Chapter 3 - Adversarial examples, solving the inner maximization

Moving to neural networks

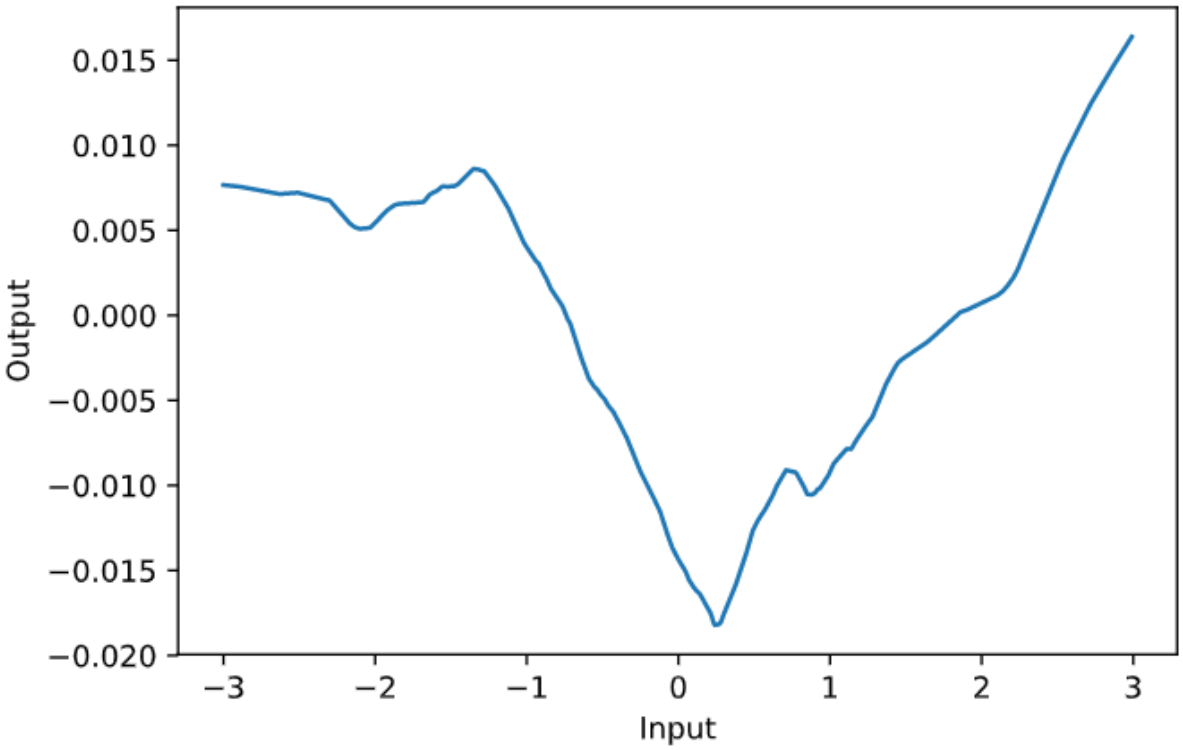
앞서 AEs와 robust optimization이 linear model에서 작동하는 맥락을 알아봤으니 이번에는 deep neural network에서의 possibility에 대해 알아보자.

다시 inner maximization problem을 살펴보자.

(1)

Neural networks의 경우 linear models 보다 loss surface가 훨씬 "irregular"하기 때문에 더 challenging하다. 아래 코드를 통해 simple (randomly trained) model과 simple linear function을 비교해보자.

```
import torch import torch.nn as nn import torch.optim as optim
torch.manual_seed(0) # random seed를 고정하기 위한 함수 model =
nn.Sequential(nn.Linear(1,100), nn.ReLU(), nn.Linear(100,100), nn.ReLU(),
nn.Linear(100,100), nn.ReLU(), nn.Linear(100,1)) opt =
optim.SGD(model.parameters(),lr=1e-2) for _ in range(100): loss =
nn.MSELoss()(model(torch.randn(100,1)), torch.randn(100,1))
opt.zero_grad() loss.backward() opt.step() plt.plot(np.arange(-3,3,0.01),
model(torch.arange(-3,3,0.01)[:,None]).detach().numpy())
plt.xlabel("Input") plt.ylabel("Output")
```



이러한 loss surface의 특징으로 인해 2가지 어려움이 있다. 첫번째는 neural network 에서는 small perturbation이 들어가도 loss 가 엄청나게 increase한다는 점이다. 두번째는 linear case와는 달리 inner maximization problem을 해결하기 어렵다는 점이다.

Strategies for the inner maximization

h_θ가 neural network 일 때 inner optimization problem을 푸는 방법?

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

lower bounds, exact solutions, 그리고 upper bounds에 따른 3가지 전략으로 inner optimization problem을 approximate하게 해결할 수 있다.

1. Lower bound : feasible한 δ 는 lower bound를 제시한다. 그리고 이는 "Empirical 하게 optimization problem을 해결하는 것", "AE를 찾는것"과 같은 의미가 된다.
2. Exact solution : Challenging. Mixed integer programming 등의 technique으로 해결할 수 있지만 이를 large model로 확장하여 푸는 것은 매우 어렵다. 하지만 몇몇 small model에 한해서는 inner maximization problem에 대한 exact solution을 construct 할 수 있다.

3. Upper bound : 네트워크 구조의 relaxation을 이용하는 것이 핵심이다. Relaxed version의 경우 original network의 정보를 포함하고 있으면서 optimize하기 쉬운 구조로 되어있다.

Some example networks

MNIST 로 간단하게 model training을 진행해보자. MNIST의 경우 discretization과 같이 너무 간단한 strategy도 잘 통한다는 점에서 불완전한 testbed이지만 기본적인 adversarial robustenss 원칙을 포함하고 있기 때문에 진행해볼 예정.

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
class Flatten(nn.Module): def forward(self, x): return x.view(x.shape[0], -1) # two FC layer network model_dnn_2 = nn.Sequential(Flatten(), nn.Linear(784,200), nn.ReLU(), nn.Linear(200,10)).to(device) # four FC layer network 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) # four conv layer + one FC layer 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)
```

아래는 모델 훈련 코드(pre-trained model을 이용할 경우 skip 해도 됨)

```
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)
```

```
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")
```

```
0.134117 0.517457 0.081000 0.287926 0.075600 0.265121 0.064000 0.221744
0.059050 0.209306 0.051100 0.182845 0.048983 0.172932 0.047100 0.157879
0.041817 0.146712 0.039800 0.136122 0.036300 0.126980 0.036100 0.123078
0.031217 0.112031 0.034500 0.115798 0.028333 0.100209 0.031500 0.106236
0.025267 0.090479 0.030300 0.099550 0.023033 0.082725 0.028500 0.093490
```

```
opt = optim.SGD(model_dnn_4.parameters(), lr=1e-1) for _ in range(10):
train_err, train_loss = epoch(train_loader, model_dnn_4, opt) test_err,
test_loss = epoch(test_loader, model_dnn_4) print(*("{:.6f}".format(i)
for i in (train_err, train_loss, test_err, test_loss)), sep="\t")
```

```
0.234967 0.777794 0.085400 0.274186 0.065917 0.221888 0.053700 0.169755
0.042750 0.143761 0.036000 0.118373 0.031367 0.106577 0.036700 0.114283
0.024883 0.083901 0.028500 0.090540 0.021017 0.069757 0.023400 0.077370
0.017367 0.057413 0.024400 0.075043 0.014700 0.048096 0.022900 0.070490
0.012167 0.039399 0.022500 0.074383 0.009717 0.033524 0.021000 0.072392
```

```
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) if t == 4: for param_group in
opt.param_groups: param_group["lr"] = 1e-2 print(*("{:.6f}".format(i) for
i in (train_err, train_loss, test_err, test_loss)), sep="\t")
```

```
0.206483 0.609042 0.026000 0.083938 0.024883 0.081515 0.017900 0.055983
0.017100 0.053789 0.016900 0.051890 0.012867 0.039944 0.013700 0.039924
0.009650 0.030815 0.013200 0.040250 0.004333 0.015506 0.010600 0.032351
0.003300 0.012391 0.010000 0.032211 0.002900 0.010868 0.010800 0.033129
0.002400 0.009762 0.010400 0.033581 0.001983 0.008888 0.010500 0.033701
```

```
torch.save(model_dnn_2.state_dict(), "model_dnn_2.pt")
torch.save(model_dnn_4.state_dict(), "model_dnn_4.pt")
torch.save(model_cnn.state_dict(), "model_cnn.pt")
```

위 코드를 모두 실행시키기 싫다면 아래 명령어를 통해 미리 train 되어있는 model을 사용할 수도 있다.

```
model_dnn_2.load_state_dict(torch.load("model_dnn_2.pt"))
model_dnn_4.load_state_dict(torch.load("model_dnn_4.pt"))
model_cnn.load_state_dict(torch.load("model_cnn.pt"))
```

The Fast Gradient Sign Method (FGSM)

이전에 우리는 δ 를 gradient 방향으로 줘야 함을 살펴봤다. Gradient는 아래와 같은 식으로 계산할 수 있다.

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

$$\delta := \delta + \alpha g$$

α : step size 만큼 δ를 gradient 방향으로 이동시키고 $||\delta|| <= \epsilon$ 에 따라 정의된 norm ball project 한다.

α 의 크기는 어떻게 정할까?

$\ell_\infty norm ||\delta||_\infty <= \epsilon$ 의 상황을 고려해보자. Norm ball로 project 하기 전에 δ 의 범위가 $[-\epsilon, \epsilon]$ 이 되게끔 clipping을 할 수 있다.

만약 δ 의 초기값이 0이라면 다음과 같이 update를 해줄 수 있다.

$$\delta := clip(\alpha g, [-\epsilon, \epsilon]).$$

만약 loss값을 최대한 많이 늘리고 싶다면 step size를 가능한한 크게 잡으면 된다. α 값이 충분히 크다면 g 값의 상대적인 entries 크기는 그다지 중요하지 않다 : 단순히 δ_i 를 g_i의 부호에 따라 $+\epsilon$ 혹은 $-\epsilon$ 으로 설정해주기만 하면 된다.

$$\delta := \epsilon \cdot sign(g)$$

이는 Fast Gradient Sign Method (FGSM)으로 알려져 있다.

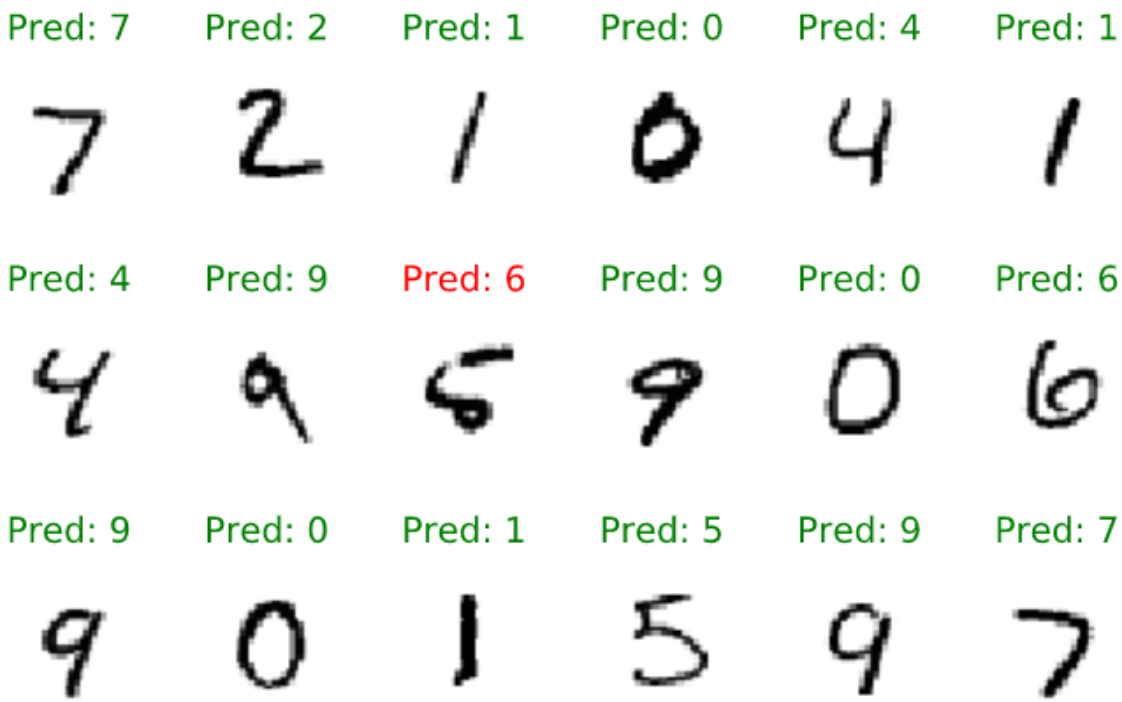
PyTorch로 살펴보자.

```
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()
```

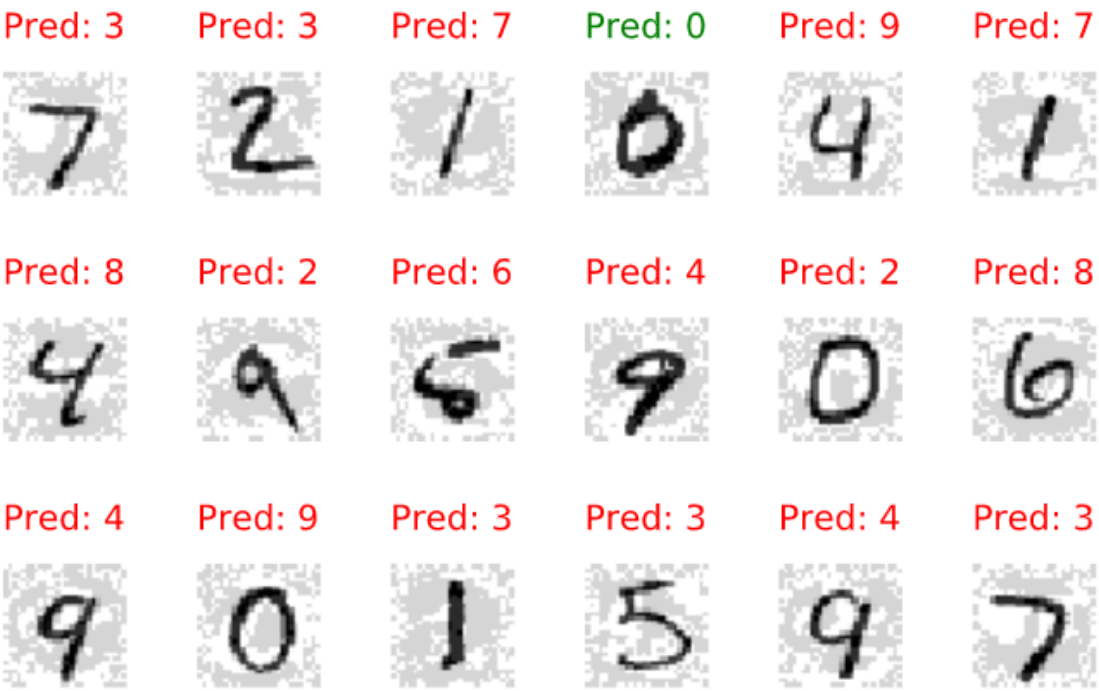
Classifier가 original images와 FGSM method로 attacked 된 images에 대해 어떻게 예측하고 있는지 살펴보자.

```
for X,y in test_loader: X,y = X.to(device), y.to(device) break def
plot_images(X,y,yp,M,N): f,ax = plt.subplots(M,N, sharex=True,
sharey=True, figsize=(N,M*1.3)) for i in range(M): for j in range(N):
ax[i][j].imshow(1-X[i*N+j][0].cpu().numpy(), cmap="gray") title = ax[i]
[j].set_title("Pred: {}".format(yp[i*N+j].max(dim=0)[1])) plt.setp(title,
color=('g' if yp[i*N+j].max(dim=0)[1] == y[i*N+j] else 'r')) ax[i]
[j].set_axis_off() plt.tight_layout()
```

```
### Illustrate original predictions yp = model_dnn_2(X) plot_images(X, y, yp, 3, 6)
```




```
### Illustrate attacked images delta = fgsm(model_dnn_2, X, y, 0.1) yp = model_dnn_2(X + delta) plot_images(X+delta, y, yp, 3, 6)
```



위처럼 2개의 FC layer로 이루어진 model의 경우 perturbation에 매우 취약하다.
ConvNet은 상대적으로 덜 하지만 여전히 민감하다.

```
### Illustrate attacked images delta = fgsm(model_cnn, X, y, 0.1) yp =
model_cnn(X + delta) plot_images(X+delta, y, yp, 3, 6)
```

Classifiers의 test errors는 어떻게 나오는지 살펴보자.

```
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])
```

```
2-layer DNN: 0.9259 4-layer DNN: 0.8827 CNN: 0.4173
```

FGSM에 대해 내릴 수 있는 결론

1. FGSM은 ℓ_∞ constraint 기반의 single projected gradient descent 이라고 볼 수 있다. 즉, FGSM은 ℓ_∞ attack이라고 볼 수 있다.
2. FGSM은 linear binary classification model 에 대해서는 ℓ_∞ norm 기반의 optimal attack임을 알 수 있다. 하지만 실제로 neural networks 은 사실상 linear 하다고 볼 수 없기 때문에 더 나은 방법을 고려해봐야 한다.

Projected gradient descent

다음으로 살펴볼 inner optimization problem을 maximize 하기 위한 방법은 단순히 projected gradient descent를 하는 것이다. (사실상 maximization을 하는 것이기 때문에 gradient ascent 라는 말이 맞긴하다.)

기본 PGD algorithm은 단순히 update를 반복하게 된다.

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

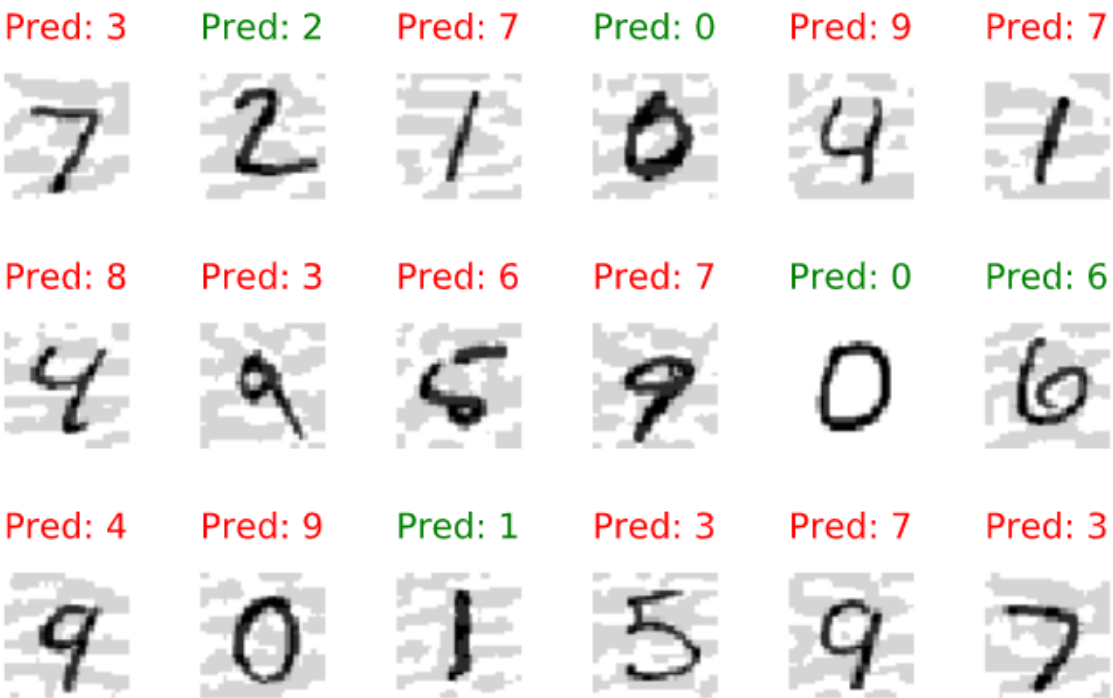
$\mathcal{P}$는 ball of interest 로의 transform을 의미한다(ℓ_∞norm 에서의 clipping과 같은 의미)

PGD에서는 step size나 iteration 횟수 등 더 특정 지을 수 있는 요소가 많다. 이 방법은 아래 코드와 같이 구현할 수 있다.

```
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()
```

CNN model에서 PGD attack이 들어간 sample이 어떻게 분류되는지 살펴보자.

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



FGSM과 비교했을 때 더 effective 한지도 애매하고 step size가 $\alpha = 1e4$ 로 매우 큰 값임을 볼 수 있다. 그 이유는 δ 및 $\delta = 0$ 에 대해서 gradient 값이 매우 작기 때문이다.

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

```
1.8276920172866085e-06
```

Gradient의 평균 절댓값은 초기 zero point에서 약 10^{-6} 정도 밖에 되지 않아서 이에 따라 α 값 까지 크게 scaling 되는 결과가 나온 것이다. 즉, boundary를 향해 굉장히 큰 step으로 접근하게 되며 이는 FGSM 방법과 거의 유사하다.

Aside : steepest descent

- Traditional gradient descent algorithm

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

- Normalized steepest descent

$$z := z - \arg\max_{||v|| \leq \alpha} v^T \nabla_z f(z)$$

- ℓ_2 norm constraint on v

$$\arg\max_{||v||_2 \leq \alpha} v^T \nabla_z f(z) = \alpha \frac{\nabla_z f(z)}{||\nabla_z f(z)||_2}$$

- ℓ_∞ norm constraint on v

$$\arg\max_{||v||_\infty \leq \alpha} v^T \nabla_z f(z) = \alpha \cdot sign(\nabla_z f(z))$$

ℓ_∞ norm constraint를 사용하는 경우 이는 mini-FGSM 을 여러번 반복하는 것과 같다.

이 방법은 현대 attack에서 사용되는 actual PGD 방법이다. PyTorch로 구현해보자.

```
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()
```

```
### Illustrate attacked images delta = pgd_linf(model_cnn, X, y,
epsilon=0.1, alpha=1e-2, num_iter=40) yp = model_cnn(X + delta)
plot_images(X+delta, y, yp, 3, 6)
```

FGSM 에 비해 더 공격이 잘 됨을 알 수 있다. step size α 의 경우 total perturbation bound인 ϵ 값에 비례해서 scaling 되기 때문에 α 는 ϵ 의 small fraction 값으로 해주는 것이 reasonable 하고 iteration number의 경우 ϵ/α 의 small multiple 값으로 선택해 주는 것이 타당하다.

전체 test set에 대해 공격이 어떻게 작동하는지 살펴보자.

```
print("2-layer DNN:", epoch_adversarial(model_dnn_2, test_loader,
pgd_linf, 0.1, 1e-2, 40)[0]) print("4-layer DNN:",
epoch_adversarial(model_dnn_4, test_loader, pgd_linf, 0.1, 1e-2, 40)[0])
print("CNN:", epoch_adversarial(model_cnn, test_loader, pgd_linf, 0.1,
1e-2, 40)[0])
```

```
2-layer DNN: 0.9637 4-layer DNN: 0.9838 CNN: 0.7432
```

FGSM attack에 비해 improve 되었음을 알 수 있다. 마지막으로 randomization을 통해 성능을 더 이끌어낼 수 있는 방법이 있다. 실제로는 잘 사용되지 않는 방법이긴 하다.

PGD의 경우 여전히 local optima에 의해 성능 한계가 존재한다. local optima를 피할 수 없기 때문에 이를 random restart를 통해 간접적으로 해결해볼 수 있다. 즉, PGD를 한번 실행 시키는 것이 아니라 random location으로 여러 번 실행 시키는 것이다.

```
def pgd_linf_rand(model, X, y, epsilon, alpha, num_iter, restarts): """
Construct PGD adversarial examples on the samples X, with random
restarts""" max_loss = torch.zeros(y.shape[0]).to(y.device) max_delta =
torch.zeros_like(X) for i in range(restarts): delta = torch.rand_like(X,
requires_grad=True) delta.data = delta.data * 2 * epsilon - epsilon 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_() all_loss = nn.CrossEntropyLoss(reduction='none')
(model(X+delta),y) max_delta[all_loss >= max_loss] = delta.detach()
[all_loss >= max_loss] max_loss = torch.max(max_loss, all_loss) return
max_delta
```

```
print("CNN:", epoch_adversarial(model_cnn, test_loader, pgd_linf_rand,
0.1, 1e-2, 40, 10)[0])
```

CNN: 0.7648

Targeted attacks

Targeted attack의 경우 true label 의 loss값을 최대화 하고 target label의 loss값을 최소화해야한다. 즉, 이는 다음과 같은 inner optimization problem을 해결하는 것과 같은 의미가 된다.

$$\max_{||\delta||\leq\epsilon}(\ell(h_\theta(x+\delta),y)-\ell(h_\theta(x+\delta),y_{targ}))\equiv\max_{||\delta||\leq\epsilon}(h_\theta(x+\delta)_{y_{targ}}-h_\theta(x+\delta)_y)$$

PGD attack으로 어떻게 구현하는지 살펴보자. (약간 더 큰 perturbation region $\epsilon = 0.2$ 를 사용할 예정. randomized restarts는 사용하지 않을 예정)

```
def pgd_linf_targ(model, X, y, epsilon, alpha, num_iter, y_targ): """
Construct targeted adversarial examples on the examples X""" delta =
torch.zeros_like(X, requires_grad=True) for t in range(num_iter): yp =
model(X + delta) loss = (yp[:,y_targ] - yp.gather(1,y[:,None])
[:,0]).sum() loss.backward() delta.data = (delta +
alpha*delta.grad.detach().sign()).clamp(-epsilon,epsilon)
delta.grad.zero_() return delta.detach()
```

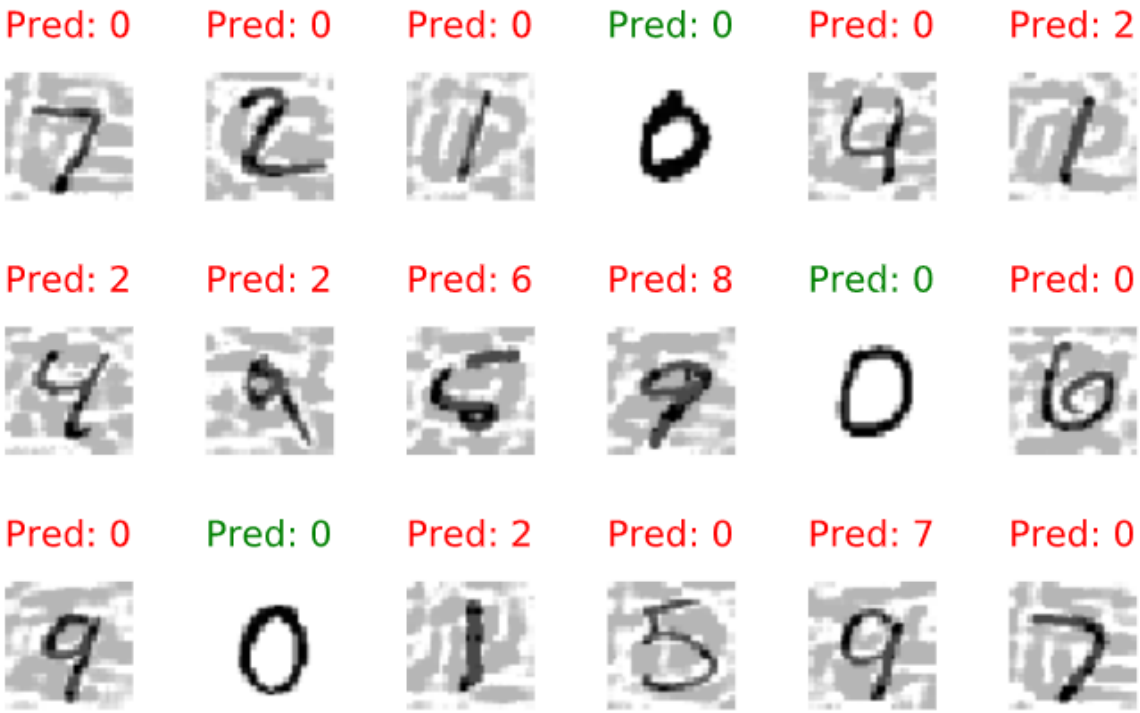
Target label을 2로 잡아보자.

```
# y_targ = 2 delta = pgd_linf_targ(model_cnn, X, y, epsilon=0.2,
alpha=1e-2, num_iter=40, y_targ=2) yp = model_cnn(X + delta)
plot_images(X+delta, y, yp, 3, 6)
```

ϵ 값이 조금 크지만 target class인 2로 했을 때 공격이 잘 되었음을 알 수 있다.

다음으로 target class를 0으로 해보자.

```
# y_targ=0 delta = pgd_linf_targ(model_cnn, X, y, epsilon=0.2, alpha=1e-
2, num_iter=40, y_targ=0) yp = model_cnn(X + delta) plot_images(X+delta,
y, yp, 3, 6)
```

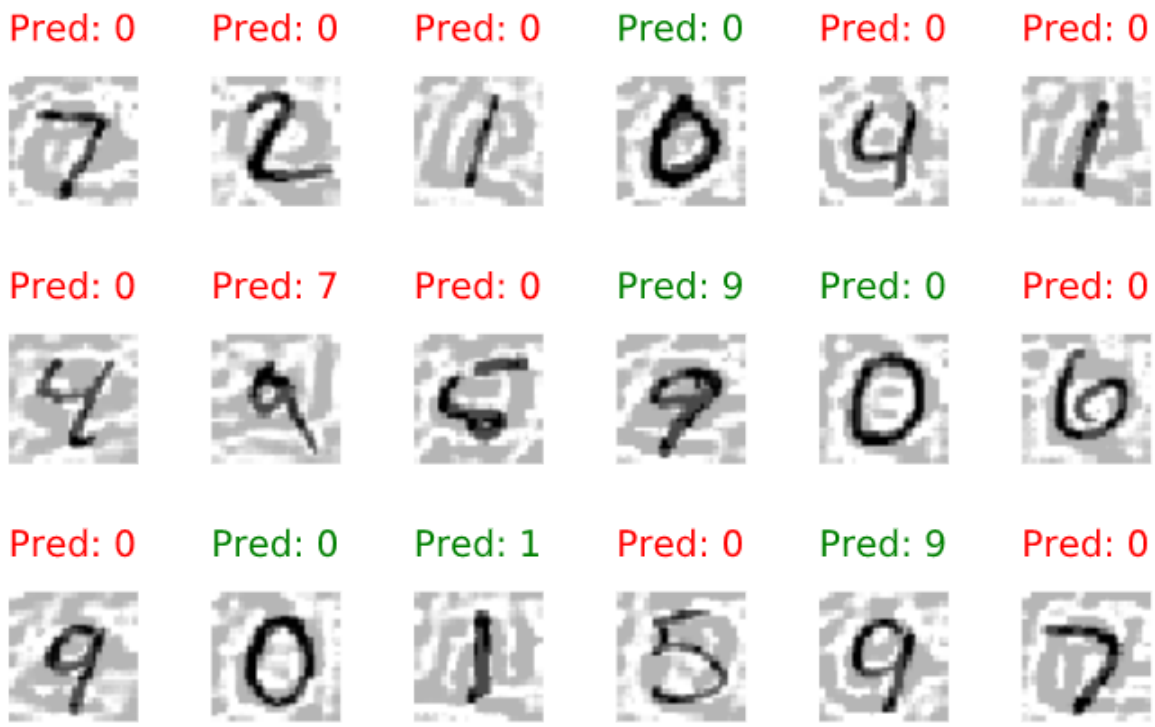


몇몇 case의 경우 target class로 분류가 되지 않았음을 볼 수 있는데 그 이유는 위 경우 0에 대한 logit 값과 true class에 대한 logit값의 차를 maximize 하는 것이 optimization 목표였고 다른 class에 대해서는 고려하지 않았기 때문이다. 따라서 이를 target class에 대한 logit을 maximize하면서 target이 아닌 다른 class에 대한 logit값은 minimize하면서 이 문제를 해결할 수 있다. 이를 식으로 표현하면 아래와 같이 표현할 수 있다.

$$\underset{||\delta|| \leq \epsilon}{\text{maximize}} (h_\theta(x+\delta)_{y_{targ}} - \sum_{y' \neq y_{targ}} h_\theta(x+\delta)_{y'}$$

```
def pgd_linf_targ2(model, X, y, epsilon, alpha, num_iter, y_targ): """
Construct targeted adversarial examples on the examples X""" delta =
torch.zeros_like(X, requires_grad=True) for t in range(num_iter): yp =
model(X + delta) loss = 2*yp[:,y_targ].sum() - yp.sum() loss.backward()
delta.data = (delta + alpha*delta.grad.detach().sign()).clamp(-
epsilon,epsilon) delta.grad.zero_() return delta.detach()
```

```
delta = pgd_linf_targ2(model_cnn, X, y, epsilon=0.2, alpha=1e-2,
num_iter=40, y_targ=0) yp = model_cnn(X + delta) plot_images(X+delta, y,
yp, 3, 6)
```



더 어려운 objective 이기 때문에 classifier를 완전히 fool 할 수는 없었다.

Non-ℓ_∞ norm

이번 section에서는 ℓ_2 norm 까지 고려해볼 예정. 이 때도 $x+\delta$ 의 범위가 $[0,1]$ 이어야 한다.

앞서 steepest descent method에서 살펴본 바를 기초하여 ℓ_2 ball의 projected normalized steepest descent method 는 아래와 같이 작성할 수 있다.

$$\delta := \mathcal{P}_\epsilon(\delta - \alpha \frac{\nabla_\delta \ell(h_\theta(x+\delta), y)}{||\nabla_\delta \ell(h_\theta(x+\delta), y)||_2})$$

$\mathcal{P}_\epsilon$ 의 의미는 반지름이 ϵ 인 ℓ_2 ball 로의 projection을 의미한다.

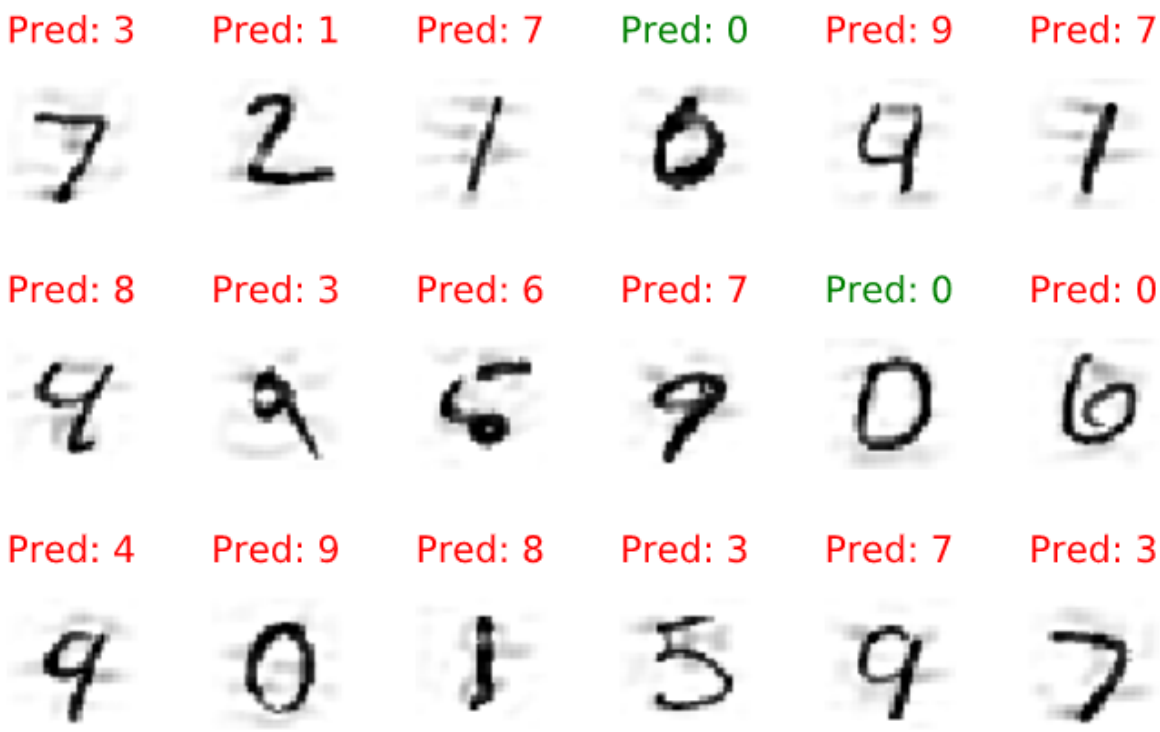
$$\mathcal{P}_\epsilon(z) = \epsilon \frac{z}{\max(\epsilon, ||z||_2)}$$

(2)

이에 따라 최종적으로 attack은 다음과 같이 할 수 있다.

```
def norms(Z): """Compute norms over all but the first dimension""" return
Z.view(Z.shape[0], -1).norm(dim=1)[:,None,None,None] def pgd_l2(model, X,
y, epsilon, alpha, num_iter): 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 +=
alpha*delta.grad.detach() / norms(delta.grad.detach()) delta.data =
torch.min(torch.max(delta.detach(), -X), 1-X) # clip X+delta to [0,1]
delta.data *= epsilon / norms(delta.detach()).clamp(min=epsilon)
delta.grad.zero_() return delta.detach()
```

```
delta = pgd_l2(model_cnn, X, y, epsilon=2, alpha=0.1, num_iter=40) yp =
model_cnn(X + delta) plot_images(X+delta, y, yp, 3, 6)
```




```
print("2-layer DNN:", epoch_adversarial(model_dnn_2, test_loader, pgd_l2,
2, 0.1, 40)[0]) print("4-layer DNN:", epoch_adversarial(model_dnn_4,
test_loader, pgd_l2, 2, 0.1, 40)[0]) print("CNN:",
epoch_adversarial(model_cnn, test_loader, pgd_l2, 2, 0.1, 40)[0])
```

```
2-layer DNN: 0.9232 4-layer DNN: 0.9435 CNN: 0.7854
```

여기서 주목해야할 점은 ℓ_2 norm perturbation에 사용되는 ϵ 의 크기는 ℓ_∞ norm perturbation 보다 커야한다는 것이다. 왜냐하면 ℓ_2 ball의 부피는 $\sqrt{n}$ 에 비례하고 ℓ_∞ ball의 부피는 n 에 비례하기 때문이다. (이 때 n은 input dimension 을 의미한다.) 예를들어 ℓ_∞ ball의 반지름이 $\epsilon = 0.1$ 인 경우 ℓ_2 ball에서는 $\frac{\sqrt{2.784}}{\sqrt{\pi e}} \cdot 0.1 \approx 1.35$ 가 되고 이것보다 약간 큰 $\epsilon = 2$ 를 사용하게 된다.

ℓ_∞ attack의 경우 image의 모든 곳에 small noise를 주는 방식이고 ℓ_2 attack의 경우 더 local한 부분에다가 더 큰 perturbation을 주게 되는 것이다.

What about all those other attacks you've read about?

FGSM 외에도 CW, DeepFool 등 여러 attack들이 존재하는데 결국 이 모든 attack 방법론들은 다음 2가지가 핵심이다.

1. 어떤 norm ball perturbation을 이용하는지
2. 그 norm ball을 optimize 하기 위해 어떤 method를 사용하는지

Exactly solving the inner maximization (combinatorial optimization)

Combinatorial optimization을 통해 inner maximization problem을 exactly solving 할 수 있는 방법을 알아봅시다.

Linear case와 마찬가지로 multiclass case에서 inner maximization problem을 exactly solve 하지는 않지만 특정 radius 내에 AE가 존재하는지 아닌지를 정확히 판단할 수 있다.

Mixed integer linear programming (combinatorial approach)를 사용할 예정이며 small model에 적용하여 neural network에 대한 정확한 추론을 수행할 수 있는 방법을 알아볼 예정이다.

An constrained formulation of targetted attacks

우선 deep classifier에 대한 targeted attack을 고려해보자. ReLU 기반의 feedforward network를 가정해보자.

$$
\begin{gathered}
z_1 = x \\
z_{i+1} = f_i(W_i z_i + b_i), i, ..., d \\
h_\theta(x) = z_{d+1}
\end{gathered}
$$

$f_i(z) = ReLU(z)$ for $i = 1, ..., d-1$

$f_d(x) = x$ (마지막 layer는 class logit을 의미한다.)

Targeted attack의 경우 target class logit $h_\theta(x+\delta)_{y_{targ}}$ 을 최대화하고 true class logit $h_\theta(x+\delta)_y$ 를 최소화하는 optimization problem으로 볼 수 있다.

이를 network의 구조를 고려하여 다시 optimization problem으로 구성해보면 아래와 같이 작성할 수 있다. input x와 intermediate actions z_i 로 표현해보자.

또한 ℓ_∞를 고려할 예정이다. (다른 norm 들도 사용할 수는 있지만 적용할 수 없는 경우도 있고 풀기 어려운 경우들도 있다.)

$$\underset{z_1,\cdots,d+1}{\text{minimize}}(e_y - e_{y_{targ}})^T z_{d+1}$$
$$subject\ to ||z_1 - x||_\infty \leq \epsilon$$
$$z_{i+1} = max(0, W_i z_i + b_i), i = 1, \ldots, d-1$$
$$z_{d+1} = W_d z_d + b_d$$

e_i : unit basis를 의미 (i번째에 1이 있고 나머지는 0인 벡터)

A mixed integer programming formulation

Binary Mixed Integer Linear Program (MILP)

- linear objective

$c^T z$ (z :optimization variable), c^T : coefficient vector)

- linear equality constraint

$Az = b$ (A : matrix, b : vector)

- linear inequality constraint

$Gz \leq h$ (G : matrix, h : vector)

- binary constraint

$z_i \in \{0, 1\} \forall_i \in \mathcal{I}$ ($\mathcal{I}$ 는 optimization variable의 subset이다)

이러한 문제는 $z_i \in \{0, 1\}$ 의 non-convex integer constraint 으로 인해 어려움이 발생한다. 이러한 제약 조건이 없었다면 이 문제는 linear program이 되어 time polynomial 하게 solve 될 수 있다.

MILP 문제 자체는 일반적으로 NP-hard 이기 때문에 현대에서 많이 사용되는 neural networks의 size만큼 확장될 수 있을 것이라고 기대하지 않는다.

하지만 MILP 자체는 매우 잘 연구된 영역이며 나이브하게 brute-force 방법으로 접근하는 것보다 훨씬 더 확장성이 뛰어난 접근 방식이 존재한다.

그래서 $z_{i+1} = max\{0, W_i z_i + b_i\}$ 제약조건을 어떻게 linear constraints와 binary integer constraint로 표현할 수 있을까?

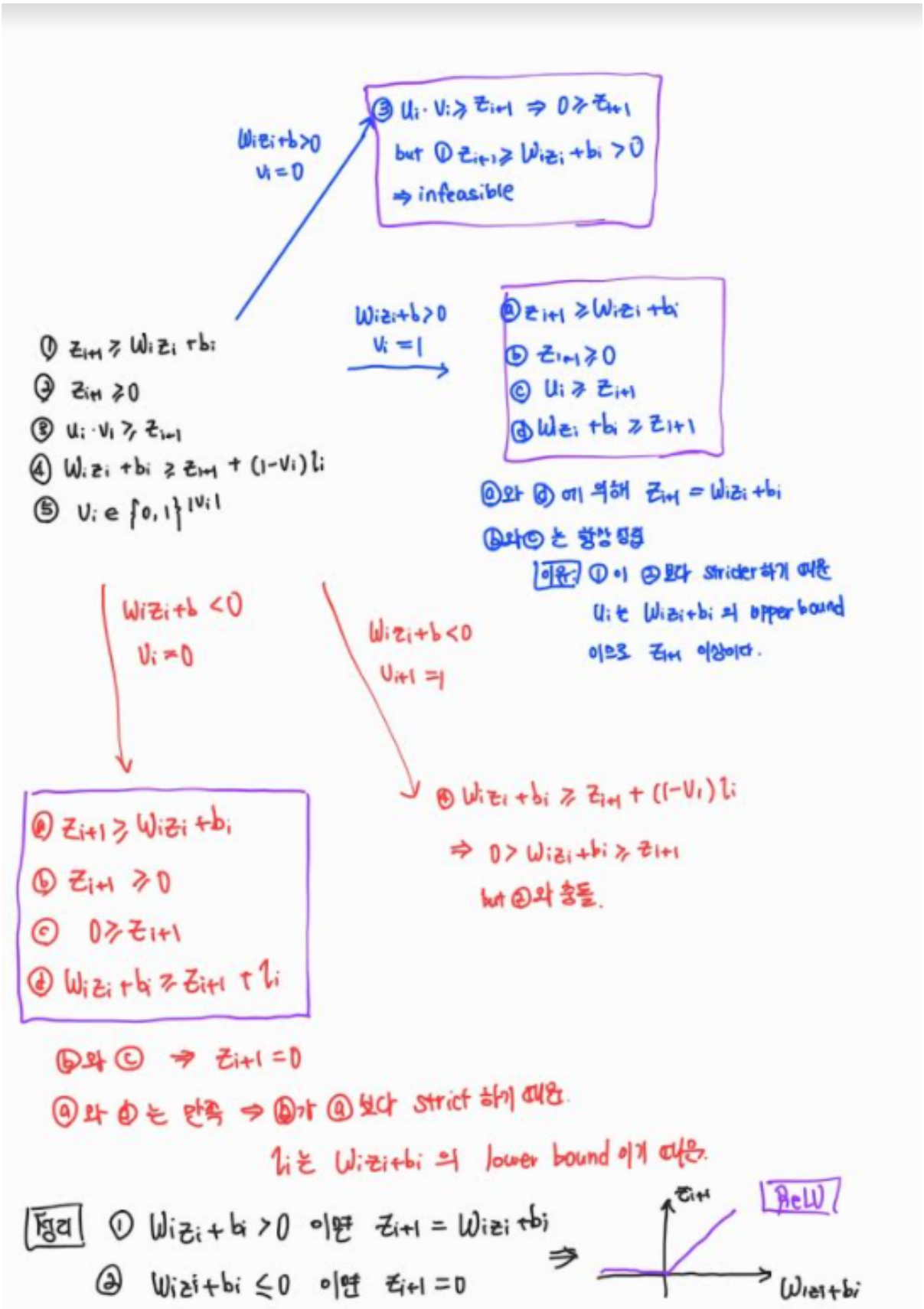
Linearization 이라는 방법을 사용할 예정.

$W_i z_i + b_i$ 가 가질 수 있는 upper and lower bound를 알 수 있다고 가정하고 lower bound value를 l_i , upper bound value를 u_i 라고 쓰자. 그리고 v_i 를 z_{i+1} 와 같은 size의 binary variable 집합이라고 놓자.

따라서 아래 부등식은 제약조건 $z_{i+1} = max\{0, W_i z_i + b_i\}$ 와 같다고 볼 수 있다.

$$
\begin{gathered}
z_{i+1} \geq W_i z_i + b_i \\
z_{i+1} \geq 0 \\
u_i \cdot v_i \geq z_{i+1} \\
W_i z_i + b_i \geq z_{i+1} + (1 - v_i) l_i \\
v_i \in \{0, 1\}^{|v_i|}
\end{gathered}
$$

위 식을 이해하기 위해 아래 과정에 대해 생각해볼 필요가 있다.



Finding upper and lower bounds

Upper and lower bounds l_i 와 u_i 는 어떻게 찾을 것인가?

사실 위에 나온 formulation의 경우 upper and lower bound 값이 어느 값이 되든지 간에 성립하는 식이기 때문에 크게 상관은 없다.

$l_i = -10^{100}$ 으로, $u_i = 10^{100}$ 으로 설정할 수 있는데 그 이유는 이 값은 웬만한 nerual network로 도달할 수 없는 값이기 때문이다.

하지만 integer programming solver의 solution time은 얼마나 좋은 upper and lower bound를 찾는지에 따라 결정이 된다.

흥미롭게도 우리는 upper and lower bound의 exact value를 앞서 얘기한 targeted attacks를 구하는 방법과 같은 방법으로 구할 수 있다. Targeted attack에서는 마지막 layer의 weight를 minimize 했듯이 upper/lower bound를 compute 할 때 intermediate layer의 single activation value $(l_k)_j$ 가 optimization problem의 solution이 된다.

$$\begin{gathered}\underset{z_1,\cdots k+1}{\text{minimize}}\ e_j^T z_{k+1} \\ \text{subject to} \quad ||z_1 - x||_\infty \leq \epsilon \\ z_{i+1} = max\{0, W_i z_i + b_i\}, i = 1, \cdots, k-1 \\ z_{k+1} = W_k z_k + b_k\end{gathered}$$

위 방법으로 upper and lower bound에 대한 exact solution을 구할 수 있다고 하더라도 매우 impractical하다. 왜냐하면 two integer programming problems를 network의 각 hidden unit에 대해 수행해줘야 하기 때문이다.

따라서 더 loose 하지만 빠르게 compute할 수 있는 bound를 찾는 것이 더 낫다. 이를 아래와 같이 표현할 수 있다.

$$\hat{l} \leq z \leq \hat{u}$$

이는 곧 $Wz + b$ 가 얼마나 클지 혹은 얼마나 작을지에 관한 물음으로 이어진다.

$(Wz + b)_i$ 라는 single entry를 고려해보자. 이는 아래와 같이 풀어쓸 수 있다.

$$(Wz + b)_i = \sum_j w_{ij} z_j + b_i$$

W_{ij} : entry in the i-th row and j-th column of W

z_j 와 b_j : j-th and i-th entries of z and b

z를 우리가 선택할 수 있다면
$W_{ij} < 0$ 인 경우에는 $z_j = \hat{u}_j$
$W_{ij} > 0$ 인 경우에는 $z_j = \hat{l}_j$

따라서 $Wz + b$ 의 bound는 아래와 같이 쓸 수 있다.

$$max\{W, 0\}\hat{l} + min\{W, 0\}\hat{u} + b \leq (Wz + b) \leq max\{W, 0\}\hat{u} + min\{W, 0\}\hat{l} + b$$

- $W < 0$

$$\hat{u} + b \leq (Wz + b) \leq \hat{l} + b$$

- $W > 0$

$$\hat{l} + b \leq (Wz + b) \leq \hat{u} + b$$

bound 자체는 매우 loose 해지지만 계산하기 빨라진다.

위에서 설명한 bound 알고리즘을 통해 bound code를 구현해보면 아래와 같다.

```
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
```

interval bound의 범위를 [0,1]로 잡고 bound_propagation을 해보면 결과는 아래와 같다.

```
epsilon = 0.1 bounds = bound_propagation(model_cnn, ((X -
epsilon).clamp(min=0), (X + epsilon).clamp(max=1))) print("lower bound:
", bounds[-1][0][0].detach().cpu().numpy()) print("upper bound: ",
bounds[-1][1][0].detach().cpu().numpy())
```

```
lower bound: [-2998.6592 -2728.1714 -2460.193 -2534.023 -3073.9976 -2425.
4656 -2874.6404 -2319.738 -2541.7148 -2524.9368] upper bound: [2305.799 2
738.133 2543.169 2636.1572 1995.432 2955.592 2503.6426 2955.3687 2904.122
3 2960.8352]
```

위 bound의 의미는 $\epsilon = 0.1$ 로 perturbation이 들어갔을 때 zero class 에대한 logit의 범위는 $[-2998, 2305]$ 가 됨을 알려준다. 많이 loose 한 범위이지만 $[-10^{100}, 10^{100}]$ 보다는 범위를 많이 좁힐 수 있었다.

다음으로 CNN이 아닌 2-layer, 4-layer model에 대한 bound를 구해보자.

2-layer

```
epsilon = 0.1 bounds = bound_propagation(model_dnn_2, ((X -
epsilon).clamp(min=0), (X + epsilon).clamp(max=1))) print("lower bound:
", bounds[-1][0][0].detach().cpu().numpy()) print("upper bound: ",
bounds[-1][1][0].detach().cpu().numpy())
```

```
lower bound: [-20.794655 -23.554483 -17.501461 -16.71551 -32.42357 -22.67
279 -30.79689 -14.9981 -23.627762 -24.76149 ] upper bound: [18.328127 14.
5928 26.335987 28.887863 18.655684 28.70531 11.502836 32.54162 24.40407 2
5.302904]
```

4-layer

```
epsilon = 0.1 bounds = bound_propagation(model_dnn_4, ((X -
epsilon).clamp(min=0), (X + epsilon).clamp(max=1))) print("lower bound:
", bounds[-1][0][0].detach().cpu().numpy()) print("upper bound: ",
bounds[-1][1][0].detach().cpu().numpy())
```

```
lower bound: [-218.17267 -195.3326 -176.88179 -214.7243 -221.61171 -173.7
9388 -233.49634 -201.39543 -208.68443 -197.08893] upper bound: [183.82648
207.23276 176.97774 205.42944 190.99425 209.2204 186.59421 231.4344 200.6
849 213.99307]
```

이제 위에서 구한 bound를 어떻게 활용할 수 있을지를 살펴보자.

A final integer programming formulation

최종적인 integer programming formulation 을 작성해보면 아래와 같이 쓸 수 있다.

$$
\begin{aligned}
\underset{z_{1,\ldots,d+1}, v_{1,\ldots,d-1}}{\text{minimize}} \quad & (e_y - e_{y_{\text{targ}}})^T z_{d+1} \\
\text{subject to} \quad & z_{i+1} \geq W_i z_i + b_i, \quad i = 1 \ldots, d-1 \\
& z_{i+1} \geq 0, \quad i = 1 \ldots, d-1 \\
& u_i \cdot v_i \geq z_{i+1}, \quad i = 1 \ldots, d-1 \\
& W_i z_i + b_i \geq z_{i+1} + (1 - v_i) l_i, \quad i = 1 \ldots, d-1 \\
& v_i \in \{0,1\}^{|v_i|}, \quad i = 1 \ldots, d-1 \\
& z_1 \leq x + \delta \\
& z_1 \geq x - \delta \\
& z_{d+1} = W_d z_d + b_d.
\end{aligned}
$$

위 integer programming을 cvxpy 를 이용해 해결해볼 예정이다. (cvxpy는 시중의 free solver보다는 속도가 느린 편이라고 한다.)

```
import cvxpy as cp def form_milp(model, c, initial_bounds, bounds):
linear_layers = [(layer, bound) for layer, bound in zip(model,bounds) if
isinstance(layer, nn.Linear)] d = len(linear_layers)-1 # create cvxpy
variables z = ([cp.Variable(layer.in_features) for layer, _ in
linear_layers] + [cp.Variable(linear_layers[-1][0].out_features)]) v =
[cp.Variable(layer.out_features, boolean=True) for layer, _ in
linear_layers[:-1]] # extract relevant matrices W =
[layer.weight.detach().cpu().numpy() for layer,_ in linear_layers] b =
[layer.bias.detach().cpu().numpy() for layer,_ in linear_layers] l =
[l[0].detach().cpu().numpy() for _, (l,_) in linear_layers] u =
[u[0].detach().cpu().numpy() for _, (_,u) in linear_layers] l0 =
initial_bound[0][0].view(-1).detach().cpu().numpy() u0 = initial_bound[1]
[0].view(-1).detach().cpu().numpy() # add ReLU constraints constraints =
[] for i in range(len(linear_layers)-1): constraints += [z[i+1] >= W[i] @
z[i] + b[i], z[i+1] >= 0, cp.multiply(v[i], u[i]) >= z[i+1], W[i] @ z[i]
+ b[i] >= z[i+1] + cp.multiply((1-v[i]), l[i])] # final linear constraint
constraints += [z[d+1] == W[d] @ z[d] + b[d]] # initial bound constraints
constraints += [z[0] >= l0, z[0] <= u0] return cp.Problem(cp.Minimize(c @
z[d+1]), constraints), (z, v)
```

MILP를 빠르게 해결하기 위해서는 더 작은 network를 고려해볼 필요가 있다. 그래서 이번에는 3-layer network를 사용해볼 예정이다. Hidden unit의 size는 50과 20으로 설정할 것이다.

굳이 새로 만들어서 train 하지 않고 이미 만들어져 있는 것을 load 해도 된다.

```
model_small = nn.Sequential(Flatten(), nn.Linear(784,50), nn.ReLU(),
nn.Linear(50,20), nn.ReLU(), nn.Linear(20,10)).to(device)
```

```
# train model and save to disk opt = optim.SGD(model_small.parameters(),
lr=1e-1) for _ in range(10): train_err, train_loss = epoch(train_loader,
model_small, opt) test_err, test_loss = epoch(test_loader, model_small)
print(*("{:.6f}".format(i) for i in (train_err, train_loss, test_err,
test_loss)), sep="\t") torch.save(model_small.state_dict(),
"model_small.pt")
```

```
0.203017 0.663095 0.086800 0.301833 0.078033 0.271283 0.063700 0.219485
0.058133 0.198609 0.050100 0.174730 0.046533 0.159953 0.045300 0.144941
0.039200 0.132439 0.039100 0.128218 0.034117 0.114285 0.037100 0.115507
0.029283 0.100041 0.035600 0.110929 0.026867 0.090279 0.031200 0.101059
0.024233 0.080470 0.030700 0.098896 0.022300 0.073255 0.032100 0.098225
```

```
# load model from disk
model_small.load_state_dict(torch.load("model_small.pt"))
```

testing set의 첫번째 example에 대해 integer program을 만들어보자.

True label은 7이고 0으로 label을 바꾸는 것이 목적이다.

```
initial_bound = ((X[0:1] - epsilon).clamp(min=0), (X[0:1] +
epsilon).clamp(max=1)) bounds = bound_propagation(model_small,
initial_bound) c = np.zeros(10) c[y[0].item()] = 1 c[2] = -1 prob, (z, v)
= form_milp(model_small, c, initial_bound, bounds)
```

마지막으로 Gurobi를 이용해서 integer program을 해결해보자.

```
prob.solve(solver=cp.GUROBI, verbose=True)
```

```
- 4.966118334049208
```

음수의 결과값이 나왔다는 의미는 target class의 class logit을 original class의 class logit 보다 더 크게 만들 수 있는 perturbation을 찾을 수 있다는 의미가 된다.

MILP를 통해 구한 값과 initial value를 model에다가 넣었을 때 마지막 layer 값과 비교를 해보자.

```
print("Last layer values from MILP:", z[3].value)
```

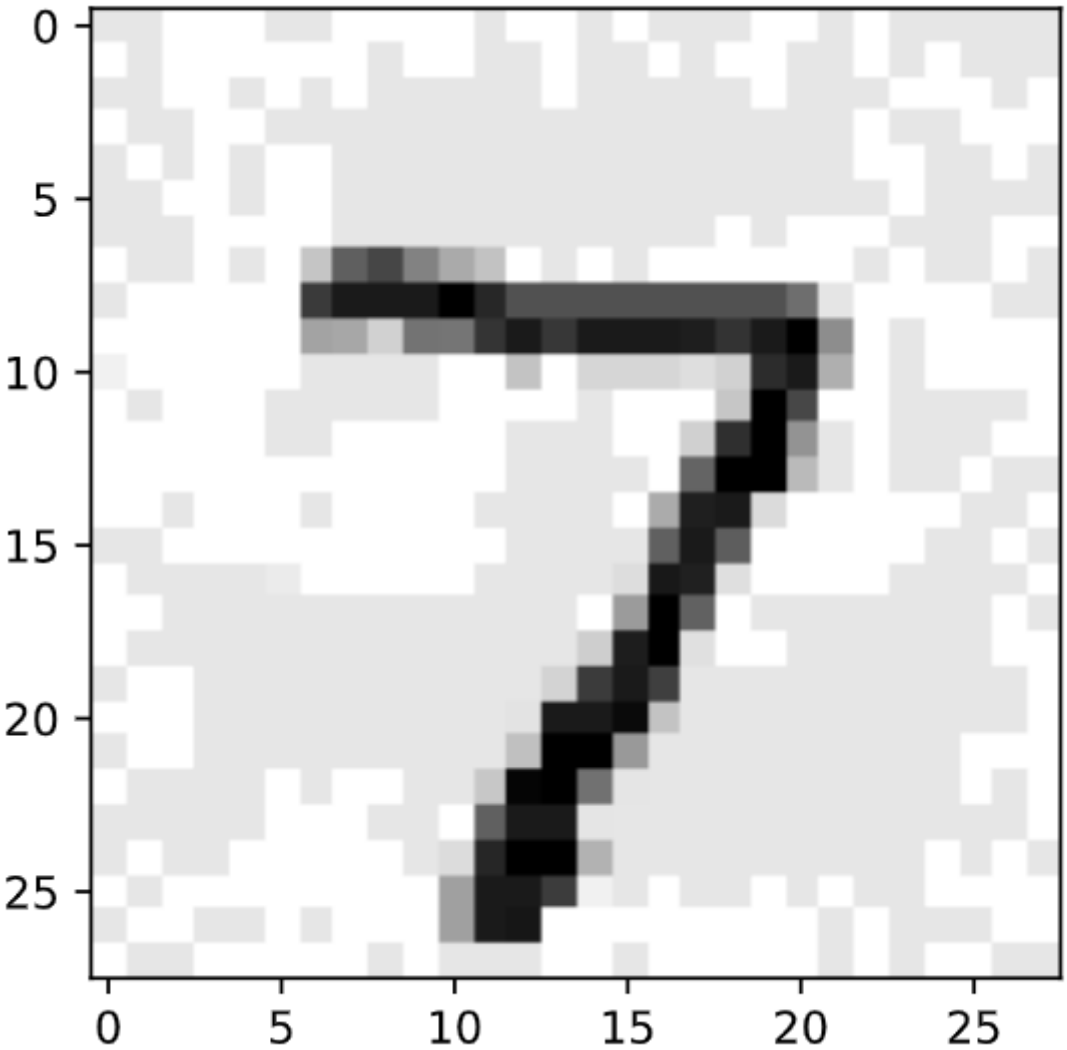
```
Last layer values from MILP: [ 1.0796434 -2.62372054 8.66100952 7.0128640
8 -7.5661212 1.30622329 -9.93690445 3.69489119 7.47907522 -1.05974112]
```

```
print("Last layer from model:",
model_small(torch.tensor(z[0].value).float().view(1,1,28,28).to(device))[0].
```

```
Last layer from model: [ 1.0796432 -2.6237202 8.661009 7.012866 -7.56612
1.3062226 -9.936907 3.6948912 7.4790754 -1.0597415]
```

perturbed image를 확인해보자.

```
plt.imshow(1-z[0].value.reshape(28,28), cmap="gray")
```



Certifying robustness

Adversarial example이 존재하는지 파악하려면 가능한 모든 alternative class label에 대해 targeted attack을 사용하여 integer programming solution을 실행하면 된다.

만약 optimization objectives 중에 하나라도 음수값이 존재하면 Adversarial Example이 존재한다는 의미가 된다. 반대로 음수값이 존재하지 않는다면 classifier가 robust하다고 볼 수 있다.

```
epsilon = 0.05 initial_bound = ((X[0:1] - epsilon).clamp(min=0), (X[0:1]
+ epsilon).clamp(max=1)) bounds = bound_propagation(model_small,
initial_bound) for y_targ in range(10): if y_targ != y[0].item(): c =
np.eye(10)[y[0].item()] - np.eye(10)[y_targ] prob, _ =
form_milp(model_small, c, initial_bound, bounds) print("Targeted attack
{} objective: {}".format(y_targ, prob.solve(solver=cp.GUROBI)))
```

```
Targeted attack 0 objective: 5.810042236946861 Targeted attack 1 objectiv
e: 10.836173372728197 Targeted attack 2 objective: 2.039339255684599 Targ
eted attack 3 objective: 0.018872730723538567 Targeted attack 4 objectiv
e: 13.52836423718531 Targeted attack 5 objective: 5.146072840088203 Targe
ted attack 6 objective: 20.45439065167794 Targeted attack 8 objective: 3.
5428745576535814 Targeted attack 9 objective: 4.274027214275465
```

모든 objective 값이 양수임을 보아 $\epsilon = 0.05$ size의 ℓ_∞ adversarial perturbation이 존재하지 않음을 알 수 있다. 만약 AEs가 존재한다면 구하는 과정 속에서 어떤 class label을 target으로 삼아야 할지도 알 수 있다.

```
epsilon = 0.1 initial_bound = ((X[0:1] - epsilon).clamp(min=0), (X[0:1] +
epsilon).clamp(max=1)) bounds = bound_propagation(model_small,
initial_bound) for y_targ in range(10): if y_targ != y[0].item(): c =
np.eye(10)[y[0].item()] - np.eye(10)[y_targ] prob, _ =
form_milp(model_small, c, initial_bound, bounds) print("Targeted attack
{} objective: {}".format(y_targ, prob.solve(solver=cp.GUROBI))
```

```
Targeted attack 0 objective: -2.545012509042315 Targeted attack 1 objecti
ve: 3.043376725812322 Targeted attack 2 objective: -4.966118334049208 Tar
geted attack 3 objective: -7.309837079755624 Targeted attack 4 objective:
4.741011344284811 Targeted attack 5 objective: -4.870590800081836 Targete
d attack 6 objective: 7.980441149609722 Targeted attack 8 objective: -6.9
536251954142365 Targeted attack 9 objective: -3.314370640372009
```

위 결과를 해석하자면 classifier가 1로 분류할 수 있도록 만들 수 있는 $\epsilon = 0.1$ 크기의
ℓ_∞ adversarial perturbation은 존재하지 않는다는 것이다.

Upper bounding the inner maximization (convex relaxations)

Robustness를 guarantee하고 싶다면 inner maximization problem에 대한 fast upper bounds 값을 아는 것이 중요하다. 예를들어서 targeted attack으로 바꿀 수 있는 class label이 없음을 알려주는 upper bound에 도달하는 경우 공격이 불가능함을 알려주게 된다.

아래의 2가지 방법을 통해 upper bound를 정할 것이다.

1. Convex relaxation (tighter bound, but expensive to compute)
2. Bound propagation (looser bound, but much faster to compute)

Convex relaxations of the verification problem

아래의 Binary integer constraint 로 인해 integer program이 해결하기 어려웠었다.

$$v_i \in \{0,1\}^{|v_i|}$$

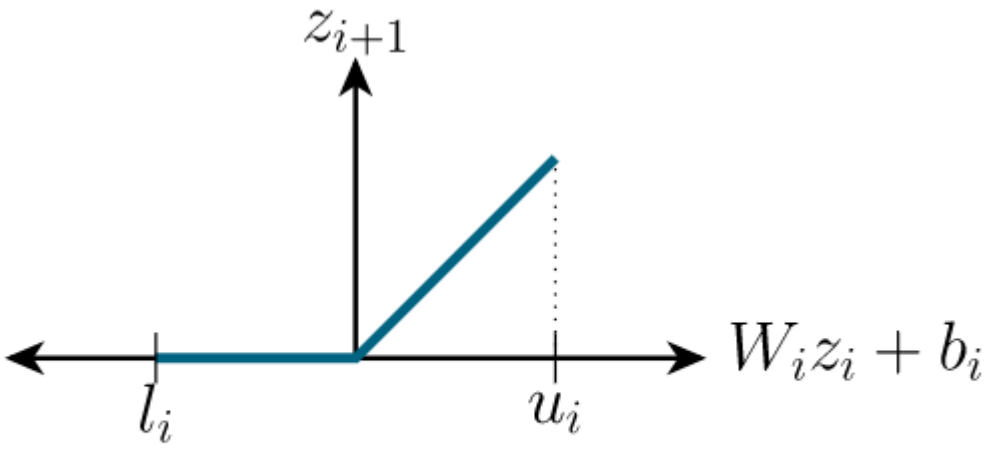
이는 convex set이 아니므로 optimize 하기 어렵다. 이는 유일한 difficult constraint 이기 때문에 이를 제거해주면 우리는 linear program을 얻을 수 있게 된다.

v_i 의 값이 0 또는 1로 이루어져야 한다는 조건을 relaxation을 해주게 된다면 v_i 의 값이 0에서 1사이의 값을 가지도록 바꿔주는 것이다.

$$0 \leq v_i \leq 1$$

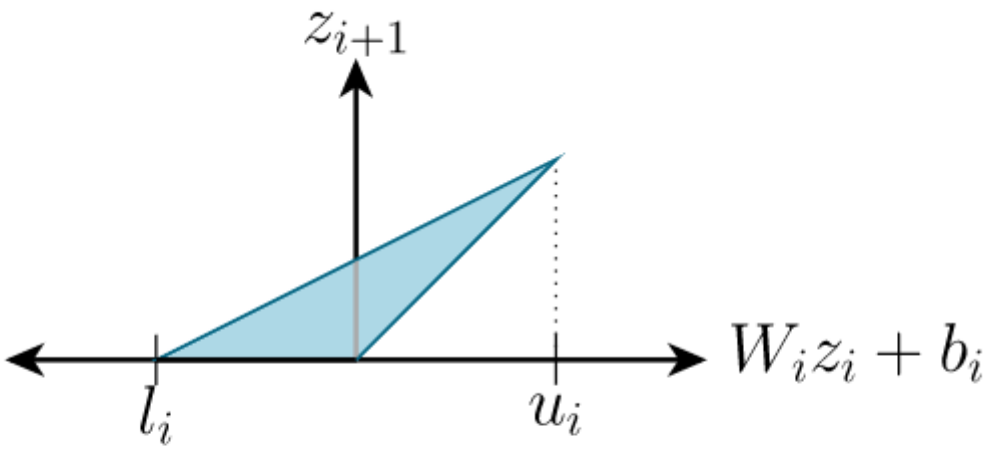
relaxed version에 대해 linear program을 해결하고 objective 값이 여전히 양수라면 이는 attack이 불가능함을 의미한다. 왜냐하면 relaxed set이 original set보다 strictly larger 하기 때문이다. (더 큰 범위를 갖고 있음). 즉, "AE"가 relaxed set에 존재하지 않는다면 original set에도 존재하지 않는다는 뜻으로 볼 수 있다.

Binary integer constraint에 의해 제약조건이 ReLU form을 띄고 있었는데 이를 relaxed version으로 바꾸게 되면 그 의미는 ReLU의 일부만을 사용하겠다는 의미가 된다. 즉, "bounded ReLU" set을 보겠다는 의미가 된다.



Bounded ReLU set.

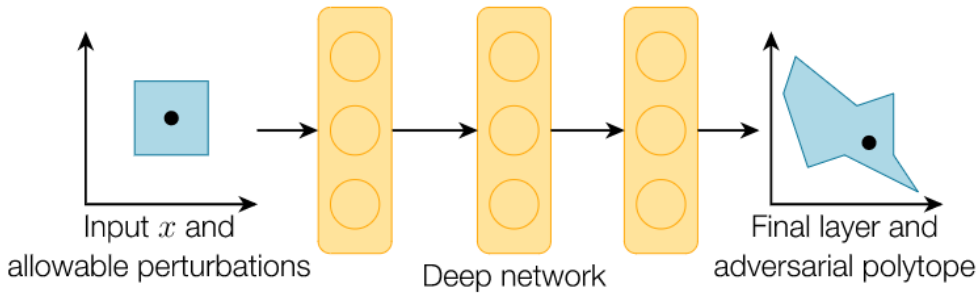
v_i 를 fractional valued로 relax 한다는 것은 bounded ReLU set을 그것의 convex hull 로 relax 한다는 것과 같은 의미가 된다.



Convex hull of the bounded ReLU set.

Relaxation을 통해 얻을 수 있는 것은 AE가 없다는 것을 보증할 수 있는 optimization problem의 objective value가 되는 것이다. 즉, 여전히 relaxation은 negative objective 한 example을 만들 수 있지만 실제로는 그러한 example이 없을 수도 있다는 뜻이다.

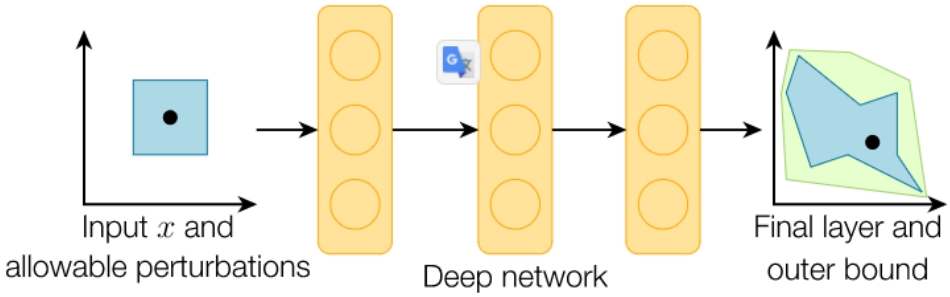
다른 방법으로 살펴보자. ℓ_∞ ball을 deep network에다가 그려보면 마지막 layer (adversarial polytope) 에서는 nasty, non-convex set으로 바뀌게 된다.



Norm-bounded input, and resulting adversarial polytope.

즉, 위 집합에서 target class logit을 최대화하고 true class logit을 최소화하는 방향의 worst-case point를 찾다보면 이는 앞에 MILP를 설명하면서 본 것과 같은 맥락이 된다.

하지만 convex relation set의 경우 adversarial polytope 의 outer bound까지 고려해야한다.



Convex outer bound on the adversarial polytope

이 집합은 최적화하기 더 쉽고(convex 하기 때문에) strictly larger하다. Outer bound의 경우 좀더 loose 해지지만 이는 더 큰 network를 고려했을 때 deep classifier의 robustness를 더 강력하게 보증할 수 있는 전략이 될 수 있다.

실제 convex relaxation이 어떻게 작동하는지 살펴보자.

```
def form_lp(model, c, initial_bounds, bounds): linear_layers = [(layer,
bound) for layer, bound in zip(model,bounds) if isinstance(layer,
nn.Linear)] d = len(linear_layers)-1 # create cvxpy variables z =
([cp.Variable(layer.in_features) for layer, _ in linear_layers] +
[cp.Variable(linear_layers[-1][0].out_features)]) v =
[cp.Variable(layer.out_features) for layer, _ in linear_layers[:-1]] #
extract relevant matrices W = [layer.weight.detach().cpu().numpy() for
layer,_ in linear_layers] b = [layer.bias.detach().cpu().numpy() for
layer,_ in linear_layers] l = [l[0].detach().cpu().numpy() for _, (l,_)
in linear_layers] u = [u[0].detach().cpu().numpy() for _, (_,u) in
linear_layers] l0 = initial_bound[0][0].view(-1).detach().cpu().numpy()
u0 = initial_bound[1][0].view(-1).detach().cpu().numpy() # add ReLU
constraints constraints = [] for i in range(len(linear_layers)-1):
constraints += [z[i+1] >= W[i] @ z[i] + b[i], z[i+1] >= 0,
cp.multiply(v[i], u[i]) >= z[i+1], W[i] @ z[i] + b[i] >= z[i+1] +
cp.multiply((1-v[i]), l[i]), v[i] >= 0, v[i] <= 1] # final linear
constraint constraints += [z[d+1] == W[d] @ z[d] + b[d]] # initial bound
constraints constraints += [z[0] >= l0, z[0] <= u0] return
cp.Problem(cp.Minimize(c @ z[d+1]), constraints), (z, v)
```

Integer programming solution에서 사용했던 example을 이용해서 2개의 bound를 비교해보자.

```
initial_bound = ((X[0:1] - epsilon).clamp(min=0), (X[0:1] +
epsilon).clamp(max=1)) bounds = bound_propagation(model_small,
initial_bound) c = np.eye(10)[y[0].item()] - np.eye(10)[2] prob, (z, v) =
form_lp(model_small, c, initial_bound, bounds) print("Objective value:",
prob.solve(solver=cp.GUROBI)) print("Last layer from relaxation:",
z[3].value)
```

```
Objective value: -26.119774634225454 Last layer from relaxation: [-0.5423
9629 -3.68197597 21.97748102 6.17979398 -6.16385997 -0.27719886 -2.256319
31 -4.14229362 9.61796699 -5.31716916]
```

Relaxation 의 경우 class 2의 logit을 truel class인 7 logit 보다 26.1198 더 크게 만들 수 있음을 보고 있다.

이번에는 perturbation을 실제 network에 넣어보면 activation 값이 relaxation과 같은 결과로 나오지 않는다는 것을 볼 수 있다.

```
print("Last layer from model:",
model_small(torch.tensor(z[0].value).float().view(1,1,28,28).to(device))[0].
```

```
Last layer from model: [ 1.1390655 -2.760746 8.2507715 5.9955297 -5.96594
57 0.492265 -8.71024 3.8318188 6.594626 -1.3704427]
```

아주 작은 perturbation 을 주었을 때의 결과도 확인해보자. (convex relaxation)

$\epsilon = 0.02$

```
epsilon = 0.02 initial_bound = ((X[0:1] - epsilon).clamp(min=0), (X[0:1]
+ epsilon).clamp(max=1)) bounds = bound_propagation(model_small,
initial_bound) for y_targ in range(10): if y_targ != y[0].item(): c =
np.eye(10)[y[0].item()] - np.eye(10)[y_targ] prob, _ =
form_lp(model_small, c, initial_bound, bounds) print("Targeted attack {}
objective: {}".format(y_targ, prob.solve(solver=cp.GUROBI)))
```

```
Targeted attack 0 objective: 8.679119731697458 Targeted attack 1 objectiv
e: 12.233313266959202 Targeted attack 2 objective: 4.120827654631229 Targ
eted attack 3 objective: 1.857739424985871 Targeted attack 4 objective: 1
6.236514322580383 Targeted attack 5 objective: 7.964241745873003 Targeted
attack 6 objective: 23.459304334257965 Targeted attack 8 objective: 6.103
14963105955 Targeted attack 9 objective: 5.581247281946292
```

위 결과를 통해 $\epsilon = 0.02$ 로는 classifier를 fool 할 수 있는 ℓ_∞ perturbation은 없다는 것을 알 수 있다.

Integer programming approach로는 풀 수 없었던 4-layer DNN에서의 결과를 살펴보자. (마찬가지로 convex relaxation)

```
epsilon = 0.01 initial_bound = ((X[0:1] - epsilon).clamp(min=0), (X[0:1]
+ epsilon).clamp(max=1)) bounds = bound_propagation(model_dnn_4,
initial_bound) for y_targ in range(10): if y_targ != y[0].item(): c =
np.eye(10)[y[0].item()] - np.eye(10)[y_targ] prob, _ =
form_lp(model_dnn_4, c, initial_bound, bounds) print("Targeted attack {}
objective: {}".format(y_targ, prob.solve(solver=cp.GUROBI)))
```

```
Targeted attack 0 objective: 5.480101103982182 Targeted attack 1 objectiv
e: 2.196709825912893 Targeted attack 2 objective: 0.4646296745679468 Targ
eted attack 3 objective: -2.264958127925656 Targeted attack 4 objective:
9.8073503196978 Targeted attack 5 objective: 1.4951834763453595 Targeted
attack 6 objective: 12.835283301376197 Targeted attack 8 objective: -0.39
325769328255866 Targeted attack 9 objective: -1.9890586254866953
```

Faster solutions of the convex relaxation

위 예제는 single MNIST example을 사용하였기 때문에 몇초 밖에 걸리지 않았지만 만약 모든 MNIST example을 사용하고 더 큰 model을 이용했어야 했다면 feasible 하지 않았을 것이다.

Verification procedure의 복잡성과 Bound의 tightness 사이의 적절한 trade-off를 찾는 것은 아직 open research question이다. 즉, computationally efficient 하면서도 large-scale network에 대해 tight bound를 제공하는 solution은 아직 발견되지 않았다.

Interval-propagation-based bounds

이 방법은 convex relaxation 보다 더 loose한 bound를 제공하고 convex relaxation 처럼 "fake" adversarial example 조차 제공하지는 않지만 매우 효율적이라는 장점이 있다.

다시 앞에서 interval bounds를 propagating 할 때를 생각해보자. $\hat{l} \leq z \leq \hat{u}$ 에 의해 우리는 아래 식을 유도해낼 수 있었다.

$$l \leq Wx + b \leq u$$

where

$$l = \max\{W, 0\}\hat{l} + \min\{W, 0\}\hat{u} + b$$
$$u = \max\{W, 0\}\hat{u} + \min\{W, 0\}\hat{l} + b.$$

이 bounds를 마지막 layer까지 propagating 하는 것은 매우 비효율적이다. (어차피 마지막 layer에서는 특정 linear function을 minimize 하려는 것이 목적임을 알고 있기 때문에) 따라서 마지막 layer 바로 전 layer 까지만 propagate 하고 마지막 layer에서는 minimization problem을 분석적으로 해결하면 된다.

좀 더 자세히 예기해보면 우리는 앞서 d-layer network에서는 마지막 layer 의 linear function $c^T z_{d+1}$ 을 minimize 하는 것이 목표임을 살펴봤다. 만약 마지막 layer의 이전 layer에서 interval bound $\hat{l} \leq z_d \leq \hat{u}$ 를 가진다면 우리는 optimization problem을 아래와 같이 간단히 작성할 수 있다.

$$\begin{aligned} \underset{z_d, z_{d+1}}{\text{minimize}} \;\; & c^T z_{d+1} \\ \text{subject to} \;\; & z_{d+1} = W_d z_d + b_d \\ & \hat{l} \leq z_d \leq \hat{u}. \end{aligned}$$

이 문제는 앞에서 bounds를 구할 때 했던 과정과 같은 전략을 이용하면 간단한 analytical solution을 가짐을 알 수 있다.

첫번째 제약조건을 이용해 z_{d+1} 을 없애면 아래와 같이 쓸 수 있다.

$$\begin{aligned} \underset{z_d}{\text{minimize}} \;\; & c^T (W_d z_d + b_d) \equiv (W_d^T c)^T z_d + c^T b_d \\ \text{subject to} \;\; & \hat{l} \leq z_d \leq \hat{u}. \end{aligned}$$

이는 곧 앞에서 했던 것과 같은 특정 bound constraint 에 대해서 linear function을 최소화하는 것과 같은 문제이다.

따라서 analytic solution은 단지 만약 $(W_d^T c) > 0$ 일때는 $(z_d)_j = \hat{l}_j$ 을 선택하고 그게 아니면 $(z_d)_j = \hat{u}_j$ 를 선택하는 것이다. 즉, 아래와 같은 optimal objective value를 갖게 된다.

$$\max\{c^T W_d c, 0\}\hat{l} + \min\{c^T W_d, 0\}\hat{u} + c^T b_d$$

코드로 구현해보자.

```
def interval_based_bound(model, c, bounds): # 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.t() + cW.clamp(max=0) @ u.t() +
cb[:,None]).t()
```

이를 앞에 convex relaxation 에서 했던것과 동일한 조건으로 $\epsilon = 0.02$ 에서의 bound를 살펴보자.

```
epsilon = 0.02 initial_bound = ((X[0:1] - epsilon).clamp(min=0), (X[0:1]
+ epsilon).clamp(max=1)) bounds = bound_propagation(model_small,
initial_bound) C = -torch.eye(10).to(device) C[y[0].item(),:] += 1
print(interval_based_bound(model_small, C,
bounds).detach().cpu().numpy())
```

```
[[ -7.426559 -2.0292485 -8.31501 -15.399408 2.3813596 -10.803006 5.575859
0. -13.850781 -11.277755 ]]
```

위 결과는 각 target class에 대한 optimization objective의 lower bounds이다.

Target class가 7인 경우는 true class가 7이므로 $c = 0$ 이 되어 objective 값이 0으로 나온다.

Convex relaxation 에 비해 더 loose 한 bound를 찾아준 것으로 예측된다.

이 방법의 장점은 fast compute 가 가능하다는 것이고 PyTorch로 모두 구현이 되어있기 때문에 실제 robust criterion 하에 network를 훈련시킬 때 backpropagate를 해줄 수 있다.

Summary

여기서 다룬 모든 내용들은 기본적으로 inner optimization problem을 최적화하는 방법 (혹은 last level logit의 linear function을 최적화하는 것; 같은 말임)을 다루고 있음을 알아야 한다.

몇몇 의 경우 실제 attack perturbation을 제안하지만 몇몇의 경우 단순히 optimization objective 의 bound만을 제공한다(이는 robust classifier인지를 검증할 때 유용하다(훈련할 때도 유용)).

최적화 문제는 단순히 Adversarial example을 찾는 작업 뿐만 아니라 Adversarial robustness를 다룰 때 중요하게 봐야 한다.