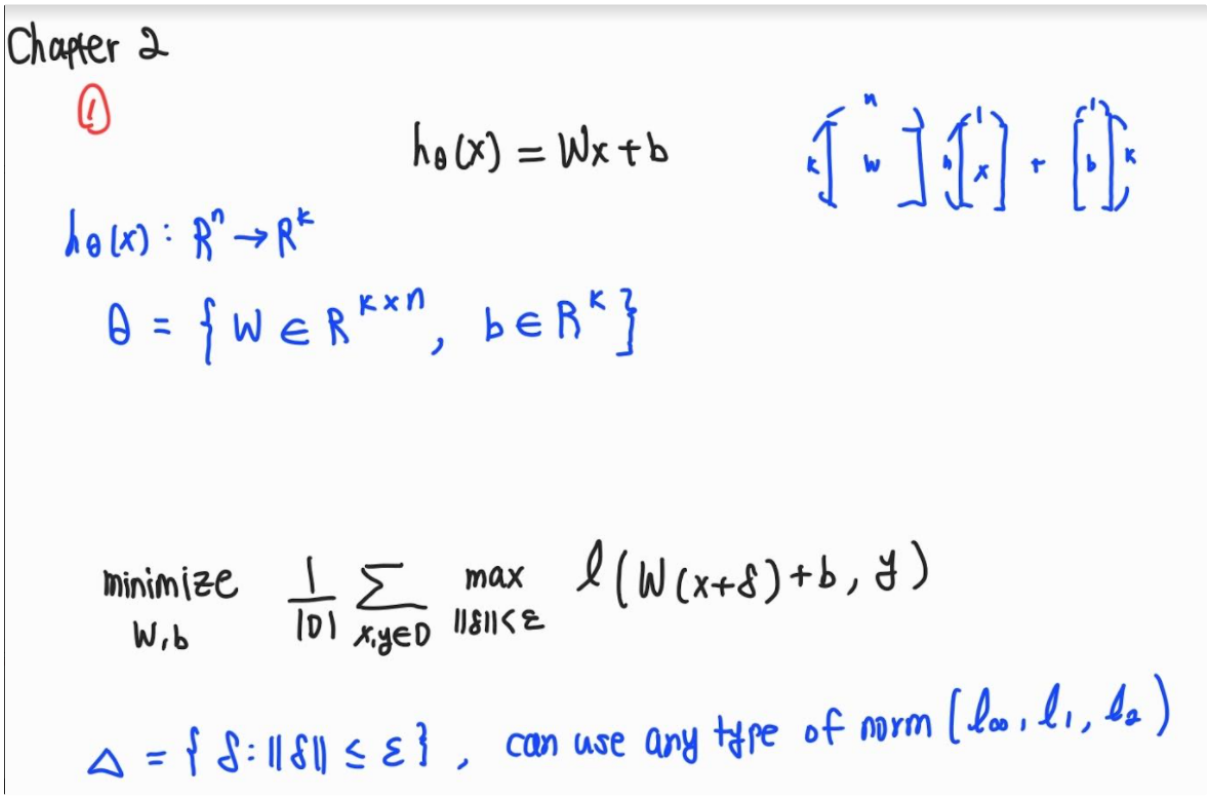
Chapter 2 - Linear models

Chapter 2 Linear models

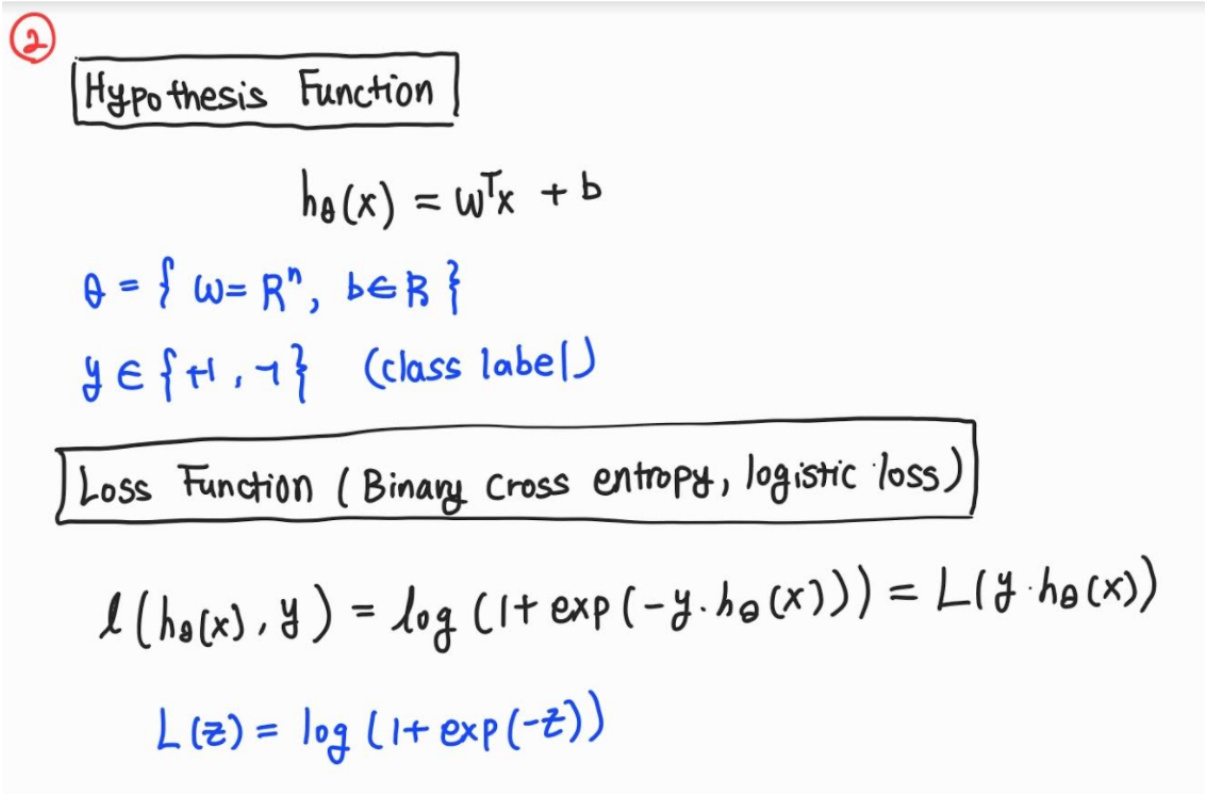


Binary optimization에 대해서는 inner maximization problem을 정확히 해결할 수 있고 multi-class classification에 대해서는 relatively tight upper bound를 구할 수 있다.

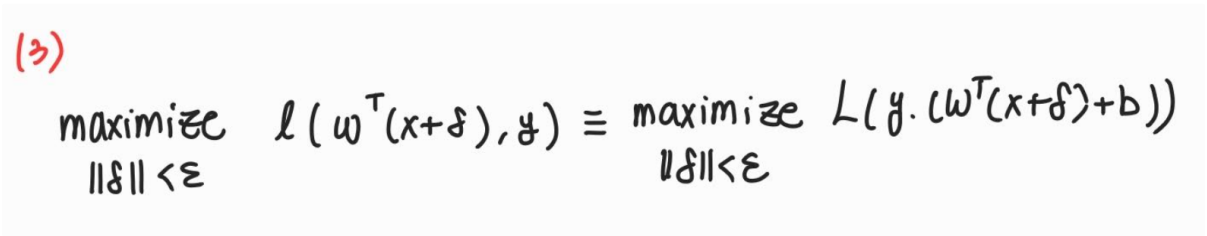
Minimization problem에 대해서는 convex하고 $x + \delta$ 에 대해 maximizing을 하고 나서도 convex 하므로 robust training procedure의 결과는 위 문제를 optimal 하게 해결할 수 있다 (적어도 binary classification에 대해서는).

반면 Deep network 의 경우에는 inner maximization problem과 outer minimization problem 모두 global 하게 해결될 수 없다. (inner problem이 exact solution을 가진다고 하더라도 network 자체의 non-convexity 때문에 outer minimization에서 optimal 한 solution을 구하기 어렵다.)

Binary classification

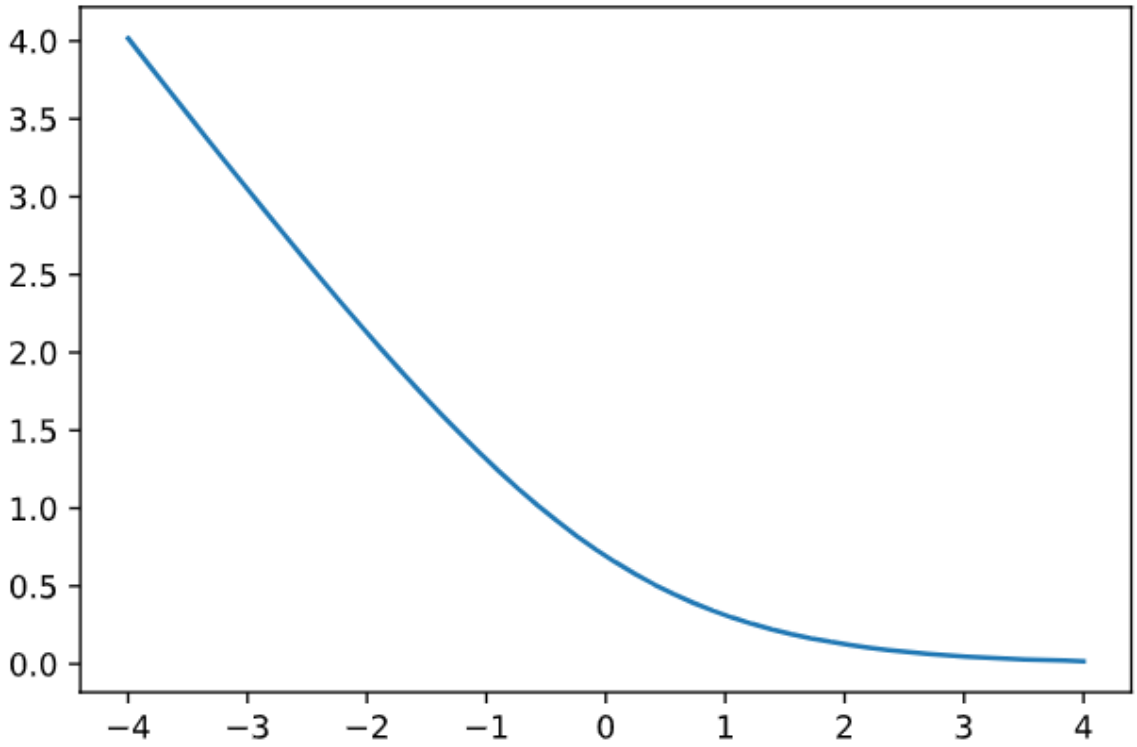


Solving the inner maximization problem



앞서 정의한 loss function을 plot 해보면

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



Plot된 결과를 보면 알 수 있듯이 해당 loss function은 (단조)감소 함수이므로 Loss function을 최대화 하기 위해서는 function의 input 값을 최소화 해주는 것과 같은 작업이다. 이를 아래와 같이 표현할 수 있다.

(4)

※ Monotonic Decreasing (단조감소)

임의의 $x, y \in R$ 에 대하여 $x \le y$ 이면 $f(x) \ge f(y)$ 인 경우

f를 감소함수라고 하고 f가 단조 감소한다고 함.

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

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

따라서 결론적으로 풀어야 할 부분은

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

(6) .

$y=+1$ 이고 l_∞ norm constraint ($\|\delta\|_\infty < \varepsilon$) 를 적용한다고 가정해보자.

$\|\delta\|_\infty < \varepsilon$ 의 의미는 δ의 각 원소의 절댓값이 ε 보다 작거나 같아야 한다는 의미가 된다.

따라서 위 예시에서 최소화를 해주기 위해서는.

(1) $\delta_i = -\varepsilon$, $w_i \geq 0$

(2) $\delta_i = \varepsilon$, $w_i \leq 0$

과 같이 설정해 줄 수 있다.

$y=-1$ 인 상황에서는 반대로 설정해주면 된다.

이는 곧 l_∞ norm constraint를 사용하는 상황에서는 optimal solution을 아래와 같이 정의할 수 있다.

$$\delta^* = -y\varepsilon \cdot \text{sign}(w)$$

sign(w) 방향으로 ε 만큼 perturbation을 가한다.

δ^* 를 적용했을 때 $y \cdot w^T \delta^*$ 를 아래와 같이 풀어쓸 수 있다.

(7)

$$y \cdot w^T \delta^* = y \cdot \sum_{i=1} -y\varepsilon \cdot \mathrm{sign}(w_i) w_i$$

$$= -y^2 \varepsilon \sum_i |w_i| = -\varepsilon \|w\|_1$$

따라서 inner maximization problem은 아래와 같이 다시 풀어 쓸 수 있다.

(8)

$$\max_{\|\delta\|_\infty < \varepsilon} L(y \cdot (w^T(x+\delta)+b)) = L(y \cdot (w^T x + b) - \varepsilon \|w\|_1)$$

결론적으로는 맨 앞에서 살펴본 robust min-max problem을 실제로 min-max problem으로 접근하는 것이 아니라 아래 식과 같이 순수 minimization problem으로 접근할 수 있다.

(9)

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

w, b에 대해서는 convex 하므로 optimal 한 solution을 구할 수 있는 문제가 된다. 이 optimization problem을 좀 더 general 하게 작성해보자면

(10)

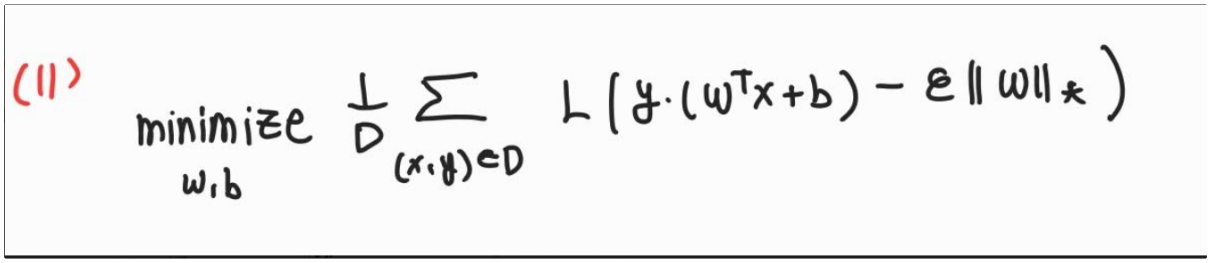
$$\min_{\|\delta\| < \varepsilon} \ y \cdot w^T \delta = -\varepsilon \|w\|_*$$

$\|\cdot\|_*$: dual norm of our original norm bound on θ

$\|\cdot\|_p$ and $\|\cdot\|_q$ are dual norms for $\frac{1}{p} + \frac{1}{q} = 1$

이로써 어떤 norm constraint를 쓰는지에 관계없이 single minimization problem으로 robust optimization problem을 해결할 수 있다.

최종 robust optimization problem을 적어보면 아래와 같다.



이는 곧 point가 decision boundary와 멀다면 parameter norm을 penalize 하지 않고 가깝다면 penalize를 한다는 의미가 되기도 한다.

Illustration of binary classification setting

실제 Linear classifier가 얼마나 AE에 강한지 알아봅시다 (스포일러 : regularize를 하지 않는 한 잘 작동하지 않는다.)

MNIST dataset을 기반으로 0과 1을 구분하는 binary classification으로 바꿔 풀어봅시다.

PyTorch 라이브러리를 사용하여 data를 load한 후 gradient descent를 이용하년 간단한 linear classifier를 build 해봅시다.

```
import torch import torch.nn as nn import torch.optim as optim # do a
single pass over the data 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 = nn.BCEWithLogitsLoss()(yp, y.float()) 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)
```

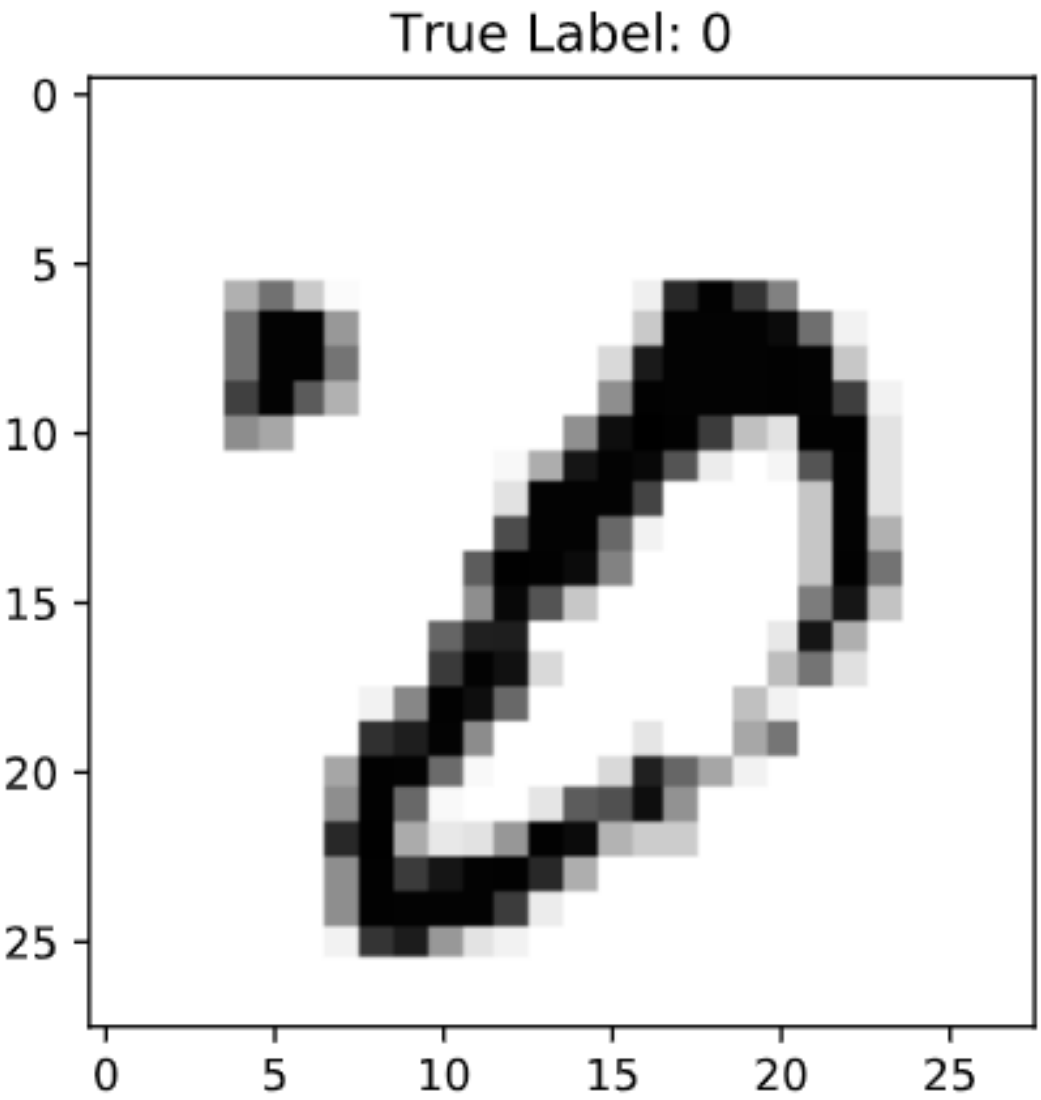
10 epoch로 classifier를 훈련시켜보자. (MNIST 0/1 이진 분류 문제는 매우 간단한 문제이므로 one epoch 만으로도 거의 final test error로 수렴하게 된다. 최종적으로는 0.0004의 error에 도달하게 된다.)

```
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")
```

```
Train Err Train Loss Test Err Test Loss 0.007501 0.015405 0.000946 0.0032
78 0.001342 0.005392 0.000946 0.002892 0.001342 0.004438 0.000473 0.00256
0 0.001105 0.003788 0.000946 0.002495 0.000947 0.003478 0.000946 0.002297
0.000947 0.003251 0.000946 0.002161 0.000711 0.002940 0.000473 0.002159
0.000790 0.002793 0.000946 0.002109 0.000711 0.002650 0.000946 0.002107
0.000790 0.002529 0.000946 0.001997
```

classifier가 오분류를 일으킨 하나의 예시를 살펴보자.

```
X_test =
(test_loader.dataset.test_data.float()/255).view(len(test_loader.dataset),-1
y_test = test_loader.dataset.test_labels 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].item()))
```



앞서 optimal perturbation은 아래와 같이 쓸 수 있음을 살펴보았다.

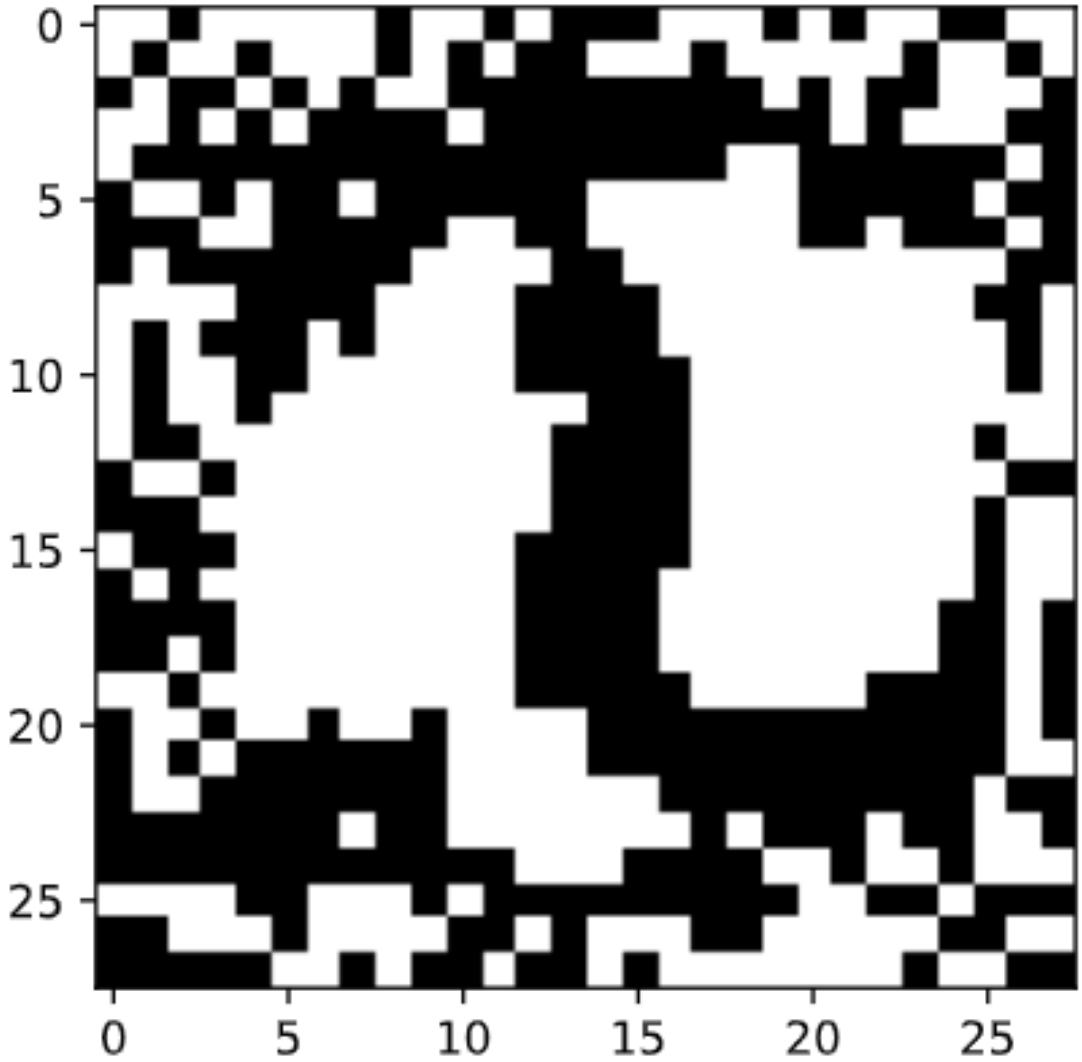
$\delta^* = -y\epsilon \cdot sign(w)$

즉, x에 의존하지 않음을 알 수 있다. 이는 곧 모든 example에 대해 동일한 perturbation이 들어간다는 의미가 된다. 이 때 우리는 $x + \delta$ 의 범위가 $[0, 1]$ 가 되도록 유의해야한다.

하지만 이번 예시에서는 이를 고려하지 않고 진행할 예정.

실제 perturbation이 어떻게 들어갔는지 살펴보자.

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



위 그림을 살펴보면 숫자 1의 이미지처럼 가운데에 검은색 vertical 선이 그어졌음을 볼 수 있다. 즉, 검은색 픽셀로 이루어진 부분이 classifier가 해당 그림을 1로 분류하게끔 유도했음을 알 수 있다.

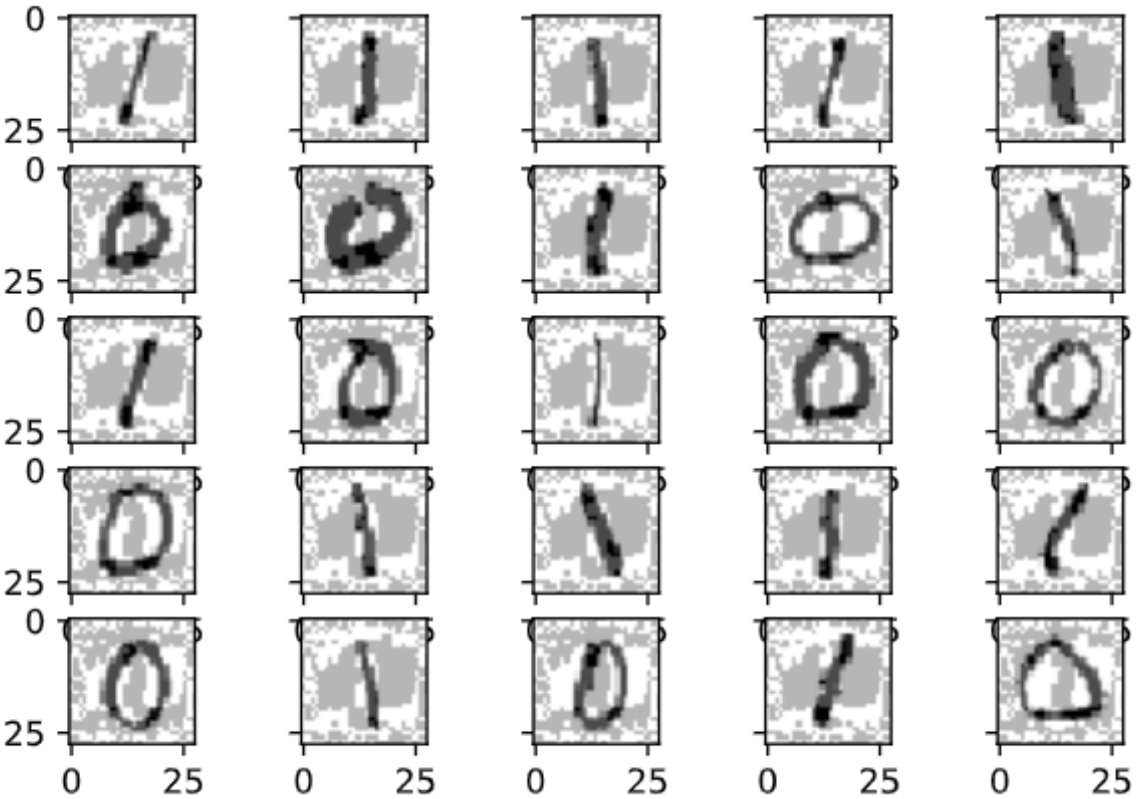
다음으로 (optimal) adversarial attack이 test set의 image에 적용되었을 때 test accuracy가 어떻게 되는지 살펴보자.

```
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 =
nn.BCEWithLogitsLoss()(yp, y.float()) 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,:,:]))
```

```
(0.8458628841607565, 3.4517438034075654)
```

l_∞ ball로 $\epsilon = 0.2$ 만큼 perturbation을 줬을 때 84.5%의 error 를 갖게된다.

```
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") ax
```



Training robust linear models

Exact robust optimization을 위해 (solving the equivalent of the min-max problem) 단순히 l_1 norm을 이용하면 됨을 알 수 있다. 이를 standard binary cross entropy loss에 적용하게 된다면 (이 때 label은 -1/+1 이 아니라 0/1 을 사용) 약간의 핸들링이 필요하지만 training procedure은 매우 simple하다.

단순히 prediction에서 $\epsilon(2y-1)||w||_1$을 빼면 된다. ($2y-1$ 은 0/1을 -1/1로 바꿔주는 역할)

```
# do a single pass over the data 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 = nn.BCEWithLogitsLoss()(yp,
y.float()) 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")
```

```
Rob. Train Err  Rob. Train Loss Rob. Test Err   Rob. Test Loss
0.147414        0.376791        0.073759        0.228654
0.073352        0.223381        0.053901        0.176481
0.062929        0.197301        0.043026        0.154818
0.057008        0.183879        0.038298        0.139773
0.052981        0.174964        0.040662        0.143639
0.050059        0.167973        0.037352        0.132365
0.048164        0.162836        0.032624        0.119755
0.046190        0.158340        0.033570        0.123211
0.044769        0.154719        0.029787        0.118066
0.043979        0.152048        0.027423        0.118974
0.041058        0.149381        0.026478        0.110074
0.040268        0.147034        0.027423        0.114998
0.039874        0.145070        0.026950        0.109395
0.038452        0.143232        0.026950        0.109015
0.037663        0.141919        0.027896        0.113093
0.036715        0.140546        0.026478        0.103066
0.036321        0.139162        0.026478        0.107028
0.035610        0.138088        0.025059        0.104717
0.035215        0.137290        0.025059        0.104803
0.034741        0.136175        0.025059        0.106629
```

위 수치들은 robust (i.e., worst case adversaial) errors와 losses 를 나타낸다. 즉, robust optimization problem을 통해 training을 함으로써 $\epsilon = 0.2$ 로 들어갔을 때 test set에서 2.5% 이상의 error로 이어지는 adversarial attack 이 될 수 없음을 보이고 있다. 이전에 살펴본 85%의 error와 비교하면 상당히 개선되었음을 알 수 있다.

그렇다면 non-adversarial training set에 대해서는 얼마나 잘 작동할까?

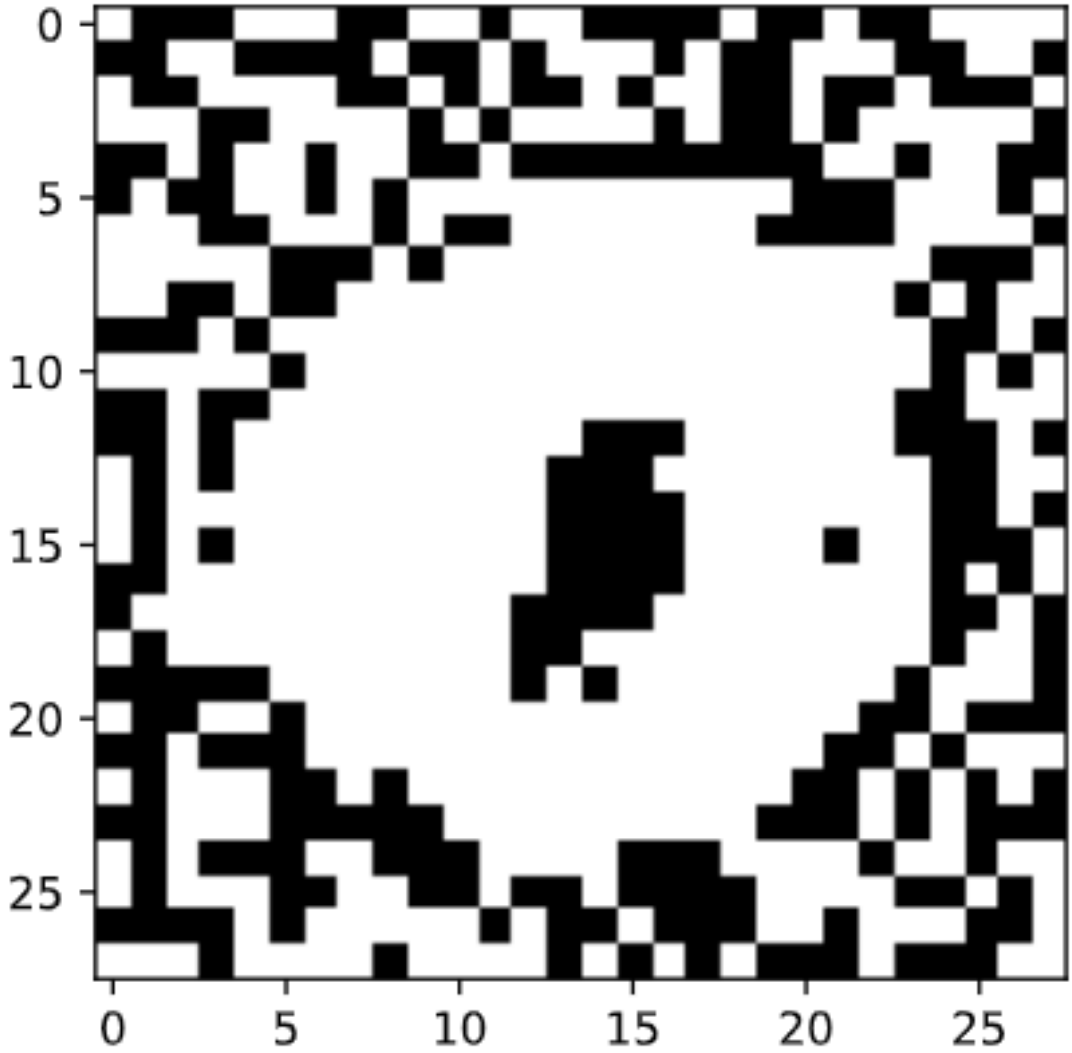
```
train_err, train_loss = epoch(train_loader, model) test_err, test_loss =
epoch(test_loader, model) print("Train Err", "Train Loss", "Test Err",
"Test Loss", sep="\t") print(*("{:.6f}".format(i) for i in (train_err,
train_loss, test_err, test_loss)), sep="\t")
```

Train Err	Train Loss	Test Err	Test Loss
0.006080	0.015129	0.003783	0.008186

Test set에서 0.3%의 error가 발생했다. 잘 되었다고 볼 수는 있지만 앞서 살펴본 standard training 만큼 좋진 않다.

최종적으로 이 robust model의 optimal perturbation을 살펴보면

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



이전에 본 결과보다 더 zero에 가까운 형태가 나온다.

따라서 우리는 robust training은 "adversarial directions"로 이어질 수 있다는 (현 시점에서는 약한)증거를 갖고 있다. "Random noise"를 추가함으로써 classifier를 fooling하는 것이 아니라 image를 actual new image의 방향으로 이동시켜야 한다. 이러한 방법은 epsilon 크기만큼 이동을 하더라도 classifier를 fooling하는데에는 그다지 성공적이지 못하다.)