

Agenda

1. Learning Rate: 경사하강알고리즘 $W_{t+1} := W_t - \alpha \frac{\partial}{\partial W_t} cost(W_t)$ 에서 α

lr이 너무 크면 발산한다, 너무 작으면 너무 오래 걸린다.

실험을 통해 적절한 lr을 찾아야 한다.

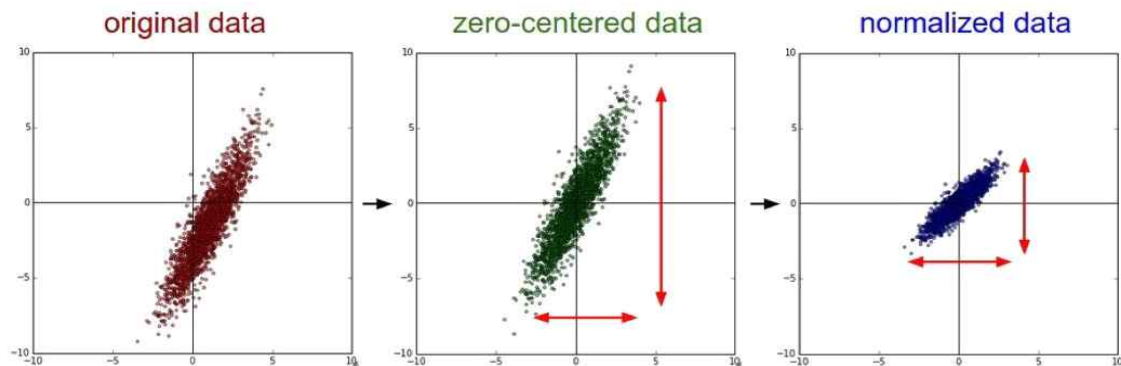
핑퐁되어 갇힐 수 있기 때문에 iteration에 따라 lr을 줄여준다.

2. Data Preprocessing

lr은 모든 feature의 가중치에 동일하게 적용되므로

각 feature 스케일의 차이가 크면 동일한 영향을 주지 못한다.

이러한 이유로 data preprocessing을 통해 스케일을 맞춰준다.



3. Avoid Overfitting

- More training data

- Reduce the number of features

- Regularization

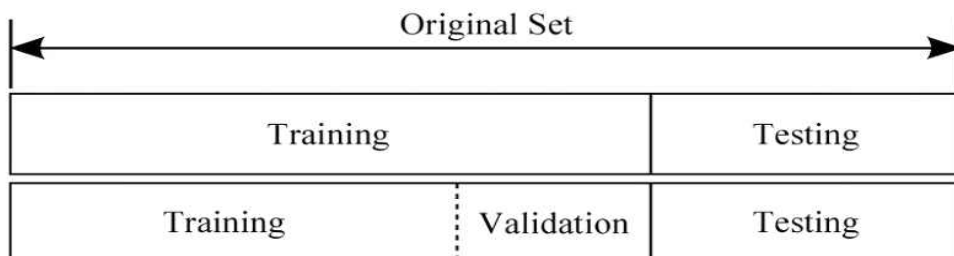
: let's not have too big numbers in the weight
cost function + regularization

$$\frac{1}{N} \sum_i^N D(S(WX_i + b), L_i) + \lambda \sum W^2, \lambda \text{는 정규화의 강도}$$

4. Performance evaluation

- use the learned parameter(W, b)

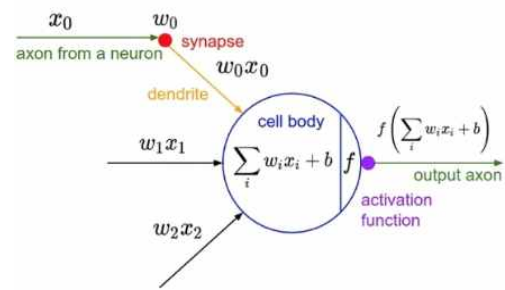
- use the unseen data = do not use training data



신경망(Neural Network)

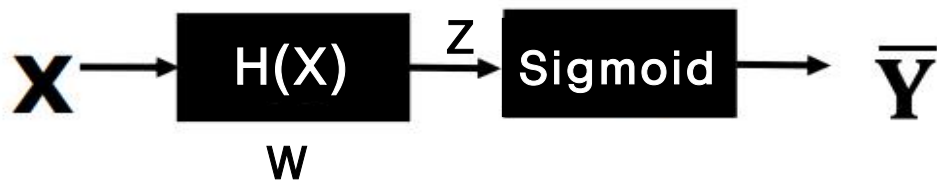
activation function

: 일정 기준 이상이면 1, 아니면 0



1. Perceptron

= neuron = 선형분류기 = Feed-forward network

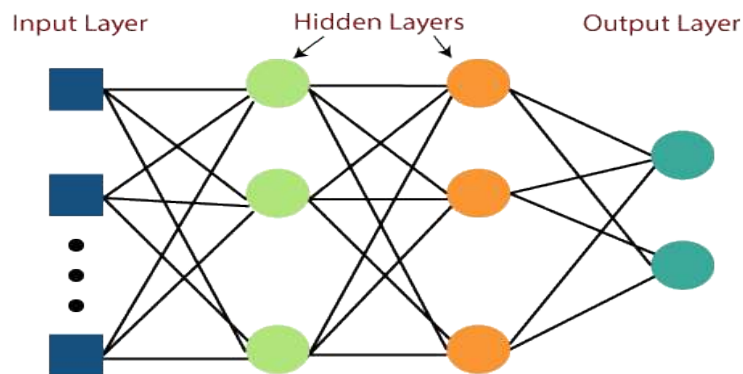


BUT One logistic regression unit cannot separate XOR

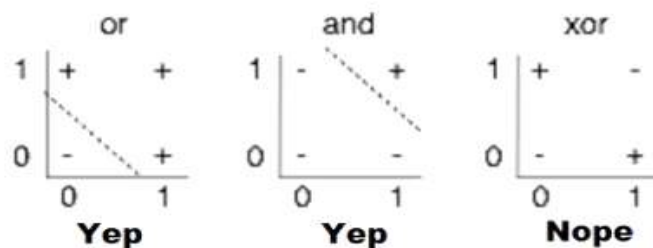
2. MLP(Multi-Layer Perceptron)

: 퍼셉트론(뉴런) 하나가 노드 하나가 되어 구성한다.

1. 입력층(Input Layer) 2. 은닉층(Hidden Layer) 3. 출력층(Output Layer)

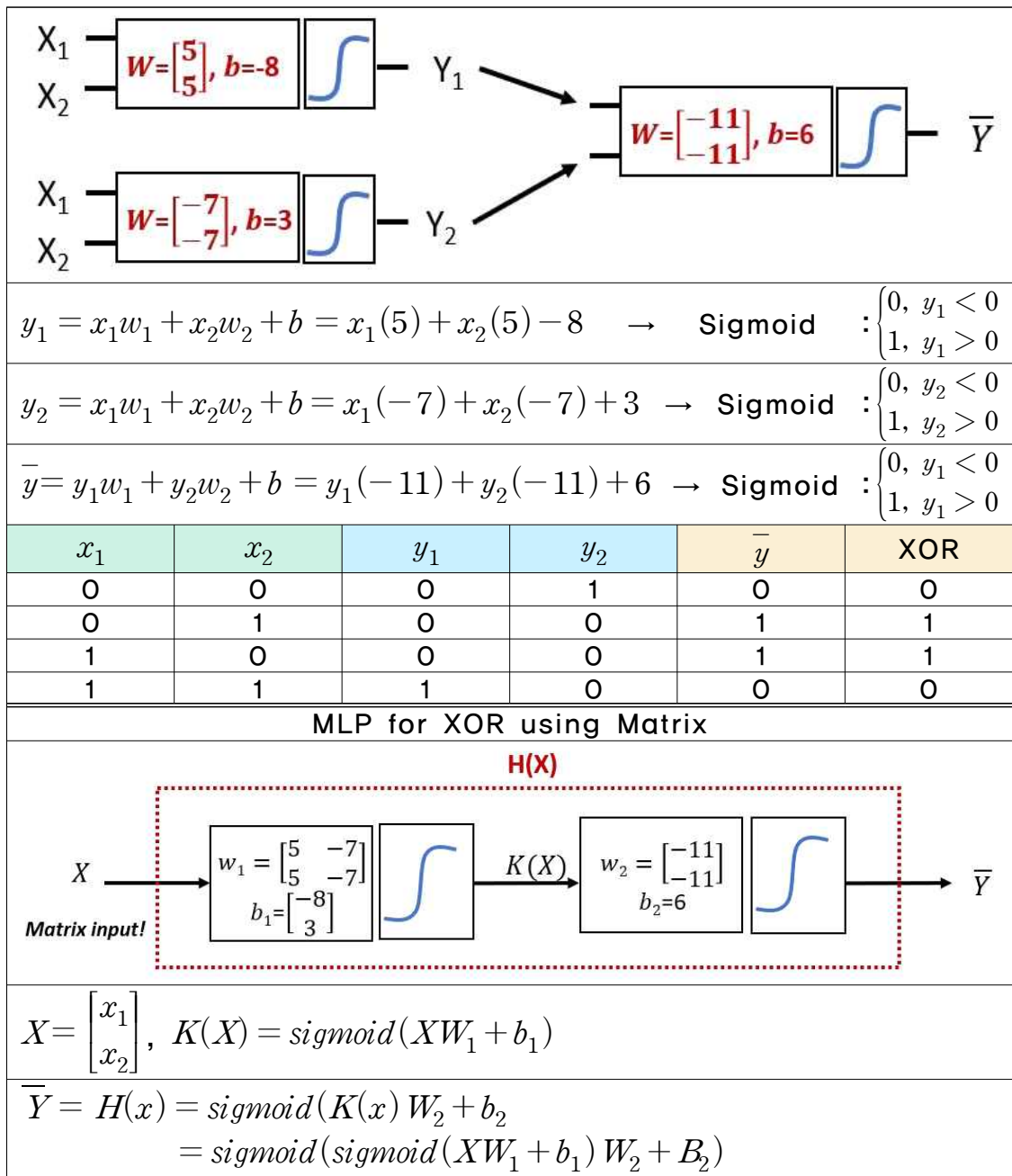


※ AND/OR 문제와 XOR 문제를 선형으로 풀 수 있을까?



Multiple logistic regression unit can separate XOR

Feed Forward(Forward Propagation)을 통한 XOR 풀이 가능

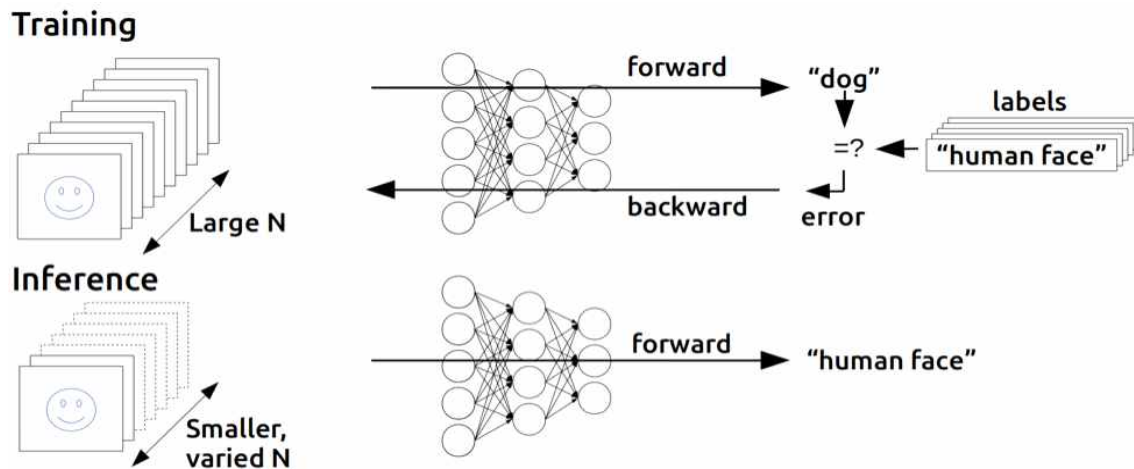


BUT too complicated to apply GD for finding optimal parameters

3. 역전파(Backpropagation)

: 역방향으로 오차를 전파하면서 각 층의 가중치를 업데이트하는 방법

- 1) W, b 초기화
- 2) Forward 계산 & Error 측정
- 3) Error 값을 Backward 하면서 W, b 갱신(error를 최소화하는 방향)



★ CNN(Convolutional Neural Network)

영상 전체로 feature가 기술되는 것이 아니라

영상의 부분과 부분의 결합을 통해서 최종적인 feature가 만들어진다는

영상의 부분 = Receptive field

Problem

1. NN을 이용한 XOR 문제 해결
2. DNN을 통한 어려운 문제 도전
3. DNN에서 Backpropagation하면 Gradient Vanishing 문제 발생

※ 역전파가 레이어를 많이 쌓은 신경망에서는 정상 동작하지 않았다

Gradient vanishing 문제: 너무 깊다 보니 역전파 작업을 하면서 gradient가 소실하여 어느 순간 0이 된다.

Other rising ML algorithms SVM, Random Forest

Geoffrey Hinton's summary

1. Labeled datasets were too small. → 빅데이터
2. Computers were too slow. → GPU
3. We initialed the weights in a stupid way. → w , b 의 초기화
4. We used the wrong type of non-linearity.

활성 함수(Activation Function)

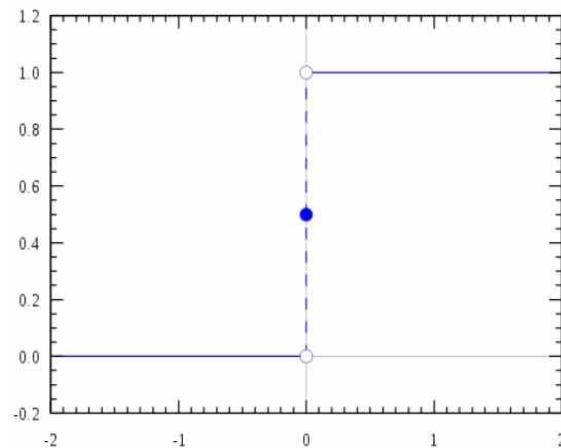
: 신경학적으로 뉴런 발사 과정

⇒ 입력 신호가 일정 기준을 만족하면 출력 신호를 다음 뉴런으로 보낼것 결정

1. Step

- 입력이 양수일 때 1, 음수일 때는 0

BUT 미분 불가능으로 모델 Optimization에 사용하기 어려워 활성 함수로 사용X



2. Sigmoid

- 단일 퍼셉트론에서 사용한 활성 함수

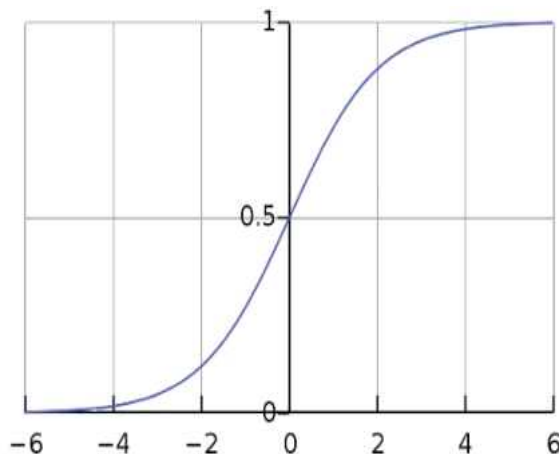
- 입력을 (0, 1) 사이로 정규화(normalization)

BUT $(-\infty, -2)$ 와 $(2, \infty)$ 에서 미분 값이 0에 가깝게 비슷하다

plus. $(-2, 2)$ 에서 미분 값이 작다

⇒ Backpropagation에서 layer를 거칠 때마다 작은 미분 값이 곱해져

Gradient Vanishing을 야기한다(MLP에서 학습이 잘 안 된 원인)

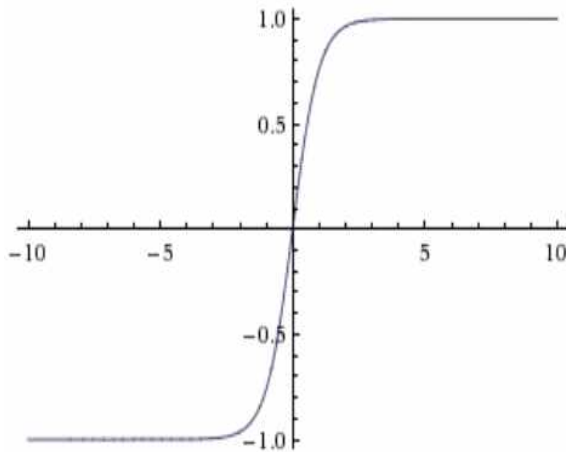


$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

3. tanh

- Sigmoid를 보완하기 위해 제안된 활성화 함수
plus. (-2, 2)에서 미분 값 크다
- 입력을(-1, 1) 사이의 값으로 정규화
- Sigmoid 보다 전반적으로 성능이 좋다

BUT Sigmoid보다는 덜 하지만 여전히 Gradient Vanishing 문제는 발생한다



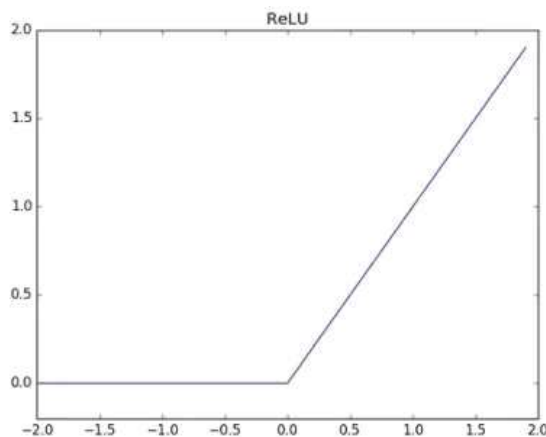
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

4. ReLU(Rectified Linear Unit)

- 가장 인기있는 활성화 함수
- 양수에서는 linear function과 같고, 음수에서는 0을 출력
- 미분 값을 0 또는 1의 값을 가지므로 Gradient Vanishing 발생X
- linear function과 같은 문제 발생X

엄연히 non-linear function이므로 layer를 깊게 쌓을 수 있다

- exp() 함수를 실행하지 않아 sigmoid나 tanh보다 6배 정보 빠르다

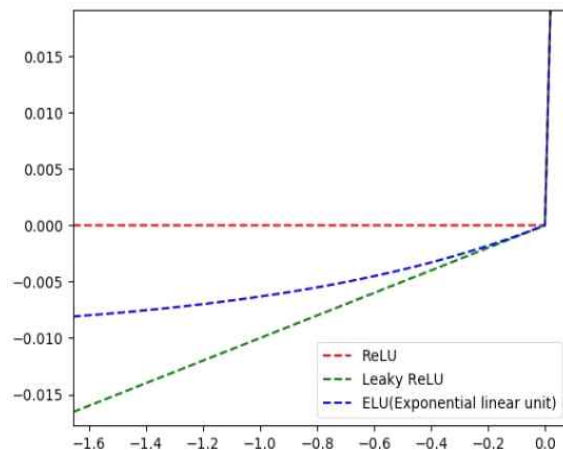


$$Relu(x) = \max(0, x)$$

But "dying ReLU" 현상 발생: $x < 0$ 에서 $Relu(x) = 0$

5. Leaky ReLU

- "dying ReLU" 현상 해결하기 위해 제시된 활성화 함수
- $x < 0$ 에서 작은 기울기 부여한다(보통 작은 기울기 = 0.01)
- Leaky ReLU로 성능 향상했다는 보고가 있으나 항상 그렇지는 않다

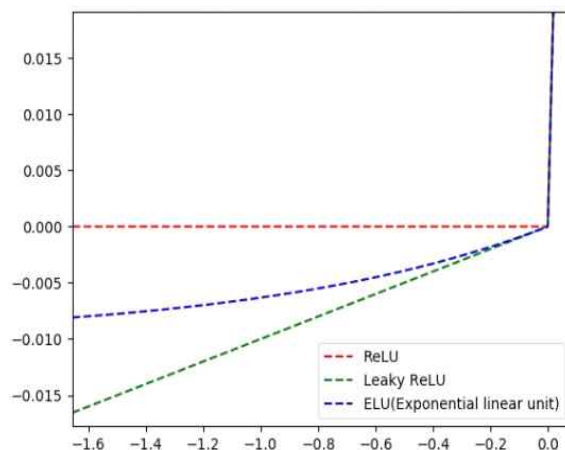


$$\text{Leaky Relu}(x) = \max(0.01x, x)$$

6. ELU(Exponential Linear Units)

- ReLU의 threshold를 -1로 낮춘 함수를 e^x 를 이용하여 근사한 활성화 함수
- "dying ReLU" 문제를 해결한다
- 출력 값이 거의 zero-centered에 가깝다

BUT ReLU, Leaky ReLU와 달리 e^x 를 계산해야 하는 비용이 든다



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

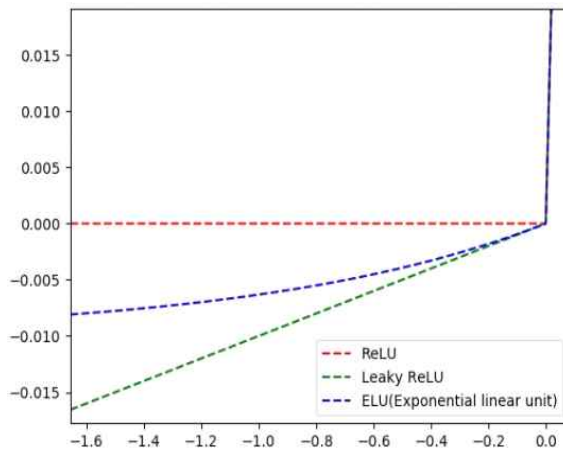
7. Maxout

- ReLU와 Leaky ReLU를 일반화한 활성화 함수

ReLU와 Leaky ReLU는 Maxout의 특수한 경우

- ReLU의 장점을 모두 갖고 있고 "dying ReLU" 문제도 해결

BUT ReLU와 달리 한 뉴런에 대해 파라미터가 2배이므로 전체 파라미터가 증가



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

1. 가장 먼저 ReLU 시도한다

2. 다음으로 Leaky ReLU, Maxout, ELU 시도한다

성능이 좋아질 가능성이 있다 but 반드시 좋아지는 것은 아니다

3 tanh 사용해도 되지만 성능이 개선될 확률은 적다

※ 앞으로 DNN에서 Sigmoid는 피한다

성능: $Maxout > ELU, Leaky ReLU \geq ReLU > tanh \geq Sigmoid$

Init method	maxout	ReLU	VLReLU	tanh	Sigmoid
LSUV	93.94	92.11	92.97	89.28	n/c
OrthoNorm	93.78	91.74	92.40	89.48	n/c
OrthoNorm-MSRA scaled	–	91.93	93.09	–	n/c
Xavier	91.75	90.63	92.27	89.82	n/c
MSRA	n/c†	90.91	92.43	89.54	n/c

"All You Need is a Good INIT", ICLR2016

The Compatibility of activation functions and initialization

Dataset: CIFAR-10

CIFAR-10: 32×32 픽셀의 60,000개 이미지

각 이미지는 10개의 클래스로 라벨링되어 있다

32×32의 1D 벡터를 처리하는 것처럼 사용

가중치 초기화(Weight Initialization)

지금까지 기본적인 선형 회귀나 Softmax와 같은 알고리즘에서는 $-1 \sim 1$ 의 난수를 weight로 사용했지만, NN에서는 weight 선정에 주의해야 한다

⇒ $w = 0$ 이면 Backpropagation 시 gradient 값이 0이 되어

Gradient Vanishing 현상 발생

※ 절대 0으로 초기화하지 않아야 한다

1. RBM(Restricted Boltzmann Machine)

Hinton et al.(2006) "A Fast Learning Algorithm for Deep Belief Nets"

– too complicated

2. Xavier initialization

– 노드의 입력(fan_in)의 수와 출력(fan_out)의 수에 비례해서 초기값을 결정

$W = \text{np.random.randn}(fan_in, fan_out)/\text{np.sqrt}(fan_in)$

$$std = gains \times \sqrt{\frac{2}{fan_in + fan_out}}$$

X. Glorot and Y. Bengio "Understanding the difficulty of training deep feedforward neural networks," in International conference on artificial intelligence and statistics, 2010

3. He's initialization = MSRA

– 노드의 입력(fan_in)의 수와 출력(fan_out)의 수에 비례해서 초기값을 결정

$W = \text{np.random.randn}(fan_in, fan_out)\text{np.sqrt}(fan_in/2)$

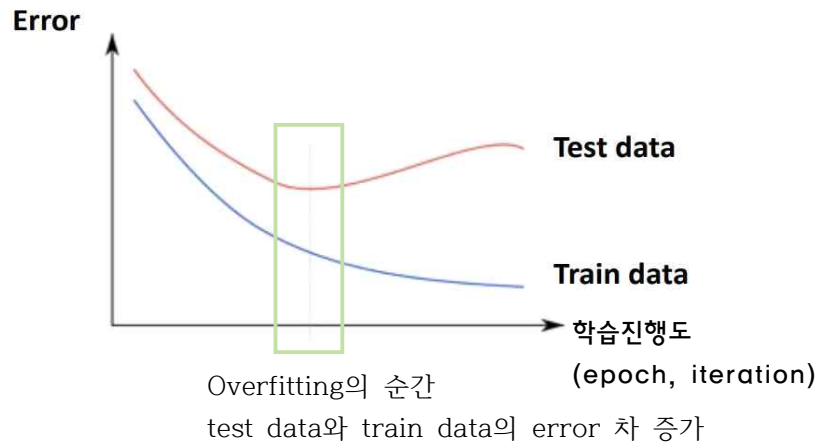
$$std = \frac{gain}{\sqrt{fan_mode}}$$

K. He, X. Zhang, S. Ren, and J. Sun "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," 2015

[PyTorch 초기화 torch.nn.init](#)

드랍아웃(Dropout)

목적: Overfitting을 줄이기



Solution

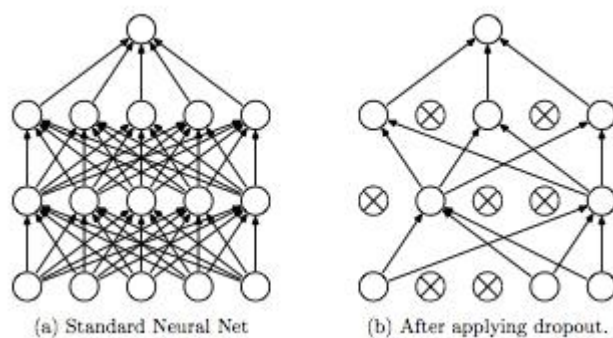
1. More Training data
2. Reduce the number of feature
3. Regularization ex. Dropout

드랍아웃(Dropout)

: 신경망에서 유닛 제거(매 학습에 비율 0.3 ~ 0.5 정도가 좋다)

훈련 데이터에 대한 복잡한 공동 적응 방지하여 Overfitting 줄이기

※ 학습에서만 적용하고 테스트에서는 모든 유닛을 사용함



매번 무작위 유닛이 Dropout되어 Ensemble과 같은 효과를 보여준다

[`torch.nn.Dropout\(p=0.5, inplace=False\)`](#)

Tips for DNN

1. 활성 함수(Activation Function)을 잘 선택한다
ReLU가 가장 널리 사용된다
2. 가중치 초기화(Weight Initialization)을 잘 선택한다
Xavier가 가장 널리 사용된다
3. 드롭 아웃(Dropout)을 잘 적용한다
NN-ReLU-Dropout를 하나의 블록으로 쌓는다

Batch_size: 몇 문제 풀고 해답을 맞추는지

Batch_size = 100 vs Batch_size = 1

100: 한 문제를 틀릴 때 이후 유사 문제를 틀릴 수 있다

100 문제를 기억해야 하므로 GPU의 메모리가 커야 한다

1: 첫 문제를 틀리더라도 바로 학습하여 나머지 문제는 맞출 수 있다

1 문제 풀고 파라미터를 갱신해야 하므로 시간이 오래 걸린다

Epoch: 전체 문제를 몇 번 풀어볼지(n회독 반복)

낮은 점수 구간에서 성능을 빠르게 오르지만

높은 점수 구간에서는 성능을 올리기는 느리므로 더 많이 반복해야 한다

Optimizer

1. Gradient Descent(GD)

학습 데이터 전체를 사용하는 최적화

2. Stochastic Gradient Descent(SGD)

학습데이터 일부(mini-batch)를 사용하는 최적화

- BGD보다 빠르게 수렴

- SGD를 여러번 반복할수록

BGD와 유사한 결과로 수렴

