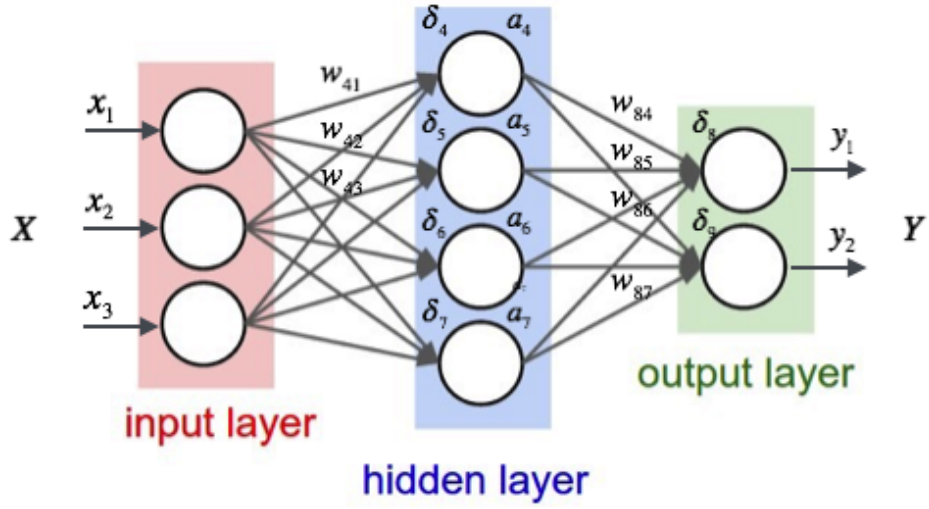
딥러닝 전반부

이 문서에서는 크게 3가지로 나뉘어서 설명이 되어있습니다. 딥러닝의 전반적인 부분을 담은 introduction, 딥러닝의 핵심부분인 backward propagation, 그리고 마지막으로 forward propagation 에 대해 설명이 있습니다. 밑에 내용은 자세한 수식이나 원리까지 설명되어 있지않지만 전반적인 딥러닝의 구성, 작동방법 등이 포함되어 있습니다.

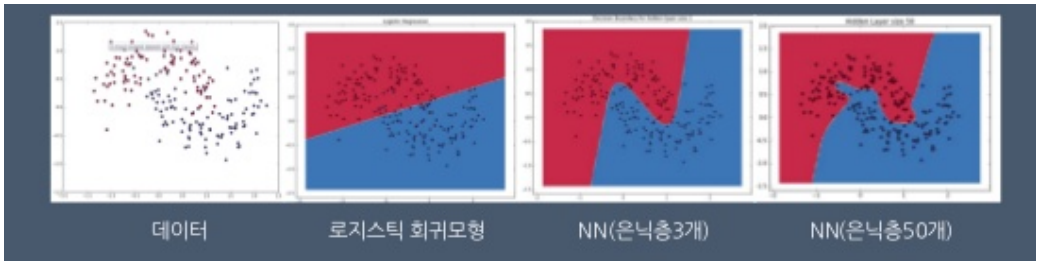
1. Introduction

딥러닝 < 머신러닝. 많은 사람들이 딥러닝을 완전히 다른 분야로 알지만 딥러닝은 머신러닝에 속해있음. 머신러닝의 한기법. 심층 신경망을 이용한 기법이다. 심층 신경망이란 영어로 deep neural network 이다. 직관적으로 해석해도 깊은 신경망이라는걸 알수있는데 이는 neural network 가 다른 기법들에 비해 deep 하다 즉 학습이 반복된다 라는것. 사실 딥러닝은 그렇게 각광받는 분야가 아니었지만 점차 발전해가며 지금 상태에 도달함. 그 대표적 예로 1986년 Backpropagation 이라는 뒤에서 앞으로 data 가 model 을 찾아내는, 결과를 보고 뒤로 가면서 몇 factor 를 조절하며 조정하는 법을 고안한 것. 이때 간단히 back propagation 에 대해 말하자면 뒤로 가면서 weight, bias 등 우리가 가장 흥미있어하는 factor 를 재계산하고 갱신하면서 모델의 성능을 높이는 것. 딥러닝이 각광받는 학문이 아니였던 이유는 층계수가 깊어질 때 마다 원하는 값을 얻어내기가 힘들었음. 그에 해결책인 효과적 알고리즘이 고안되기까지 많은 시간이 흘렀지만 결국 대표적 문제들인 time complexity, vanishing gradient, overfitting 등의 해결방안이 나오기 시작하여 딥러닝이 주목 받기 시작함. 이에대한 해결 방안은 밑에 서술할 예정. 딥러닝은 입력층 , 은닉층, 출력층에서 은닉층이 Deep, 즉 2개의 layer 이상 있는것들을 칭함.



이 그림을 보면 알수 있듯이 input 으로 들어온 데이터가 hidden data 를 거쳐서 ouput 으로 간다. 이때 hidden 이 깊어지면 즉 layer 가 2개이상이면 deep learning.

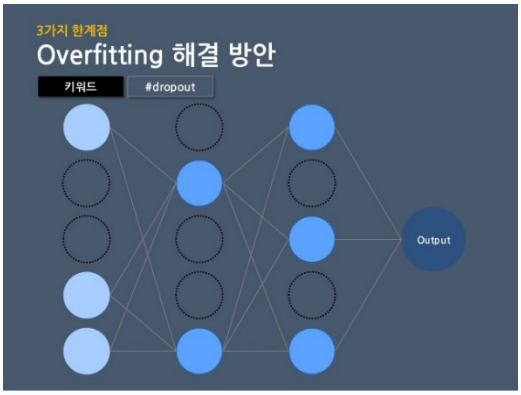
딥러닝의 핵심은 은닉층의 여러 결합(비선형 활성함수)로 비선형 영영역 표현하는 것.



이 그래프를 보면 은닉층이 늘어날수록 classification 이 더 정확한 것을 볼 수 있음. 그렇다고 무조건 은닉층을 깊게 만들어서 레이어를 더하고 더하면 높은 성능의 모델을만들수 있지않은가? 세상은 마냥 우리에게 친절하지 않다. 레이어를 계속 더하면 major 한 문제점 3가지가 돋보이게 된다. 이 위에 언급한 time complexity(느리다), vanishing gradient(덜하다) 그리고 overfitting(과하다)가 그 문제점이다. 이것에 대한 해결방안은 다음과 같음:

3가지의 문제의 각각의 현재로써의 해결방안:

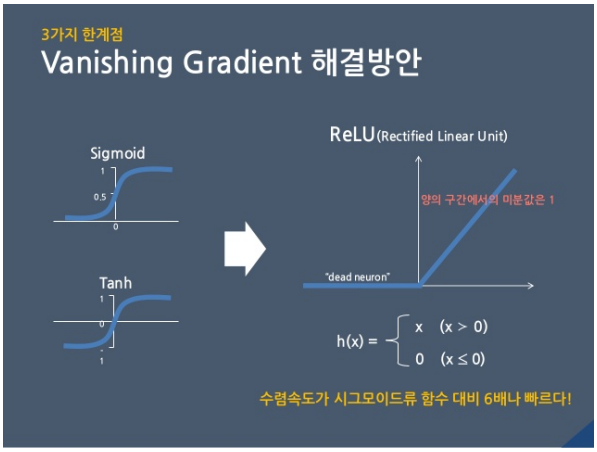
(1) Overfitting – too many neurons. 너무많은 신경. 좋지않은 network. Not enough data. 즉 결론적으로 학습 데이터(train data) 에 '만' 잘맞는 모델이 생김. 그러면 이것에 대한 해결방안은 무엇일까? 그건 Drop out 이라는 방법이다. 많은 neuron 들이 network 안에 있기에 복잡한 모델을 더 단순하게 만들기 위해서 random 한 확률로 neuron 들을 뽑아 drop out, 즉 버린다. 물론 모든 neuron 을 학습하는것보다 train set 에 대한 설명력이 떨어지지만 성능은 말이 안되게 개선된다. Drop out 할 neuron 과 살려둘 neuron 은 Bernoulli에의해 결정되기 때문에 뉴런의 개수가 n 개라면, 살아남는 neuron 은 독립적인 binomial 에 의해 결정된다.



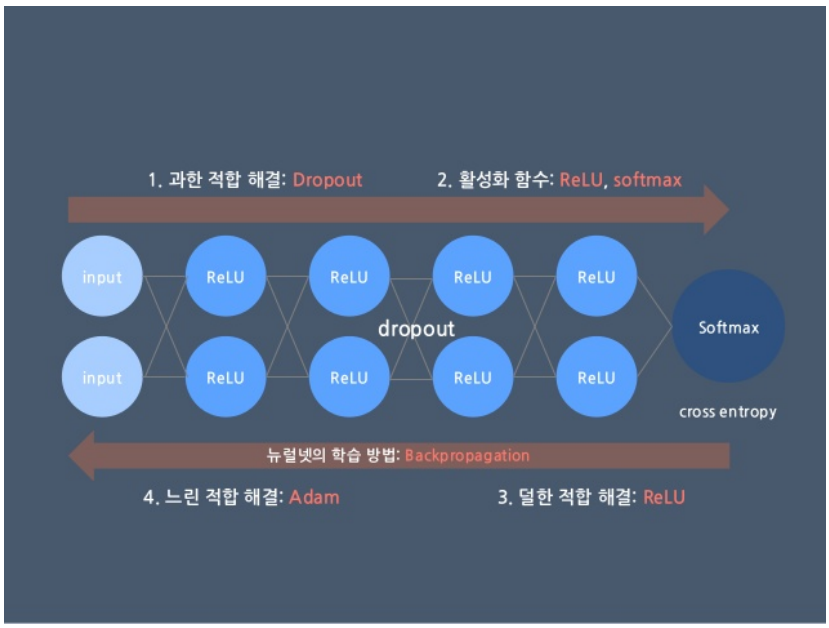
(2) Too slow- 우리의 목적은 optimize 된 bias와 weight 를 찾는 것. 그것을 어떻게 찾는가는 현재 가중치로 산출된 cost function의 gradient 를 구해서 cost function 의 값을 낮추는 방향으로 update 하는 누구나 다아는 그 방법, gradient descent 를 이용한다. 하지만 딥러닝에선 optimizer 를 이용해 speed problem 을 완화한다. 그종류로는 SGD(Stochastic Gradient descent) 확률적경사하강법이 있는데 이는 확률적으로 무작위로 골라낸 데이터에 대해 수행하는 경사하강법이다. 이에 대한 자세한 수식과 내용은 담지 않지만 전반적으로 파악을 하자면 보통의 경사하강법과는 다르게 이름 그대로 확률적으로 랜덤하게

데이터에 대해 경사하강법을 수행하는 것이다. 두드러지는 단점으로는 랜덤확률이다 보니 왔다리갔다리 하는 경우가 있고 수렴 안정성이 낮아 최적해를 찾을 수 없는 확률이 높다. 이에 대한 대안으로는 학습속도와 운동량을 고려하는 것. 다른 optimizer 로 속도에 중심 중 둔 것은 adagrad (가본곳은 더 정밀하게, 안가본곳은 대충훑기) , adadelta (계속 정밀 정밀 모든 것을 탐색하지 않고 어느 한 선에서 끊어주는 것) RMSprop(adagrad 에서 발전해서 가본곳은 더 정밀하게 가지만 그순간에 gradient 를 재계산하여 속도를 조절함) 등이 있고 운동량에 중심을 둔 것은 momentum(물리에서 따온 그것, 영아닌길이 아닌 진짜 길 같은 곳으로), NAG(momentum 에서 나아가 순간 순간 gradient 를 다시계산하여 운동량 높이기) 등이 있는데 현재로써 가장 보편적이고 성능이 우수한 optimizer 는 adam 이긴하지만 다양한 알고리즘을 이용하여 많은 모델을 적합하는 것이 중요.

(3) Vanishing gradient – 말그대로 gradient 가 사라지는 것을 의미하는데 이에 해결책은 Relu activation 함수를 이용하는것. Relu는 입력값이 음수일 때 0을 출력하고, 그렇지 않으면 자기자신을 출력하는 함수이다. 미분계수가 0과 1의 값만 가지게 되므로, 학습속도 정체의 문제가 발생하지 않는다는 장점을 가짐. 그러나 음수에서 미분계수가 0이기 때문에, 음수의 입력값이 주어질 때 학습이 이루어지지 않는다는 단점이 있다.



요약그림:



2. Backward propagation

결과, 즉 아웃풋이 나온후 오차가 관측됨. 이때 틀린 정도의 기울기를 거꾸로 전달하는 것이 백프로파게이션. 그후 weight, bias 를 다시 갱신함. 우리의 목적은 최적화된 weight 와 bias 를 찾는 것. 즉 cost function 의 기울기를 구해서 cost 를 최소화 할수 있는방향으로 갱신. 신경망은 구체적으로 어떻게 학습되는가? 신경망이 학습하는 것을 순서대로 나열해보면, Input layer로부터 각 layer를 지나치며 weight들을 곱하고, activation function을 지난 다음, 마지막 output layer에서 오차 값을 계산한 이후(feed forward), 업데이트를 위해 순차적으로 거꾸로(back) 돌아가야(propagation)함. 그러면서 weight 값과 bias 를 다시 갱신하는 것..

밑에를 보면 더 자세한 수식과 알고리즘의 순서를 알수있다.

1. 전방향 연산(Forwardpropagate)을 수행해서 Hidden Layer 1-L_2, Hidden Layer 2-L_3, 계속해서 최종적으로 Output Layer-L_n까지 노드들의 activation을 계산한다.
2. Output Layer-n_l의 각각의 노드(output unit) i에 대해서 아래와 같이 에러값($\delta_i^{(n_l)}$)을 계산한다.
$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2}||y - h_{W,b}(x)||^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$
3. Output Layer를 제외한 각각의 노드(output unit) i에 대해서 아래와 같이 에러값($\delta_i^{(n_l)}$)을 계산한다.
$\delta_i^{(n_l)} = (\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}) f'(z_i^{(n_l)})$
4. Neural Networks의 파라미터들(parameters)-W, b-에 대한 미분값(derivative)를 아래와 같이 계산한다.
$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)}$
$\frac{\partial}{\partial b_{ij}^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}$

코스트펑션을 최소화 시키기 위해 에러를 아웃풋에서 구해서 인풋을 변환시킬수없으니 그사이 은닉층들에서 weight 를 미적의 chain rule 로 계산하며 최소화 시키며 갱신시키고 결과를 개선시키는 것. 자세한 수식과 설명은 상당히 수학적이고 많은이론이 들어가기에 참고를 위해 링크를 첨부함: https://kakalabblog.wordpress.com/2017/04/04/understanding-of-backpropagation/

3. Forward propagation

Activation function 을 each layer 에 적용시켜 아웃풋까지 반복. 그후 error 를 구하는 것. 가장 큰 activate neuron 을 가진 것을 classify 한다. 이것은 은닉층에서 이루어짐. 그후 아웃풋과 은닉층 사이에 classify함.

신경망 모델에서의 계산 과정을 정리하면, 1)입력층에 데이터가 입력되면 각 뉴런들은 간선에 부여된 가중치를 고려하여 다음 층으로 데이터를 전달한다. 2)데이터를 전달받은 뉴런들은 다음 층으로 데이터를 전달하기 위해 활성화 함수를 통해 전달할 값을 계산하고, 3)다시 다음 간선에 부여된 가중치를 고려하여 데이터를 전달한다. 4)이러한 과정이 출력층까지 이루어지면 출력층 또한 활성화 함수를 통해 출력할 값을 계산하여 최종적인 값을 출력한다

활성화 함수로는 relu 함수를 주로 사용한다. 활성화 함수를 쓰지 않으면 linear regression 과 똑같기 때문에 은닉층에서 사용한다.

출력층에서는 softmax 라는 함수를 사용하는데 softmax 함수는 분류를 위해 사용된다. 이게 무슨 뜻이냐면, 쉬운 예를 들어보자. 우리는 어떠한 사진이 고양이인지를 인식할수있게 모델을 잘 학습시키고싶다. 그래서 열심히 딥러닝 기법을 사용하여 학습시키고 이제 데이터를 줘서 이게 고양이인지 아닌지 결정해야하는데 이때 softmax 를 사용하여 출력(즉 아웃풋)한다. 더 자세히 말하자면 softmax 함수는 출력층에 전파된 입력값들을 벡터로 받아 각 성분의 크기 비율 (출력 벡터의 성분합이 1) 을 출력하는 함수이다

밑에는 총정리 그림: