

# Adversarial Robustness

Shusen Wang

# An Adversarial Example



+ .007 ×



=



**“panda”**  
57.7% confidence

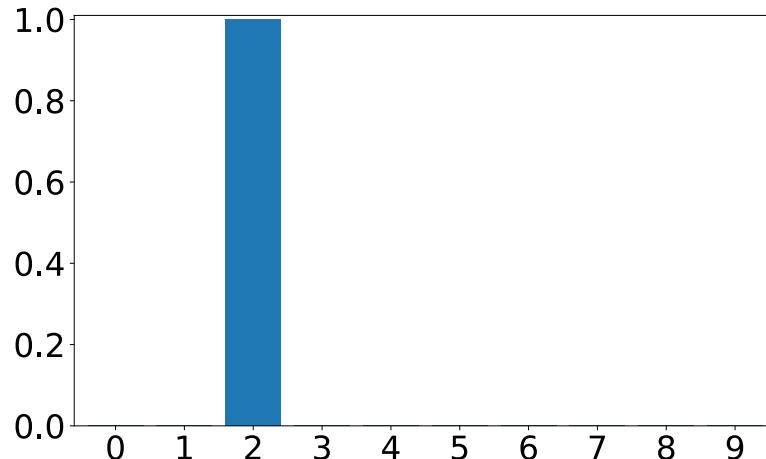
**“gibbon”**  
99.3% confidence



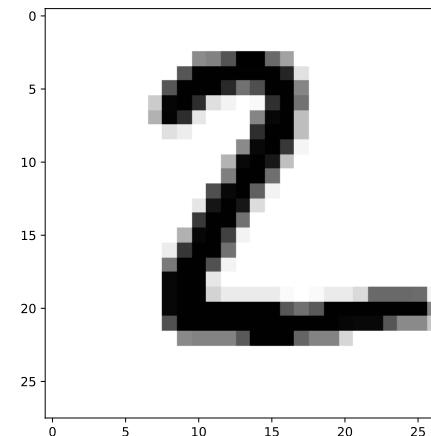
# **Revisit CNN for MNIST Classification**

# Neural Network for MNIST

- Neural network:  $\mathbf{p} = \mathbf{f}(\mathbf{x}; \mathbf{W})$   
Trainable parameters of the neural network.
- The prediction --- a 10-dim vector.
- $p_j$  indicates how likely  $\mathbf{x}$  is the digit  $j$ .



28×28 input image.



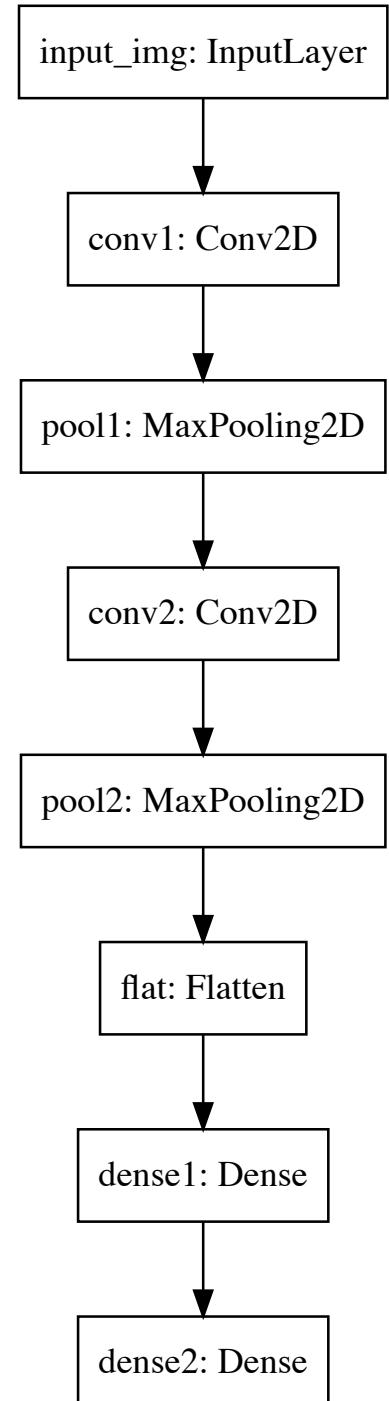
# CNN for MNIST

```
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense
from keras import models

# input
input_img = Input(shape=(28, 28, 1), name='input_img')

# layers
conv1 = Conv2D(10, (5, 5), activation='relu', name='conv1')(input_img)
pool1 = MaxPooling2D((2, 2), name='pool1')(conv1)
conv2 = Conv2D(20, (5, 5), activation='relu', name='conv2')(pool1)
pool2 = MaxPooling2D((2, 2), name='pool2')(conv2)
flat = Flatten(name='flat')(pool2)
dense1 = Dense(100, activation='relu', name='dense1')(flat)
pred = Dense(10, activation='softmax', name='dense2')(dense1)

# model
model = models.Model(inputs=input_img, outputs=pred)
```

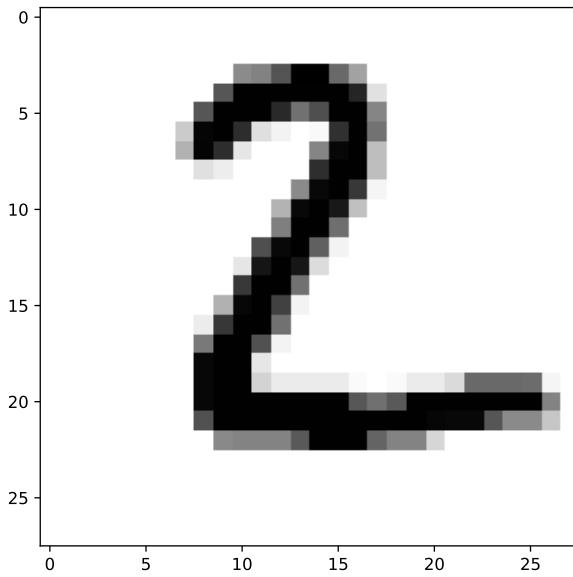


# CNN for MNIST

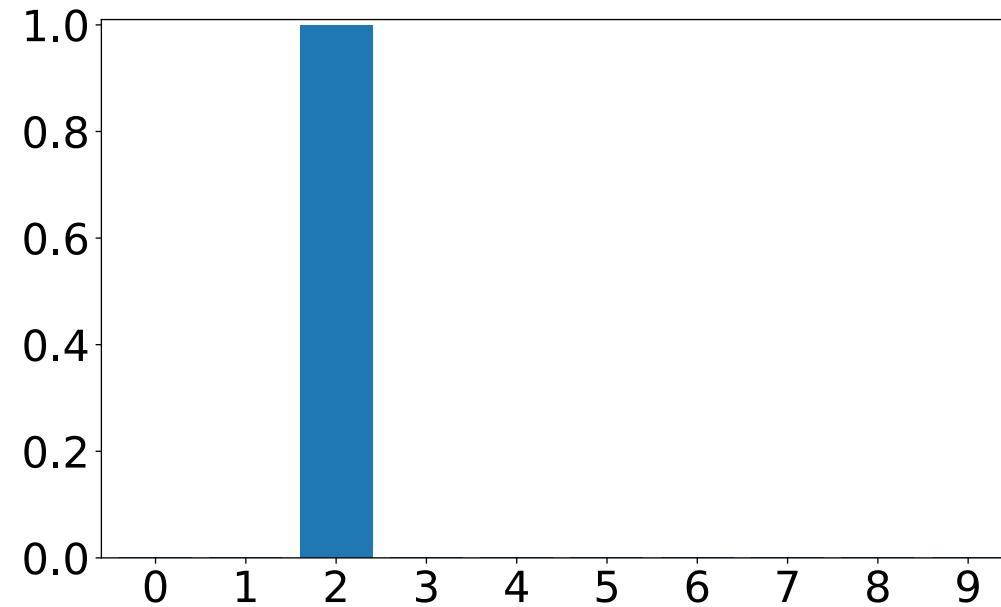
```
model.compile(optimizer='RMSprop',  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])  
  
history = model.fit(  
    x_train,  
    y_train_vec,  
    shuffle=True,  
    epochs=10,  
    batch_size=128,  
    validation_data=(x_test, y_test_vec)  
)
```

# CNN for MNIST

- Training accuracy = 99.5%
- Validation accuracy = 99.1%



28×28 input image

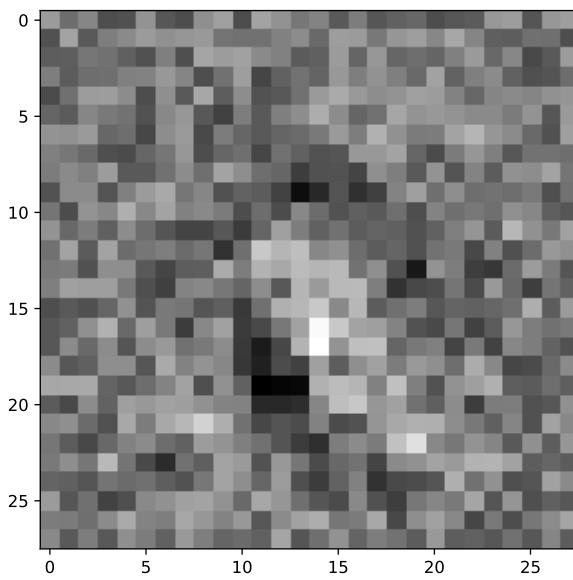


10-dim prediction vector

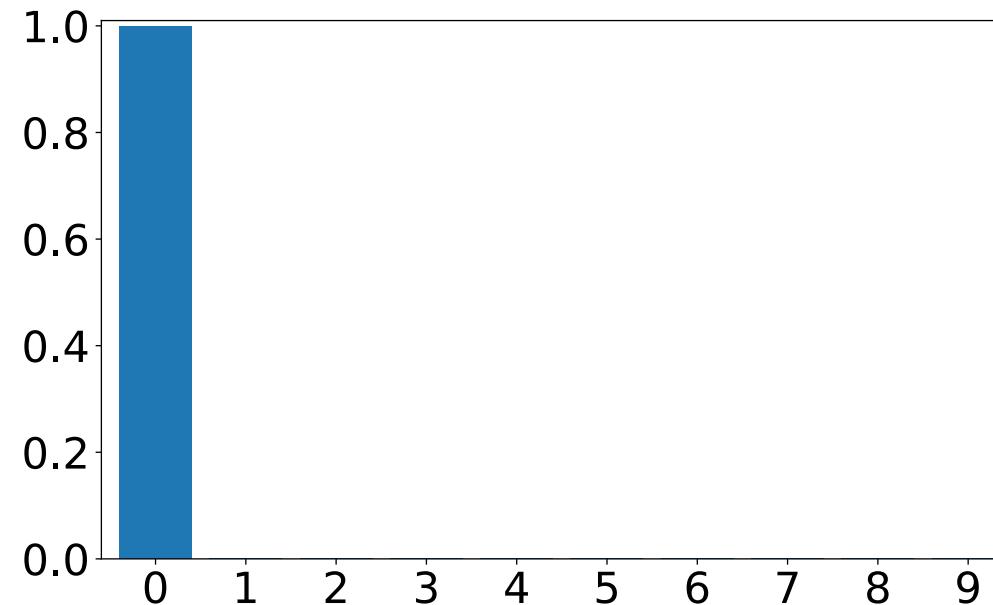
# **Optimization w.r.t. X**

# Fake Image

- Our trained CNN thinks the  $28 \times 28$  input image is digit “0”.



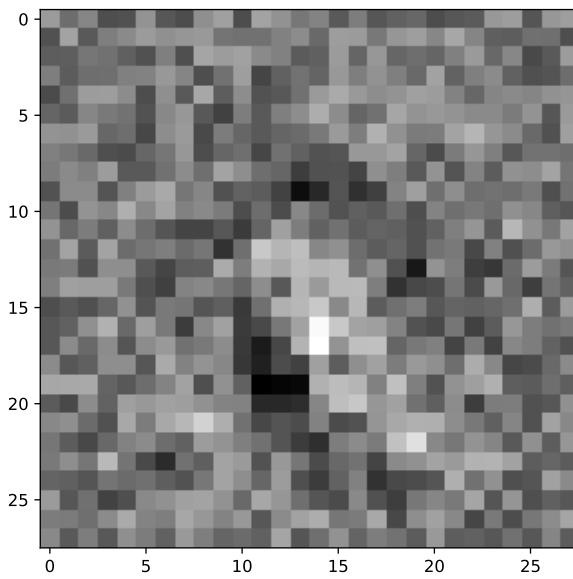
**28×28 input image**



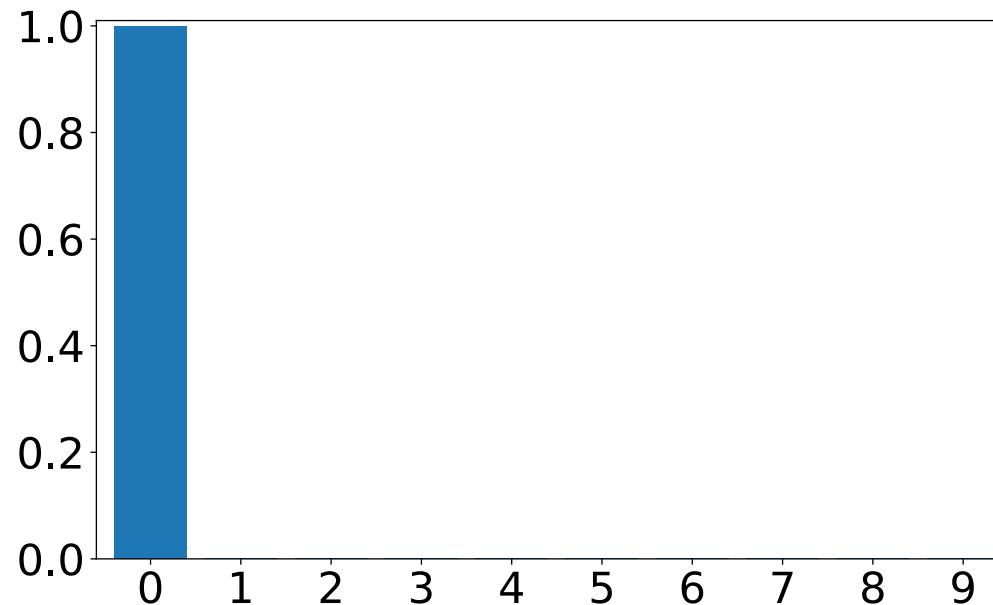
**10-dim prediction vector**

# Fake Image

**Question:** How is the fake image generated?



**28×28 input image**



**10-dim prediction vector**

# Generate Fake Image

**Question:** How is the fake image generated?

- Neural network:  $\mathbf{p} = \mathbf{f}(\mathbf{x}; \mathbf{W})$ .
- Previously, we fix  $\mathbf{x}_1, \dots, \mathbf{x}_n$  and solve

$$\mathbf{W}^* = \operatorname*{argmin}_{\mathbf{W}} \sum_{j=1}^n \text{Dist}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j; \mathbf{W})).$$

# Generate Fake Image

**Question:** How is the fake image generated?

- Neural network:  $\mathbf{p} = \mathbf{f}(\mathbf{x}; \mathbf{W})$ .
- Previously, we fix  $\mathbf{x}_1, \dots, \mathbf{x}_n$  and solve

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \sum_{j=1}^n \operatorname{Dist}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j; \mathbf{W})).$$

- To generate a fake image, we do the following:
  1. Set a fake target, e.g.,  $\tilde{\mathbf{y}} = [1, 0, 0, \dots, 0]$ .
  2. Fix the network parameters to  $\mathbf{W}^*$ .
  3. Generate a fake image  $\tilde{\mathbf{x}}$  by

$$\tilde{\mathbf{x}} = \operatorname{argmin}_{\mathbf{x}} \operatorname{Dist}(\tilde{\mathbf{y}}, \mathbf{f}(\mathbf{x}; \mathbf{W}^*)).$$

# Generate Fake Image

**Step 1:** Set a fake target.

```
import numpy as np
from keras import backend as K
from keras.layers import Input

j = 0 # the fake label
y_tilde = np.zeros((1, 10))
y_tilde[0, j] = 1
print(y_tilde)

fake_target = Input(tensor=K.constant(y_tilde))

[[1. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

# Generate Fake Image

**Step 2:** Fix the trained network's parameters,  $\mathbf{W}$ .

```
from keras.layers import Input  
  
model.trainable = False  
input_img = Input(shape=(28, 28, 1))  
pred = model(input_img)
```

# Generate Fake Image

## Step 3: Generate a fake image.

- Define the loss function and evaluate the gradient.
  - `pred`: the output of the pre-trained model.
  - `fake_target`:  $[1, 0, 0, \dots, 0]$ .
  - Loss function: `CrossEntropy(pred, fake_target)`.

```
import keras
from keras import backend as K

loss = keras.metrics.binary_crossentropy(pred, fake_target)
grads = K.gradients(loss, [input_img])[0]
fetch_loss_and_grads = K.function([input_img], [loss, grads])
```

# Generate Fake Image

## Step 3: Generate a fake image.

initialization

```
import numpy as np

fake_img = np.random.rand(1, 28, 28, 1)
learn_rate = 0.5

for i in range(10):
    l, g = fetch_loss_and_grads([fake_img])
    print('iter ' + str(i) + ': loss = ' + str(l))
    fake_img -= learn_rate * g
```

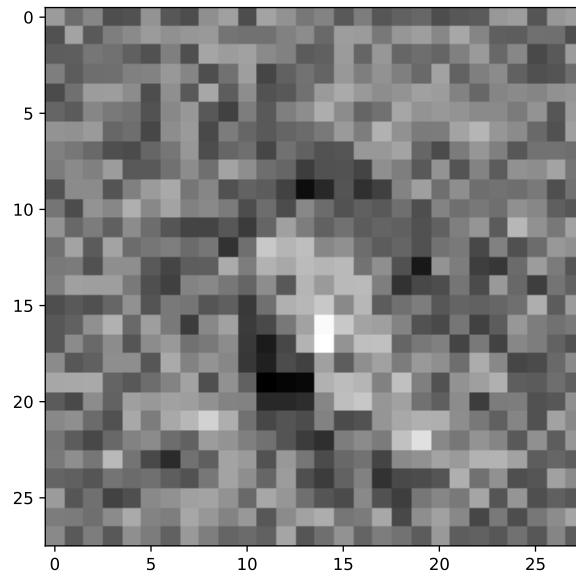
gradient  
descent

```
iter 0: loss = [3.039003]
iter 1: loss = [2.6458066]
iter 2: loss = [1.0068675]
iter 3: loss = [0.22192626]
iter 4: loss = [0.01752027]
iter 5: loss = [0.01353873]
iter 6: loss = [0.01130872]
iter 7: loss = [0.00976203]
iter 8: loss = [0.00869376]
iter 9: loss = [0.00787604]
```

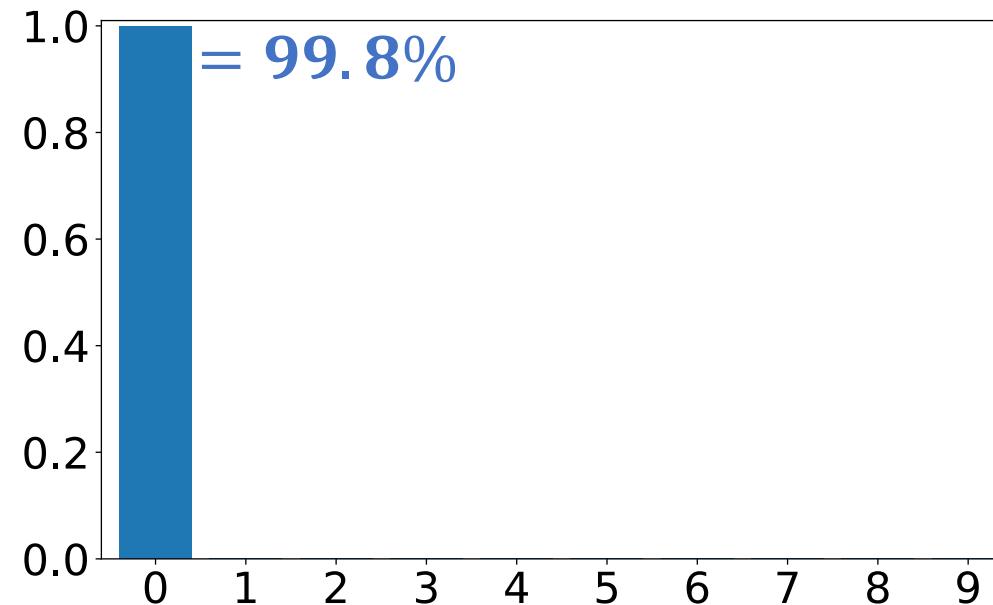
# Generate Fake Image

```
print(model.predict(fake_img)[0])
```

```
[9.9776781e-01 4.1152799e-04 4.0594186e-04 8.5301363e-05 6.2220497e-05  
 7.1304827e-04 2.1397672e-04 1.0672671e-04 1.9847257e-04 3.4956032e-05]
```



28×28 fake image



10-dim prediction vector

# Untargeted Attack

**Read the blog for details:**

<https://medium.com/onfido-tech/adversarial-attacks-and-defences-for-convolutional-neural-networks-66915ece52e7>

# Untargeted Attack

- $\mathbf{x}$ : a real image.
- $\mathbf{y}$ : true label.
- Maximize  $\text{Dist}(\mathbf{y}, \mathbf{f}(\mathbf{x}; \mathbf{W}^*))$  w.r.t.  $\mathbf{x}$ .
  - Make the prediction,  $\mathbf{f}(\mathbf{x}; \mathbf{W}^*)$ , far from  $\mathbf{y}$ .
  - → Wrong prediction.

# Untargeted Attack

- $\mathbf{x}$ : a real image.
- $\mathbf{y}$ : true label.
- Maximize  $\text{Dist}(\mathbf{y}, \mathbf{f}(\mathbf{x}; \mathbf{W}^*))$  w.r.t.  $\mathbf{x}$ .
  - Make the prediction,  $\mathbf{f}(\mathbf{x}; \mathbf{W}^*)$ , far from  $\mathbf{y}$ .
  - → Wrong prediction.
- Gradient ascent:
  - Repeat:  $\mathbf{x} \leftarrow \mathbf{x} + \alpha \frac{\partial \text{dist}(\mathbf{y}, \mathbf{f}(\mathbf{x}; \mathbf{W}^*))}{\partial \mathbf{x}}$ .
  - Stop if the prediction,  $\mathbf{f}(\mathbf{x}; \mathbf{W}^*)$ , is incorrect.

# Untargeted Attack

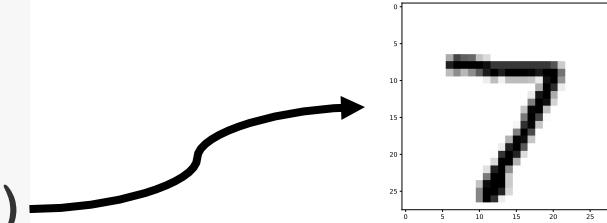
**Step 1:** Get an image and its true target.

```
# get the i-th test sample
i = 0
digit = x_test[i].reshape((1, 28, 28, 1))
label = y_test[i]
print('The true label is ' + str(label))
```

```
# one-hot encode the label
y_true = np.zeros((1, 10))
y_true[0, label] = 1
print('One-hot encode: ' + str(y_true))
```

The true label is 7

One-hot encode: [[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]



# Untargeted Attack

**Step 2:** Fix the trained network's parameters.

```
from keras.layers import Input  
  
input_img = Input(shape=(28, 28, 1))  
true_target = Input(tensor=K.constant(y_true))  
pred = model(input_img)
```

# Untargeted Attack

**Step 3:** Generate a fake image using gradient ascent.

```
import keras
from keras import backend as K

loss = keras.metrics.binary_crossentropy(pred, true_target)
grads = K.gradients(loss, [input_img])[0]
fetch_grads = K.function([input_img], [grads])
```

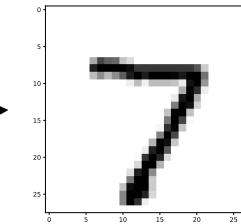
# Untargeted Attack

**Step 3:** Generate a fake image.

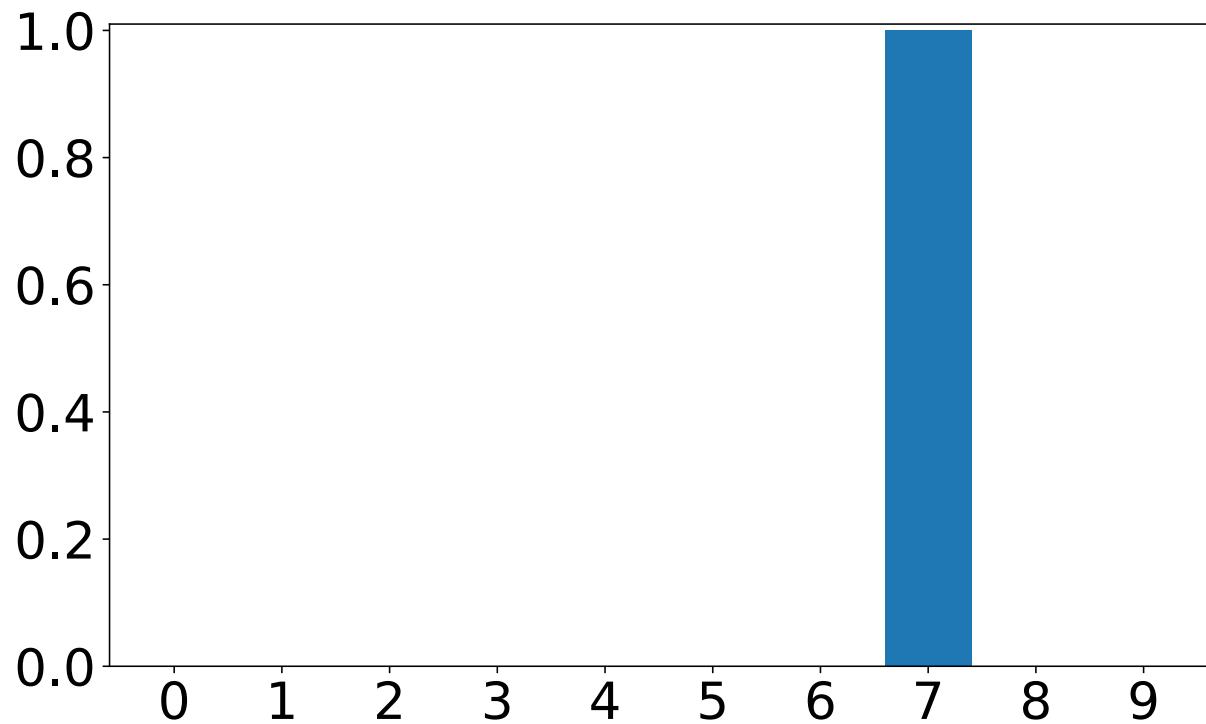
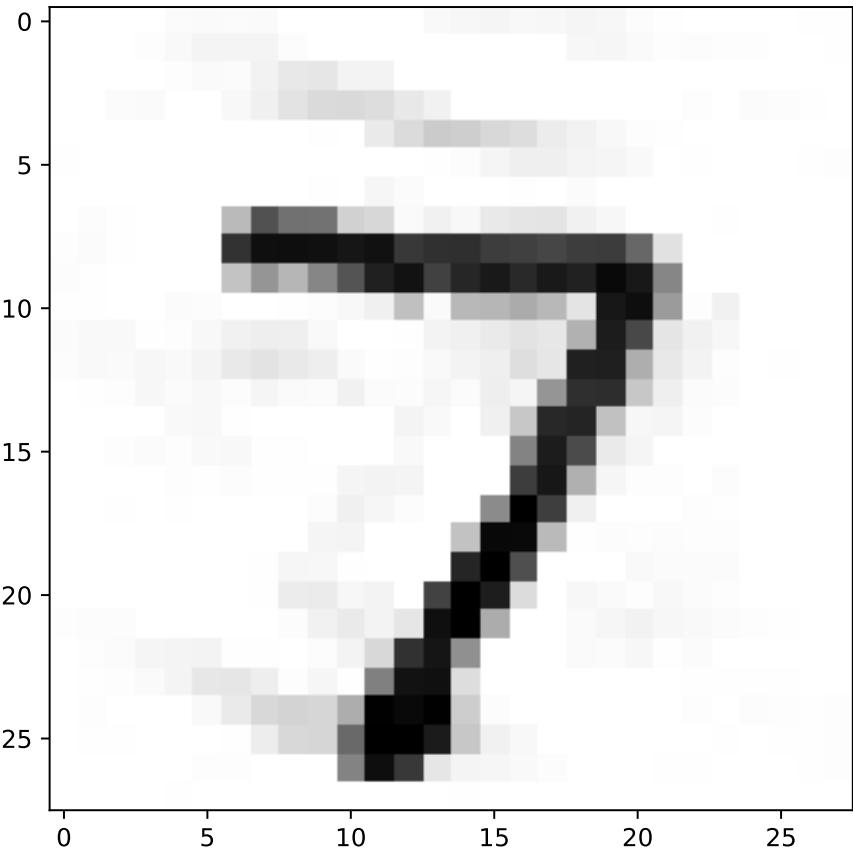
```
import numpy

fake_img = digit.copy() # initialize

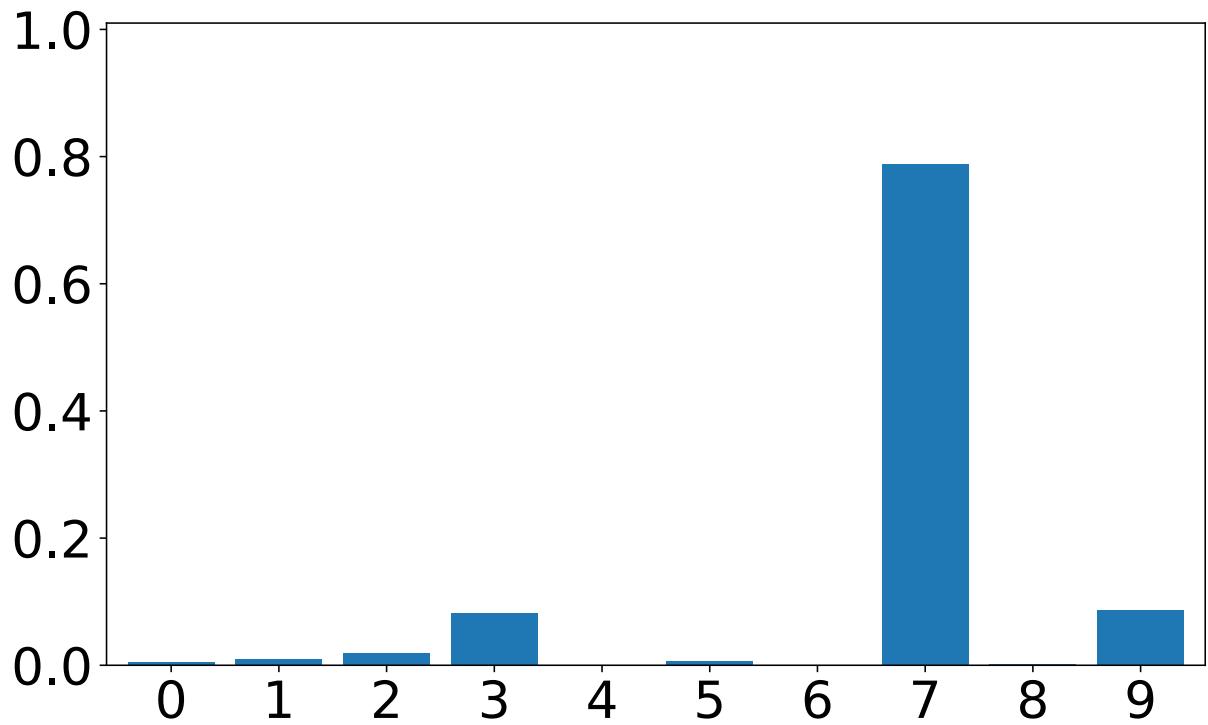
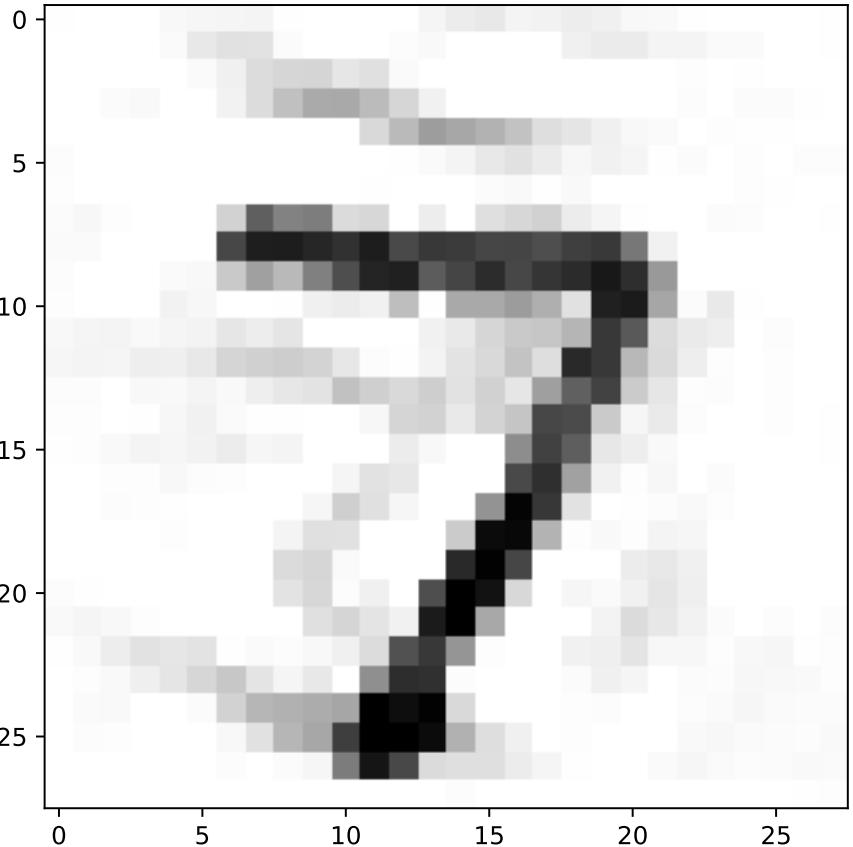
for t in range(20):
    # gradient descent
    grad = fetch_grads([fake_img])[0]
    g = grad / numpy.max(numpy.abs(grad))
    fake_img += 0.05 * g
    # project to [0, 1]
    fake_img[fake_img < 0] = 0
    fake_img[fake_img > 1] = 1
```



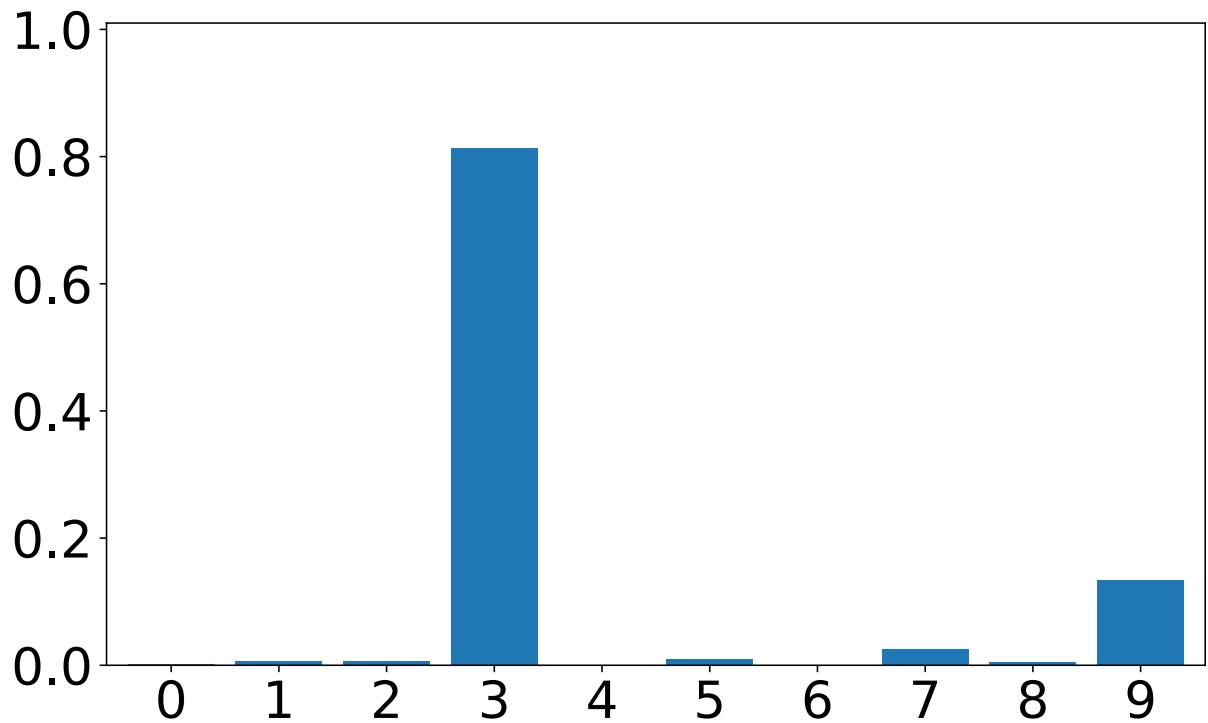
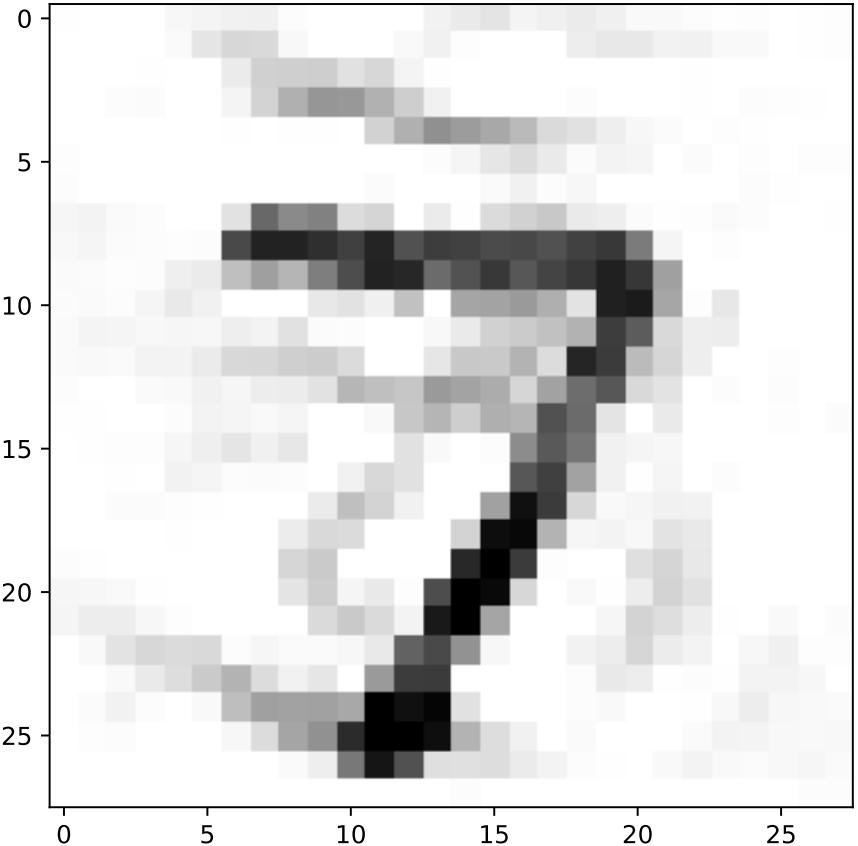
# Results after 5 Steps



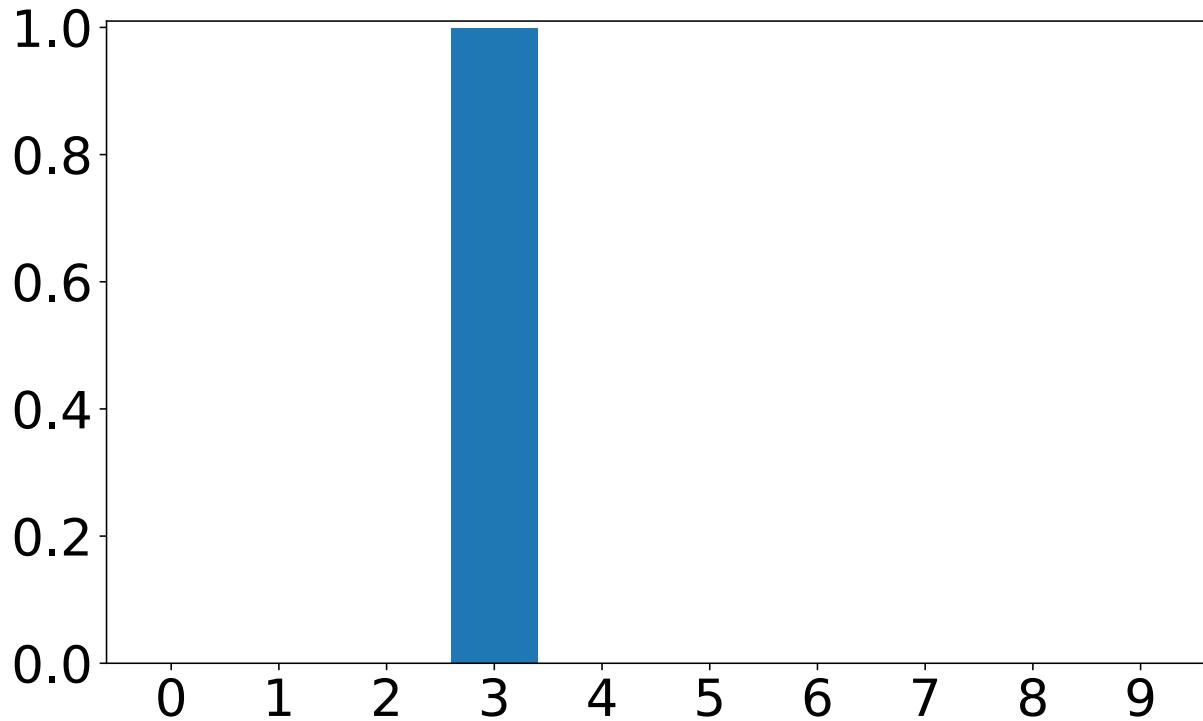
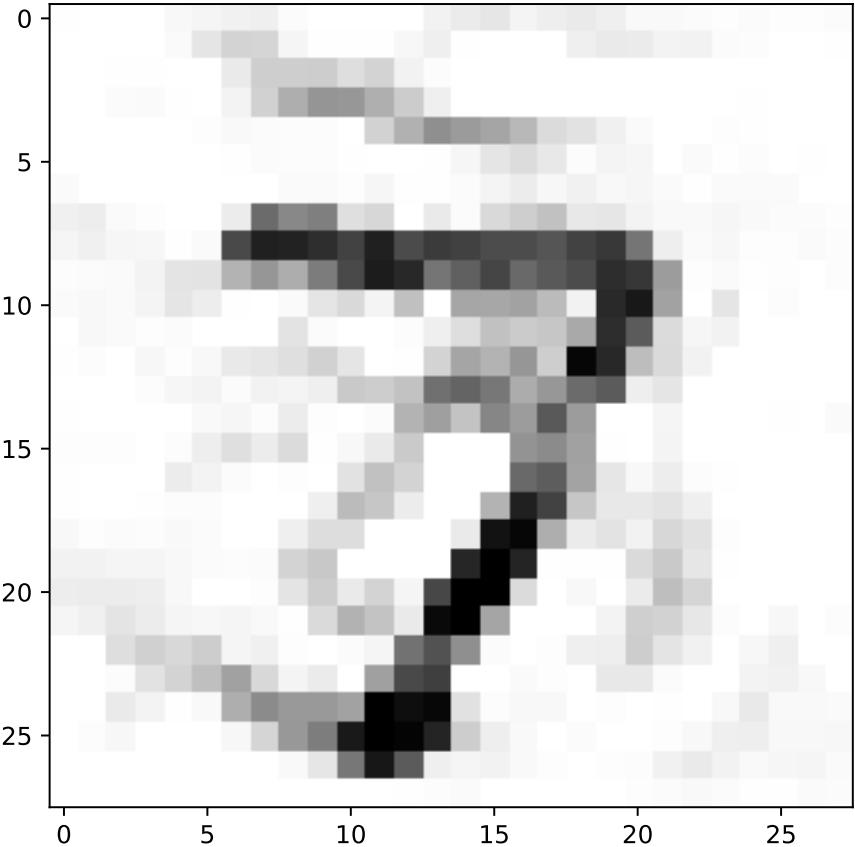
# Results after 10 Steps



# Results after 15 Steps



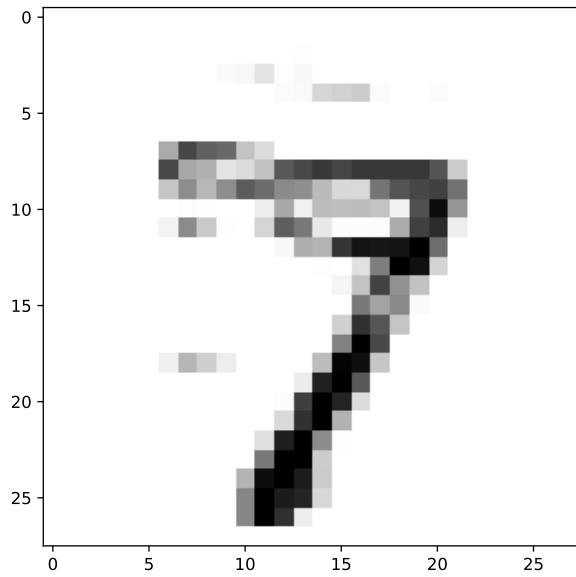
# Results after 20 Steps



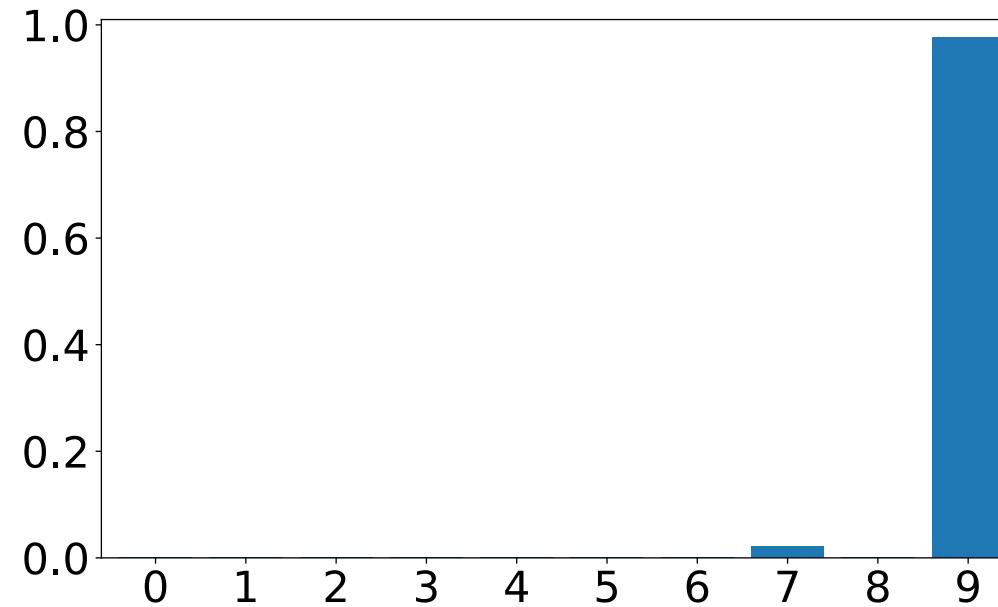
# **Targeted Attack**

# Targeted Adversarial Example

- Our trained CNN thinks the 28×28 input image is digit “9”.



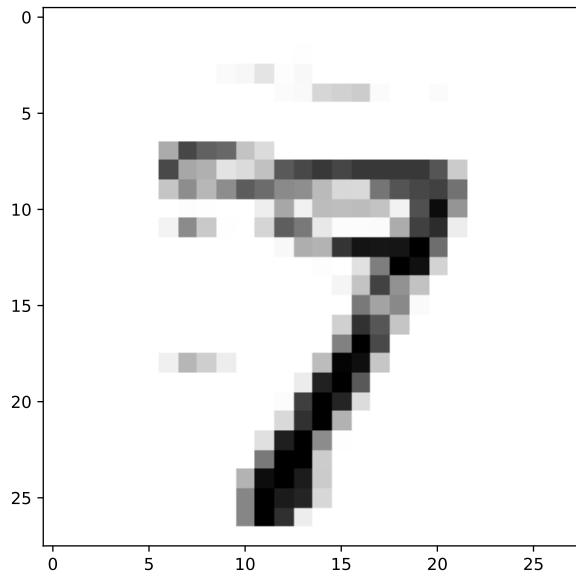
28×28 input image



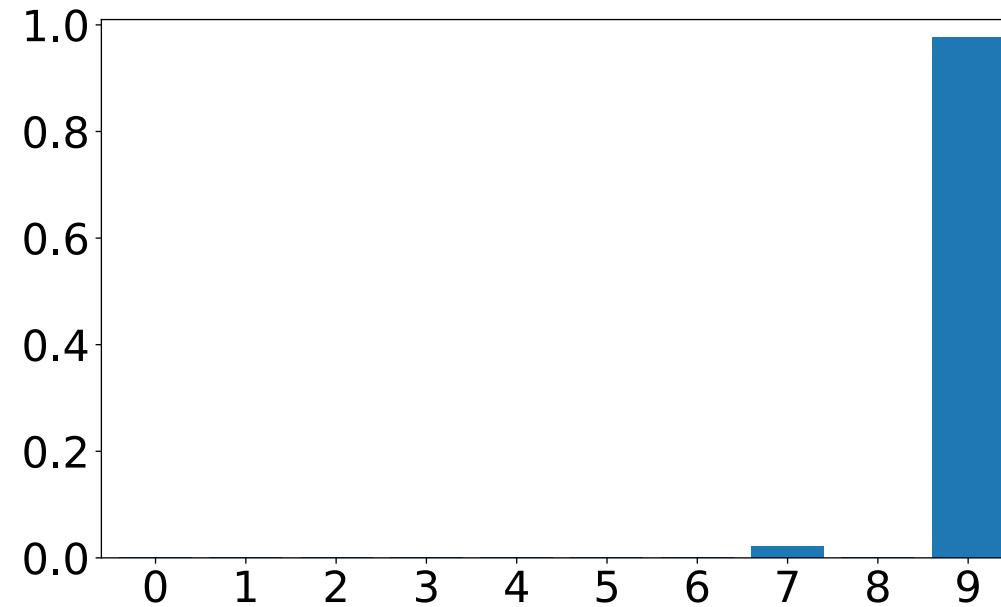
10-dim prediction vector

# Targeted Adversarial Example

**Question:** How is the fake image generated?



28×28 input image



10-dim prediction vector

# Targeted Attack

**Question:** How is the fake image generated?

1. Given a real image  $\mathbf{x}^*$  which we want to attack.
2. Set a fake target, e.g.,  $\tilde{\mathbf{y}} = [1, 0, 0, \dots, 0]$ .
3. Fix the network parameters to  $\mathbf{W}^*$ .
4. Set up the loss function

$$\text{Loss}(\tilde{\mathbf{x}}, \mathbf{x}^*, \tilde{\mathbf{y}}) = \text{Dist}(\tilde{\mathbf{y}}, \mathbf{f}(\tilde{\mathbf{x}}; \mathbf{W}^*)) + \lambda \|\tilde{\mathbf{x}} - \mathbf{x}^*\|_2^2$$

make the neural net think the generated image  $\tilde{\mathbf{x}}$  is digit “0”.

the generated image  $\tilde{\mathbf{x}}$  should look similar to the real one  $\mathbf{x}^*$ .

# Targeted Attack

**Question:** How is the fake image generated?

1. Given a real image  $\mathbf{x}^*$  which we want to attack.
2. Set a fake target, e.g.,  $\tilde{\mathbf{y}} = [1, 0, 0, \dots, 0]$ .
3. Fix the network parameters to  $\mathbf{W}^*$ .
4. Set up the loss function

$$\text{Loss}(\tilde{\mathbf{x}}, \mathbf{x}^*, \tilde{\mathbf{y}}) = \text{Dist}(\tilde{\mathbf{y}}, \mathbf{f}(\tilde{\mathbf{x}}; \mathbf{W}^*)) + \lambda \|\tilde{\mathbf{x}} - \mathbf{x}^*\|_2^2$$

make the neural net think the generated image  $\tilde{\mathbf{x}}$  is digit “0”.

the generated image  $\tilde{\mathbf{x}}$  should look similar to the real one  $\mathbf{x}^*$ .

5. Generate a fake image  $\tilde{\mathbf{x}}$  by  $\tilde{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} \text{Loss}(\mathbf{x}, \mathbf{x}^*, \tilde{\mathbf{y}})$ .

# Targeted Attack

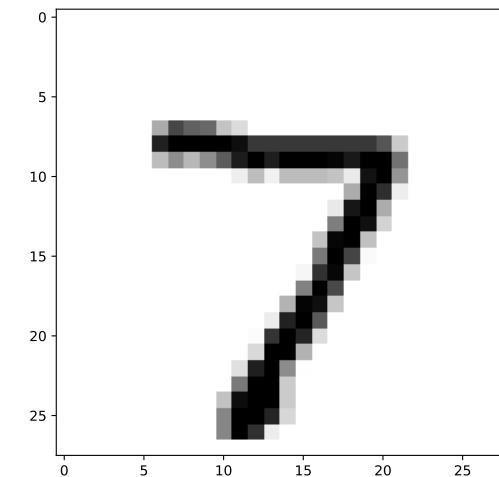
**Step 1:** Given a real image which we want to attack.

```
from keras.layers import Input
from keras import backend as K

i = 0
digit = x_test[i].reshape((1, 28, 28, 1))
print('The true label is ' + str(y_test[i]))

target_img = Input(tensor=K.constant(digit))
```

The true label is 7



# Targeted Attack

**Step 2:** Set a fake target to be digit “0”.

```
import numpy as np
from keras import backend as K
from keras.layers import Input

j = 0 # the fake label
y_tilde = np.zeros((1, 10))
y_tilde[0, j] = 1
print(y_tilde)

fake_target = Input(tensor=K.constant(y_tilde))

[[1. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

# Targeted Attack

**Step 3:** Freeze the parameters. Define input image and prediction.

```
from keras.layers import Input  
  
model.trainable = False  
input_img = Input(shape=(28, 28, 1))  
pred = model(input_img)
```

# Targeted Attack

## Step 4: Set the loss function.

- $\text{Loss}(\tilde{\mathbf{x}}, \mathbf{x}^*, \tilde{\mathbf{y}}) = \text{CrossEntropy}(\tilde{\mathbf{y}}, \mathbf{f}(\tilde{\mathbf{x}}; \mathbf{W}^*)) + \lambda \|\tilde{\mathbf{x}} - \mathbf{x}^*\|_2^2$ 



make the neural net think the generated image  $\tilde{\mathbf{x}}$  is digit “ $\tilde{\mathbf{y}}$ ”.

the generated image  $\tilde{\mathbf{x}}$  should look similar to the real one  $\mathbf{x}^*$ .

# Targeted Attack

## Step 4: Set the loss function.

- $\text{Loss}(\tilde{\mathbf{x}}, \mathbf{x}^*, \tilde{\mathbf{y}}) = \text{CrossEntropy}(\tilde{\mathbf{y}}, \mathbf{f}(\tilde{\mathbf{x}}; \mathbf{W}^*)) + \lambda \|\tilde{\mathbf{x}} - \mathbf{x}^*\|_2^2$ 

make the neural net think the generated image  $\tilde{\mathbf{x}}$  is digit “0”.

the generated image  $\tilde{\mathbf{x}}$  should look similar to the real one  $\mathbf{x}^*$ .

```
import keras
from keras import backend as K

param = 60

loss1 = keras.metrics.binary_crossentropy(pred, fake_target)
loss2 = K.mean(K.square(target_img - input_img))

loss = loss1 + param * loss2

grads = K.gradients(loss, [input_img])[0]
fetch_loss_and_grads = K.function([input_img], [loss, grads])
```

# Targeted Attack

## Step 4: Set the loss function.

$$\bullet \text{Loss}(\tilde{\mathbf{x}}, \mathbf{x}^*, \tilde{\mathbf{y}}) = \text{CrossEntropy}(\tilde{\mathbf{y}}, \mathbf{f}(\tilde{\mathbf{x}}; \mathbf{W}^*)) + \lambda \|\tilde{\mathbf{x}} - \mathbf{x}^*\|_1$$

make the neural net think the generated image  $\tilde{\mathbf{x}}$  is digit “0”.

the generated image  $\tilde{\mathbf{x}}$  should look similar to the real one  $\mathbf{x}^*$ .

```
import keras
from keras import backend as K

param = 60

loss1 = keras.metrics.binary_crossentropy(pred, fake_target)
loss2 = K.mean K.square target_img - input_img))

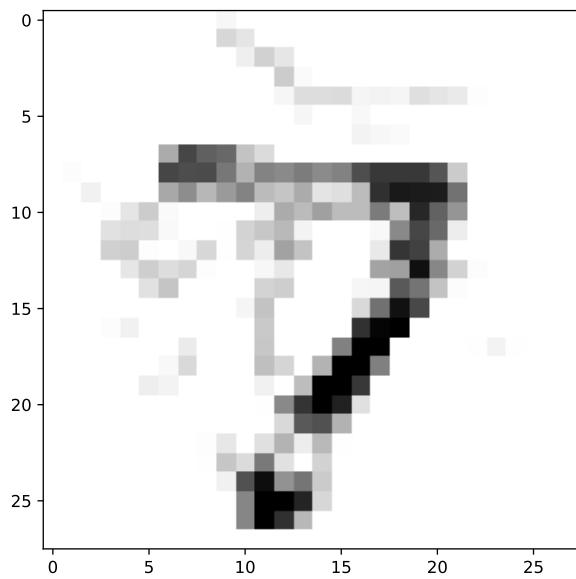
loss = loss1 + param * loss2
```

Replace `square()` by `abs()`

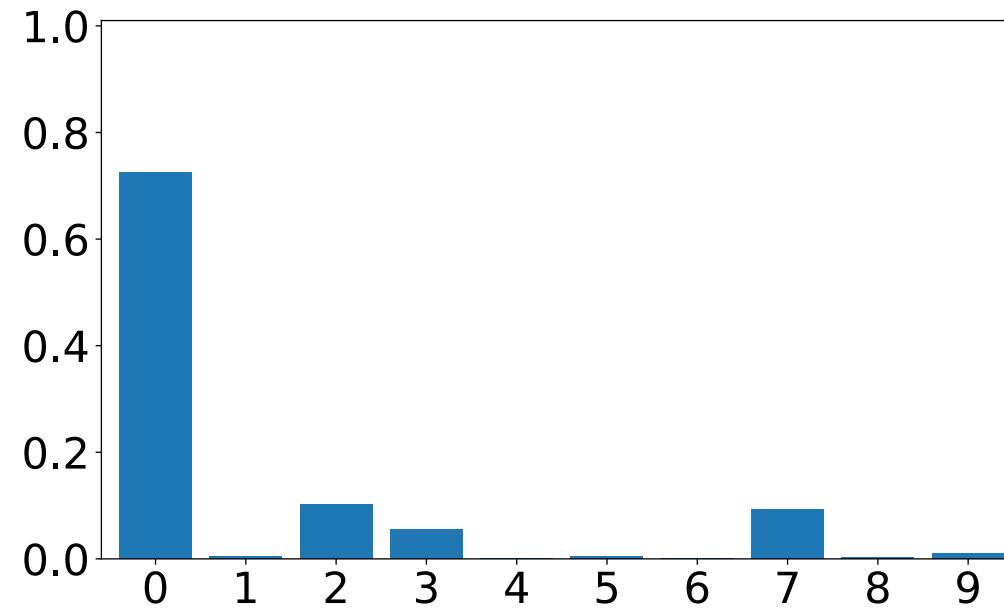
```
grads = K.gradients(loss, [input_img])[0]
fetch_loss_and_grads = K.function([input_img], [loss, grads])
```

# Targeted Attack

- Our trained CNN thinks the  $28 \times 28$  input image is digit “0”.



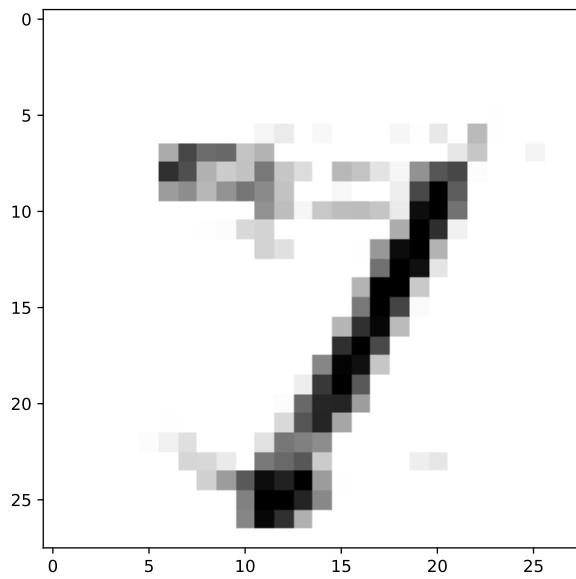
28×28 generated image



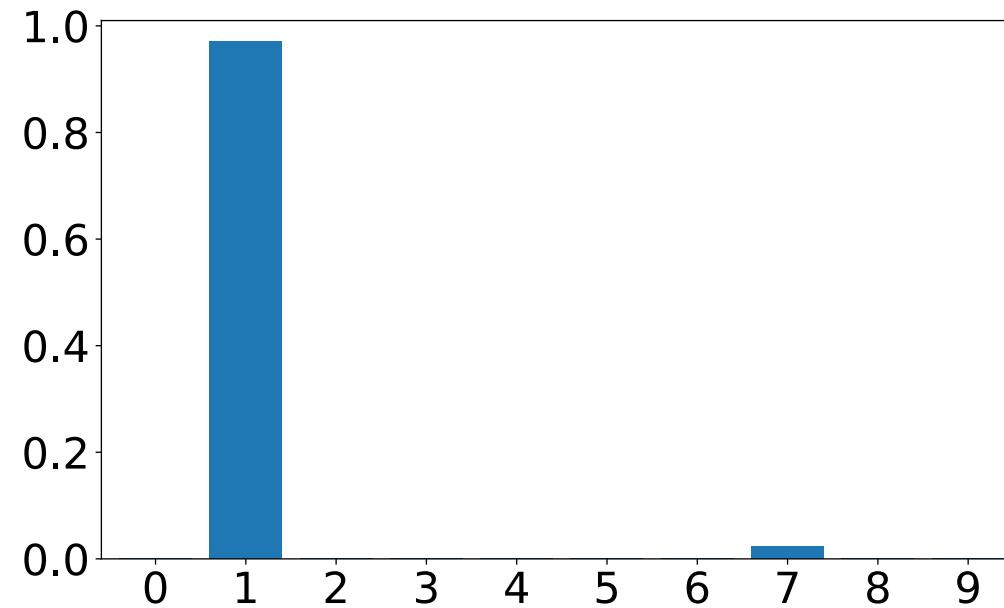
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the 28×28 input image is digit “1”.



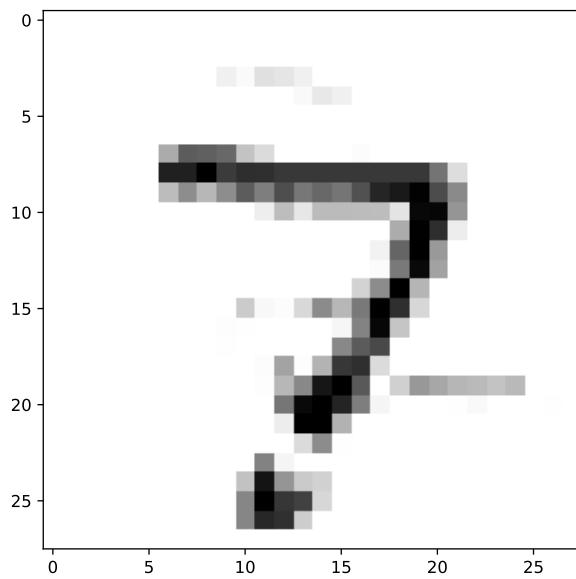
28×28 generated image



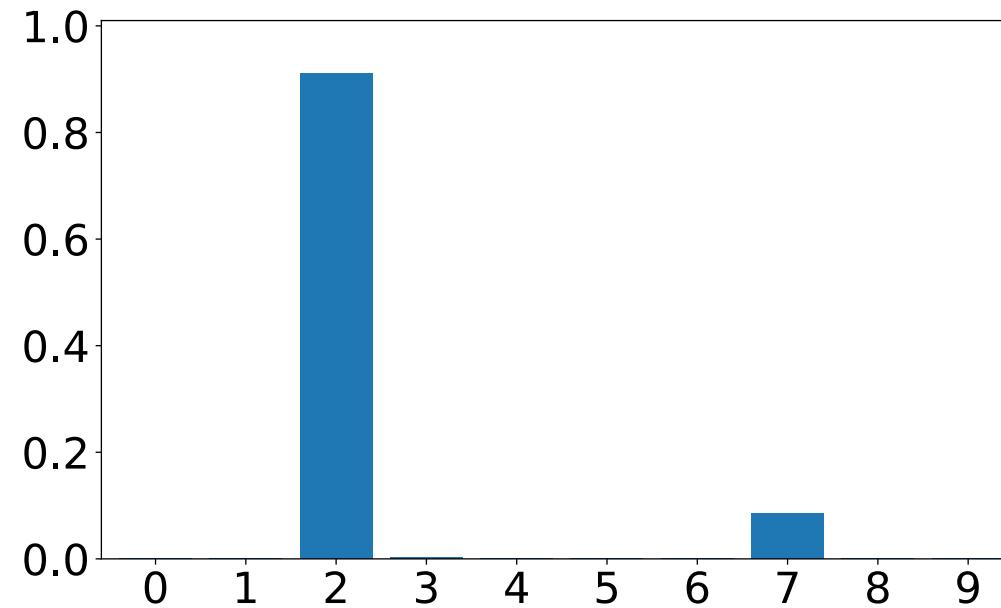
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the 28×28 input image is digit “**2**”.



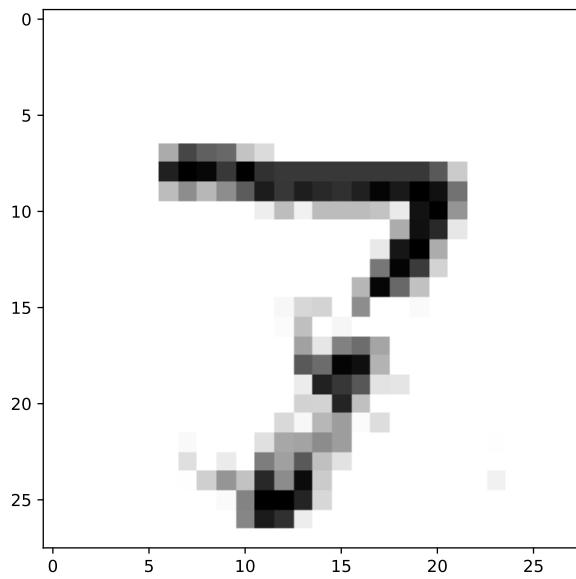
28×28 generated image



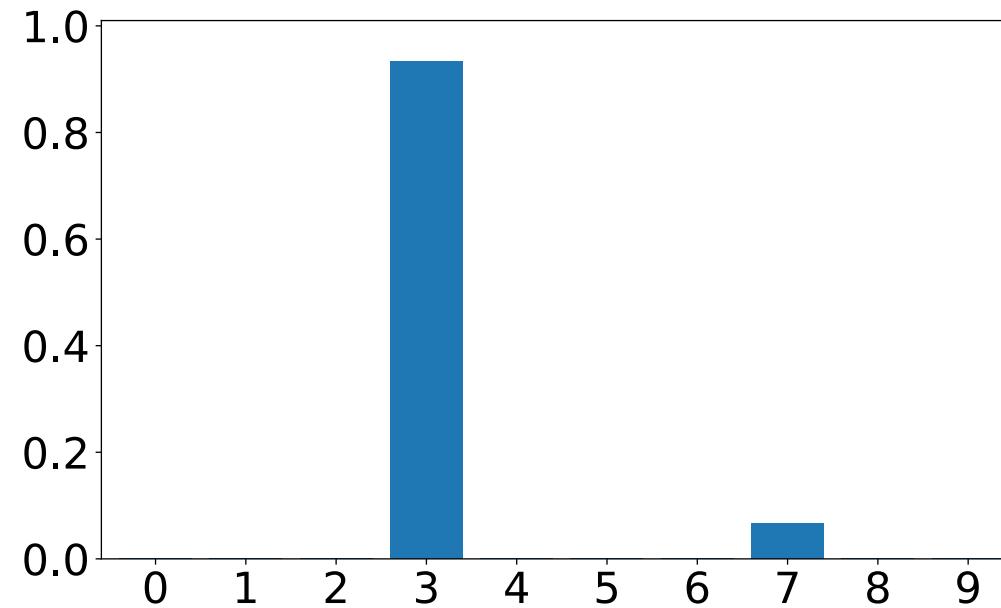
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the 28×28 input image is digit “3”.



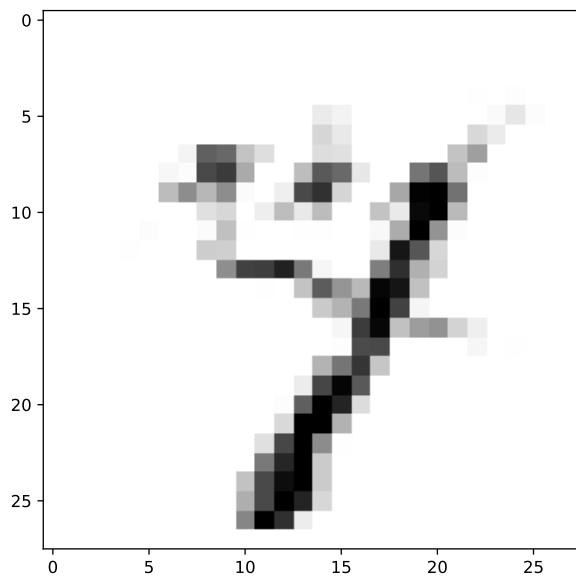
28×28 generated image



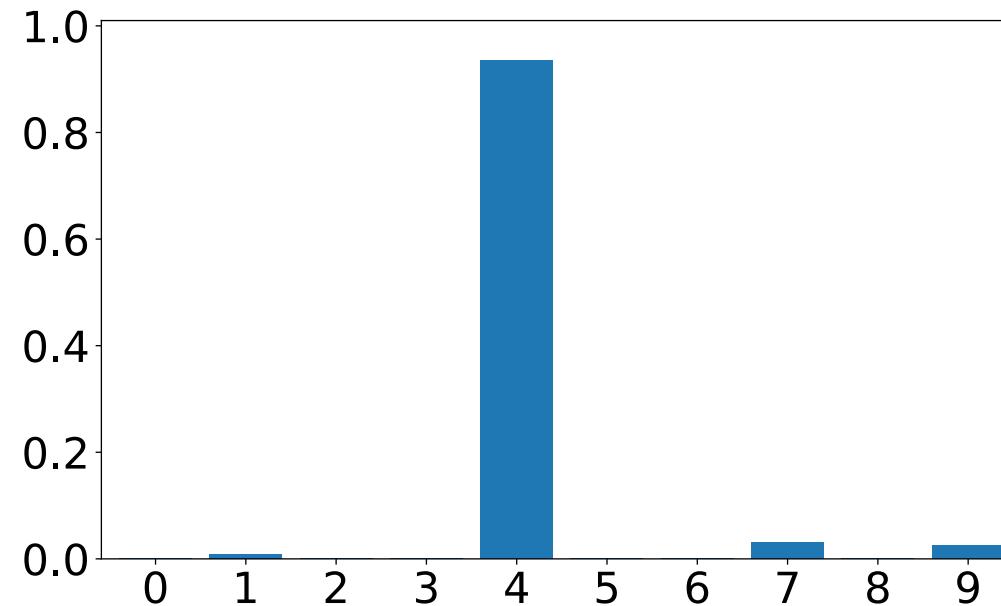
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the 28×28 input image is digit “4”.



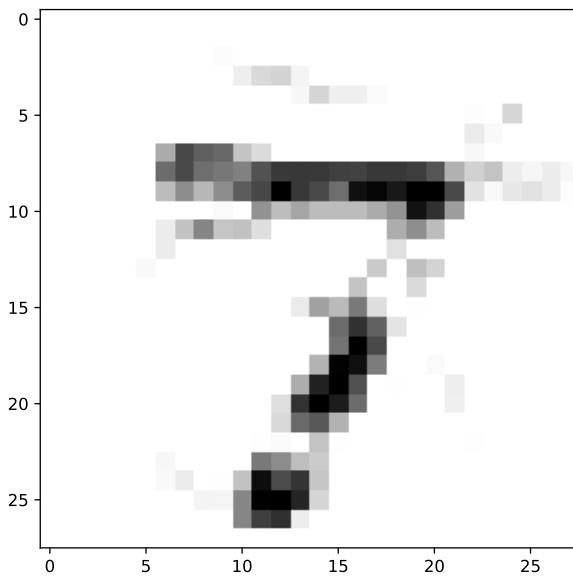
28×28 generated image



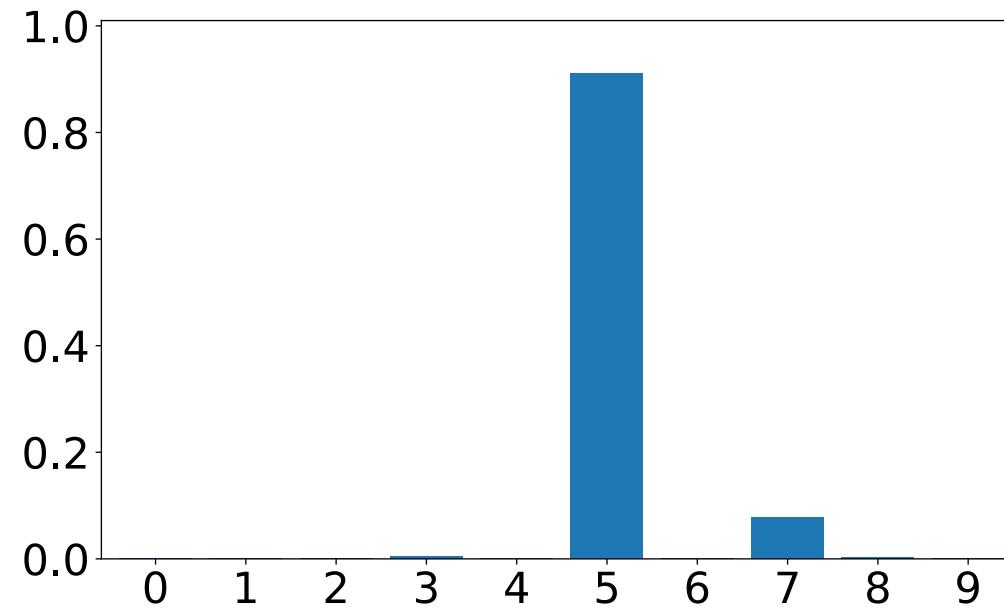
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the  $28 \times 28$  input image is digit “5”.



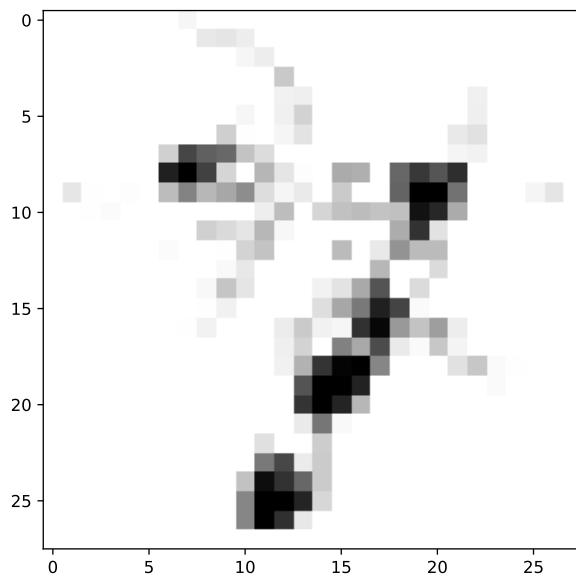
28×28 generated image



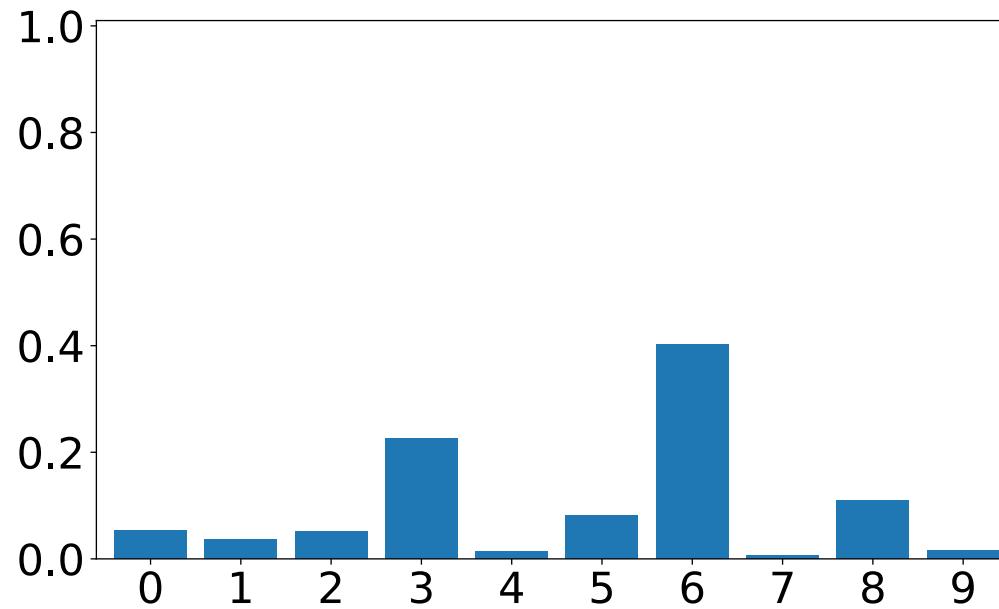
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the  $28 \times 28$  input image is digit “6”.



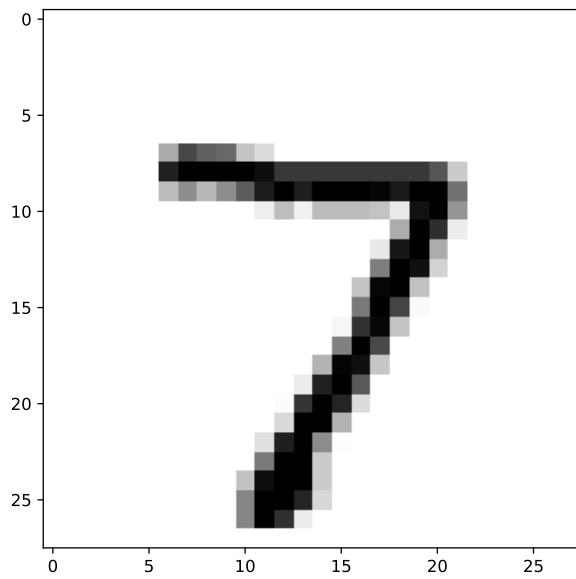
28×28 generated image



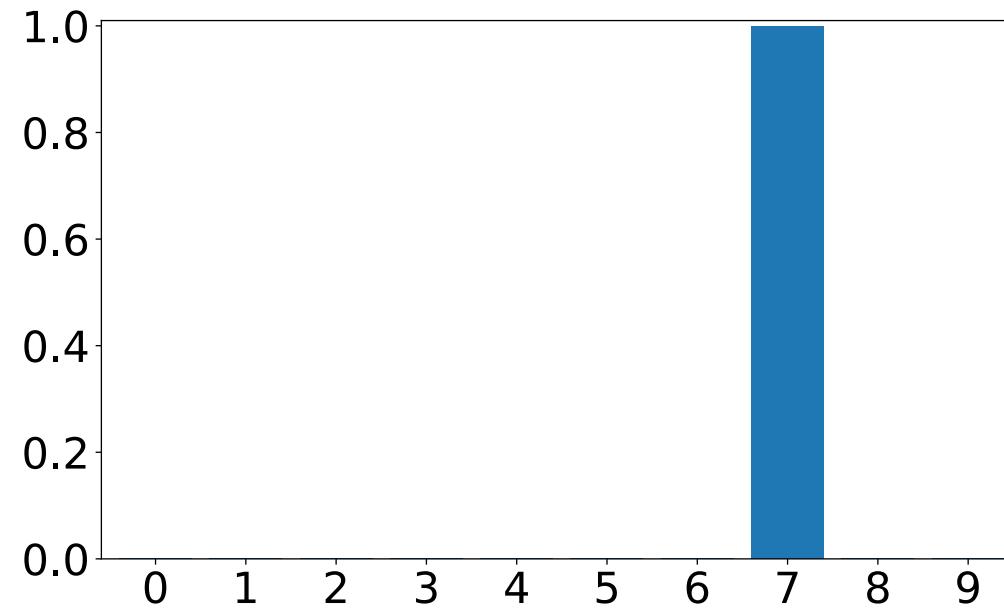
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the 28×28 input image is digit “7”.



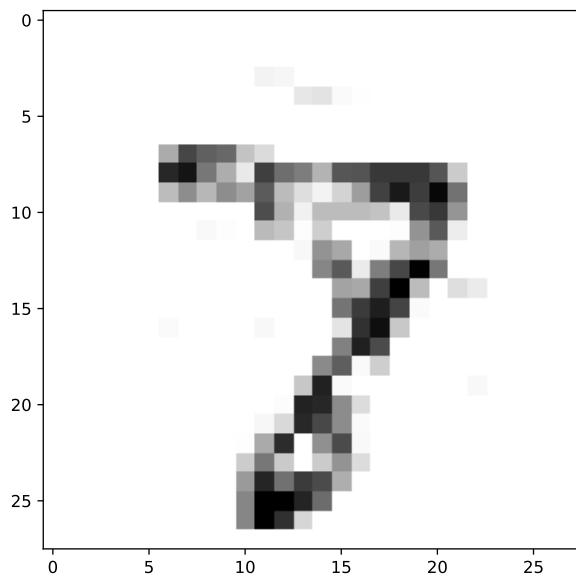
28×28 generated image



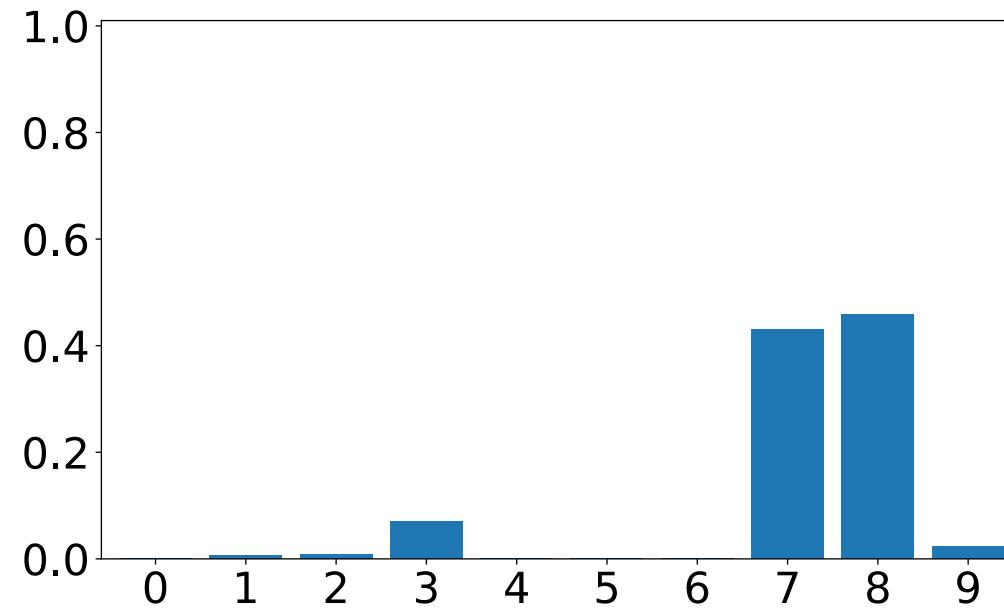
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the 28×28 input image is digit “8”.



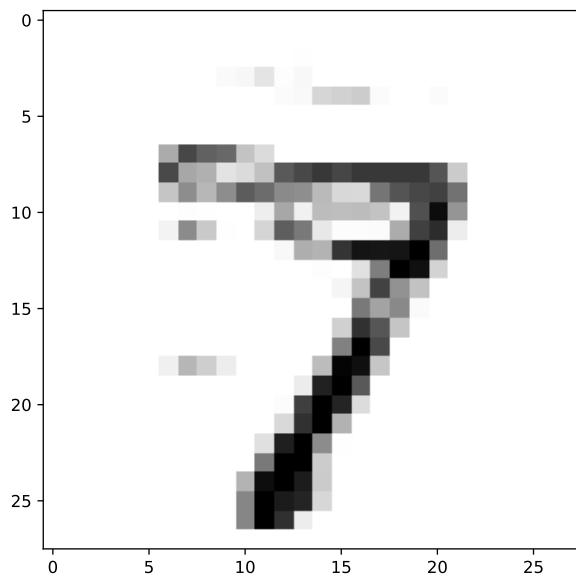
28×28 generated image



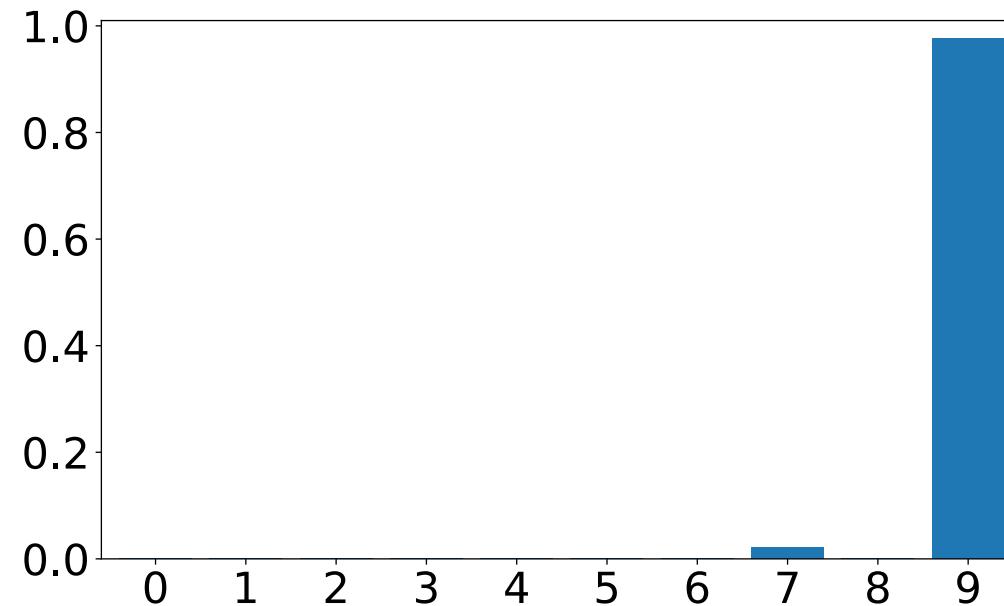
10-dim prediction vector

# Targeted Attack

- Our trained CNN thinks the 28×28 input image is digit “9”.



28×28 generated image



10-dim prediction vector

# Adversarial Training

Read the articles and slides for details: <https://adversarial-ml-tutorial.org/>

# Adversarial Robust Model

- Standard model:

$$\min_{\mathbf{W}} \sum_{j=1}^n \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j ; \mathbf{W})).$$

# Adversarial Robust Model

- Standard model:

$$\min_{\mathbf{W}} \sum_{j=1}^n \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j ; \mathbf{W})).$$

- Min-max model (robust to adversary):

$$\min_{\mathbf{W}} \sum_{j=1}^n \left\{ \max_{\|\boldsymbol{\delta}\| < \sigma} \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j + \boldsymbol{\delta} ; \mathbf{W})) \right\}.$$

# Min-Max Model: Optimization

Min-max model:

$$\min_{\mathbf{W}} \sum_{j=1}^n \left\{ \max_{\|\boldsymbol{\delta}\| < \sigma} \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j + \boldsymbol{\delta}; \mathbf{W})) \right\}.$$

Equivalent to:

$$\min_{\mathbf{W}} \sum_{j=1}^n \text{Loss}(\mathbf{y}_j, \mathbf{f}(\tilde{\mathbf{x}}_j; \mathbf{W})).$$

- Here,  $\tilde{\mathbf{x}}_j = \mathbf{x}_j + \boldsymbol{\delta}^*$  is an adversarial example, where

$$\boldsymbol{\delta}^* = \operatorname*{argmax}_{\|\boldsymbol{\delta}\| < \sigma} \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j + \boldsymbol{\delta}; \mathbf{W})).$$

# Min-Max Model: Optimization

**Min-max model:**

$$\min_{\mathbf{W}} \sum_{j=1}^n \left\{ \max_{\|\boldsymbol{\delta}\| < \sigma} \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j + \boldsymbol{\delta}; \mathbf{W})) \right\}.$$

**Repeat:**

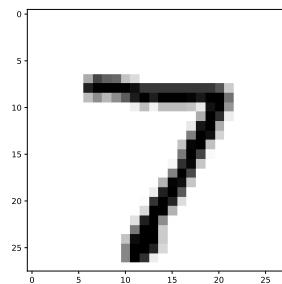
1. Randomly sample an index  $j$  from  $\{1, \dots, n\}$ .
2. Adversary:  $\boldsymbol{\delta}^* = \underset{\|\boldsymbol{\delta}\| < \sigma}{\operatorname{argmax}} \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j + \boldsymbol{\delta}; \mathbf{W}))$ .
3. Update  $\mathbf{W}$  using the derivative of  $\text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j + \boldsymbol{\delta}^*; \mathbf{W}))$  w.r.t.  $\mathbf{W}$ .

# Min-Max Model: Optimization

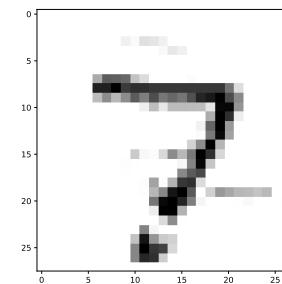
Another way to understand adversarial training.

**Repeat:**

1. Randomly select a real sample; generate an **adversarial sample**.
2. Replace real samples by **adversarial sample**. (Use the **true label**, e.g., “7”.)
3. Update the model parameters using the **adversarial sample** and **true label**.



**Real sample**



**Adversarial sample**

# Another Approach: Gradient Regularization

- Standard model:

$$\min_{\mathbf{W}} \sum_{j=1}^n \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j; \mathbf{W})).$$

- Gradient regularization model:

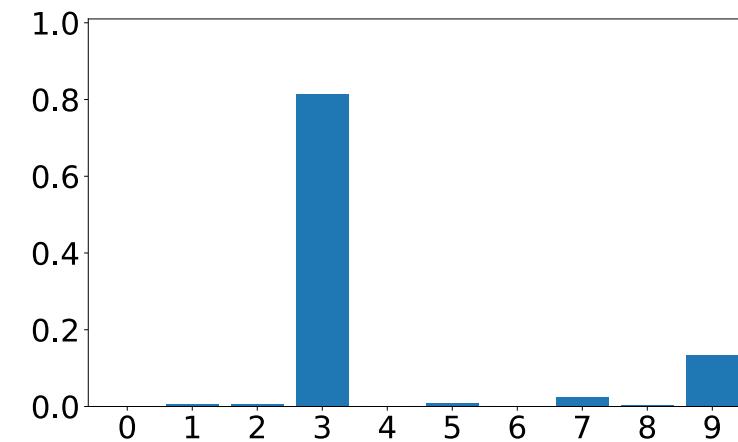
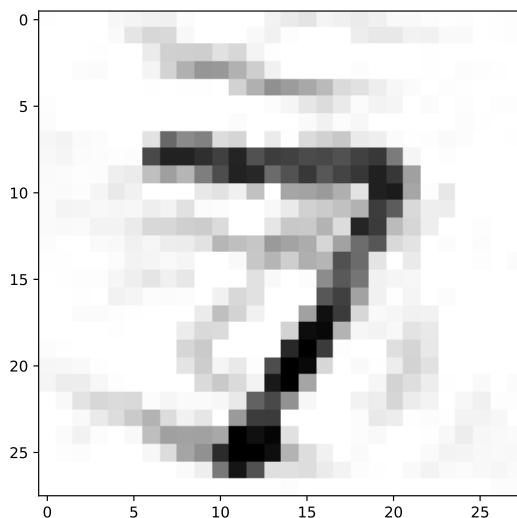
$$\min_{\mathbf{W}} \sum_{j=1}^n \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j; \mathbf{W})) + \lambda \left\| g(\mathbf{x}_j) \right\|_2^2.$$

- Here  $g(\mathbf{x})$  is the gradient of Loss at  $\mathbf{x}$ .

# Summary

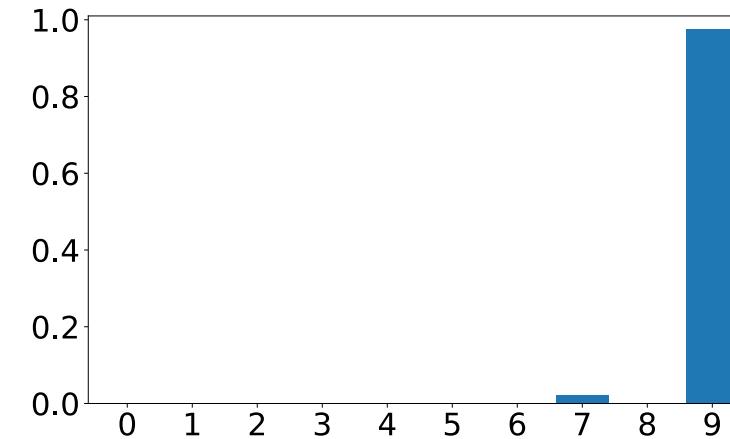
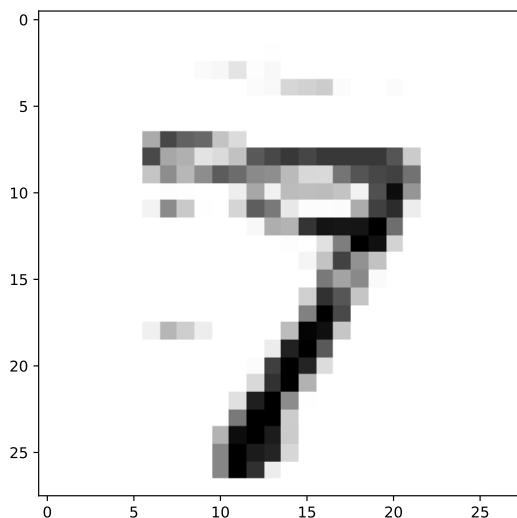
# Attack Neural Networks

- Untargeted attack.
  - Perturb the image to increase `Dist(true_target, prediction)`.
  - Just let the neural network make wrong prediction.
  - Do not know what the outcome will be.



# Attack Neural Networks

- Untargeted attack.
- Targeted attack.
  - Set a `fake_label` and minimize `Dist(fake_target, prediction)`.
  - Also keep it small: `Dist(fake_image, true image)`.
  - The neural network will make the wrong prediction as we expected.



# Adversarial Training

- Defend from attacks.
  - Why?
  - Slightly change a stop sign.
  - A self-driving car can ignore the stop sign.



# Adversarial Training

- Defend from attacks.
- Min-max model:

$$\min_{\mathbf{W}} \sum_{j=1}^n \left\{ \max_{\|\boldsymbol{\delta}\| < \sigma} \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j + \boldsymbol{\delta}; \mathbf{W})) \right\}.$$

- Gradient regularization model:

$$\min_{\mathbf{W}} \sum_{j=1}^n \text{Loss}(\mathbf{y}_j, \mathbf{f}(\mathbf{x}_j; \mathbf{W})) + \lambda \left\| \mathbf{g}(\mathbf{x}_j) \right\|_2^2,$$

where  $\mathbf{g}(\mathbf{x})$  is the gradient of Loss at  $\mathbf{x}$ .