Chapter 4 - Adversarial training, solving the outer minimization

From adversarial examples to training robust models

앞장에서는 inner maximization problem을 푸는 방법에 집중하였다. 즉, 아래와 같이 표현할 수 있다.

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

아래 3가지 주요 기술을 다뤘다.

1. local gradient-based search(objective 에 대한 lower bound를 제안)
2. exact combinatorial optimization(objective를 exactly solving)
3. convex relaxztion (objective 에 대한 입증 가능한 upper bound를 제안)

이번 챕터에서는 우선 앞에서 다룬 min-max problem으로 다시 돌아와서 생각해보자. 즉, adversarial attack에 robust한 모델을 training하는 task. 어떤 adversary를 사용하든지 간에(attacker가 어떤 strategy를 사용할지 모르는 상황에서) 잘 작동하는 model을 만드는 것.

input, output 쌍의 집합인 S에 대해 우리는 아래와 같은 outer minimization problem을 해결하고자 한다.

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

Robust optimization formulation 의 목표는 공격자가 classifier의 parameters 에 대한 모든 정보를 알고 있고 어떤 parameter를 선택하든지간에 attack 방법을 특정 지을 수 있음을 가정했을 때도 공격이 불가능한 모델을 보증하는 것에 있다.

공격자가 어떤 "power"를 가졌는지는 특정 지을 수 없으니까 나름의 현실적인 공격 상황을 가정하고 모델을 평가해야 한다.

inner maximization problem을 할 때의 과정을 그대로 Adversarially robust system을 훈련할 때 사용할 것이다.

inner problem에서는 3가지 방법을 사용했는데

1. Local search를 통한 lower bounding
2. Combinator optimization을 통한 exact solutions
3. Convex relaxations를 통한 upper bound

하지만 실제로는 Training 과정에서는 2번을 적용할 수 없다.

이미 integer program solving 자체가 매우 time consuming 하고 이를 training procedure에 통합시키는 것은 practical 하지 않은 접근 방법이다.

따라서 2가지 선택지만이 남게 된다.

1. local search method를 통해 찾은 lower bounds와 examples를 사용해 (Empirical 하게) adversarially robust classifier를 훈련시키기
2. convex upper bounds를 통해 provably robust classifier를 훈련시키기

Adversarial training with adversarial examples

Adversarial training의 기본적인 idea는 훈련 과정에 Adversarial examples를 포함시키는 것이다. 즉, "standard" 훈련 과정은 Adversarial examples에 취약한걸 아니까 그냥 훈련 할 때 몇몇의 Adversarial examples를 포함 시키는 것이다.

이는 곧 어떤 Adversarial examples를 가지고 훈련해야 되는 물음을 가져온다.

먼저 앞에서 다룬 min-max objective를 다시 생각해보자. 우리는 아래와 같은 min-max objective를 gradient descent를 통해 최적화하는 것이 목적이다.

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

만약 θ 를 SGD를 통해 최적화하고 싶다면 단순히 몇몇 minibatch의 loss function을 θ 에 대한 gradient를 구해서 음의 방향으로 update시키면 된다. 즉, 아래와 같은 update를 반복하면 되는 것이다.

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

그렇다면 inner gradient는 어떻게 계산할까?

이 때 Danskin's Theorem을 이용하면 된다.

최대항을 포함하는 함수의 (sub)gradient를 계산하기 위해서는

1. 최대값을 찾고
2. 그 지점에 대해 gradient를 계산한다

즉, 아래와 같이 쓸 수 있다.

$$\nabla_\theta \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y) = \nabla_\theta \ell(h_\theta(x+\delta^\star(x)), y)$$

where

$$\delta^\star(x) = \operatorname*{argmax}_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y).$$

여기서 Danskin's theorem의 경우 최대값을 정확히 계산할 수 있는 경우에만 적용 가능하다. 하지만 정확한 최대값을 찾는 것은 쉬운 일이 아니다. 그리고 문제를 optimal하게 solve하지 못한다면 gradient의 특성에 대해 공식적으로 말하는 것 또한 어렵다.

그럼에도 불구하고 실제로는 robust gradient descent 과정의 질은 얼마나 maximization 문제를 잘 수행하는지에 따라 달려있다. 즉, adversarial training의 핵심은 inner maximization procedure에 strong attack을 잘 통합하는 것이다. 이 때 Projected gradient descent 방식이 우리가 알고 있는 가장 강력한 attack이다.

즉, adversarial training 전략은 아래와 같이 쓸 수 있다.

Repeat:

1. Select minibatch B, initialize gradient vector $g := 0$
2. For each (x, y) in B:
   a. Find an attack perturbation $\delta^\star$ by (approximately) optimizing
   $$\delta^\star = \operatorname*{argmax}_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$$
   b. Add gradient at $\delta^\star$
   $$g := g + \nabla_\theta \ell(h_\theta(x+\delta^\star), y)$$
3. Update parameters θ
   $$\theta := \theta - \frac{\alpha}{|B|} g$$

1. minibatch B를 선택, gradient vector를 $g := 0$으로 초기화

2. minibatch B에 있는 (x, y)에 대해

   a. $\delta^* = \underset{||\delta|| \leq \epsilon}{\text{argmax}} \ell(h_\theta(x+\delta), y)$ 를 (approximate하게) 최적화하는 attack perturbation δ^*을 찾는것

   b. δ^* 에 gradient를 합치는 것

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

3. 파라미터 θ 업데이트

$$\theta := \theta - \frac{\alpha}{|B|} g$$

위 과정을 코드로 구현해보자.

```
import torch import torch.nn as nn import torch.optim as optim from
torchvision import datasets, transforms from torch.utils.data import
DataLoader mnist_train = datasets.MNIST("../data", train=True,
download=True, transform=transforms.ToTensor()) mnist_test =
datasets.MNIST("../data", train=False, download=True,
transform=transforms.ToTensor()) train_loader = DataLoader(mnist_train,
batch_size = 100, shuffle=True) test_loader = DataLoader(mnist_test,
batch_size = 100, shuffle=False) device = torch.device("cuda:0" if
torch.cuda.is_available() else "cpu")
```

```
torch.manual_seed(0) class Flatten(nn.Module): def forward(self, x):
return x.view(x.shape[0], -1) 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)
```

```
def fgsm(model, X, y, epsilon=0.1): """ 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() def
pgd_linf(model, X, y, epsilon=0.1, alpha=0.01, num_iter=20,
randomize=False): """ Construct FGSM adversarial examples on the examples
X""" if randomize: delta = torch.rand_like(X, requires_grad=True)
delta.data = delta.data * 2 * epsilon - epsilon else: 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()
```

```
def epoch(loader, model, opt=None): """Standard 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) 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) 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)
```

Standard model에 대해 training을 한후 adversarial error를 평가해보자.

```
opt = optim.SGD(model_cnn.parameters(), lr=1e-1) for t in range(10):
train_err, train_loss = epoch(train_loader, model_cnn, opt) test_err,
test_loss = epoch(test_loader, model_cnn) adv_err, adv_loss =
epoch_adversarial(test_loader, model_cnn, 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")
torch.save(model_cnn.state_dict(), "model_cnn.pt")
```

```
# train error, test error, adversarial error 0.272300 0.031000 0.666900
0.026417 0.022000 0.687600 0.017250 0.020300 0.601500 0.012533 0.016100
0.673000 0.009733 0.014400 0.696600 0.003850 0.011000 0.705400 0.002833
0.010800 0.696800 0.002350 0.010600 0.707500 0.002033 0.010900 0.714600
0.001783 0.010600 0.708300#
```

```
model_cnn.load_state_dict(torch.load("model_cnn.pt"))
```

train error와 test error는 낮지만 adversarial error는 매우 높음을 알 수 있다.

이제는 adversarial training을 했을 때의 결과를 보자.

```
model_cnn_robust = 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)
```

```
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")
torch.save(model_cnn_robust.state_dict(), "model_cnn_robust.pt")
```

```
# train error, test error, adversarial error 0.715433 0.068900 0.170200
0.100933 0.020500 0.062300 0.053983 0.016100 0.046200 0.040100 0.011700
0.036400 0.031683 0.010700 0.034100 0.021767 0.008800 0.029000 0.020300
0.008700 0.027600 0.019050 0.008700 0.027900 0.019150 0.008600 0.028200
0.018250 0.008500 0.028300
```

```
model_cnn_robust.load_state_dict(torch.load("model_cnn_robust.pt"))
```

Evaluating robust models

Adversarial training을 한 결과 error rate가 2.8%로 original model의 error rate인 71%와 비교했을 때 많이 개선되었음을 알 수 있다(test accuracy 또한 개선 되었다).

과연 FGSM 이나 PGD, randomization이 적용된 PGD 등의 공격이 가해졌을 때도 error rate가 낮게 나오는지 살펴보자.

FGSM

```
print("FGSM: ", epoch_adversarial(test_loader, model_cnn_robust, fgsm)
[0])
```

```
FGSM: 0.0258
```

FGSM의 경우 잘 막아내는 모습이다. FGSM은 PGD의 step size $\alpha = \epsilon$ 만큼 한번 반복하는 방법이기 때문에 PGD보다 더 잘 안된다. PGD도 해보자.

PGD

```
print("PGD, 40 iter: ", epoch_adversarial(test_loader, model_cnn_robust,
pgd_linf, num_iter=40)[0])
```

```
PGD, 40 iter: 0.0286
```

PGD 에 대해서도 잘 막아낸다.

만약 step size를 더 줄인다면 어떻게 될까?

```
print("PGD, small_alpha: ", epoch_adversarial(test_loader,
model_cnn_robust, pgd_linf, num_iter=40, alpha=0.05)[0])
```

```
PGD, 40 iter: 0.0284
```

Randomization도 적용해보자.

```
print("PGD, randomized: ", epoch_adversarial(test_loader,
model_cnn_robust, pgd_linf, num_iter=40, randomize=True)[0])
```

PGD, randomized: 0.0284

이는 ℓ_∞ attack에 대한 adversarial training을 한 것이다. 즉, ℓ_1, ℓ_2, ℓ_∞에 모두 robust한 model을 만드는 것은 또다른 문제가 될 것이다.

What is happening with these robust models?

왜 이 모델들은 robust attack에 대해 잘 작동할까? 왜 다른 기법들은 robust model training에 대해 부족할까?

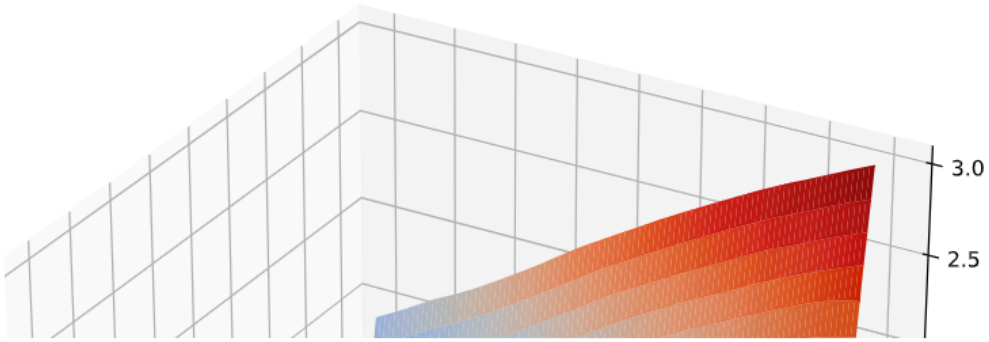
이 물음들에 답하기 위해 우리는 우선 trained classifier의 loss surface를 살펴볼 필요가 있다.

```
for X,y in test_loader: X,y = X.to(device), y.to(device) break
```

```
def draw_loss(model, X, epsilon): Xi, Yi = np.meshgrid(np.linspace(-
epsilon, epsilon,100), np.linspace(-epsilon,epsilon,100)) def
grad_at_delta(delta): delta.requires_grad_(True) nn.CrossEntropyLoss()
(model(X+delta), y[0:1]).backward() return
delta.grad.detach().sign().view(-1).cpu().numpy() dir1 =
grad_at_delta(torch.zeros_like(X, requires_grad=True)) delta2 =
torch.zeros_like(X, requires_grad=True) delta2.data =
torch.tensor(dir1).view_as(X).to(device) dir2 = grad_at_delta(delta2)
np.random.seed(0) dir2 = np.sign(np.random.randn(dir1.shape[0]))
all_deltas = torch.tensor((np.array([Xi.flatten(), Yi.flatten()]).T @
np.array([dir2, dir1])).astype(np.float32)).to(device) yp =
model(all_deltas.view(-1,1,28,28) + X) Zi =
nn.CrossEntropyLoss(reduction="none")(yp,
y[0:1].repeat(yp.shape[0])).detach().cpu().numpy() Zi =
Zi.reshape(*Xi.shape) #Zi = (Zi-Zi.min())/(Zi.max() - Zi.min()) fig =
plt.figure(figsize=(10,10)) ax = fig.gca(projection='3d') ls =
LightSource(azdeg=0, altdeg=200) rgb = ls.shade(Zi, plt.cm.coolwarm) surf
= ax.plot_surface(Xi, Yi, Zi, rstride=1, cstride=1, linewidth=0,
antialiased=True, facecolors=rgb)
```

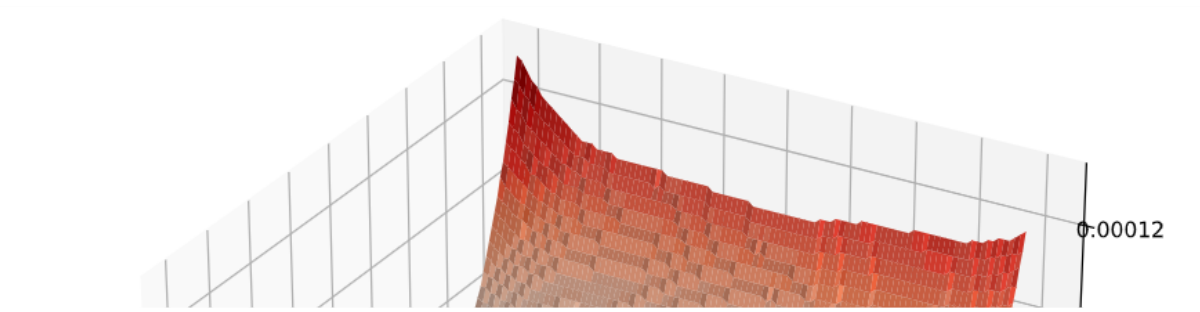
Standard network에 대한 loss surface를 살펴보자.

```
draw_loss(model_cnn, X[0:1], 0.1)
```



Robust model의 loss surface를 살펴보자.

```
draw_loss(model_cnn_robust, X[0:1], 0.1)
```



그림을 보면 robust model의 loss surface가 bumpy한데 그 이유는 z축의 scale이 더 작기 때문이다. Standard model과 같은 scale에 놓으면 평평해질 것이다.

Robust model의 loss surface는 gradient 방향과 random 방향으로 모두 평평하다는 특징을 보인다.

반면에 Traditionally trained model의 경우 gradient direction과 random direction에 대해 모두 다양함을 알 수 있다.

정리하자면 PGD 기반 adversarial trained 된 모델은 smooth한 loss surface를 가지고 있다는 점에서 truly robust함을 알 수 있다.

Relaxation-based robust training

마지막으로 network 검증뿐만 아니라 훈련을 위해서 convex relaxation을 사용해보자.

우리가 훈련한 robust classifier의 robustness를 입증하기 위해 interval bound를 고려해야 한다. 만약 optimization objective가 모든 targeted class에 대해 positive하면 classifier가 adversarial attack에 대해 robust함을 의미한다는 것을 기억해야한다.

아래 코드를 통해 살펴보자.

```
def bound_propagation(model, initial_bound): l, u = initial_bound bounds
= [] for layer in model: if isinstance(layer, Flatten): l_ = Flatten()(l)
u_ = Flatten()(u) elif isinstance(layer, nn.Linear): l_ =
(layer.weight.clamp(min=0) @ l.t() + layer.weight.clamp(max=0) @ u.t() +
layer.bias[:,None]).t() u_ = (layer.weight.clamp(min=0) @ u.t() +
layer.weight.clamp(max=0) @ l.t() + layer.bias[:,None]).t() elif
isinstance(layer, nn.Conv2d): l_ = (nn.functional.conv2d(l,
layer.weight.clamp(min=0), bias=None, stride=layer.stride,
padding=layer.padding, dilation=layer.dilation, groups=layer.groups) +
nn.functional.conv2d(u, layer.weight.clamp(max=0), bias=None,
stride=layer.stride, padding=layer.padding, dilation=layer.dilation,
groups=layer.groups) + layer.bias[None,:,None,None]) u_ =
(nn.functional.conv2d(u, layer.weight.clamp(min=0), bias=None,
stride=layer.stride, padding=layer.padding, dilation=layer.dilation,
groups=layer.groups) + nn.functional.conv2d(l, layer.weight.clamp(max=0),
bias=None, stride=layer.stride, padding=layer.padding,
dilation=layer.dilation, groups=layer.groups) +
layer.bias[None,:,None,None]) elif isinstance(layer, nn.ReLU): l_ =
l.clamp(min=0) u_ = u.clamp(min=0) bounds.append((l_, u_)) l,u = l_, u_
return bounds def interval_based_bound(model, c, bounds, idx): # requires
last layer to be linear cW = c.t() @ model[-1].weight cb = c.t() @
model[-1].bias l,u = bounds[-2] return (cW.clamp(min=0) @ l[idx].t() +
cW.clamp(max=0) @ u[idx].t() + cb[:,None]).t() def
robust_bound_error(model, X, y, epsilon): initial_bound = (X - epsilon, X
+ epsilon) err = 0 for y0 in range(10): C = -torch.eye(10).to(device)
C[y0,:] += 1 err += (interval_based_bound(model, C, bounds,
y==y0).min(dim=1)[0] < 0).sum().item() return err def
epoch_robust_bound(loader, model, epsilon): total_err = 0 C = [-
torch.eye(10).to(device) for _ in range(10)] for y0 in range(10): C[y0]
[y0,:] += 1 for X,y in loader: X,y = X.to(device), y.to(device)
initial_bound = (X - epsilon, X + epsilon) bounds =
bound_propagation(model, initial_bound) for y0 in range(10): lower_bound
= interval_based_bound(model, C[y0], bounds, y==y0) total_err +=
(lower_bound.min(dim=1)[0] < 0).sum().item() return total_err /
len(loader.dataset)
```

이 bound를 사용했을 때 robustly trained model이 어떤 경우에는 AEs에 대해 민감하지 않은지를 입증할 수 있는지를 생각해보자.

```
# epsilon=0.1 epoch_robust_err(test_loader, model_cnn_robust, 0.1)
```

```
1.0
```

불행히도 위 interval-based bound는 훈련된 classifier에 대해 적합하지 않음을 알 수 있다. 더 작은 ϵ 값을 넣어서 이 방법에 대한 검증을 진행해보자.

예를들어 $\epsilon = 0.0001$ 을 넣었을 때 우리는 최종적으로 "reasonable"한 bound를 달성하게 된다.

```
# epsilon=0.0001 epoch_robust_err(test_loader, model_cnn_robust, 0.0001)
```

```
0.0261
```

이러한 접근 방식은 그다지 유용한 것 같지 않다. 이러한 오차는 network의 깊이에 따라 누적되는 경향이 있는데 그 이유는 interval bounds는 이전에 다룬 것처럼 layer가 쌓일 때 마다 점점 더 loose 해지기 때문이다. (그로 인해 이전 chapter에서 3-layer 짜리 model에다가 적용했을 때는 bounds를 그다지 나쁘지 않게 구할 수 있었던 것이다.)

Training using provable criteria

Upper bound에 기반한 loss를 최소화하기 위한 방향으로 network를 training 하면 우리는 의미있는 bounds를 얻을 수 있게 된다.

이를 하기 위해서는, classifier의 cross entropy loss의 손실 상한을 최소화하기 위해 interval bounds를 이용해야 한다.

logit vector를 만들어 targeted attack의 음의 objective 값들을 대체하고 이 vector를 cross entropy loss값으로 사용하게 되면 이는 original loss의 strict upper bound로써 작용한다.

코드로 구현하면 아래와 같다.

```
def epoch_robust_bound(loader, model, epsilon, opt=None): total_err = 0
total_loss = 0 C = [-torch.eye(10).to(device) for _ in range(10)] for y0
in range(10): C[y0][y0,:] += 1 for X,y in loader: X,y = X.to(device),
y.to(device) initial_bound = (X - epsilon, X + epsilon) bounds =
bound_propagation(model, initial_bound) loss = 0 for y0 in range(10): if
sum(y==y0) > 0: lower_bound = interval_based_bound(model, C[y0], bounds,
y==y0) loss += nn.CrossEntropyLoss(reduction='sum')(-lower_bound,
y[y==y0]) / X.shape[0] total_err += (lower_bound.min(dim=1)[0] <
0).sum().item() total_loss += loss.item() * X.shape[0]
#print(loss.item()) if opt: opt.zero_grad() loss.backward() opt.step()
return total_err / len(loader.dataset), total_loss / len(loader.dataset)
```

이제 이 bound를 이용해서 model을 훈련시켜보자. ϵ 값을 처음부터 0.1로 설정하면 모든 숫자에 대해 같은 probability로 예측하게 되고 다시 복구할 수 없게 된다. 따라서 ϵ값을 작은 값부터 시작해서 차근히 늘려가는 것이 좋은 방법이다.

```
torch.manual_seed(0) model_cnn_robust_2 = nn.Sequential(nn.Conv2d(1, 32,
3, padding=1, stride=2), nn.ReLU(), nn.Conv2d(32, 32, 3, padding=1, ),
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)
```

```
opt = optim.SGD(model_cnn_robust_2.parameters(), lr=1e-1) eps_schedule =
[0.0, 0.0001, 0.001, 0.01, 0.01, 0.05, 0.05, 0.05, 0.05, 0.05] + 15*[0.1]
print("Train Eps", "Train Loss*", "Test Err", "Test Robust Err",
sep="\t") for t in range(len(eps_schedule)): train_err, train_loss =
epoch_robust_bound(train_loader, model_cnn_robust_2, eps_schedule[t],
opt) test_err, test_loss = epoch(test_loader, model_cnn_robust_2)
adv_err, adv_loss = epoch_robust_bound(test_loader, model_cnn_robust_2,
0.1) #if t == 4: # for param_group in opt.param_groups: #
param_group["lr"] = 1e-2 print(*("{:.6f}".format(i) for i in
(eps_schedule[t], train_loss, test_err, adv_err)), sep="\t")
torch.save(model_cnn_robust_2.state_dict(), "model_cnn_robust_2.pt")
```

```
Train Eps Train Loss* Test Err Test Robust Err 0.000000 0.829700 0.033800
1.000000 0.000100 0.126095 0.022200 1.000000 0.001000 0.119049 0.021500
1.000000 0.010000 0.227829 0.019100 1.000000 0.010000 0.129322 0.022900
1.000000 0.050000 1.716497 0.162200 0.828500 0.050000 0.744732 0.092100
0.625100 0.050000 0.486411 0.073800 0.309600 0.050000 0.393822 0.068100
0.197800 0.050000 0.345183 0.057100 0.169200 0.100000 0.493925 0.068400
0.129900 0.100000 0.444281 0.067200 0.122300 0.100000 0.419961 0.063300
0.117400 0.100000 0.406877 0.061300 0.114700 0.100000 0.401603 0.061500
0.116400 0.100000 0.387260 0.059600 0.111100 0.100000 0.383182 0.059400
0.108500 0.100000 0.375468 0.057900 0.107200 0.100000 0.369453 0.056800
0.107000 0.100000 0.365821 0.061300 0.116300 0.100000 0.359339 0.053600
0.104200 0.100000 0.358043 0.053000 0.097500 0.100000 0.354643 0.055700
0.101500 0.100000 0.352465 0.053500 0.096800 0.100000 0.348765 0.051500
0.096700
```

그 어떤 $\epsilon = 0.1$ 로 bound 된 ℓ_∞ attack은 classifier를 9.67% 이상의 error를 낼 수 없을 것이다. (clean error의 경우 5.15%에 그쳤다.)

PGD에 한해서 adversarial attack이 얼마나 나쁜 영향을 미치게 되는지 살펴보자.

```
print("PGD, 40 iter: ", epoch_adversarial(test_loader,
model_cnn_robust_2, pgd_linf, num_iter=40)[0])
```

```
PGD, 40 iter: 0.0779
```