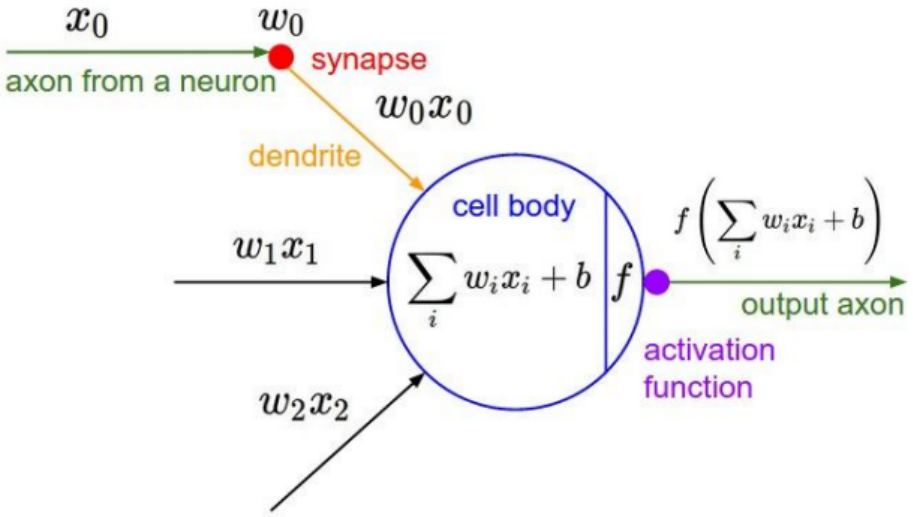
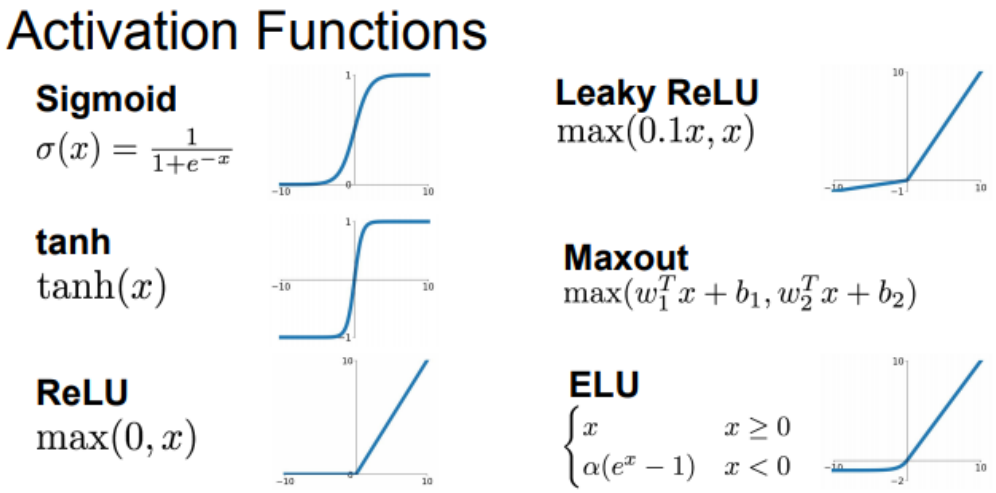
# Activation Functions

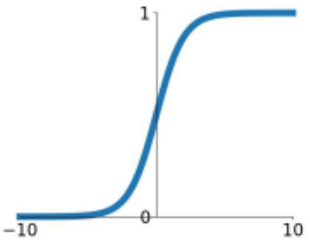
input이 들어오면 가중치와 곱해지고, 비선형 함수인 활성함수를 거쳐 해당 데이터의 활성화여부를 결정해준다.



# Sigmoid

: 각 입력을 받아서 그 입력을 [0, 1] 사이의 값이 되도록 해줌. input의 값이 크면 output이 1에 가깝고, 값이 작으면 0에 가까움. 0 근처 구간을 보면 선형함수 같아 보이는 linear 구간 존재. 뉴런의 firing rate를 포화시키는 것으로 해석할 수 있음.




**Sigmoid**

$$\sigma(x) = 1/(1 + e^{-x})$$

문제점 1

- 음/양의 큰값에서 Saturation(기울기가 0에 가까워지는 현상)되는 것이 gradient를 없앤다.

- x가 0에 가까운건 잘 동작함

문제점 2

- 출력이 zero centered하지 않음.

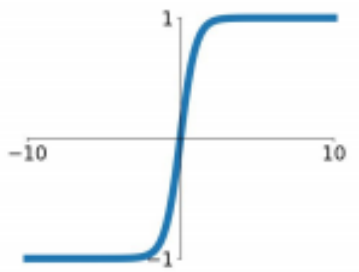
- 만약 input이 항상 양수일 때, w의 gradient와 부호가 항상 같게 됨. 이는 W로 하여금 모두 양의 방향이나, 모두 음의 방향으로밖에 업데이트가 되지 못하게 하기 때문에, zig zag path를 따르게 되어 비효율적이다.

문제점 3

- exp()로 인해 계산 비용이 큼.

## tanh

: sigmoid와 유사하나, 범위가 [-1, 1].



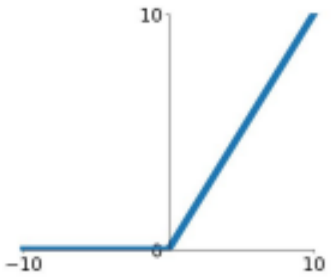

$$\tanh(x)$$

zero-centered 문제는 해결되었으나, saturation 문제는 여전히 존재하므로 gradient가 죽는다.

## ReLU(가장 많이 사용)

- x가 양수이면 saturation 되지 않음.(입력의 절반이 saturation되지 않음)

계산 효율 좋음(sigmoid나 tanh보다 수렴 속도가 약 6배 빠름) 생물학적 타당성 큼.




$$\max(0, x)$$

문제점

- zero- centered가 아님.

- 음의 영역에서는 saturation (x=0에서도 gradient 0)

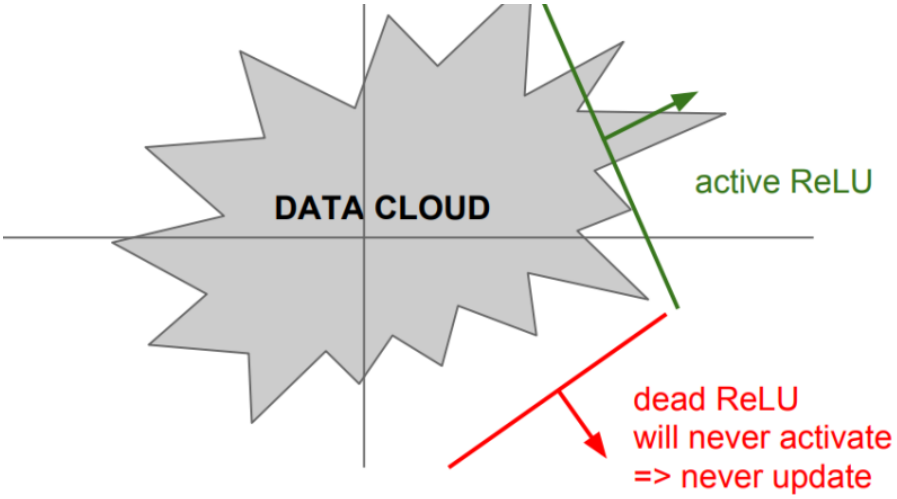
- gradient의 절반을 죽임 => dead ReLU

## Dead ReLU

: activate가 일어나지 않고 update되지 않음.

## Active ReLU

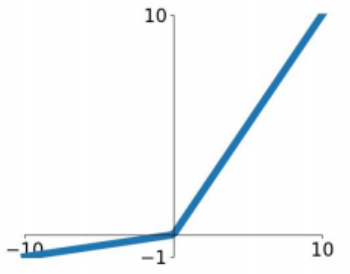
: 일부는 active되고 일부는 active하지 않음.




- 초기화를 잘못해서 가중치 평면이 data cloud에서 멀리 떨어진 경우

- Learning rate가 지나치게 높은 경우, 가중치가 날뛰게 되며 ReLU가 데이터의 manifold를 벗어나게 됨

## Leaky ReLU




**Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

- 계산 효율적 => sigmoid나 Tanh보다 빨리 수렴.

- 음의 영역에서도 이제 saturation 되지 않음

- dead ReLU 없음

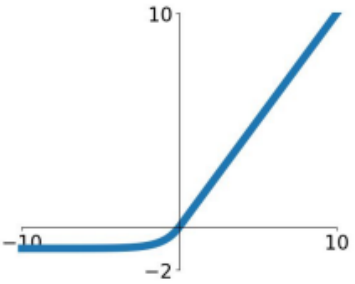
## PReLU

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

- Leaky ReLU와 유사(negative space에 기울기 존재)하지만 기울기 alpha (파라미터)로 결정됨(backprop)으로 학습시키는 파라미터로 만듦.

## ELU




$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$

- zero-mean에 가까운 출력값

- 음에서 saturation. 하지만 saturation이 노이즈에 강인하다고 생각

## Maxout Neuron

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

- 기본형식을 정의하지 않음.

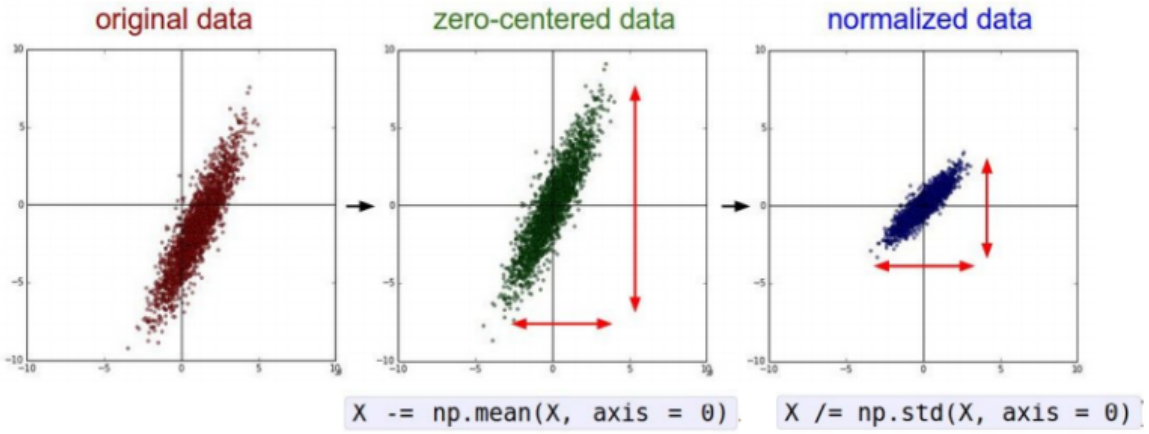
- 두개의 선형함수 중 큰 값을 선택 -> ReLU와 leaky RELU의 일반화 버전

- 선형이기에 saturation 안 되고 gradient 죽지 않음.

문제점

- W1, W2 때문에 파라미터 수 두배됨

## Data Preprocessing




- zero-mean으로 만들고 표준편차로 normalize(모든 차원이 동일한 범위 안에 있게 해줘서 전부 동등한 기여를 하게 함).

- 모든 입력 값이 positive라면 최적의 weight update를 할 수 없는 문제가 발생하기에 입력값에 zero-mean 값을 빼서 zero-centered가 되도록 해줌.

- 이미지의 경우 전처리로 zero-centering만 하고 normalization하지 않음(이미지는 각 차원 간에 스케일이 어느 정도 맞춰져 있기 때문).

## Weight Initialization

모든 가중치를 0으로 설정한다면

- 모든 뉴런이 같은 일을 함.

- 모든 가중치가 똑같은 값으로 업데이트됨

- 모든 가중치를 동일하게 초기화시키면 symmetry breaking이 일어날 수 없음(가중치를 랜덤하게 초기화 시켜 symmetry breaking을 함).

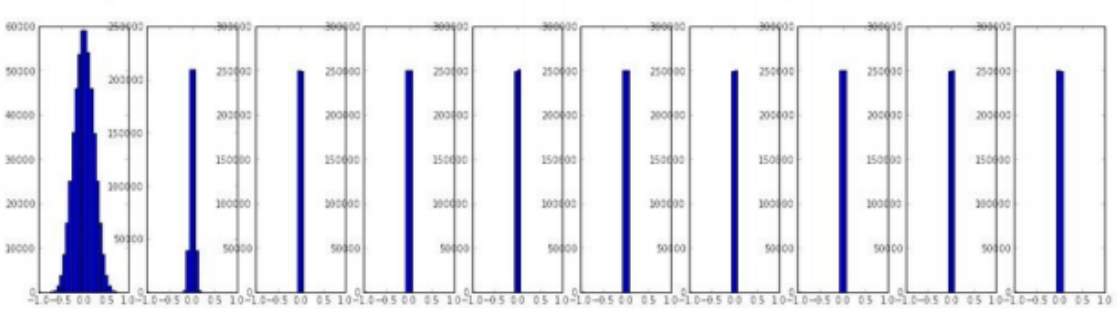
<초기화 문제 해결 방법>

임의의 작은 값으로 초기화하기

```
W = 0.01* np.random.randn(D,H)
```

- 초기 W를 표준정규분포에서 샘플링 한다

- 더 깊은 네트워크에서 문제가 발생할 수 있음.



레이어당 500개의 뉴런, 활성화 함수: tanh

위의 그래프처럼 출력값이 점점 0에 가까워짐을 알 수 있다.

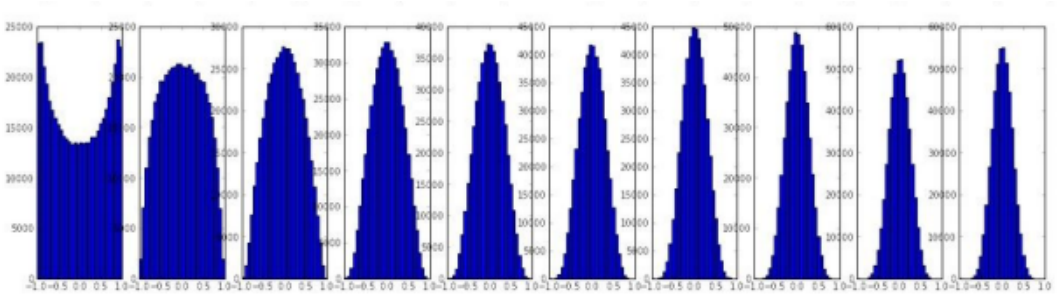
- 즉 입력 값이 0에 수렴하고, 이는 gradient를 작게 만듦, 결국 업데이트가 잘 일어나지 않음.(입력값이 점점 0에 수렴하고 가중치를 업데이트 하려면 upstream gradient에 local gradient를 곱하면 되는데 WX를 W에 대해 미분해보면 local gradient가 입력 X가 됨 => X는 매우 작은 값)

가중치를 큰 값으로 초기화

- saturation되고 gradient 0=> 가중치 업데이트가 일어나지 않음.

**Xavier Initalization - Glorot(2010)**

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```



- standard gaussian으로 뽑은 값을 입력의 수로 스케일링 해줌.

- 입출력의 분산을 맞춰주는 것

- 입력 수가 작으면 더 작은 값으로 나누고 좀 더 큰 값을 얻음. 작은 입력 수가 가중치와 곱해지기 때문에 가중치가 더 커야만 출력의 분산만큼 큰 값을 얻을 수 있기에 더 큰 가중치가 필요하다.

- ReLU는 출력의 절반을 죽이고 그 절반은 매번 0이 되므로 출력의 분산을 반토막 내버림. 따라서 점점 많은 값들이 0이 되고 비활성됨.

- 이를 위해 절반이 없어졌다는 사실을 고려하여 추가적으로 2를 더 나눠주었을 때 잘 작동됨.

## Batch Normalization

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

- 어떤 레이어로부터 나온 Batch 단위 만큼의 activations가 있다고 했을 때, 이 값들이 unit gaussian이길 바라기에 강제로 만듦

- 현재 batch에서 계산한 mean과 variance를 이용해서 normalization을 할 수 있음.

- 학습동안 모든 레이어의 입력이 unit gaussian이 됐으면 좋기에 네트워크의 forward pass 동안 그렇게 되도록 명시적으로 만들어줌.

- FC나 Cov Layer 직후에 넣어줌.

- Conv에서는 같은 Activation Map의 같은 채널에 있는 요소들은 같이 Normalize 해줌

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

하지만 unit gaussian이 되어 항상 saturation이 일어나지 않는 경우만 되는 것을 선호하진 않고, saturation을 조절하고 싶기 때문에 scaling 연산을 추가함.

BN은 gradient의 흐름을 원활하게 해주고 학습이 더 잘 되게 해줌. 또한 learning rate를 키울 수 있고, 다양한 초기화 기법들도 사용할 수 있게 됨. test time에서 추가적인 계산은 없음. 또한 regularization의 역할도 함.

# Learning Process

1. Data preprocessing : 데이터 전처리

2. Choose the architecture : hidden layer 구성

3. loss is reasonable : loss 값 확인

4. 데이터 일부만 학습시켜보기

5. training with regularization and learning rate

# Hyperparameter Optimization

cross-valindation은 training set으로 학습시키고 validation set으로 평가하는 방식

1. coarse stage: epoch 몇 번으로 좋은지 아닌 지 판단 -> 범위 결정

- log space에서 차수 값만 샘플링하는 게 좋음.

2. fine stage: 학습 좀 더 길게

- train 동안 cost 변화를 읽음. 이전 cost보다 더 커지거나 3배 높아지거나 하면 NaNs 나옴. 빠르게 오르면 멈추고 다른 거 선택

- 여기서 말하는 cost가 뭔지.

- reg범위, lr 범위 정함

- 최적 값이 범위의 중앙 쯤에 위치하도록 범위를 설정

- random search를 사용하면 important variable에서 더 다양한 값을 샘플링 할 수 있어 좋음

loss curve

-평평하다가 갑자기 가파르게 내려감-> 초기화 문제

= gradient의 역전파가 초기에는 잘 되지 않다가 학습이 진행되면서 회복

train과 va accuracy가 큰 차이면 오버핏 -> regularization의 강도 높이기

gap이 없다면 아직 overfit하지 않은 것이고 capacity를 높일 여유있는 것

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

가중치의 크기 대비 가중치 업데이트의 비율 0.001이 좋음

가중치의 크기=파라미터의 norm 구하기