Adversarial Robustness: Theory and Practice

Source: https://adversarial-ml-tutorial.org

Jiaru Zhang
AISIG Team

Version:	V1.0
Created:	Sep.19, 2020
Last Modified:	Sep.21, 2020
Report Date:	Sep.21, 2020

Contents

1 Introduction

2 Linear Models

3 Adversarial Examples

4 Adversarial Training

Introduction

Deep Learning: The Success Story

Image classification

Reinforcement Learning

Input sentence:	*Translation (PBMT):*	*Translation (GNMT):*	*Translation (human):*
李克強此行將啟動中加總理年度對話機制，與加拿大總理杜魯多舉行兩國總理首次年度對話。	Li Keqiang premier added this line to start the annual dialogue mechanism with the Canadian Prime Minister Trudeau two prime ministers held its first annual session.	Li Keqiang will start the annual dialogue mechanism with Prime Minister Trudeau of Canada and hold the first annual dialogue between the two premiers.	Li Keqiang will initiate the annual dialogue mechanism between premiers of China and Canada during this visit, and hold the first annual dialogue with Premier Trudeau of Canada.

Machine translation

Introduction

Is Deep Learning **truly** ready for real-world deployment?

Introduction

An Example

- Let’s use the (pre-trained) ResNet50 model to classify a picture.
- First step: Visualization

```
from PIL import Image
from torchvision import transforms
import matplotlib.pyplot as plt

# read the image, resize to 224 and convert to PyTorch Tensor
img = Image.open("pic.jpg")
preprocess = transforms.Compose([
   transforms.Resize(224),
   transforms.ToTensor(),
])
tensor = preprocess(img)[None,:,:,:]

# plot image
plt.imshow(tensor[0].numpy().transpose(1,2,0))
plt.show()
```

Introduction

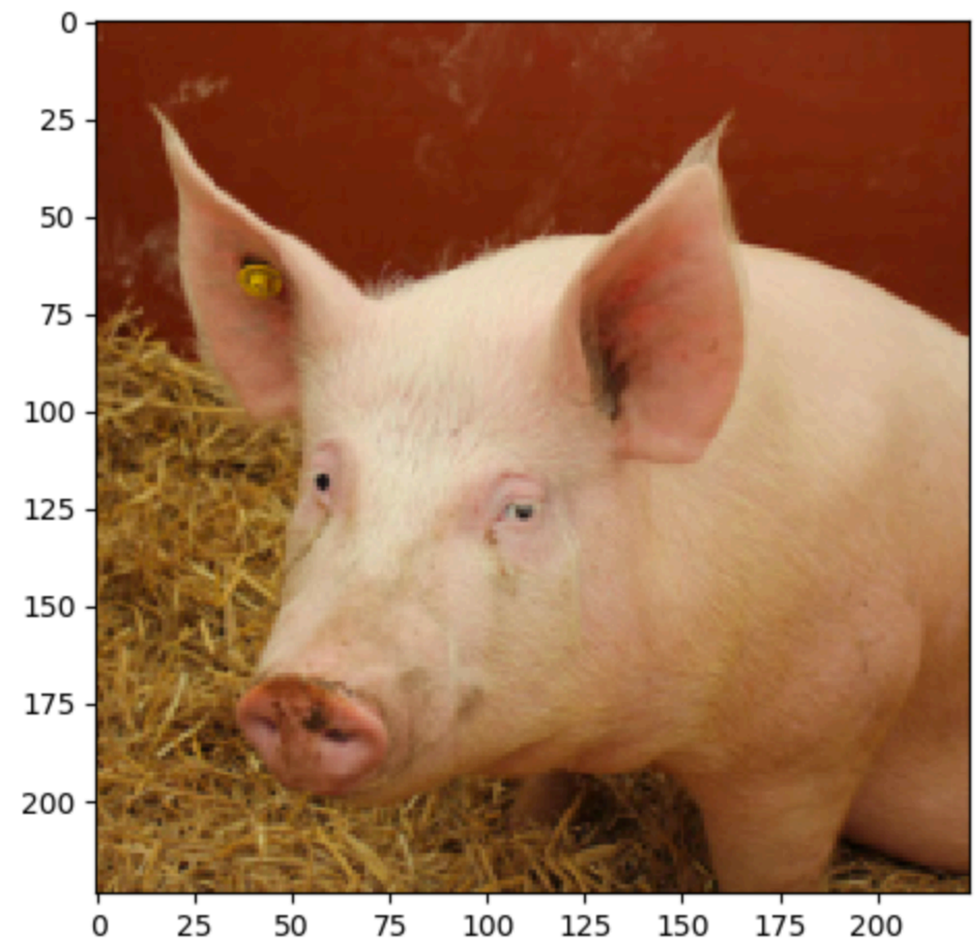
An Example

- Let's use the (pre-trained) ResNet50 model to classify a picture.
- First step: Visualization

```
from PIL import Image
from torchvision import transforms
import matplotlib.pyplot as plt

# read the image, resize to 224 and convert to PyTorch Tensor
img = Image.open("pic.jpg")
preprocess = transforms.Compose([
   transforms.Resize(224),
   transforms.ToTensor(),
])
tensor = preprocess(img)[None,:,:,:]

# plot image
plt.imshow(tensor[0].numpy().transpose(1,2,0))
plt.show()
```



- Result: What a lovely pig!

Introduction

An Example

- Let's use the (pre-trained) ResNet50 model to classify a picture.
- Second step: load the model and classify the picture:

```
import torch
import torch.nn as nn
from torchvision.models import resnet50

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]

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()
pred = model(norm(tensor))

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.max(dim=1)[1].item()])
```

- Result: hog. The image classifier works well.

Introduction

An Example

- Let's use the (pre-trained) ResNet50 model to classify a picture.
- Last step: Is our classifier confident?

```
# 341 is the class index corresponding to "hog"
print(nn.Softmax()(pred)[0, 341])
```

- Result: 0.9961. The image classifier is confident.

Introduction

Introductory Notation

- You are an attacker now. You want to fool this classifier.
- Let's define the problem formally.
- The optimization problem of classifier:

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

- It is typically solved by (stochastic) gradient descent:

$$\theta := \theta - \frac{\alpha}{|\mathscr{B}|} \sum_{i \in \mathscr{B}} \nabla_\theta \ell\left(h_\theta\left(x_i\right), y_i\right).$$

Introduction

Introductory Notation

- As an attacker, we want to adjust the image to **maximize** the loss:

$$\underset{\hat{x}}{\text{maximize}}\, \ell \left(h_\theta(\hat{x}), y\right).$$

- Of course, we cannot select arbitrary $\hat{x}$. We need to ensure it is close to x:

$$\underset{\delta \in \Delta}{\text{maximize}}\, \ell \left(h_\theta(x+\delta), y\right).$$

- Here we would like Δ to capture anything that humans visually feel to be the “same” as the original input.
- A common implementation is:

$$\Delta = \left\{\delta : \|\delta\|_\infty \leq \epsilon\right\}.$$

Introduction

Attack

- We simply PyTorch's SGD optimizer to optimize

$$\underset{\|\delta\|_\infty \le \epsilon}{\text{maximize}}\ \ell\left(h_\theta(x+\delta), y\right).$$

```
import torch.optim as optim

epsilon = 2. / 255

delta = torch.zeros_like(tensor, requires_grad=True)
opt = optim.SGD([delta], lr=1e-1)

for t in range(30):
   pred = model(norm(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 -0.003882253309711814
5 -0.0069345044903457165
10 -0.015822222456336021
15 -0.08059070259332657
20 -11.235554695129395
25 -15.812901496887207
True class probability: 6.352733521453047e-07
```

- After 30 gradient steps, the classifier thinks it cannot be a pig.

Introduction

Attack

- The classifier is quite sure the image is a wombat:

```
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: 0.9999040365219116
```

Introduction

Attack

- The classifier is quite sure the image is a wombat:

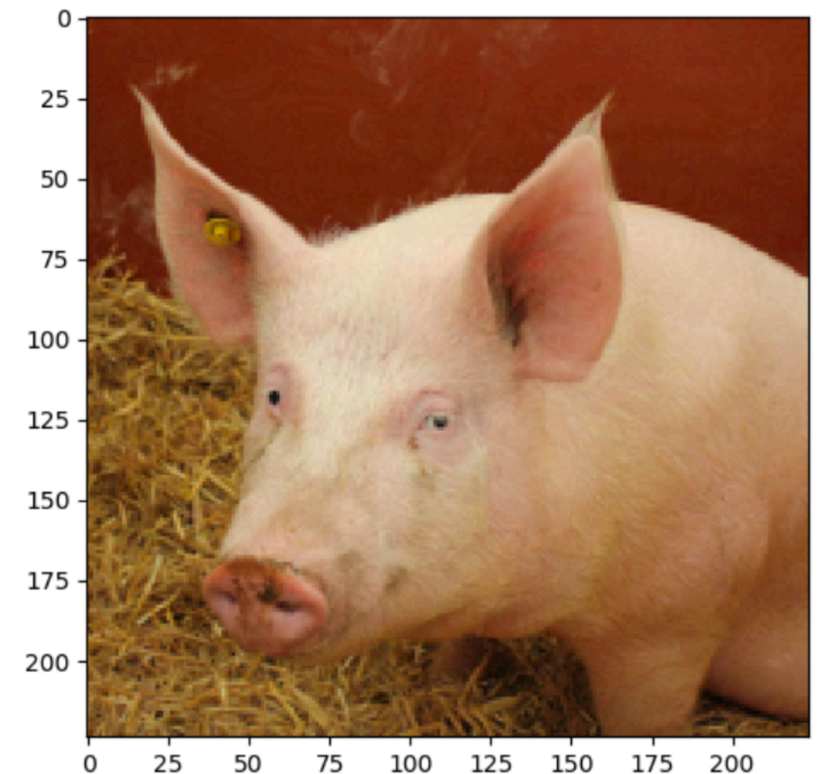
```
max_class = pred.max(dim=1)[1].item()
print("Predicted class: ", imagenet_classes[max_class])
print("Predicted probability:", nn.Softmax(dim=1)(pred)[0,max_class].item())
```

```
Predicted class:  wombat
Predicted probability: 0.9999040365219116
```

- So what does this wombat-pig look like?

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

- Extremely similar to our original pig.



Introduction

Targeted attacks

- What if we want the image to be classified as we desire?
- By modify the loss function:

$$\underset{\delta \in \Delta}{\text{maximize}} \left(\ell \left(h_\theta(x+\delta), y \right) - \ell \left(h_\theta(x+\delta), y_{\text{target}} \right) \right)$$

- Implementation:

```
for t in range(100):
   pred = model(norm(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)
```

Introduction

Targeted attacks

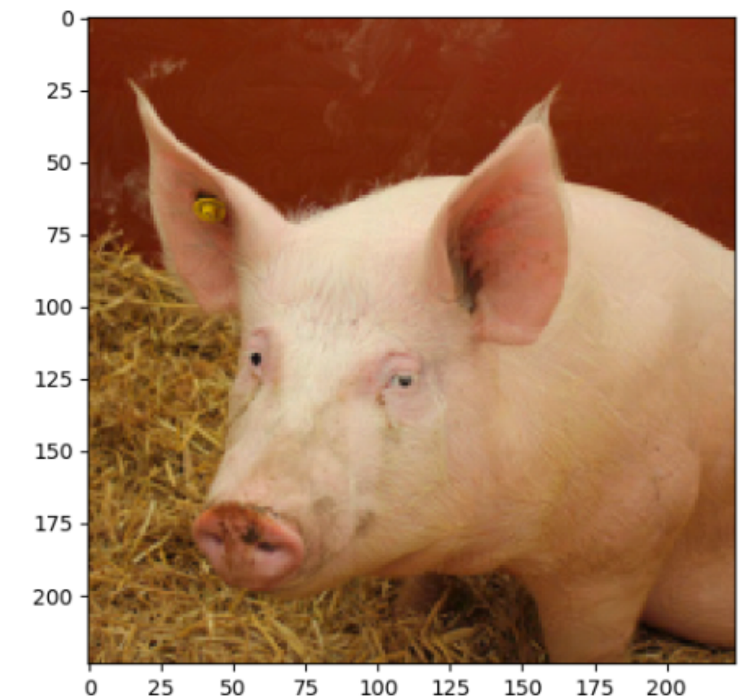
- What if we want the image to be classified as we desire?
- By modify the loss function:

$$\underset{\delta \in \Delta}{\text{maximize}} \left(\ell \left(h_\theta(x+\delta), y \right) - \ell \left(h_\theta(x+\delta), y_{\text{target}} \right) \right)$$

- Implementation:

```
for t in range(100):
    pred = model(norm(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)
```




```
Predicted class:  airliner
Predicted probability: 0.9028816223144531
```

- Results: The classifier is confident that it is an airliner.

Introduction

Training adversarially robust classifiers

- Recall: The optimization problem for attacker is

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

- Training adversarially robust classifiers can be written as the optimization problem

$$\underset{\theta}{\text{minimize}} \frac{1}{\left|D_{\text{train}}\right|} \sum_{(x,y)\in D_{\text{train}}} \max_{\delta \in \Delta(x)} \ell \left(h_\theta(x+\delta), y\right)$$

Introduction

Training adversarially robust classifiers

- Recall: The optimization problem for attacker is

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

- Training adversarially robust classifiers can be written as the optimization problem

$$\underset{\theta}{\text{minimize}}\, \frac{1}{\left|D_{\text{train}}\right|} \sum_{(x,y)\in D_{\text{train}}} \max_{\delta \in \Delta(x)} \ell\left(h_\theta(x+\delta), y\right)$$

- It is the **min-max** or **robust optimization formulation** of adversarial learning.
- Every **adversarial attack** and **defense** are a method for **approximately** solving the **inner maximization** and/or **outer minimization** problem respectively.

Introduction

Training adversarially robust classifiers

- As with traditional training, the solution of the problem is by stochastic gradient descent:

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

- But how do we compute the gradient of the inner item which contains a maximization problem?
- By Danskin's theorem, the gradient can be calculated by

$$\nabla_\theta \max_{\delta\in\Delta(x)} \ell\left(h_\theta(x+\delta), y\right) = \nabla_\theta \ell\left(h_\theta\left(x+\delta^\star\right), y\right)$$

where

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

Contents

1 Introduction

2 Linear Models

3 Adversarial Examples

4 Adversarial Training

Linear Models

Binary classification

- Consider the a simple classifier:

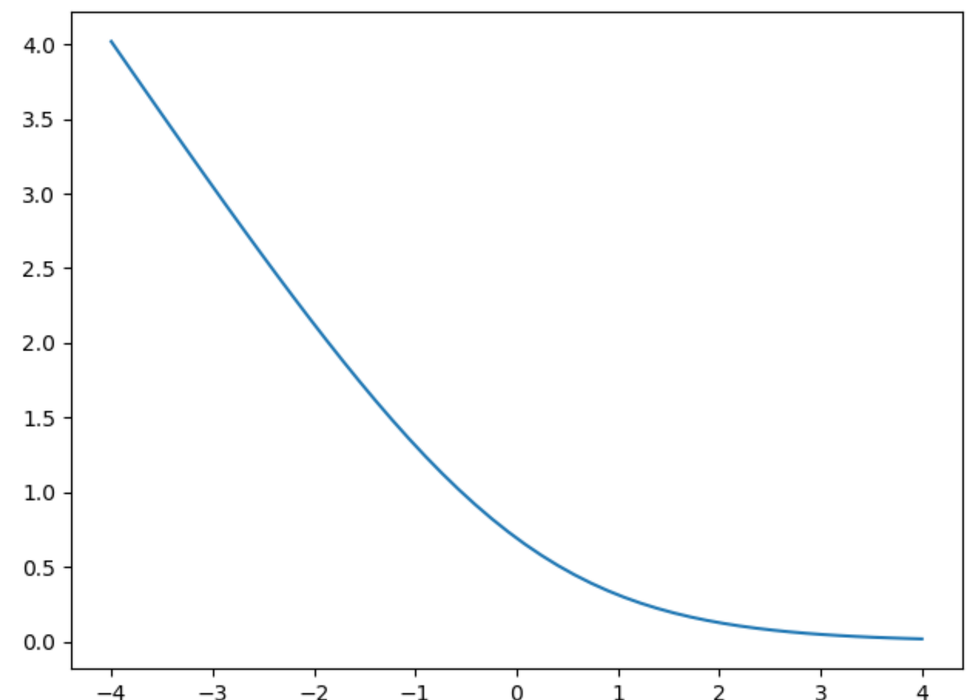
$$h_\theta(x) = w^T x + b$$

- The loss function:

$$\ell\left(h_\theta(x), y\right) = \log\left(1 + \exp\left(-y \cdot h_\theta(x)\right)\right) \equiv L(y \cdot h\theta(x))$$

- The function is a reversed softplus:

```
x = np.linspace(-4, 4)
plt.plot(x, np.log(1 + np.exp(-x)))
```



Linear Models

Inner Maximization Problem

- Here, the inner maximization problem is

- $$\underset{\|\delta\|\le\epsilon}{\text{maximize}}\,\ell\left(w^T(x+\delta),y\right) \equiv \underset{\|\delta\|\le\epsilon}{\text{maximize}}\,L\left(y\cdot\left(w^T(x+\delta)+b\right)\right)$$

$$= L\left(\min_{\|\delta\|\le\epsilon} y\cdot\left(w^T(x+\delta)+b\right)\right)$$

$$= L\left(y\cdot\left(w^Tx+b\right)+\min_{\|\delta\|\le\epsilon} y\cdot w^T\delta\right)$$

- Therefore, it is reduced to

$$\min_{\|\delta\|\le\epsilon} y\cdot w^T\delta$$

Linear Models

Inner Maximization Problem

- Therefore, it is reduced to

$$\min_{\|\delta\| \leq \epsilon} y \cdot w^T \delta$$

- Consider the ℓ_∞ norm constraint, i.e., each element of δ is in $[-\epsilon, \epsilon]$.
- We can clearly minimize this quantity when we set $\delta_i = -\epsilon$ for $w_i > 0$ and $\delta_i = \epsilon$ for $w_i < 0$ when $y = +1$.
- For $y = -1$, we just flip these quantities.
- Hence, the optimal solution is

$$\delta^\star = -y\epsilon \cdot \mathrm{sign}(w)$$

Linear Models

Implementation

- Firstly load the MNIST data with 0/1 labels. Codes omitted.
- Next, train the classifier with SGD:

```
def epoch(loader, model, opt=None):
    total_loss, total_err = 0., 0.
    for X, y in loader:
        yp = model(X.view(X.shape[0], -1))[:, 0]
        loss = torch.sum(nn.Softplus()(-yp * (2 * y - 1))) / len(X)
        if opt:
            opt.zero_grad()
            loss.backward()
            opt.step()

        total_err += ((yp > 0) * (y == 0) + (yp < 0) * (y == 1)).sum().item()
        total_loss += loss.item() * X.shape[0]
    return total_err / len(loader.dataset), total_loss / len(loader.dataset)


model = nn.Linear(784, 1)
opt = optim.SGD(model.parameters(), lr=1.)
print("Train Err", "Train Loss", "Test Err", "Test Loss", sep="\t")

for i in range(10):
    train_err, train_loss = epoch(train_loader, model, opt)
    test_err, test_loss = epoch(test_loader, model)
    print(*("{:.6f}".format(i) for i in (train_err, train_loss, test_err, test_loss)), sep="\t")
```

Linear Models

Implementation

- Here are the results:

```
Train Err    Train Loss   Test Err     Test Loss
0.000833     0.002530     0.000100     0.000575
0.000267     0.001023     0.000100     0.000465
```

...

```
0.000117     0.000449     0.000100     0.000398
0.000117     0.000407     0.000200     0.000413
```

- There are only two mis-classifying samples.
- Let's present one of them:

```
X_test = (test_loader.dataset.test_data[test_idx].float() / 255).view(len(test_idx), -1)
y_test = test_loader.dataset.test_labels[test_idx]
yp = model(X_test)[:, 0]
idx = (yp > 0) * (y_test == 0) + (yp < 0) * (y_test == 1)
plt.imshow(1 - X_test[idx][0].view(28, 28).numpy(), cmap="gray")
plt.title("True Label: {}".format(y_test[idx][0].item()))
```

Linear Models

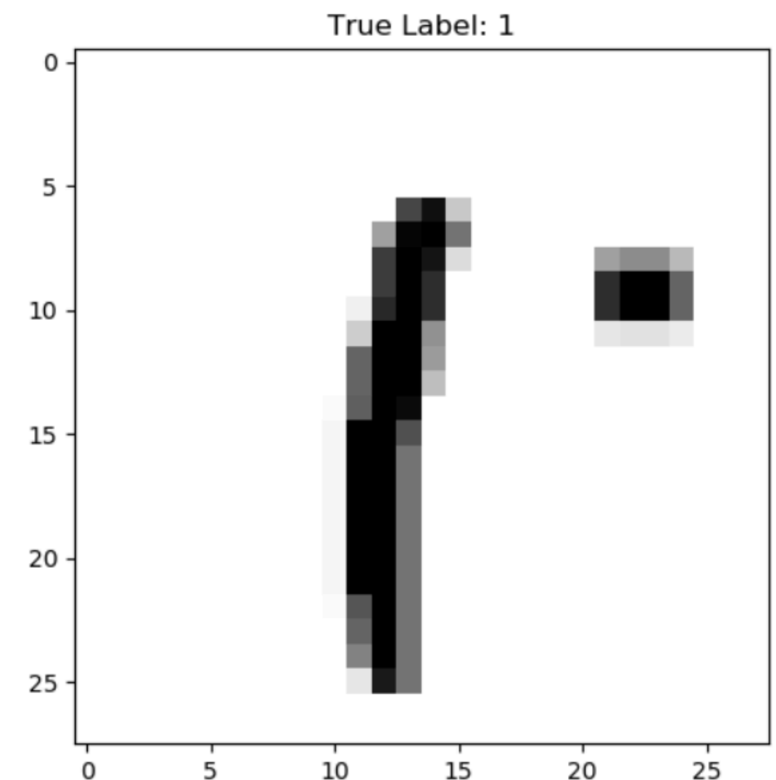
Implementation

- Here are the results:

```
Train Err    Train Loss   Test Err     Test Loss
0.000833     0.002530     0.000100     0.000575
0.000267     0.001023     0.000100     0.000465
...
0.000117     0.000449     0.000100     0.000398
0.000117     0.000407     0.000200     0.000413
```

- There are only two mis-classifying samples.
- Let's present one of them:

```
X_test = (test_loader.dataset.test_data[test_idx].float() / 255).view(l
y_test = test_loader.dataset.test_labels[test_idx]
yp = model(X_test)[:, 0]
idx = (yp > 0) * (y_test == 0) + (yp < 0) * (y_test == 1)
plt.imshow(1 - X_test[idx][0].view(28, 28).numpy(), cmap="gray")
plt.title("True Label: {}".format(y_test[idx][0].item()))
```



- It indeed seems a bit odd.

Linear Models

Implementation

- Recall the optimal perturbation is

$$\delta^\star = -y\epsilon \cdot \text{sign}(w)$$

- Let's present it:

```
epsilon = 0.2
delta = epsilon * model.weight.detach().sign().view(28, 28)
plt.imshow(1 - delta.numpy(), cmap="gray")
```

Linear Models

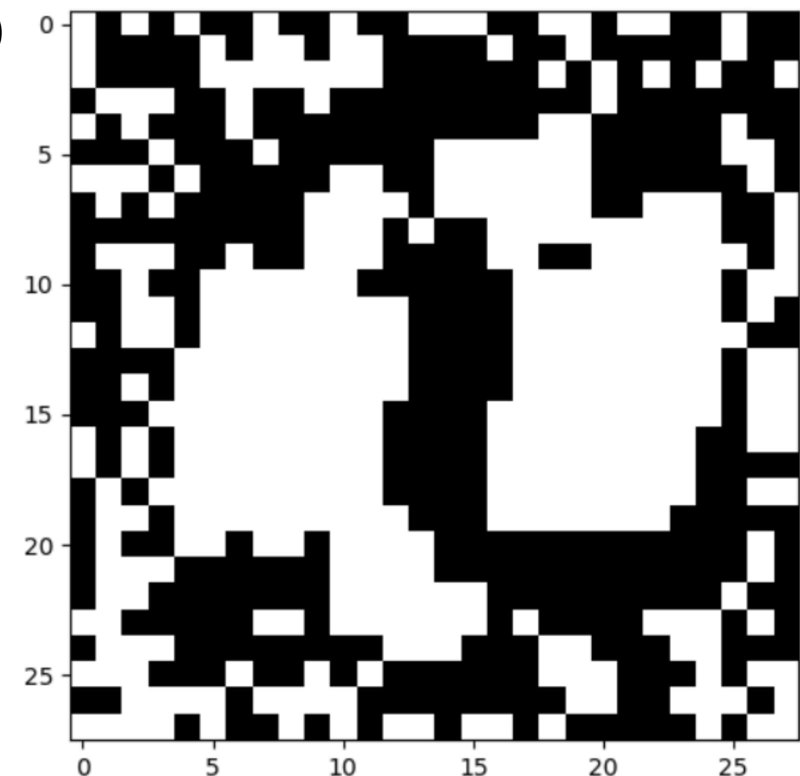
Implementation

- Recall the optimal perturbation is

$$\delta^\star = -y\epsilon \cdot \text{sign}(w)$$

- Let's present it:

```
epsilon = 0.2
delta = epsilon * model.weight.detach().sign().view(28, 28)
plt.imshow(1 - delta.numpy(), cmap="gray")
```



- We can intuitively regard it as a combination of a white "0" and a black "1", but it is not perfectly.

Linear Models

Implementation

- Next, see what happens after adding the noise:

```
def epoch_adv(loader, model, delta):
    total_loss, total_err = 0., 0.
    for X, y in loader:
        yp = model((X - (2 * y.float()[:, None, None, None] - 1) * delta).view(X.shape[0], -1))[:, 0]
        loss = torch.sum(nn.Softplus()(-yp * (2 * y - 1))) / len(X)
        total_err += ((yp > 0) * (y == 0) + (yp < 0) * (y == 1)).sum().item()
        total_loss += loss.item() * X.shape[0]
    return total_err / len(loader.dataset), total_loss / len(loader.dataset)

print(epoch_adv(test_loader, model, delta[None, None, :, :]))
```

- Result: `(0.1872, 0.9245074098825454)`
- There is an obvious error rate decrease: 0.0002 -> 0.1872
- The original tutorial reports a more sharp performance decrease, but I do not reproduce it successfully.

```
(0.8458628841607565, 3.4517438034075654)
```

Linear Models

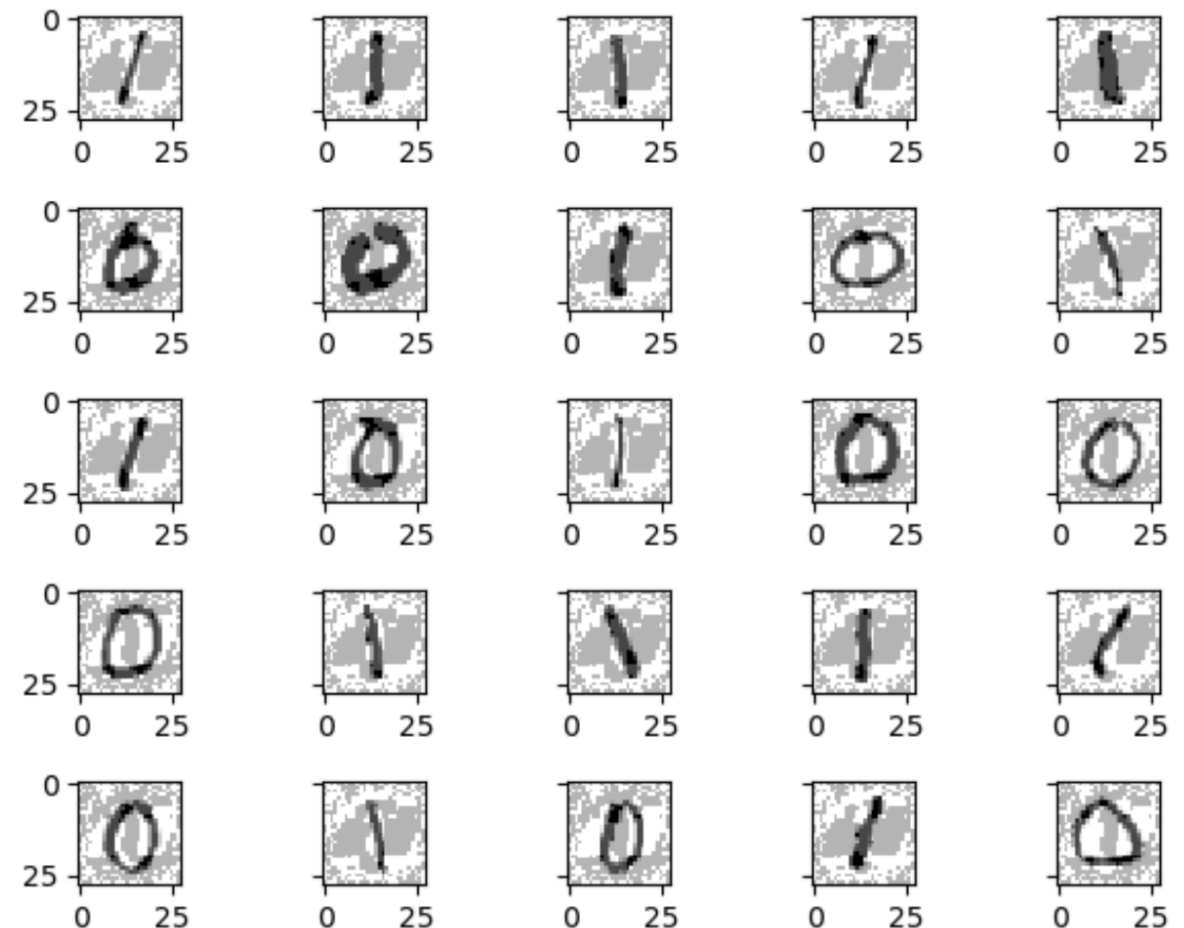
Implementation

- The perturbed images here are recognizably different:

```
f, ax = plt.subplots(5, 5, sharey=True)
for i in range(25):
    ax[i % 5][i // 5].imshow(1 - (X_test[i].view(28, 28) - (2 * y_test[i] - 1) * delta).numpy(), cmap="gray")

plt.show()
```

- But I do not think it can fool humans.



Linear Models

Outer Minimization Problem

- Recall the solution of the inner maximization is

$$L\left(y \cdot \left(w^T x + b\right) + \min_{\|\delta\| \le \epsilon} y \cdot w^T \delta\right)$$

- Here $\delta^\star = -y\epsilon \cdot \mathrm{sign}(w)$

Linear Models

Outer Minimization Problem

- Recall the solution of the inner maximization is

$$L\left(y \cdot \left(w^T x + b\right) + \min_{\|\delta\| \leq \epsilon} y \cdot w^T \delta\right)$$

- Here $\delta^\star = -y\epsilon \cdot \mathrm{sign}(w)$
- Therefore,

$$y \cdot w^T \delta^\star = y \cdot \sum_{i=1} -y\epsilon \cdot \mathrm{sign}(w_i) w_i = -y^2 \epsilon \sum_i |w_i| = -\epsilon \|w\|_1.$$

- Hence, the inner maximization is: $L\left(y \cdot \left(w^T x + b\right) - \epsilon \|w\|_1. \right)$

Linear Models

Outer Minimization Problem

- Recall the solution of the inner maximization is

$$L\left(y \cdot \left(w^T x + b\right) + \min_{\|\delta\| \leq \epsilon} y \cdot w^T \delta\right)$$

- Here $\delta^\star = -y\epsilon \cdot \mathrm{sign}(w)$

- Therefore,

$$y \cdot w^T \delta^\star = y \cdot \sum_{i=1} - y\epsilon \cdot \mathrm{sign}(w_i) w_i = -y^2 \epsilon \sum_i |w_i| = -\epsilon \|w\|_1.$$

- Hence, the inner maximization is: $L\left(y \cdot \left(w^T x + b\right) - \epsilon \|w\|_1.\right)$

- Now we can convert the outer minimization problem into:

$$\underset{w,b}{\mathrm{minimize}} \frac{1}{D} \sum_{(x,y) \in D} L\left(y \cdot \left(w^T x + b\right) - \epsilon \|w\|_1\right)$$

Linear Models

Implementation

- The robust training can be easily implemented by adding adversarial noise into the input:

```
def epoch_robust(loader, model, epsilon, opt=None):
    total_loss, total_err = 0., 0.
    for X, y in loader:
        yp = model(X.view(X.shape[0], -1))[:, 0] - epsilon * (2 * y.float() - 1) * model.weight.norm(1)
        loss = torch.sum(nn.Softplus()(-yp * (2 * y - 1))) / len(X)
        if opt:
            opt.zero_grad()
            loss.backward()
            opt.step()

        total_err += ((yp > 0) * (y == 0) + (yp < 0) * (y == 1)).sum().item()
        total_loss += loss.item() * X.shape[0]
    return total_err / len(loader.dataset), total_loss / len(loader.dataset)
model = nn.Linear(784, 1)
opt = optim.SGD(model.parameters(), lr=1e-1)
epsilon = 0.2
print("Rob. Train Err", "Rob. Train Loss", "Rob. Test Err", "Rob. Test Loss", sep="\t")
for i in range(20):
    train_err, train_loss = epoch_robust(train_loader, model, epsilon, opt)
    test_err, test_loss = epoch_robust(test_loader, model, epsilon)
    print(*("{:.6f}".format(i) for i in (train_err, train_loss, test_err, test_loss)), sep="\t")
```

- Let's see the result:

- The error rate decreases from 0.18 to 0.0045.

```
Rob. Train Err  Rob. Train Loss Rob. Test Err   Rob. Test Loss
0.026483    0.069028    0.012600    0.038966
...
0.006617    0.027721    0.004700    0.021207
0.006533    0.027627    0.004500    0.020877
```

Linear Models

Implementation

- What about the performance on normal samples?

```
train_err, train_loss = epoch(train_loader, model)
test_err, test_loss = epoch(test_loader, model)
print("Train Err", "Train Loss", "Test Err", "Test Loss", sep="\t")
print(*("{:.6f}".format(i) for i in (train_err, train_loss, test_err, test_loss)), sep="\t")
```

- Results:

Train Err	Train Loss	Test Err	Test Loss
0.001483	0.003676	0.000900	0.002072

- Now the model mis-classify 9 samples, instead of 2 before.
- In fact, it is a fundamental tradeoff between clean accuracy and robust accuracy.

Linear Models

Implementation

- Finally, let's look at the perturbation for the robust model:

```
delta = epsilon * model.weight.detach().sign().view(28,28)
plt.imshow(1-delta.numpy(), cmap="gray")
```

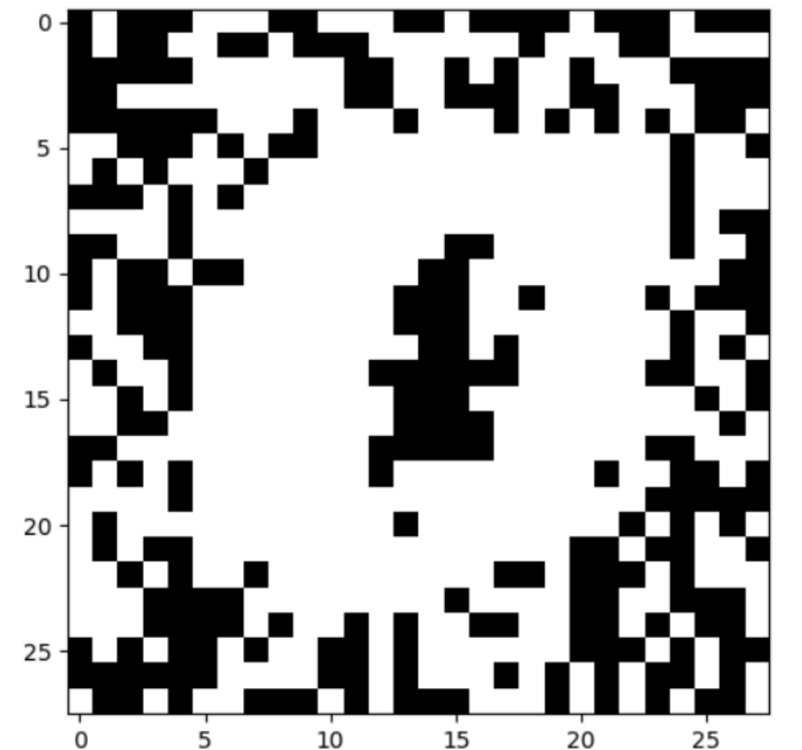
Linear Models

Implementation

- Finally, let's look at the perturbation for the robust model:

```
delta = epsilon * model.weight.detach().sign().view(28,28)
plt.imshow(1-delta.numpy(), cmap="gray")
```

- Result:
- It is more like a zero than the perturbation before.
- It means the model becomes more smart.
- We can never use "random noises" to fool it!



Contents

1 Introduction

2 Linear Models

3 Adversarial Examples

4 Adversarial Training

Adversarial Samples

Example Networks

- We use three different networks as example Networks:

```
model_dnn_2 = nn.Sequential(Flatten(), nn.Linear(784,200), nn.ReLU(),
                            nn.Linear(200,10)).to(device)

model_dnn_4 = nn.Sequential(Flatten(), nn.Linear(784,200), nn.ReLU(),
                            nn.Linear(200,100), nn.ReLU(),
                            nn.Linear(100,100), nn.ReLU(),
                            nn.Linear(100,10)).to(device)

model_cnn = nn.Sequential(nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(),
                          nn.Conv2d(32, 32, 3, padding=1, stride=2), nn.ReLU(),
                          nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
                          nn.Conv2d(64, 64, 3, padding=1, stride=2), nn.ReLU(),
                          Flatten(),
                          nn.Linear(7*7*64, 100), nn.ReLU(),
                          nn.Linear(100, 10)).to(device)
```

- We denote them as DNN2, DNN4, and CNN.

Adversarial Samples

Example Networks

- We trained them on the whole MNIST dataset:

```
def epoch(loader, model, opt=None):
    total_loss, total_err = 0., 0.
    for X, y in loader:
        X, y = X.to(device), y.to(device)
        yp = model(X)
        loss = nn.CrossEntropyLoss()(yp, y)
        if opt:
            opt.zero_grad()
            loss.backward()
            opt.step()

        total_err += (yp.max(dim=1)[1] != y).sum().item()
        total_loss += loss.item() * X.shape[0]
    return total_err / len(loader.dataset), total_loss / len(loader.dataset)

opt = optim.SGD(model_dnn_2.parameters(), lr=1e-1)
for _ in range(10):
    train_err, train_loss = epoch(train_loader, model_dnn_2, opt)
    test_err, test_loss = epoch(test_loader, model_dnn_2)
    print(*("{:.6f}".format(i) for i in (train_err, train_loss, test_err, test_loss)), sep="\t")
```

- The final test errors are: 0.0285, 0.0210, 0.0105.

Adversarial Samples

Moving to Neural Networks

- Returning back to the inner maximization problem again:

$$\underset{\|\delta\| \le \epsilon}{\text{maximize}}\, \ell \left(h_\theta(x), y\right)$$

- Here $h_\theta(x)$ is a neural network, so we cannot solve it analytically.
- However, we can find a lower bound on the optimization objective, by empirically finding feasible δ.

Adversarial Samples

The Fast Gradient Sign Method (FGSM)

- A direct method is to adjust δ on the direction of the gradient of x, i.e., we first compute the gradient

$$g := \nabla_\delta \ell \left(h_\theta(x + \delta), y \right)$$

- Then take a step $\delta := \delta + \alpha g$ for some step size α, then project back into the norm ball defined by $\|\delta\| \leq \epsilon$.
- We still consider the ℓ_∞ norm.
- If our initial δ is zero, and the step size is as large as possible, we have

$$\delta := \epsilon \cdot \text{sign}(g)$$

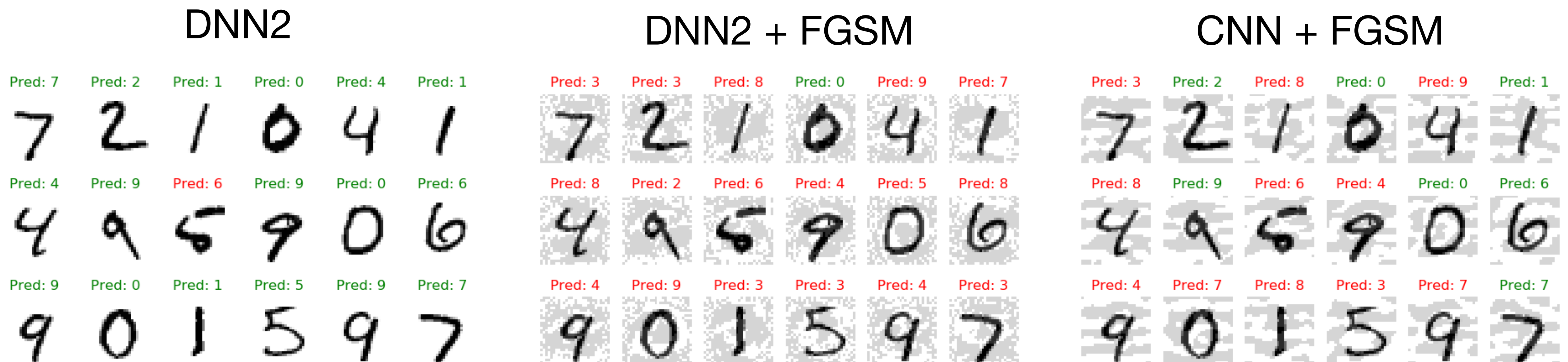
Adversarial Samples

Implementation

- It is easy to implement the FGSM method:

```
def fgsm(model, X, y, epsilon):
    """ Construct FGSM adversarial examples on the examples X"""
    delta = torch.zeros_like(X, requires_grad=True)
    loss = nn.CrossEntropyLoss()(model(X + delta), y)
    loss.backward()
    return epsilon * delta.grad.detach().sign()
```

- Let’s see the predictions firstly:



Adversarial Samples

Implementation

- It is easy to implement the FGSM method:

```
def fgsm(model, X, y, epsilon):
    """ Construct FGSM adversarial examples on the examples X"""
    delta = torch.zeros_like(X, requires_grad=True)
    loss = nn.CrossEntropyLoss()(model(X + delta), y)
    loss.backward()
    return epsilon * delta.grad.detach().sign()
```

- What about the error rate?

```
def epoch_adversarial(model, loader, attack, *args):
    total_loss, total_err = 0., 0.
    for X, y in loader:
        X, y = X.to(device), y.to(device)
        delta = attack(model, X, y, *args)
        yp = model(X + delta)
        loss = nn.CrossEntropyLoss()(yp, y)

        total_err += (yp.max(dim=1)[1] != y).sum().item()
        total_loss += loss.item() * X.shape[0]
    return total_err / len(loader.dataset), total_loss / len(loader.dataset)

print("2-layer DNN:", epoch_adversarial(model_dnn_2, test_loader, fgsm, 0.1)[0])
print("4-layer DNN:", epoch_adversarial(model_dnn_4, test_loader, fgsm, 0.1)[0])
print("        CNN:", epoch_adversarial(model_cnn, test_loader, fgsm, 0.1)[0])
```

- Results:

```
2-layer DNN: 0.933
4-layer DNN: 0.8918
        CNN: 0.4601
```

Very obvious decrease!

Adversarial Samples

Discussion on FGSM

- FGSM is one of the first methods for constructing adversarial examples.
- FGSM is specifically an attack under an ℓ_∞ norm bound but is easy to be generalized.
- Recall that FGSM is **exactly** the optimal attack against the linear model.
- It means that it assumes that the linear approximation of the classifier, hence it performs a single projected gradient step.

Adversarial Samples

Projected Gradient Descent (PGD)

- The basic Projected Gradient Descent (PGD) method is a simple multi-step generalization of FGSM:

$$\text{Repeat:} \quad \delta := \mathcal{P}(\delta + \alpha \nabla_\delta \ell(h_\theta(x+\delta), y))$$

- It is also easy to implement:

```
def pgd(model, X, y, epsilon, alpha, num_iter):
    """ Construct FGSM adversarial examples on the examples X"""
    delta = torch.zeros_like(X, requires_grad=True)
    for t in range(num_iter):
        loss = nn.CrossEntropyLoss()(model(X + delta), y)
        loss.backward()
        delta.data = (delta + X.shape[0]*alpha*delta.grad.data).clamp(-epsilon,epsilon)
        delta.grad.zero_()
    return delta.detach()
```

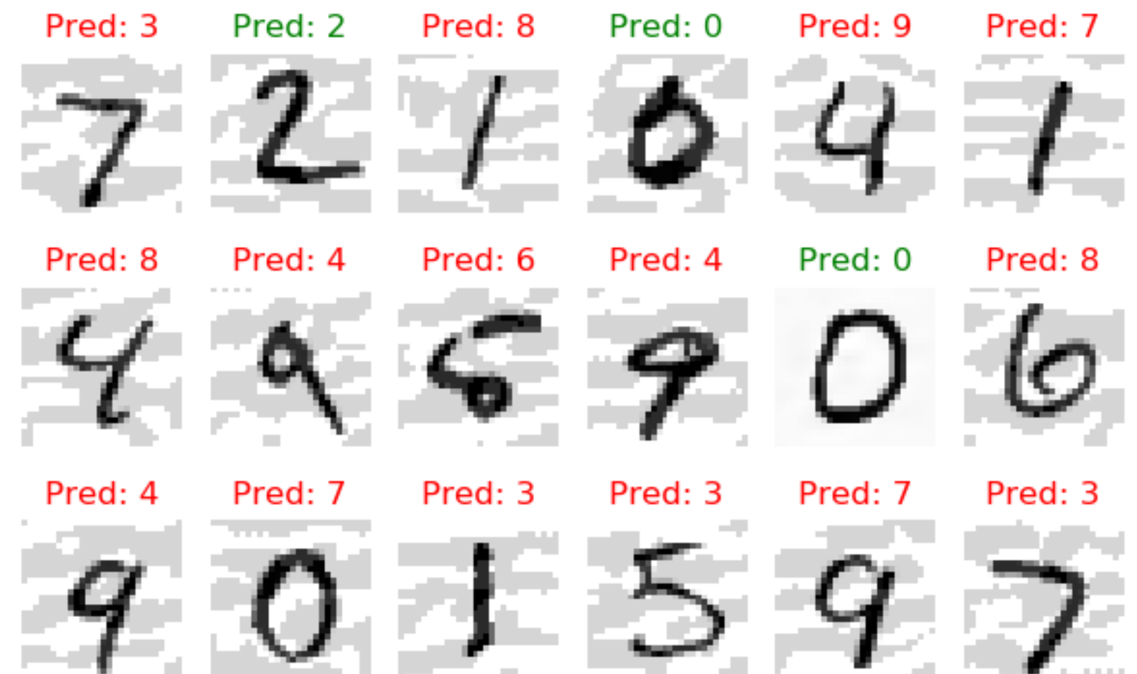
Adversarial Samples

Projected Gradient Descent (PGD)

- Let's take a look at the results on CNN:

```
delta = pgd(model_cnn, X, y, 0.1, 1e4, 1000)
yp = model_cnn(X + delta)
plot_images(X+delta, y, yp, 3, 6)
```

- It is not so much more effective than FGSM.
- There is one 0 digit where oddly the attack didn't seem to do anything at all.
- We use a very large α in PGD.



Adversarial Samples

Projected Gradient Descent (PGD)

- The reason is that the gradient of δ is very small:

```
delta = torch.zeros_like(X, requires_grad=True)
loss = nn.CrossEntropyLoss()(model_cnn(X + delta), y)
loss.backward()
print(delta.grad.abs().mean().item())
```

- Result:

```
5.1862962209270336e-06
```

- Therefore, the gradient-based updating like

$$z := z - \alpha \nabla_z f(z)$$

will be very small.

- Solution: normalization on the updating!

$$z := z - \underset{\|v\| \leq \alpha}{\operatorname{argmax}}\, v^T \nabla_z f(z)$$

Adversarial Samples

Projected Gradient Descent (PGD)

- Under the ℓ_∞ norm restriction, it is

$$\underset{\|v\|_\infty \le \alpha}{\operatorname{argmax}}\, v^T \nabla_z f(z) = \alpha \cdot \operatorname{sign}\left(\nabla_z f(z)\right)$$

- Implementation:

```
def pgd_linf(model, X, y, epsilon, alpha, num_iter):
    """ Construct FGSM adversarial examples on the examples X"""
    delta = torch.zeros_like(X, requires_grad=True)
    for t in range(num_iter):
        loss = nn.CrossEntropyLoss()(model(X + delta), y)
        loss.backward()
        delta.data = (delta + alpha*delta.grad.detach().sign()).clamp(-epsilon,epsilon)
        delta.grad.zero_()
    return delta.detach()
```

Adversarial Samples

Projected Gradient Descent (PGD)

- Results:

- Error rate comparison:

PGD	FGSM	None
2-layer DNN: 0.9685	2-layer DNN: 0.933	2-layer DNN: 0.0285
4-layer DNN: 0.9858	4-layer DNN: 0.8918	4-layer DNN: 0.0210
CNN: 0.7634	CNN: 0.4601	CNN: 0.0105

Adversarial Samples

Extensions

- Targeted attacks:

- Non-ℓ_∞ norms
- Exactly solving the inner maximization (combinatorial optimization)
- Upper bounding the inner maximization (convex relaxations)

Contents

1 Introduction

2 Linear Models

3 Adversarial Examples

4 Adversarial Training

Adversarial Training

Training robust models

- In the previous section, we focus on the inner maximization problem:

$$\underset{\|\delta\| \leq \epsilon}{\text{maximize}}\, \ell\left(h_\theta(x+\delta), y\right)$$

- Now we return to the min-max problem, which corresponds to the task of training robust models:

$$\underset{\theta}{\text{minimize}}\, \frac{1}{|S|} \sum_{x,y \in S} \max_{\|\delta\| \leq \epsilon} \ell\left(h_\theta(x+\delta), y\right)$$

- Two methods:
    - Train an (empirically) adversarially robust classifier by adversarial training.
    - Train a provably robust classifier using convex upper bounds.

Adversarial Training

Adversarial Training

- As we have discussed before, we can optimize θ by stochastic gradient descent:

$$\theta := \theta - \alpha \frac{1}{|B|} \sum_{x,y \in B} \nabla_\theta \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y).$$

- According to Danskin's Theorem, we need to
  1. Find the maximum,
  2. Compute the gradient at this point.
- However, we cannot find the maximum now. We can only find the approximate solutions.
- In other words, a "better" maximum point provides a more robust gradient descent procedure.

Adversarial Training

Implementation

- As we have discussed before, we can optimize θ by stochastic gradient descent.

```
def epoch_adversarial(loader, model, attack, opt=None, **kwargs):
    """Adversarial training/evaluation epoch over the dataset"""
    total_loss, total_err = 0., 0.
    for X, y in loader:
        X, y = X.to(device), y.to(device)
        delta = attack(model, X, y, **kwargs)
        yp = model(X + delta)
        loss = nn.CrossEntropyLoss()(yp, y)
        if opt:
            opt.zero_grad()
            loss.backward()
            opt.step()

        total_err += (yp.max(dim=1)[1] != y).sum().item()
        total_loss += loss.item() * X.shape[0]
    return total_err / len(loader.dataset), total_loss / len(loader.dataset)
```

```
opt = optim.SGD(model_cnn_robust.parameters(), lr=1e-1)
for t in range(10):
    train_err, train_loss = epoch_adversarial(train_loader, model_cnn_robust, pgd_linf, opt)
    test_err, test_loss = epoch(test_loader, model_cnn_robust)
    adv_err, adv_loss = epoch_adversarial(test_loader, model_cnn_robust, pgd_linf)
    if t == 4:
        for param_group in opt.param_groups:
            param_group["lr"] = 1e-2
    print(*("{:.6f}".format(i) for i in (train_err, test_err, adv_err)), sep="\t")
```

Adversarial Training

Implementation

- Results: (train_err, test_err, adv_err)

adv_training

train_err	test_err	adv_err
0.888133	0.886500	0.886500
0.662717	0.047000	0.129300
0.098050	0.024100	0.076100
0.056617	0.017800	0.052900
0.042133	0.012800	0.038700
0.027933	0.009600	0.034800
0.025950	0.010000	0.034200
0.025583	0.009300	0.032900
0.024400	0.009200	0.032300
0.023533	0.009300	0.033000

normal training

train_err	test_err	adv_err
0.244500	0.037000	0.719300
0.026350	0.019600	0.664600
0.017233	0.017700	0.684600
0.013133	0.014900	0.711200
0.010017	0.014000	0.674300
0.004567	0.011000	0.719700
0.003683	0.011000	0.719100
0.003050	0.010900	0.720600
0.002700	0.011400	0.717500
0.002350	0.011400	0.728900

- Adversarial training decreases the adversarial error.

Adversarial Training

Evaluating

- Does this model performs well facing other adversarial attacks? For example, FGSM, PGD with more iterative numbers?

```
print("FGSM: ", epoch_adversarial(test_loader, model_cnn_robust, fgsm)[0])
print("PGD, 40 iter: ", epoch_adversarial(test_loader, model_cnn_robust, pgd_linf, num_iter=40)[0])
print("PGD, small_alpha: ", epoch_adversarial(test_loader, model_cnn_robust, pgd_linf, num_iter=40, alpha=0.05)[0])
print("PGD, randomized: ", epoch_adversarial(test_loader, model_cnn_robust, pgd_linf,
                                             num_iter=40, randomize=True)[0])
```

- Results:

```
FGSM:  0.0301
PGD, 40 iter:  0.033
PGD, small_alpha:  0.0331
PGD, randomized:  0.0333
```

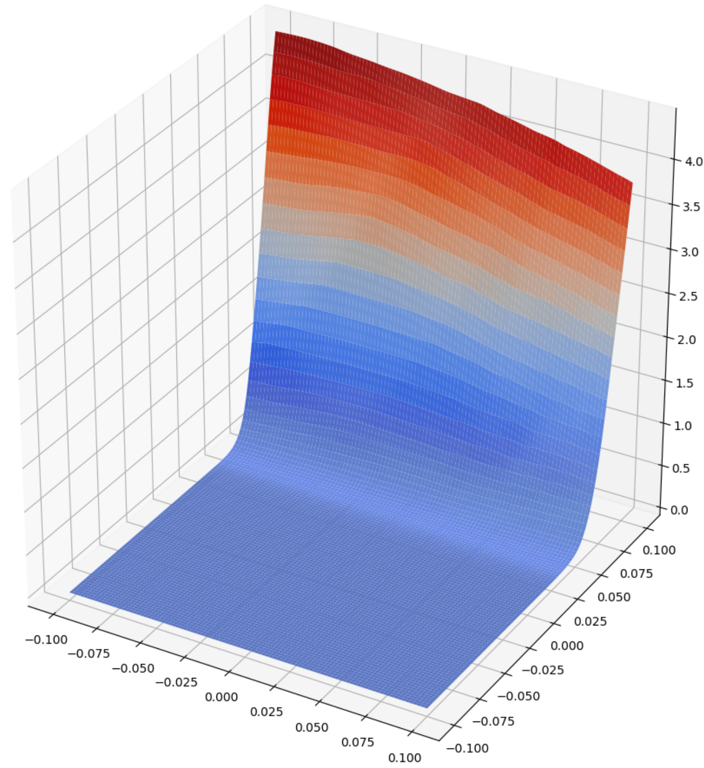
- Now maybe we are confident enough to make the model public and see whether anyone else can break it!

Adversarial Training

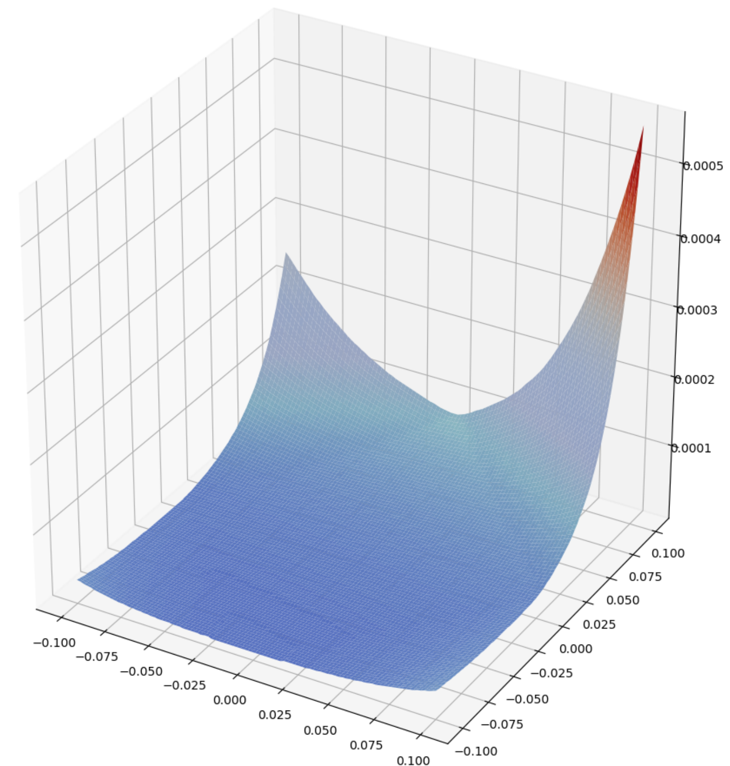
What is happening with these robust models?

- Why do these models work well against robust attacks?
- Let's look at the loss surface for CNN and the robust CNN (codes omitted):

CNN



Robust CNN



- The loss change of Robust CNN is very small.

Conclusion

- Review:

- Q & A

Appendix

Implementation

- Codes for loading datasets in Linear Models:

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim

mnist_train = datasets.MNIST("./data", train=True, download=True, transform=transforms.ToTensor())
mnist_test = datasets.MNIST("./data", train=False, download=True, transform=transforms.ToTensor())


def get_indices(dataset, class_names):
    indices = []
    for i in range(len(dataset.targets)):
        if dataset.targets[i] in class_names:
            indices.append(i)
    return indices



train_idx = get_indices(mnist_train, [0, 1])
train_loader = DataLoader(mnist_train, batch_size=64, sampler=torch.utils.data.sampler.SubsetRandomSampler(train_idx))

test_idx = get_indices(mnist_test, [0, 1])
test_loader = DataLoader(mnist_test, batch_size=64, sampler=torch.utils.data.sampler.SubsetRandomSampler(test_idx))
```

- Return