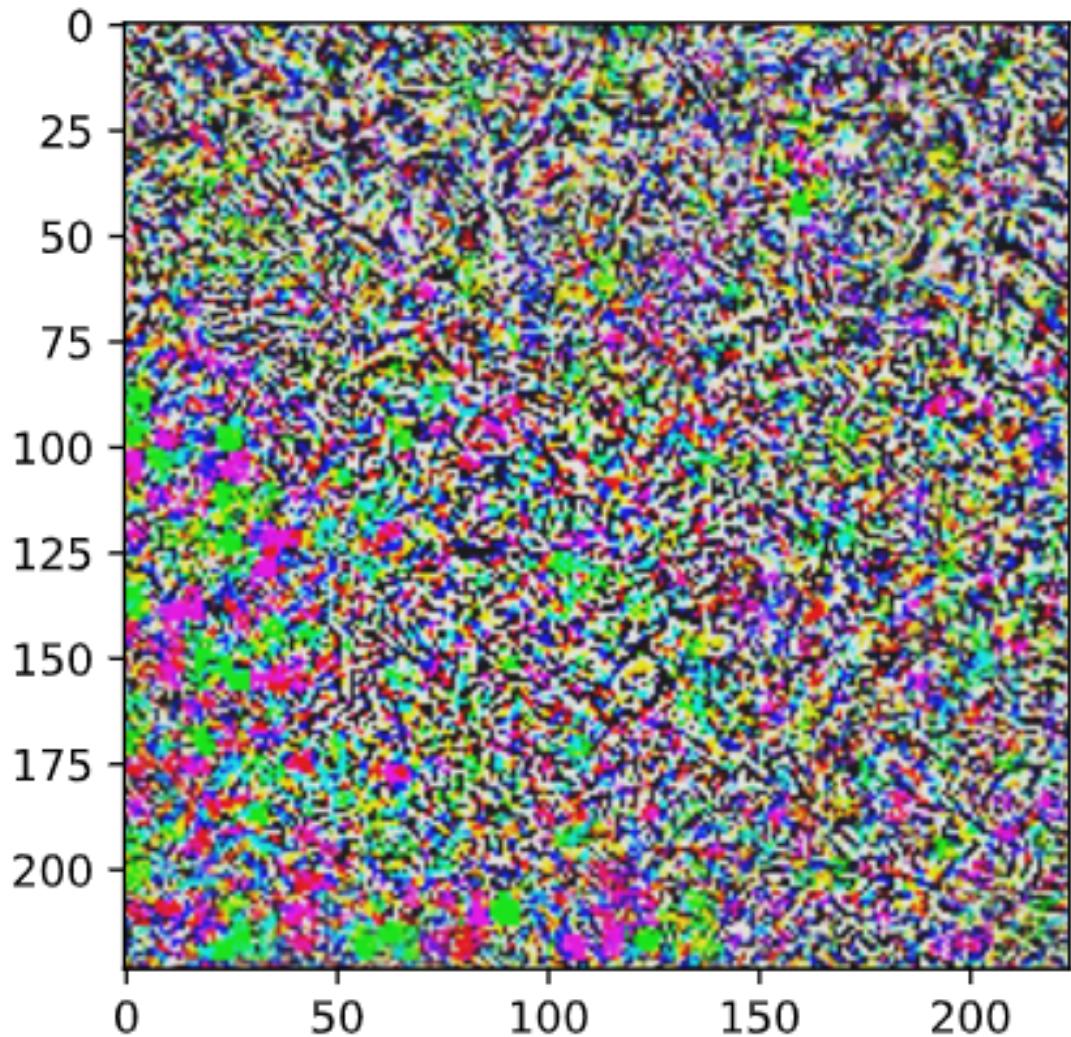
하지만 pig\_tensor에다가 delta만큼의 perturbation을 넣은 이미지를 출력했을 때 사람 눈에는 돼지처럼 보인다.

```
plt.imshow((pig_tensor + delta)[0].detach().numpy().transpose(1,2,0))
```



delta 값을 50배 zoom해서 출력해보면 아래와 같이 나온다.

```
plt.imshow((50*delta+0.5)[0].detach().numpy().transpose(1,2,0))
```



이러한 tiny multiple random-looking noise를 추가함으로써 우리 눈에는 original image와 동일하게 보이지만 잘못 분류되는 이미지를 만들 수 있다.

## Targeted attacks

어떤 이미지를 우리가 원하는 class로 misclassify 시킬 수도 있다. 앞서 살펴본 바와 한 가지 차이점은 그냥 단순히 correct class의 loss값을 maximize 하는 것이 아니라 거기 에다가 추가로 우리가 원하는 class의 loss값을 minimize하는 것이다.

해당 optimization problem은 아래와 같이 쓸 수 있다.

$$\begin{aligned}
 & \underset{\delta \in \Delta}{\text{maximize}} \left( l(h_\theta(x+\delta), y) - l(h_\theta(x+\delta), y_{\text{target}}) \right) \\
 & \quad \rightarrow \text{true label에 대한 loss} \uparrow \quad \rightarrow \text{target label에 대한 loss} \downarrow \\
 & \equiv \underset{\delta \in \Delta}{\text{maximize}} \left( h_\theta(x+\delta)_{y_{\text{target}}} - h_\theta(x+\delta)_{y} \right)
 \end{aligned}$$

```

delta = torch.zeros_like(pig_tensor, requires_grad=True) opt =
optim.SGD([delta], lr=5e-3) for t in range(100): pred =
model(norm(pig_tensor + delta)) loss = (-nn.CrossEntropyLoss()(pred,
torch.LongTensor([341])) + nn.CrossEntropyLoss()(pred,
torch.LongTensor([404]))) if t % 10 == 0: print(t, loss.item())
opt.zero_grad() loss.backward() opt.step() delta.data.clamp_(-epsilon,
epsilon)

```

```

0 24.00604820251465 10 -0.1628284454345703 20 -8.026773452758789 30 -15.6
77117347717285 40 -20.60370635986328 50 -24.99606704711914 60 -31.0098495
48339844 70 -34.80946350097656 80 -37.928680419921875 90 -40.323955535888
67

```

`torch.LongTensor([341])`에 해당하는 class가 pig이고 `torch.LongTensor([404])`에 해당하는 class가 airliner이다. 즉, target class는 airliner이다.

```

max_class = pred.max(dim=1)[1].item() print("Predicted class: ",
imagenet_classes[max_class]) print("Predicted probability:",
nn.Softmax(dim=1)(pred)[0,max_class].item())

```

```

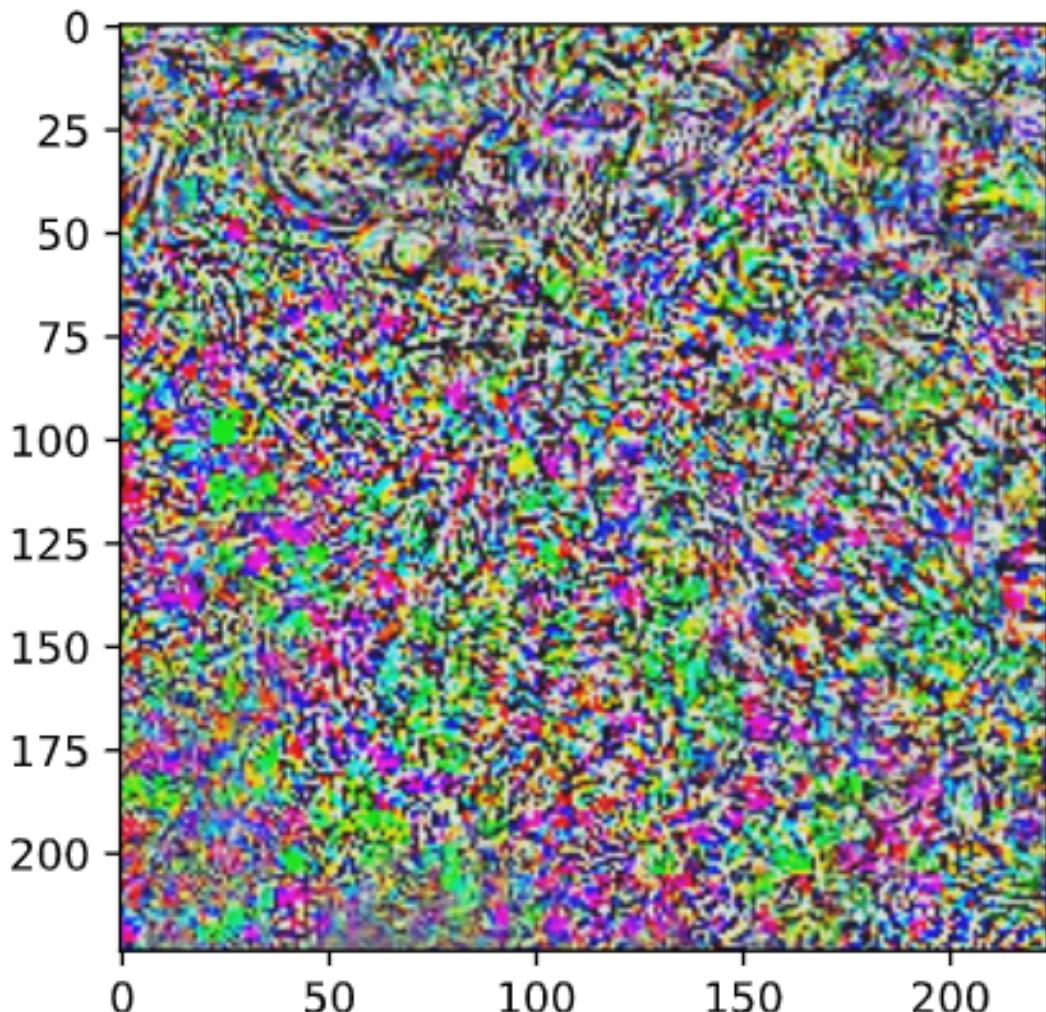
Predicted class: airliner Predicted probability: 0.9679961204528809

```

predicted class가 airliner로 나온것을 보아 targeted attack이 잘 된 것을 알 수 있다.

Airliner noise를 출력해보면 아래와 같다.

```
plt.imshow((50*delta+0.5)[0].detach().numpy().transpose(1,2,0))
```



## A brief (incomplete) history of adversarial robustness

- Origins (robust optimization)
- Support vector machines
- Adversarial classification (e.g. Domingos 2004)
- Distinctions between different types of robustness (test test, train time, etc)
- Szegedy et al, 2003, Goodfellow et al, 2004
- Many proposed defense methods
- Many proposed attack methods
- Exact verification methods

- Convex upper bound methods
- Recent trends

## Adversarial robustness and training

### Brief review : risk, training, and testing sets

Machine learning에서 사용되는 risk라는 개념에 대해 알아봅시다. Classifier의 risk는 표본의 true distribution에 따른 expected loss를 의미한다.

$$R(h_\theta) = E_{(x,y) \sim D} [l(h_\theta(x)), y]$$

D : true distribution over samples

$$D = \{(x_i, y_i) \sim D\}, i=1, \dots, m$$

finite set of samples to approximate distribution  
of the actual data

### Empirical Risk

$$\hat{R}(h_\theta, D) = \frac{1}{|D|} \sum_{(x,y) \in D} l(h_\theta(x), y)$$

$$\underset{\theta}{\text{minimize}} \hat{R}(h_\theta, D_{\text{train}})$$

Training set ( $D_{\text{train}}$ )의 empirical risk를 minimize 하는 parameters<sup>3</sup>을 찾는 것이 machine learning algorithm training의 전통적인 process이다.

But  $D_{\text{train}}$  을 통해 결정된 후에는 true risk  $R(h_\theta)$ 를 구하기 위해

$\hat{R}(h_\theta, D_{\text{test}})$ 을 대신 사용한다.

## Adversarial risk

앞서 살펴본 traditional risk와의 차이점이라하면은 traditional risk에서는 sample point에 대한 loss를 다뤘다면 adversarial risk에서는 sample point 주변의 몇몇 영역에 대한 worst case loss를 다루게 된다.

$$R_{\text{adv}}(h_\theta) = E_{(x,y) \sim D} \left[ \max_{\delta \in \Delta(x)} \ell(h_\theta(x+\delta)), y \right]$$

$\Delta(x)$ : perturbation region  
depends on sample point itself.

$R_{\text{adv}}(h_\theta)$ 는 각 이미지에 대해 어떤 종류의 perturbation이 허용되는지에 대한 semantic information을 가지게 된다.

Empirical analog of the adversarial risk.

$$\hat{R}_{\text{adv}}(h_\theta, D) = \frac{1}{|D|} \sum_{(x,y) \in D} \max_{f \in \Delta(x)} \ell(h_\theta(x+\delta), y)$$

Allowable set( $\Delta(x)$ ) 안에 있는 모든 input을 adversarially manipulate 할 수 있다면 위 같은 classifier의 worst-case empirical loss  $\hat{R}_{\text{adv}}$ 의 수치를 뜻하게 된다.

## Training adversarially robust classifiers

## Min-max or Robust optimization formulation of adversarial learning

$$\underset{\theta}{\text{minimize}} \hat{R}_{\text{adv}}(h_{\theta}, D_{\text{train}}) \equiv \underset{\theta}{\text{minimize}} \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} \max_{\delta \in \Delta(x)} l(h_{\theta}(x+\delta), y)$$

minimizes the empirical adversarial risk

$$\theta := \theta - \frac{\alpha}{|B|} \sum_{(x,y) \in B} \nabla_{\theta} \max_{\delta \in \Delta(x)} l(h_{\theta}(x+\delta), y)$$

Traditional training; solve optimization problem by SGD over  $\theta$ .

repeatedly choose a minibatch  $B \subseteq D_{\text{train}}$

update  $\theta$  according to its gradient

Inner term의 maximization problem은 고려해보자. Danskin's theorem에 의하면 maximization term을 포함하는 inner function의 gradient는 단순히 함수의 최대값이며 gradient로 구할 수 있다.

$$\delta^* = \operatorname{argmax}_{\delta \in \Delta(x)} l(h_{\theta}(x+\delta), y)$$

$\delta^*$ : optimum of the inner-optimization problem.

$$\nabla_{\theta} \max_{\delta \in \Delta(x)} l(h_{\theta}(x+\delta), y) = \nabla_{\theta} l(h_{\theta}(x+\delta^*), y) \text{ gradient } \delta^*$$

Process of gradient descent on the empirical adversarial risk

1. Solve the inner maximization problem (compute an adversarial example)

$$\delta^*(s) = \operatorname{argmax}_{\delta \in \Delta(x)} l(h_\theta(x + \delta), y)$$

2. Compute the gradient of the empirical adversarial risk, and update  $\theta$

$$\theta := \theta - \frac{\alpha}{|\mathcal{B}|} \sum_{(x,y) \in \mathcal{B}} \nabla_\theta l(h_\theta(x + \delta^*(x)), y)$$

그래서 Adversarial training 이란

1. iteratively compute adversarial examples
2. update the classifier based not upon the original data points, but upon these adversarial examples

One of the most effective empirical methods we have for training adversarially robust models