Deep Learning

Adversarial Machine Learning

U Kang
Seoul National University

In This Lecture

- Adversarial machine learning
- Generating adversarial examples
  - Fast gradient sign method (FGSM)
  - Momentum iterative FGSM (MI-FGSM)
  - One-step target class method (OTCM)
- Implementations
  - Attacking trained models by adv. examples
  - Training models robust to the adv. attacks

Outline

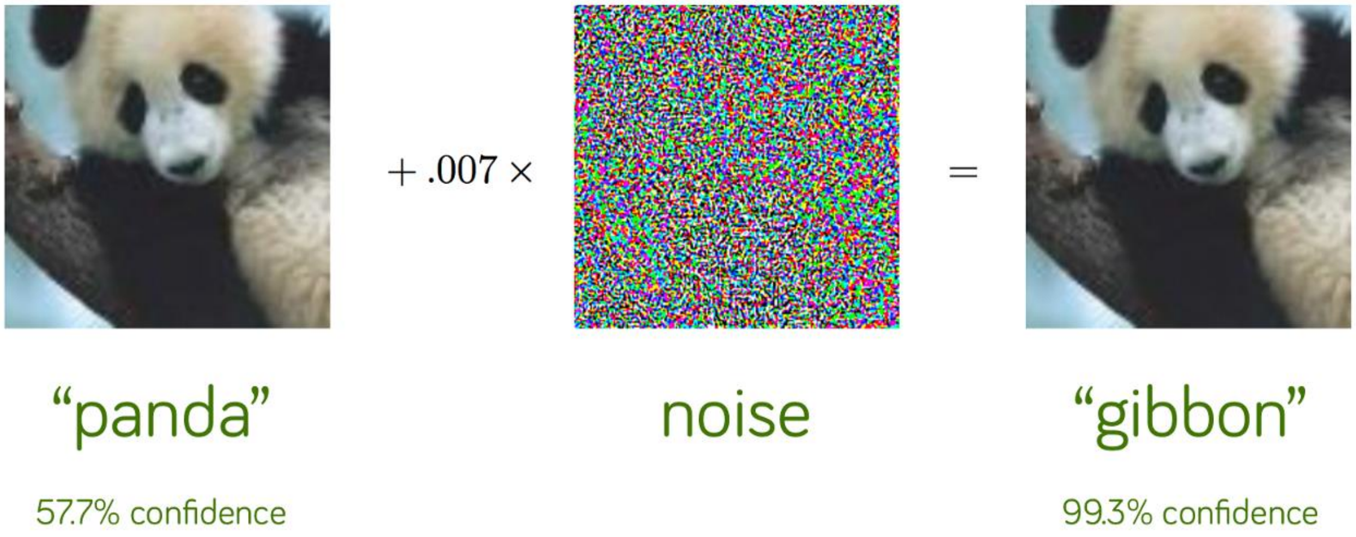
- ☐ **Adversarial Machine Learning**
- ☐ Practice: CNN

Adversarial Machine Learning

- **Adversarial machine learning**
  - A technique that attempts to fool (trained) models through malicious input
- **Why is it important?**
  - Trained ML models are used in various applications
    - Autonomous driving, factory automation, etc.
  - Attacking trained models may ruin a whole system
  - ML models should be safe from malicious attacks

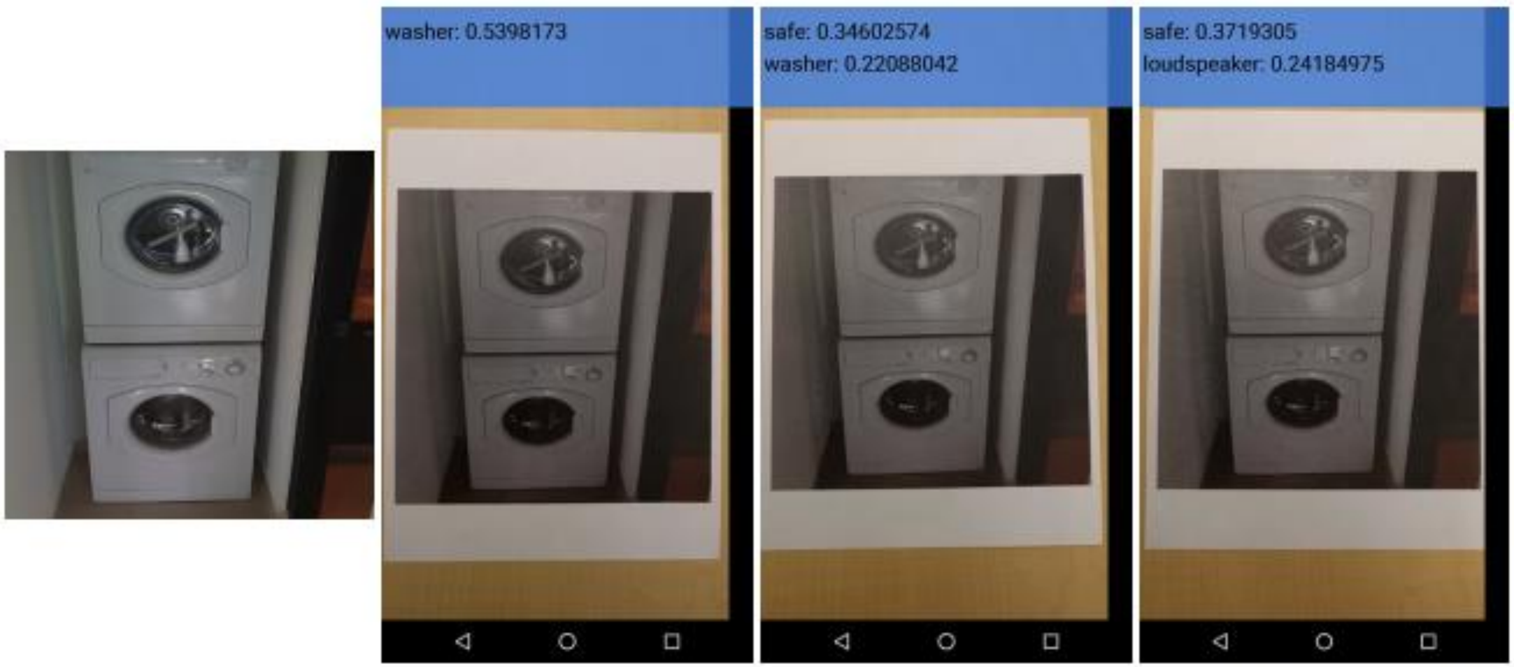
Adversarial Examples (1)

- **Adversarial examples**
  - Inputs to machine learning models that an attacker has intentionally designed to cause a mistake
  - The following is a famous example of classification



Adversarial Examples (2)

- We can fool a classifier by taking a picture of a washer and using the captured screen



(a) Image from dataset (b) Clean image (c) Adv. image, $\epsilon = 4$ (d) Adv. image, $\epsilon = 8$

Adversarial Attacks

- We introduce three simple attack methods
  - Fast gradient sign method (FGSM)
  - Momentum iterative FGSM (MI-FGSM)
  - One-step target class method (OTCM)
- All three approaches are **white-box** methods
  - The structure of a model is fully visible
  - An attacker can compute the gradient of the model

FGSM (1)

- **Fast gradient sign method (FGSM)**
  - A simple but powerful technique for adv. attacks
  - Described in [Goodfellow et al., ICLR 2015]
- FGSM works as follows for an input image x:
  - Computes the gradient of the loss with respect to x
  - Creates a new image that locally maximizes the loss
  - This new image becomes an adversarial image

FGSM (2)

- FGSM is summarized as follows:

$$x' = x + \epsilon \cdot \text{sign}\left(\nabla_x J_\theta(x, y)\right)$$

  - x': Adversarial image
  - x, y: Original image and label
  - ϵ: Multiplier to ensure the perturbations are small
  - θ: Model parameters
  - J: Loss function

MI-FGSM (1)

- **Momentum Iterative FGSM (MI-FGSM)**
  - Proposed in [Dong et al., CVPR 2018]
  - FGSM is easily defended as it is a one-step approach
    - An adversarial example x' is derived easily from x
  - Iterative methods naturally make x' far from x
    - Not far by the Euclidean distance
  - MI-FGSM adds a *momentum* to iterative methods

MI-FGSM (2)

- MI-FGSM keeps track of two kinds of variables:

$$g_{t+1} = \mu g_t + \frac{\nabla_x J_\theta(x'_t, y)}{\|\nabla_x J_\theta(x'_t, y)\|}$$

$$x'_{t+1} = x'_t + \alpha \cdot \mathrm{sign}(g_{t+1})$$

  - μ is a decay factor and normally set to 1
  - $\alpha = \epsilon / T$ with T being the number of iterations

MI-FGSM (3)

- The algorithm in the original paper:

Algorithm 1 MI-FGSM

Input: A classifier f with loss function J; a real example $\boldsymbol{x}$ and ground-truth label y;

Input: The size of perturbation ϵ; iterations T and decay factor μ.

Output: An adversarial example $\boldsymbol{x}^*$ with $\|\boldsymbol{x}^* - \boldsymbol{x}\|_\infty \le \epsilon$.

1: $\alpha = \epsilon / T$;

2: $\boldsymbol{g}_0 = 0$; $\boldsymbol{x}_0^* = \boldsymbol{x}$;

3: **for** $t = 0$ to $T - 1$ **do**

4: Input $\boldsymbol{x}_t^*$ to f and obtain the gradient $\nabla_{\boldsymbol{x}} J(\boldsymbol{x}_t^*, y)$;

5: Update $\boldsymbol{g}_{t+1}$ by accumulating the velocity vector in the gradient direction as

$$\boldsymbol{g}_{t+1} = \mu \cdot \boldsymbol{g}_t + \frac{\nabla_{\boldsymbol{x}} J(\boldsymbol{x}_t^*, y)}{\|\nabla_{\boldsymbol{x}} J(\boldsymbol{x}_t^*, y)\|_1}; \tag{6}$$

6: Update $\boldsymbol{x}_{t+1}^*$ by applying the sign gradient as

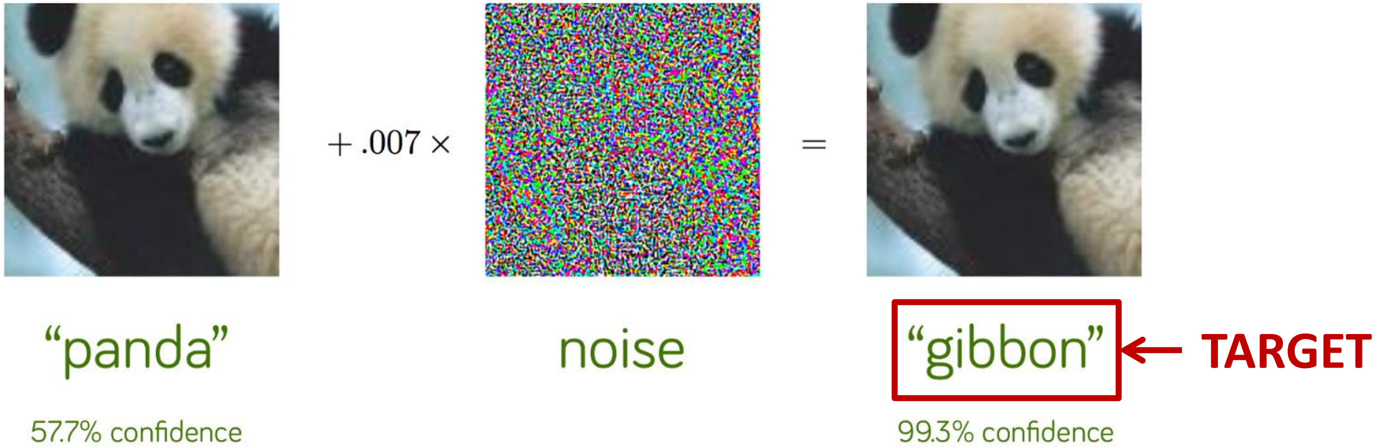
$$\boldsymbol{x}_{t+1}^* = \boldsymbol{x}_t^* + \alpha \cdot \mathrm{sign}(\boldsymbol{g}_{t+1}); \tag{7}$$

7: **end for**

8: **return** $\boldsymbol{x}^* = \boldsymbol{x}_T^*$.

OTCM (1)

- **One-step target class method (OTCM)**
  - FGSM and MI-FGSM are non-targeted methods
    - They fool a classifier but not for a specific label
  - OTCM, however, takes a specific label as input
    - It produces different results for different input labels



OTCM (2)

- OTCM is summarized as follows:

$$x' = x - \epsilon \cdot \mathrm{sign}\big(\nabla_x J_\theta(x, y')\big)$$

  - y' is a target class that given as input
- Note that the gradient is *subtracted* from x
  - FGSM adds the gradient to increase the loss
  - OTCM tries to minimize the loss for the target y'

Outline

- ☑ Adversarial Machine Learning
- ☐ **Practice: CNN**

Implementing a CNN

- We quickly implement a CNN for a practice
- Import essential libraries and download MNIST

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
import tensorflow.keras.layers as ly
from tensorflow.keras.utils import to_categorical

import numpy as np
import matplotlib.pyplot as plt
```

```
# Load dataset
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Preprocess dataset
train_images = train_images.reshape((-1, 28, 28, 1))
test_images = test_images.reshape((-1, 28, 28, 1))
train_images, test_images = train_images / 255.0, test_images / 255.0
train_labels = to_categorical(train_labels, 10)
test_labels = to_categorical(test_labels, 10)
```

Convolutional Layers

- Define convolutional layers of 3 × 3 kernels
  - ReLU and max-pooling of size 2 × 2 are used
  - The number of channels increases by layers

```
model = Sequential(
    [
        ly.Conv2D(filters=32, kernel_size=3, strides=1, padding='same',
                  activation='relu', use_bias=False, input_shape=(28,28,1)),
        ly.MaxPool2D(pool_size=2, strides=2, padding='same'),

        ly.Conv2D(filters=64, kernel_size=3, strides=1,
                  activation='relu', use_bias=False, padding='same'),
        ly.MaxPool2D(pool_size=2, strides=2, padding='same'),

        ly.Flatten(),
        ly.Dense(units=256, activation='relu'),
        ly.Dense(units=10, activation='softmax'),
    ]
)
```

Dense Layers

- Add two dense layers to the end of the network

```
model = Sequential(
    [
        ly.Conv2D(filters=32, kernel_size=3, strides=1, padding='same',
                  activation='relu', use_bias=False, input_shape=(28,28,1)),
        ly.MaxPool2D(pool_size=2, strides=2, padding='same'),

        ly.Conv2D(filters=64, kernel_size=3, strides=1,
                  activation='relu', use_bias=False, padding='same'),
        ly.MaxPool2D(pool_size=2, strides=2, padding='same'),

        ly.Flatten(),
        ly.Dense(units=256, activation='relu'),
        ly.Dense(units=10, activation='softmax'),
    ]
)
```

Model Structure

- Check model structure

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        288
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        18432
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 256)               803072
_________________________________________________________________
dense_1 (Dense)              (None, 10)                2570
=================================================================
Total params: 824,362
Trainable params: 824,362
Non-trainable params: 0
_________________________________________________________________
```

Compiling and Training

- Define cost and optimizer tensors for training
  - We use the typical cross-entropy loss function
  - We use the SGD optimizer
  - We train the network for $N = 3$ epochs

```
model.compile(optimizer='sgd', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=3)
```

```
Epoch 1/3
1875/1875 [==============================] - 7s 4ms/step - loss: 0.4881 - accuracy: 0.8621
Epoch 2/3
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1373 - accuracy: 0.9588
Epoch 3/3
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0913 - accuracy: 0.9726
```

Evaluation

- Evaluate the performance of trained model
  - We use test dataset for evaluation

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f"Test loss: {test_loss:.4f}, accuracy: {test_acc*100:.2f}")
```

```
313/313 - 1s - loss: 0.0684 - accuracy: 0.9781
Test loss: 0.0684, accuracy: 97.81
```

Questions?