20.04.20

주제 **Activation Functions**

Activation Functions

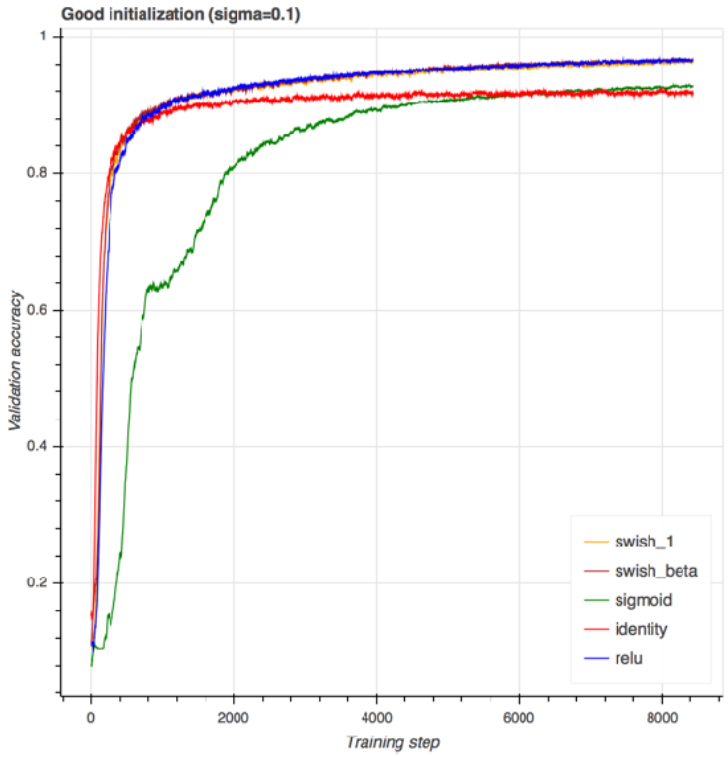
우선 짚고 넘어갈 것은, 강의에서 배우는 여러 가지 Activation Function은 **Hidden Layer의 뉴런에 적용할 때를 기준으로** 말하는 것이다. Output layer에서는 DNN 전에 하던 것처럼 classification 모델에선 softmax 함수를, regression 모델에선 linear 함수를 사용해야 '예측'으로서의 형태를 띄게 된다.

참고자료 2의 영상에 의하면, **실전에서는** 어지간하면 hidden layer에는 **ReLU를 활성화 함수로** 쓰되, Dying ReLU problem(dying neuron)이 심하게 발생할 경우 Leaky ReLU나 Maxout을 쓰라고 권한다. 그리고 다른 활성화 함수를 쓸 일이 생기더라도, 절대 Hyperbolic Tangent나 Sigmoid를 Hidden Layer의 활성화 함수로는 쓰지 말라고 강력히 권고한다.

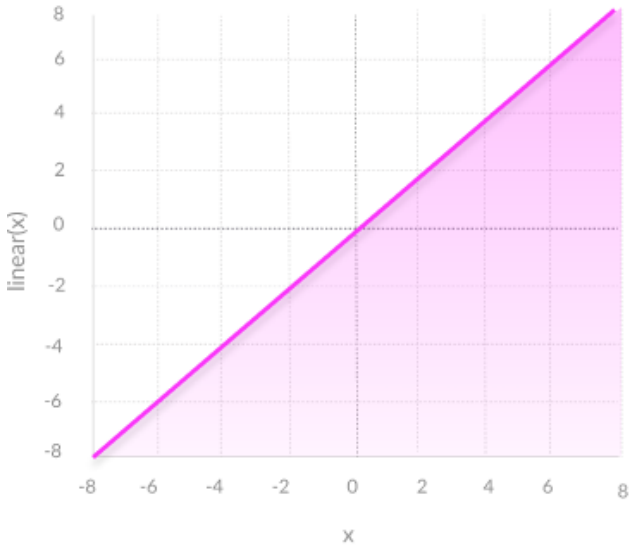
어느 활성화 함수가 가장 좋은가?

ReLU, ELU, Leaky ReLU의 효율을 실험한 글인데, 결과적으로 다 비슷하긴 한데 ELU > Leaky ReLU > ReLU 순으로 성능이 조금 차이가 났다고 한다.

- activation function 및 성능 비교



비선형적(Non-Linear)인 활성화 함수를 택해야 하는 이유



선형 함수의 경우...

- 여러 레이어로 쌓아 놓더라도 결국은 한 레이어나 다름 없다는 점. (레이어 쌓기가 불가능)

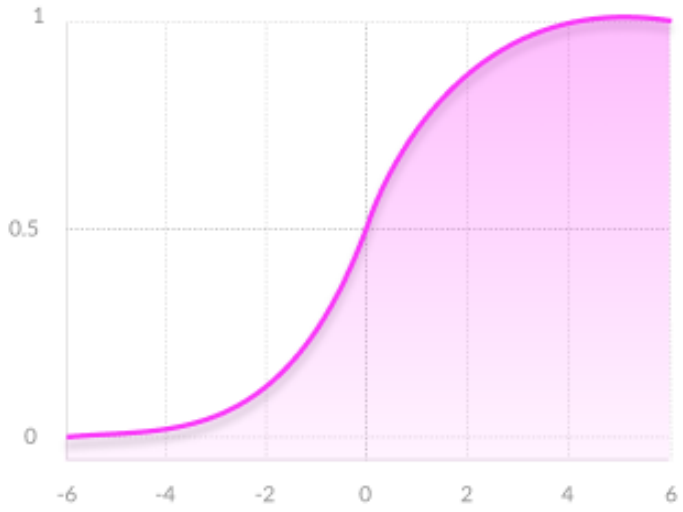
$$F_n(x) = c_n x$$
$$F_3(F_2(F_1(x))) = c_3 c_2 c_1 x = c' x = G(x)$$

- Back propagation을 쓰더라도, 어떤 **input 값이 들어오든지 상관 없이** 기울기가 **일정**하기 때문에 어떤 변화를 줘야 더 나은 모델로 발전시킬 수 있는지 알 수가 없다는 점.

따라서 Non-Linear Activation Function(비선형 활성화 함수)을 써야 사진이나 영상, 음성 등의 복잡한 데이터에 걸맞는 충분히 복잡한(레이어를 쌓을 수 있고, input에 따라 다른 기울기를 도출할 수 있는) 모델을 만들 수 있다고 한다.

Sigmoid/Logistic Activation Function

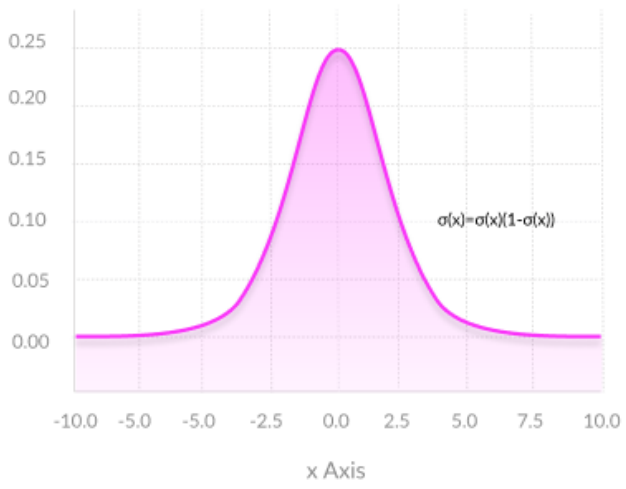
형태는 이미 알다시피...



- 이 모델의 장점:
  - 연속적인 gradient 값
  - output의 범위가 (0, 1)로 **제한** 되어 있음 (normalization; 정규화 됨)

- 이 모델의 취약점:
  - **Vanishing Gradient 현상**. 일정 범위 이상 부터는 예측의 차이가 거의 나지 않으며, gradient도 0에 가까움. 이 때문에 학습이 더디게 될 수 있음.

    (참고 자료 - sigmoid function 미분 꼴)



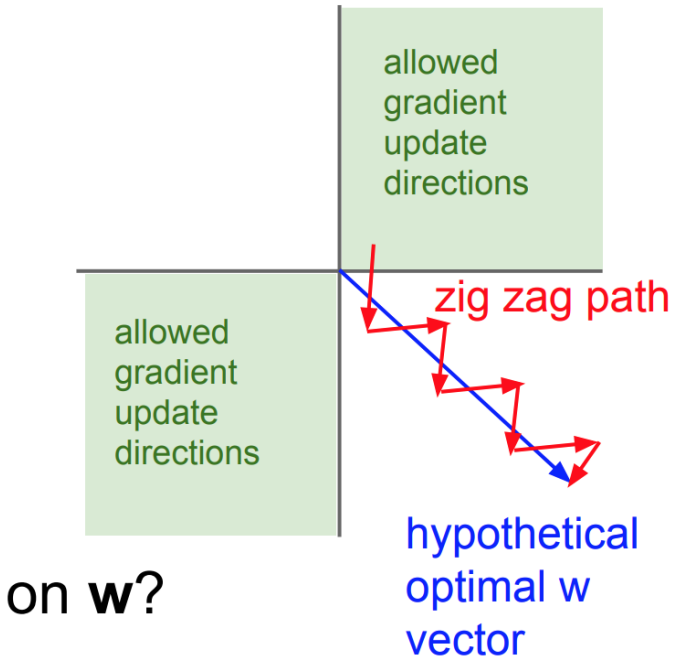
- output의 중심이 0이 아님("not zero-centered"):
  - 왜 문제가 되는가?

  활성화 함수를 f라고 한다면,

$$f(w_1x_1 + w_2x_2 + w_3x_3 + ... + b)$$
$$\frac{\partial f}{\partial w_i} = f' \cdot x_i$$

  과 같은 과정에 의해, gradient는 input 값과 활성화 함수의 미분 값에 영향을 받게 된다.

  한편 hidden layer의 활성화 함수가 전부 Sigmoid일 경우 hidden layer에선 input(x)을 무조건 양수로 받게 된다. 따라서 **모든 w에 대한 gradient의 부호도 똑같게** 되고, w를 고치는 방향도 (1) 모두가 양수 방향으로 움직임 (2) 모두가 음수 방향으로 움직임, 이라는 두 방향으로 제한되기 때문에 가장 효율적인 방향으로 나아가고 싶어도 못 나간다는 한계가 발생한다.

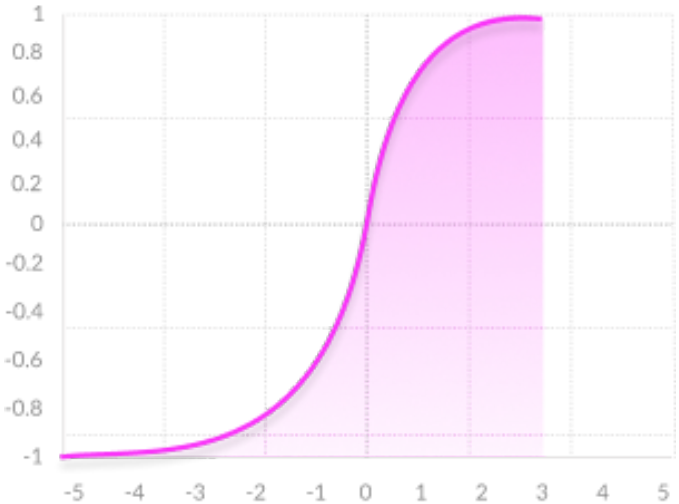


두 개의 input을 받아 두 개의 weight 값을 갖는 함수라면 대충 이런 식(zig zag path)의 움직임을 보인다. 이 때문에 w를 업데이트 하는 과정에서 효율이 저하된다.

- exponential 함수가 들어가 계산하기 복잡함.

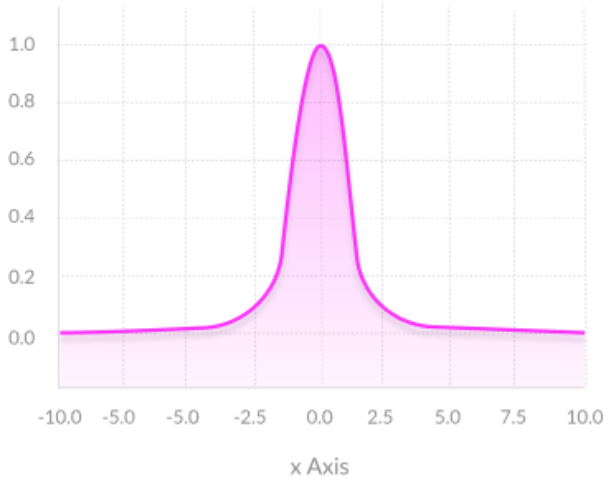
TanH(Hyperbolic Tangent)

형태는

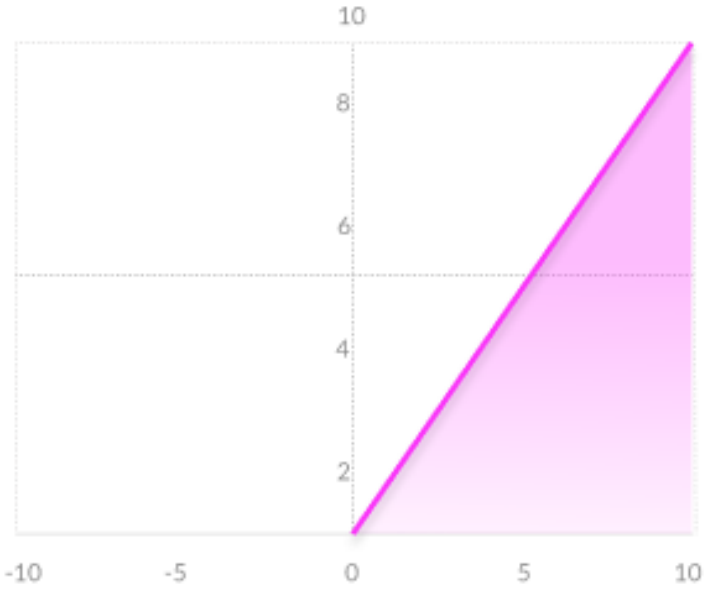


$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- 이 형태는 Sigmoid Function과 장단점이 비슷한데, 중심이 0이라는 점만 다르다. sigmoid의 한계였던 non zero-centered 모양의 단점을 극복한다.
- TanH 역시 vanishing gradient 현상을 극복하진 못한다. 또한 exponential 형태이기에 계산이 복잡하다.

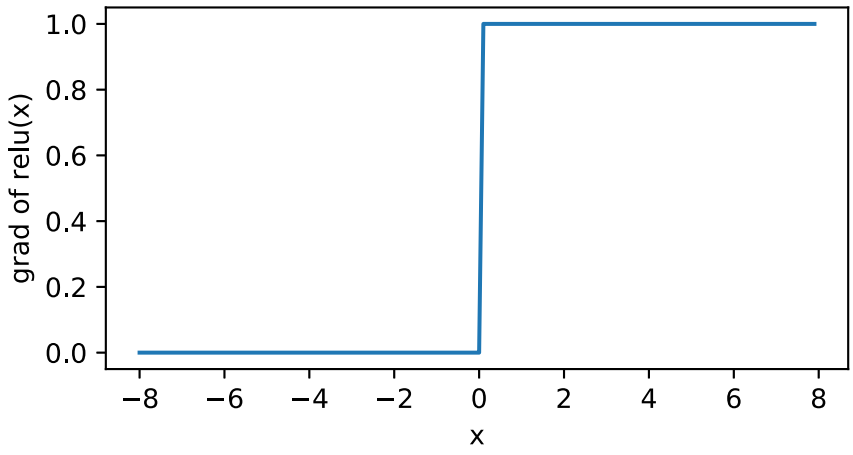


ReLU(Rectified Linear Unit; 교정된 선형 유닛)



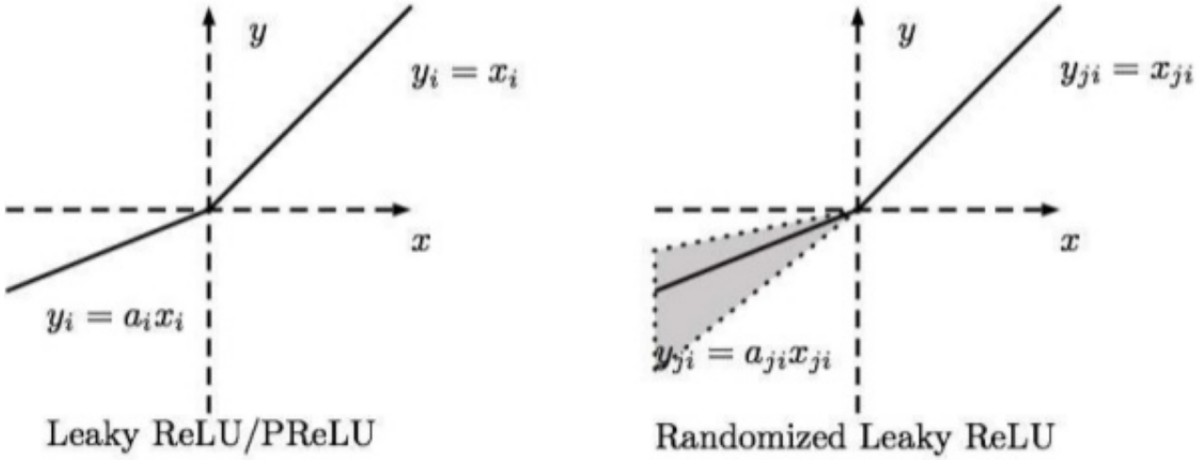
Rectified Linear Unit Function의 약자로, 아주 간단한 비선형 변환이다.

- 값이 사라지거나 그대로 전달하는 방식으로 미분이 잘 작동한다.
- 0에서는 좌미분 값인 0을 선택한다.



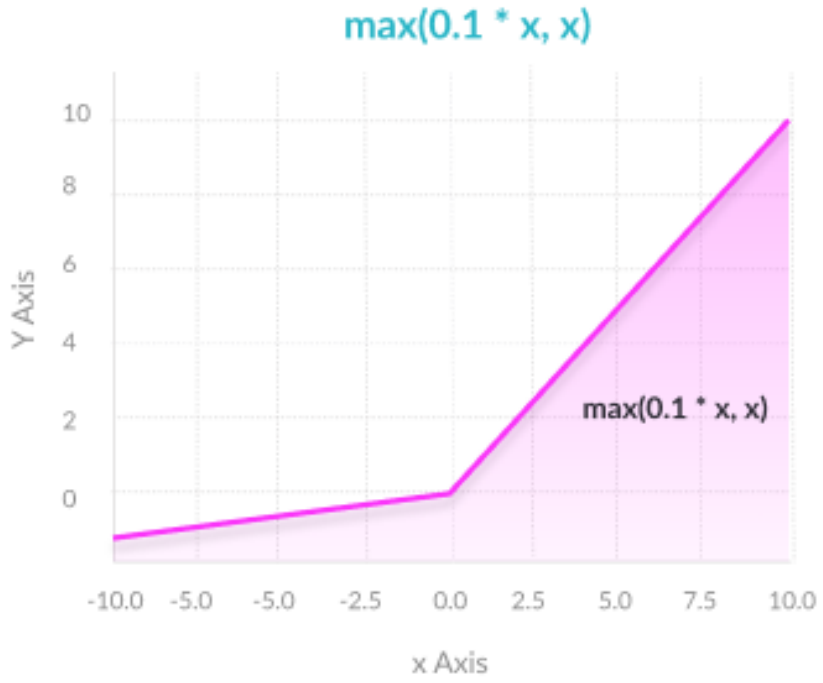
도함수가 unit step function의 형태로, 빠른 연산이 가능하다.

- 이 모델의 장점:
  - 계산하기가 간단해 효율적임. 네트워크 연산을 하기가 훨씬 빨라진다.
  - 선형 모델에서의 한계를 극복한 비선형 모델.
  - Vanishing Gradient 문제를 해결해줌.
- 이 모델의 취약점
  - The **Dying ReLU problem**: 0 이하의 input을 가질 경우 gradient가 0이 되어버려 몇몇 노드에선 학습이 불가능해짐.



ReLU의 파생 함수

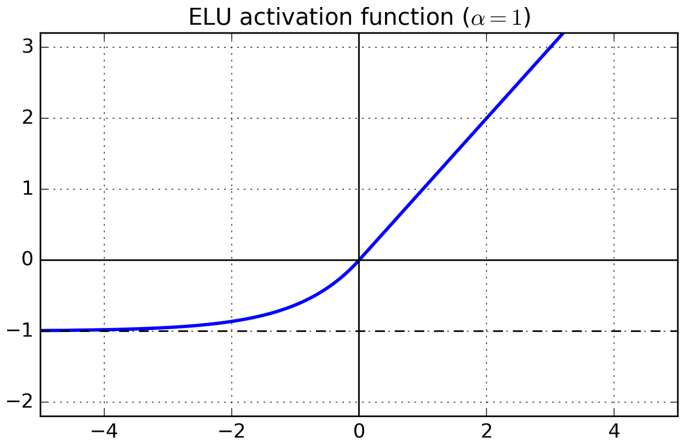
- Leaky ReLU



기존의 ReLU에서 음수 영역을 양의 기울기(0.1)를 가진 선형 함수로 만들어줌.

- 이 모델의 장점
  - 0 이하의 input을 받을 때도 back propagation이 가능하게 된다.
- 이 모델의 단점
  - 'Leaky ReLU가 ReLU의 완전한 상위호환 아니냐?' 라는 주제의 토론글. 명확한 단점은 없는 듯 보인다.
- Parametric ReLU : 음수 영역의 기울기도 학습을 통해 변경할 수 있음. 이 방법은 계산해야 할 변수가 더 많아진다는 점(기존의 w에 더해서 기울기 a까지)과, 문제에 따라 학습 효율이 달라져 예측이 어렵다는 단점이 있음.
- Exponential LU(ELU) : 음수 영역을 exponential 함수로 설정.

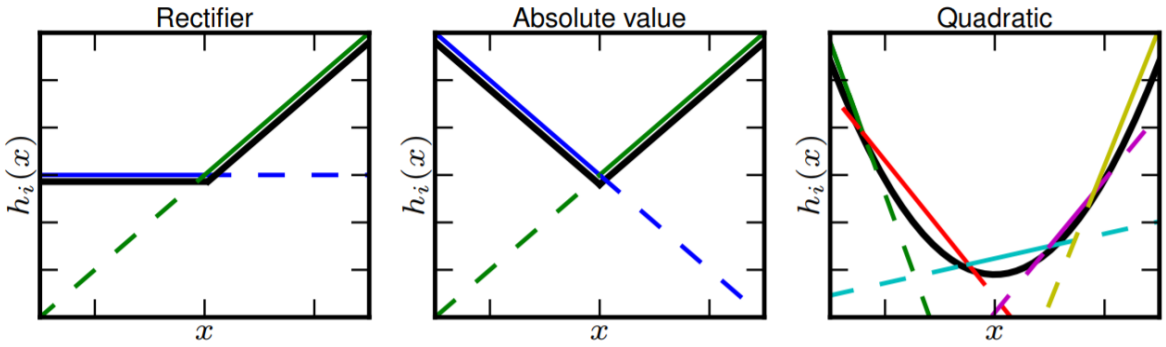
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$



Maxout

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

여러 개의 linear function(wx+b)의 결과의 총합 중 가장 큰 값을 고르는 방식의 활성화 함수다.



여러 개의 직선 중에서 가장 위에 있는 부분만 골라서 조각조각 이어붙이는(piece-wise linear), 그런 느낌이다.

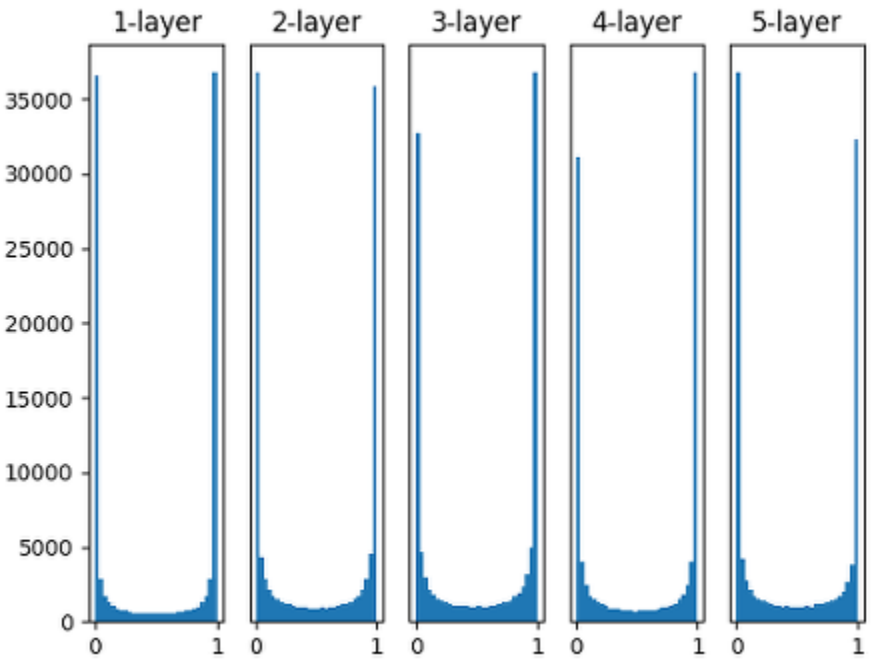
맨 왼쪽 그림에서 알 수 있듯이, ReLU와 L-ReLU도 포함하는 일반적인 모델이다. 이 모델의 장점은 ReLU의 모든 장점을 가져가면서 단점이었던 dying ReLU를 극복한다는 것이다. 한편 한 뉴런에 2개 이상의 선형 방정식(즉 더 많은 weight)을 포함해야 하므로, 계산해야 할 weight가 몇 배로 더 많아진다는 단점이 있다.

Weight Initialization

가중치 초기값이 작은 수일 경우

가중치 초기값은 작은 값으로 초기화 해야하는 데, 그 이유는 활성화 함수가 **sigmoid**일 경우 만약 가중치 초기값(절대값)을 큰 값으로 한다면 **0과 1로 수렴**하기 때문에 그래디언트 소실이 발생하게 된다. 또한 활성화 함수가 **ReLU**일 경우 절대값이 클 경우 **음수일 때는 dead ReLU 문제가 발생**하고, **양수일 때는 그래디언트 폭주**가 일어나게 된다.

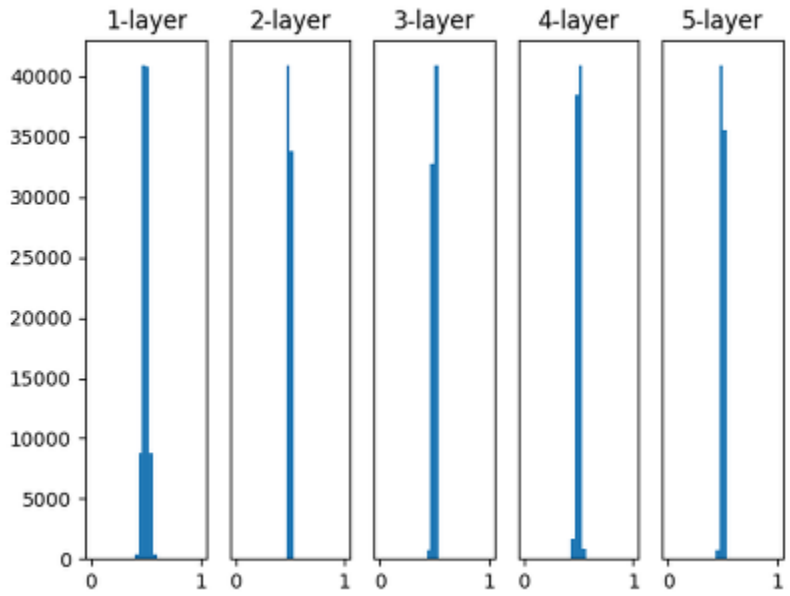
- sigmoid activation func



  sigmoid 함수의 특성상 출력이 0과 1에 몰려있다.

  → gradient가 0에 몰려있다는 의미와 같다.

- weight 편차를 줄였을 경우,



Deep Belief Network (DBN)

Restricted Boltzman Machine (RBM)

Idea

Input Data는 적합한 Output Data를 계산하기 위한 용도로만 이용된다. 이러한 관점에서는 Hidden Layer가 Output Data를 계산하는데 중요한 Feature들만 판별할 수 있으면 된다. 그런데 이왕이면 Output Data를 계산하는 용도로 효과적인 Hidden Layer로 다시 Input Data를 생성하게 할 수는 없을까?

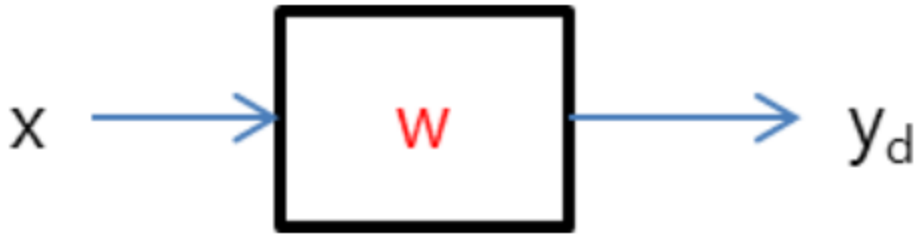
예를 들어 삼각형/사각형 이미지를 Input으로 해서 이것이 삼각형인지, 사각형인지를 분류하는 것을 Neural Network로 해결해야 한다고 가정해보자. Hidden Layer가 꼭지점의 개수만 찾아낼 수 있다면, 이것은 쉽게 해결된다. 분류하는 것만 보자면 더 이상 생각할 필요가 없는 것이다.

그런데, Hidden Layer 값을 통해 Input이미지를 재생성하려고 하면 꼭짓점의 갯수로는 부족하다. 면적 정보가 추가로 들어가면 더 Input이미지와 유사한 이미지를 생성할 수 있다. 밑변의 길이 정보가 들어가면 더더욱 유사한 이미지를 생성가능하다.

이때 Input Image를 더 잘 표한할 수 있는 정보로는 면적이 좋을지, 밑변이 좋을지 등을 선택해주는 확률 모델이 바로 RBM이다. 그런데 짚고넘어가야할 점은 꼭 Input Image를 더 잘 표현하기 위해 더 많은 정보를 필요로 하는 것은 아니다. 하나의 Feature만 사용하더라도 면적과 꼭짓점의 개수 중 어떤 것이 더 효과적이냐는 다를 수 있다.

얼굴 인식을 예로들면 MLP는 각 인식을 해야하는 사람들의 특징(차이)를 구분하기 위해 Network를 Training한다면, RBM은 각 사람들을 잘 표현할 수 있는 Feature들을 찾기 위해 Network를 Training 한다고 볼 수 있다.

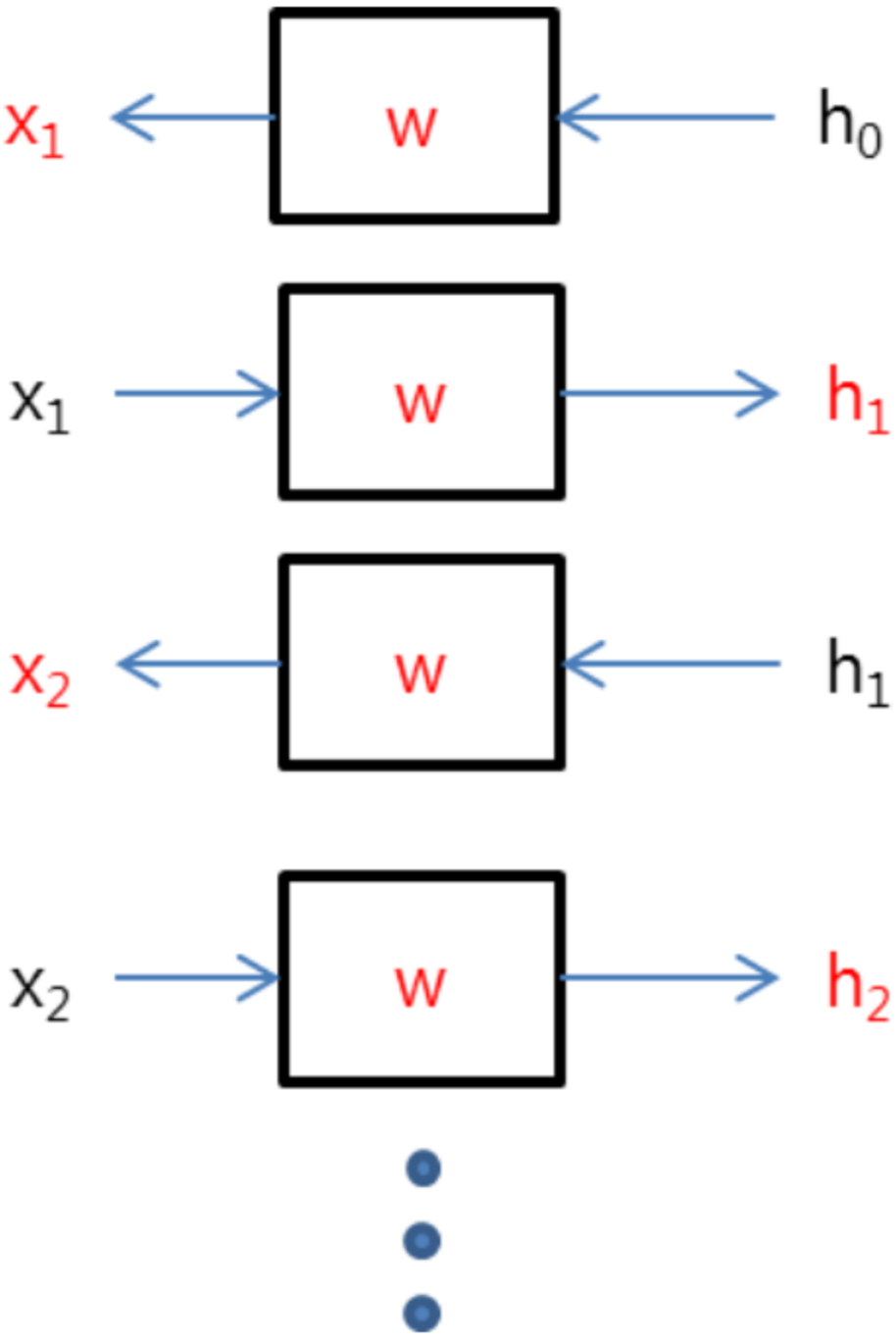
Mechanism



<기존의 Neural Network System>

<Deep Belidef Network>

기존에는 위의 그림처럼 상위 layer부터 하위 layer로 weight를 구해왔지만 DBN은 하위 layer부터 상위 layer를 만들어나간다. DBN에서는 입력 x만을 알 수 있다고 가정하고 그 입력이 어떤h(더 나아가 y)로부터 만들어졌는지 추론한다. 3*w=h이라는 조건을 만족하는 w와 h는 무수히 많은 해가 존재하고, 이를 구하기 위해 RBM을 이용한다.



<Restricted Boltzmann Machine>

w와 h 는 아무 값이나 넣어서 초기화 시켜두었다고한다면 h로부터 어떻게든 x를 만들어내고 그 x로 부터 다시 h를 만들어내고, 그렇게 만들어낸 h로부터 다시 x를 만들어내는 과정을 무수히 반복해서 점점 올바른 h와 w를 찾아나간다.

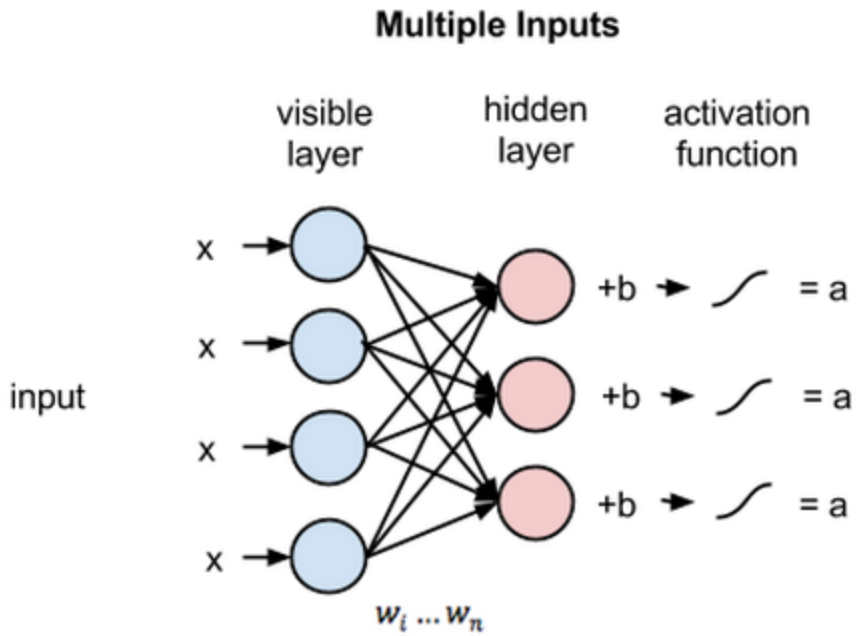
RBM은 Markov Random Fields 의 특별한 타입으로, visible layer와 hidden layer가 양방향으로 전부 연결되어 있는(fully-connected) 얕은 두 층으로 이루어진 확률론적인 모델이다.

하지만 같은 layer의 노드끼리는 연결이 제한되어 있기 때문에 이 모델의 이름에 Restricted가 붙었다고 한다.

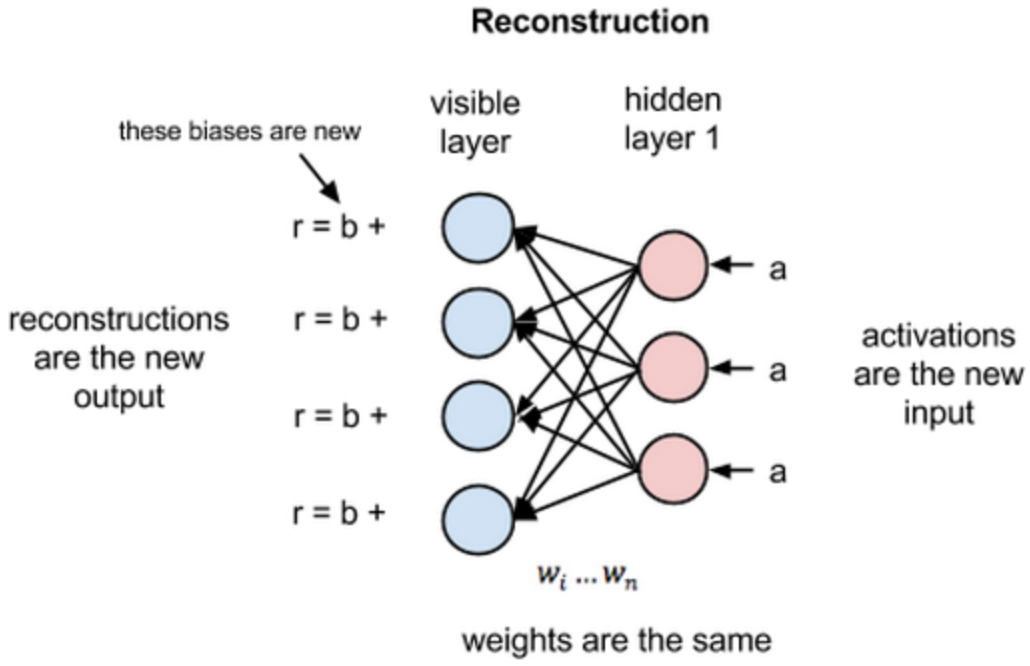
1. **Pre-Training**

RBM은 현재 layer와 다음 layer에 대해서만 동작한다.

(forward) 현재 layer에 들어온 x값에 대해 weight을 계산한 값을 다음 layer에 전달한다.

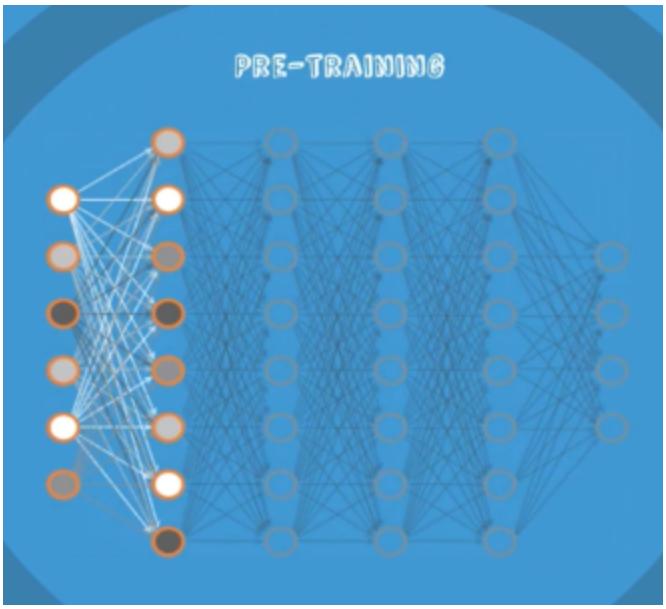


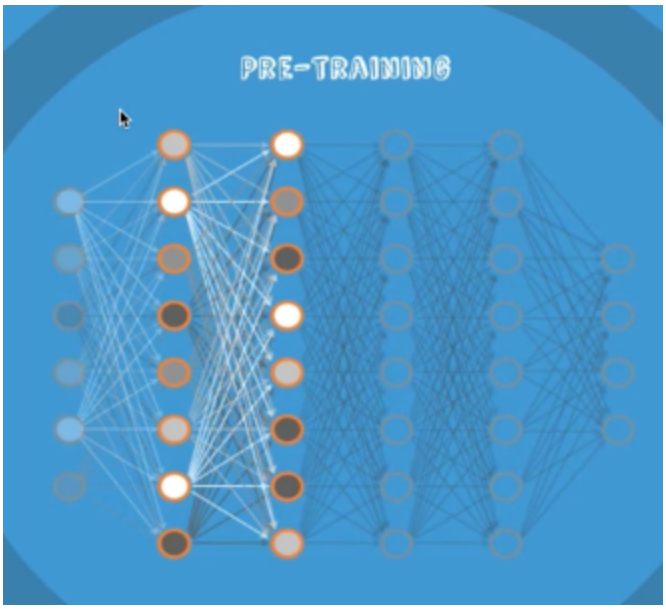
(backward) 이렇게 전달 받은 값을 이번에는 거꾸로 이전(현재) layer에 weight 값을 계산해서 전달하여 입력값에 대해 재구성한다.

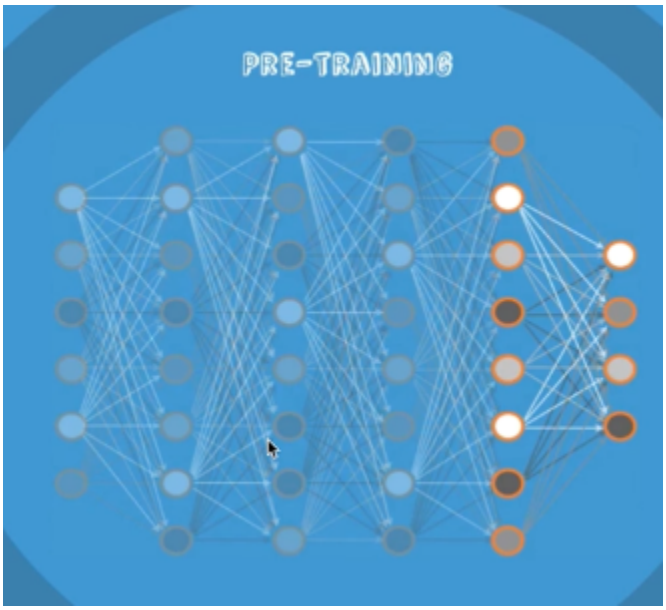


forward와 backward 계산을 반복해서 진행하면, 입력값 x와 재구성된 값(x hat)의 차이가 최소가 되는 weight을 발견하게 된다.

RBM은 이 방법을 2개 layer에 대해 초기값을 결정할 때까지 반복하고, 다음 번 2개에 대해 다시 반복하고. 이것을 마지막 layer까지 반복하는 방식이다. 이렇게 초기화된 모델을 Deep Belief Network라고 부른다. 신뢰할 수 있는 초기값을 사용하기 때문에 belief라는 단어가 들어간 것이다.


PRE-TRAINING


PRE-TRAINING



이러한 과정을 pre-training이라고 부른다. 첫 번째 그림에서 첫 번째 layer의 weight을 초기화하고, 두 번째 그림에서 두 번째 weight을 초기화하고, 세 번째 그림까지 진행하면 전체 layer에서 사용하는 모든 weight이 초기화된다.

주의해야 할 사항은 한번의 weight uptdate가 끝나면 이것을 freeze시켜놓고 다음 layer도 마찬가지로 RBM을 통해 학습히키는 것이다.

1. **Fine Tuning**

위에서 학습하여 결정된 weights이 neural network의 초기값이 된다. 여기에 error back propagation 알고리즘으로 마지막 튜닝을 하는것이 fine-tuning이라는 과정인 것이다.

그런데, 설명이 긴 만큼 구현하는 것도 쉽지 않을 건 틀림없다. 좋은 소식이 있다. 굳이 복잡한 형태의 초기화를 하지 않아도 비슷한 결과를 낼 수 있다고 한다.

Xavier/He Initialization

Xavier Initialization

2010년에 발표된 xavier 초기화와 2015년에 xavier 초기화를 응용한 He 초기화가 있다. 이들 초기화 방법은 놀랄 정도로 단순하고, 놀랄 정도로 결과가 좋다.

Xavier Initialization 혹은 Glorot Initialization라고도 불리는 초기화 방법은 이전 노드와 다음 노드의 개수에 의존하는 방법이다. Uniform 분포를 따르는 방법과 Normal분포를 따르는 두가지 방법이 사용된다.

- **Xavier Normal Initialization**

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

- **Xavier Uniform Initialization**

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

Xaiver함수는 비선형함수(ex. sigmoid, tanh)에서 효과적인 결과를 보여준다. 하지만 ReLU 함수에서 사용 시 출력 값이 0으로 수렴하게 되는 현상을 확인 할 수 있다. 따라서 ReLU함수에는 또 다른 초기화 방법을 사용해야 한다.

He Initialization

ReLU를 활성화 함수로 사용 시 Xavier 초기값 설정이 비효율적인 결과를 보이는 것을 확인했는데, 이런 경우 사용하는 초기화 방법을 He initialization이라고 한다. 이 방법 또한 정규분포와 균등분포 두가지 방법이 사용된다.

- **He Normal Initialization**

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in}}}$$

- **He Uniform Initialization**

$$W \sim U(-\sqrt{\frac{6}{n_{in}}}, +\sqrt{\frac{6}{n_{in}}})$$

이 두 가중치 초기화 함수들은 torch.nn.init 모듈에 정의되어있다.

```
def xavier_normal_(tensor, gain=1.):
    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    std = gain * math.sqrt(2.0 / float(fan_in + fan_out))

    return _no_grad_normal_(tensor, 0., std)
```

```
def xavier_uniform_(tensor, gain=1.):
    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    std = gain * math.sqrt(2.0 / float(fan_in + fan_out))
    a = math.sqrt(3.0) * std  # Calculate uniform bounds from standard deviation

    return _no_grad_uniform_(tensor, -a, a)
```

```
def kaiming_normal_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu'):
    fan = _calculate_correct_fan(tensor, mode)
    gain = calculate_gain(nonlinearity, a)
    std = gain / math.sqrt(fan)
    with torch.no_grad():
        return tensor.normal_(0, std)
```

```
def kaiming_uniform_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu'):
    fan = _calculate_correct_fan(tensor, mode)
    gain = calculate_gain(nonlinearity, a)
    std = gain / math.sqrt(fan)
    bound = math.sqrt(3.0) * std  # Calculate uniform bounds from standard deviation
    with torch.no_grad():
        return tensor.uniform_(-bound, bound)
```

kaiming 함수들이 바로 He initialization 함수들이다. torch 안에 이런식으로 정의가 되어있고, 살펴보면 위의 수학적 정의들을 코드로 잘 옮겨놨음을 확인할 수 있다. 실제 신경망을 구현할 때 학습 전에 미리 weight를 위 함수들을 이용해서 initialization하면 된다.

Conclusion

다양한 종류의 초기화 방법에 대해서 알아 보았다. 초기값 설정이 학습과정에 매우 큰 영향을 끼칠 수 있기 때문에 초기화 방법 또한 신중히 선택해야 한다.

- Sigmoid, tanh 경우 Xavier 초기화 방법이 효율적이다.
- ReLU계의 활성화 함수 사용 시 He 초기화 방법이 효율적이다.
- 최근의 대부분의 모델에서는 He초기화를 주로 선택한다.

Overfitting Solution

Overfitting

training data에 지나치게 fitting되어 test data에 제대로 반응하지 못하는 현상이다.

overfitting은 주로 다음 두 경우에 자주 발생한다.

- training data가 적은경우
- parameter와 feature가 많은 경우

기타 Overfitting에 대한 기타 자세한 설명은 생략한다. 이전 문서 참고바람.

Regularization

overfitting 방지를 위해 cost function을 조작하는 기법이다.

한국어로는 '일반화'로 번역된다. 정규화(normalization)와는 다르다.

정규화가 데이터 전처리 과정에서 데이터를 0~100 사이의 값으로 정규화 시키는 과정이었다면

일반화는 신경망이 범용성을 갖도록 처리하는 것이기 때문.

weight decay (L2 Regularization)

통상 ML에서 통계적 추론을 할 때는 cost function이나 error function이 작아지는 쪽으로 학습한다.

그런데 이 때 가중치 W값이 과도하게 커져 overfitting을 유발하는 경우가 있다.

특정 뉴런의 영향이 과도하게 증가했다는 것.

이에 대해 가중치가 클 수록 페널티를 부과해 오버피팅을 억제하는 기법이 L2 Regularization이다.

패널티를 부과한다는 것은 구체적으로 cost function에 1/2 * λ * W^2 을 더하는 것이다.

(여기서 λ는 Regularization 정도를 조절하는 parameter이다.)

$$W \leftarrow W - \eta\left(\frac{\partial L}{\partial W} + \lambda W\right)$$

back propagation을 통해 계산된 gradient에 위 항을 미분한 λW가 더해지므로 보정이 되는 것.

- 이 람다 값은 어떻게 정할까?

  위 링크의 Stack Overflow 답변에 의하면, 이 역시 Training data의 일부를 λ=0에서부터 값을 점점 키우면서 돌려 보면서 모델이 예측값을 어떻게 내놓는지를 관찰하며 결정해야 한다. 그리고 결정한 값에서 살짝 작게 잡아야 전체 데이터에 맞을 것...이라고 말하는 것 같다.

  더 나아가면, 람다 값을 직접 관찰하며 임의로 정해 줘야 한다는 모호함을 피하고 싶다면, Tikhonov Regularization 이라는 다른 일반화 방법에선 상수값에 대한 솔루션을 확실하게 정해줄 수 있으니 그 쪽을 권한다는 답변도 있다.

sparsity (L1 Regularization)

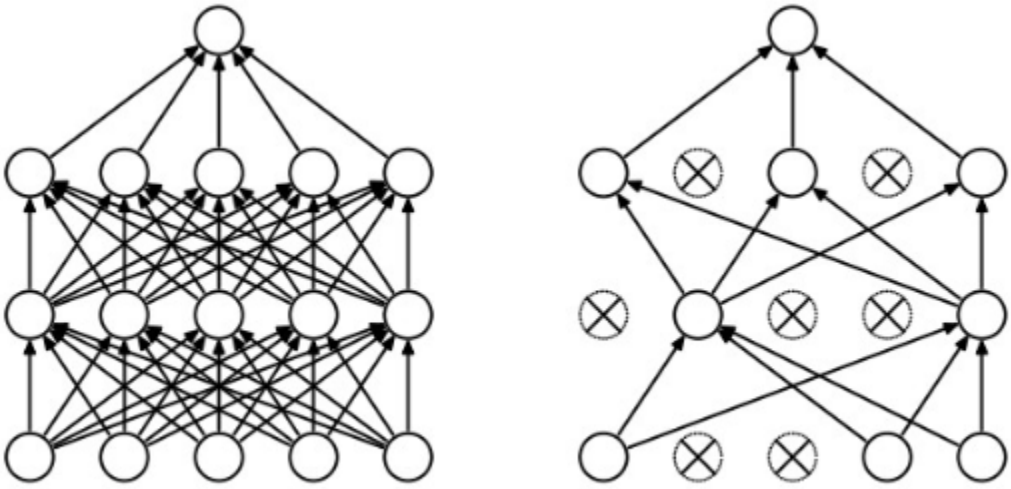
L1 에서는 2차항 대신 1차항을 더한다.

λ|w| 를 cost function에 더하는 것이다.

L2 와 달리 L1의 경우 w를 업데이트 할 때 상수값 만을 더해주게 되어 있으므로
작은 가중치들이 무시되어 몇개의 중요한 가중치만이 남게 된다고 한다.
따라서 몇개의 중요한 가중치만을 원한다면 L1이 적합하지만
미분이 불가능하다는 점에 유의하여야 하고,
따라서 주로 L2가 사용되는 편이다.

Dropout

일반적으로 DNN에서 layer의 수가 많아질 수록 학습능력이 좋아진다.
그렇지만 적절한 양의 데이터가 준비되지 않으면 overfitting에 빠질 위험이 있다.
이 경우 overfitting을 방지하기 위해 각 layer에서 임의의 뉴런들을 선택하여 drop하고
나머지 뉴런에 대해서만 학습을 진행시키는 기법이 dropout이다.



(a) Standard Neural Net (b) After applying dropout.

dropout을 통해 얻을 수 있는 이점은 다음과 같다.

Voting 효과

일정한 mini batch 기간 동안 특정 뉴런만을 택하여 줄어든 layer로 학습을 진행하면

그 때 overfitting이 발생하더라도 각각의 학습마다 임의로 overfitting이 다르게 발생하므로

이를 반복하면 평균 효과를 얻을 수 있다.

(무작위로 overfitting된 각 결과의 평균이 의도한 결과와 같아진다는 의미인 듯)

Co-adaption 회피

Co-adaption 이란 특정 뉴런에 다른 뉴런이 영향을 받는 것을 의미하는데

Regularization에서 살펴 봤듯이 특정 뉴런의 w가 과도하게 증가하면

다른 뉴런에 악영향을 끼치며 학습을 방해하는 문제가 있었다.

그런데 dropout 에서는 뉴런들을 임의로 drop하며 학습을 진행하므로

특정뉴런에 의해 다른 뉴런들이 영향을 받아 특정뉴런에 동조화 되는

Co-adaption의 위험이 적다.

즉, 데이터에 영향을 받지않는 더 강건한 시스템을 구현할 수 있는 것이다.

시그모이드를 왜 사용했는가?

- XOR 문제는 다중 퍼셉트론(MLP)를 통해 해결 가능하더라!그 많은 가중치들을 어떻게 학습시킬 수 있는가?Back Propagation각각의 가중치들이 결과에 얼마나 영향을 주었는가를 계산미분 가능한 AF(Squashing function)이 필요 → sigmoid (비선형 활성화 함수)미분이 편하다0과 1로 변환 - Activation 미분 값의 범위가

강의 요약lec10-1: 10-1 Sigmoid 보다 ReLU가 더 좋아XOR & MLPVanishing gradientactivation function**lec10-2: Weight 초기화 잘해보자**가중치 초기값이 0이거나 동일한 경우가중치 초기값이 작은 수일 경우**레퍼런스**

Optimizer

Optimizer

Momentum

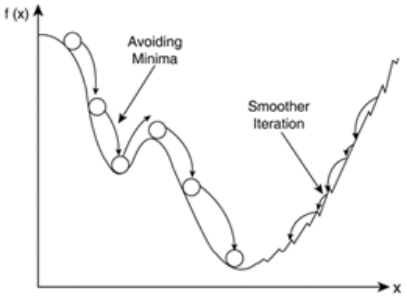
Momentum 방식은 말 그대로 Gradient Descent를 통해 이동하는 과정에 일종의 '관성'을 주는 것이다. 현재 Gradient를 통해 이동하는 방향과는 별개로, 과거에 이동했던 방식을 기억하면서 그 방향으로 일정 정도를 추가적으로 이동하는 방식이다. 수식으로 표현하면 다음과 같다. v_t를 time step t에서의 이동 벡터라고 할 때, 다음과 같은 식으로 이동을 표현할 수 있다.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

이 때, γ는 얼마나 momentum을 줄 것인지에 대한 momentum term으로서, 보통 0.9 정도의 값을 사용한다. 식을 살펴보면 과거에 얼마나 이동했는지에 대한 이동 항 v를 기억하고, 새로운 이동항을 구할 경우 과거에 이동했던 정도에 관성항만큼 곱해준 후 Gradient을 이용한 이동 step 항을 더해준다. 이렇게 할 경우 이동항 v_t 는 다음과 같은 방식으로 정리할 수 있어, Gradient들의 지수평균을 이용하여 이동한다고도 해석할 수 있다.

$$v_t = \eta \nabla_\theta J(\theta)_t + \gamma\eta \nabla_\theta J(\theta)_{t-1} + \gamma^2 \eta \nabla_\theta J(\theta)_{t-2} + \dots .$$



Optimizer

Adagrad

Adagrad(Adaptive Gradient)는 변수들을 update할 때 각각의 변수마다 step size를 다르게 설정해서 이동하는 방식이다. 이 알고리즘의 기본적인 아이디어는 '지금까지 많이 변화하지 않은 변수들은 step size를 크게 하고, 지금까지 많이 변화했던 변수들은 step size를 작게 하자' 라는 것이다. 자주 등장하거나 변화를 많이 한 변수들의 경우 optimum에 가까이 있을 확률이 높기 때문에 작은 크기로 이동하면서 세밀한 값을 조정하고, 적게 변화한 변수들은 optimum 값에 도달하기 위해서는 많이 이동해야할 확률이 높기 때문에 먼저 빠르게 loss 값을 줄이는 방향으로 이동하려는 방식이라고 생각할 수 있겠다. 특히 word2vec이나 GloVe 같이 word representation을 학습시킬 경우 단어의 등장 확률에 따라 variable의 사용 비율이 확연하게 차이나기 때문에 Adagrad와 같은 학습 방식을 이용하면 훨씬 더 좋은 성능을 거둘 수 있을 것이다.

Adagrad의 한 스텝을 수식화하여 나타내면 다음과 같다.

$$G_t = G_{t-1} + (\nabla_\theta J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_\theta J(\theta_t)$$

Optimizer

RMSProp

RMSProp은 딥러닝의 대가 제프리 힌톤이 제안한 방법으로서, Adagrad의 단점을 해결하기 위한 방법이다. Adagrad의 식에서 gradient의 제곱값을 더해나가면서 구한 G_t 부분을 합이 아니라 지수평균으로 바꾸어서 대체한 방법이다. 이렇게 대체를 할 경우 Adagrad처럼 G_t가 무한정 커지지는 않으면서 최근 변화량의 변수간 상대적인 크기 차이는 유지할 수 있다. 식으로 나타내면 다음과 같다.

$$G = \gamma G + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

$$\theta = \theta - \frac{\eta}{\sqrt{G+\epsilon}} \cdot \nabla_\theta J(\theta_t)$$

Optimizer

Adam

Adam (Adaptive Moment Estimation)은 RMSProp과 Momentum 방식을 합친 것 같은 알고리즘이다. 이 방식에서는 Momentum 방식과 유사하게 지금까지 계산해온 기울기의 지수평균을 저장하며, RMSProp과 유사하게 기울기의 제곱값의 지수평균을 저장한다.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta))^2$$

다만, Adam에서는 m과 v가 처음에 0으로 초기화되어 있기 때문에 학습의 초반부에서는 m_t, v_t가 0에 가깝게 bias 되어있을 것이라고 판단하여 이를 unbiased 하게 만들어주는 작업을 거친다. m_t 와 v_t의 식을 $\sum$ 형태로 펼친 후 양변에 expectation을 씌워서 정리해보면, 다음과 같은 보정을 통해 unbiased 된 expectation을 얻을 수 있다. 이 보정된 expectation들을 가지고 gradient가 들어갈 자리에 $\hat{m}_t$, G_t가 들어갈 자리에 $\hat{v}_t$를 넣어 계산을 진행한다.

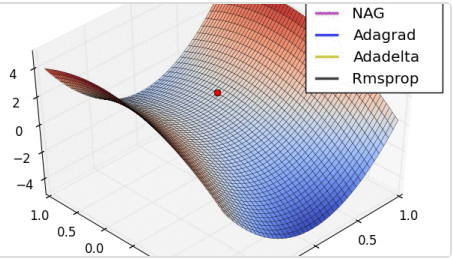
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t+\epsilon}}\hat{m}_t$$

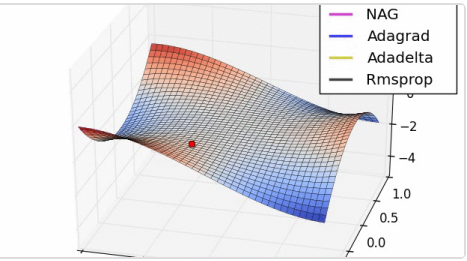
보통 β_1 로는 0.9, β_2로는 0.999, ϵ 으로는 10^{-8} 정도의 값을 사용한다고 한다.

http://i.imgur.com/2dKCQHh.gif?1



http://i.imgur.com/pD0hWu5.gif?1

http://i.imgur.com/NKsFHJb.gif?1



레퍼런스 ^—^

< 1. Activation Function>

1. < 7가지 활성화 함수 비교하기>, https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/
2. 어떤 활성화 함수를 써야 하는가?(영상), https://youtu.be/-7scQpJT7uo
3. 활성화 함수의 완벽한 가이드, https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044
4. 왜 zero-centered 형태의 활성화 함수를 써야 할까? https://rohanvarma.me/inputnormalization/

10-1

머신 러닝 - 활성화 함수(activation function)들의 특징과 코드 구현하기: Sigmoid, Tang, ReLU

3.8. 다층 퍼셉트론 (Multilayer Perceptron)

[딥러닝] 뉴럴 네트워크 Part. 5 - 새로운 활성화 함수

<3. regularization / dropout>

https://umbum.dev/222

https://blog.naver.com/laonple/220527647084