Basics of Deep Learning
2020년 봄 학기

2020년 02월 29일
https://www.github.com/KU-BIG/KUBIG_2020_Spring

1. 딥러닝이란?

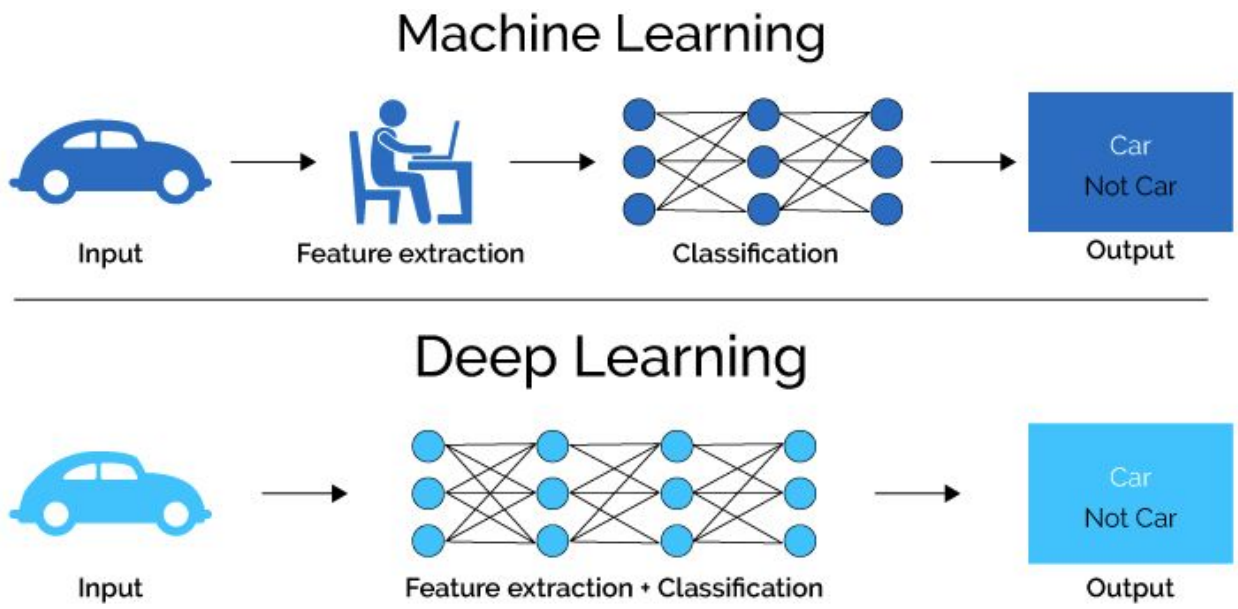
딥러닝의 정의에 대한 논의는 첫 세션때 배웠던 머신러닝과의 비교에서 부터 출발한다.

(1) 딥러닝 vs. 머신러닝

딥러닝과 머신러닝은 아예 별개의 것은 아니다. 딥러닝은 머신러닝의 일종으로 해석될 수 있다. 하지만 딥러닝은 더 많은 데이터를 처리할 때 유리하고, 인간의 힘을 덜 들여도 된다는 장점을 가지고 있다.

딥러닝과 머신러닝의 차이에 대한 논의는 아래 세 가지 측면에서 진행해 볼 것이다.

1) 데이터: 딥러닝은 머신러닝에 비해 많은 labelled 샘플을 필요로 한다.
2) 하드웨어(컴퓨터): 많은 양의 데이터를 처리하는 만큼, 딥러닝은 GPU와 같은 하드웨어를 필요로 한다.(GPU는 실제로 구비하려면 굉장히 비싸지만, Google Colab에서 무료로 사용할 수 있다!)
3) 변수 추출(Feature Extraction): 머신러닝에서는 알고리즘 안에 input 데이터를 넣기 전에, 데이터의 복잡성을 줄이고 패턴을 더 명확하게 하기 위해 domain knowledge(일종의 사전 지식)을 넣어주는 일을 말한다. 머신러닝은 features 사이의 패턴과 상관관계를 인간이 직접 넣어준다.. 즉, Feature extraction의 과정을 인간이 직접 하기 때문에, 다소 복잡한 사전작업이 필요하다. 반면에 딥러닝은 feature extraction 또한 겹겹히 쌓인 레이어들을 이용하여 진행한다. 따라서 인간이 할 일은 머신러닝에 비해 현저히 줄어든다. 다만 내부에서 어떤 일이 일어나는지 인간이 파악하기 어려워 'Black Box'라고 불리기도 한다. 아래 그림을 통해 더욱 자세히 이해할 수 있을 것이다.

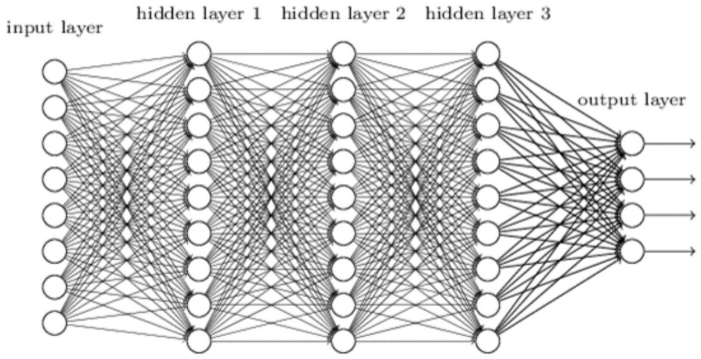


출처:https://www.quora.com/What-is-the-difference-between-deep-learning-and-usual-machine-learning

- 대표적인 Deep Learning Architecture

  그 다음으로는, 대표적인 Deep Learning의 구조에 대해 알아보자. 이 읽기자료에서는 DNN에 대해 자세히 다룰 예정이다.

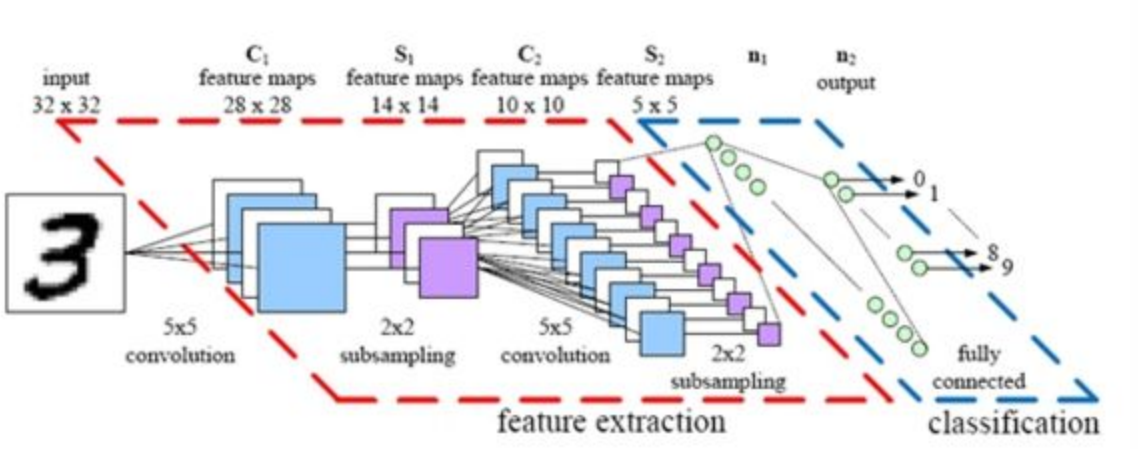
  - DNN(Deep Neural Networks)
- Hidden layer을 두개 이상 지닌 학습방법으로, 컴퓨터가 스스로 분류 레이블을 만들고, 공간을 왜곡하고, 데이터를 구분하는 과정의 반복을 통해 최적의 결과를 도출해내는 알고리즘이다. 아래 CNN과 RNN은 DNN의 응용 알고리즘이라고 볼 수 있다.



출처:http://physics2.mju.ac.kr/juhapruwp/?p=1517

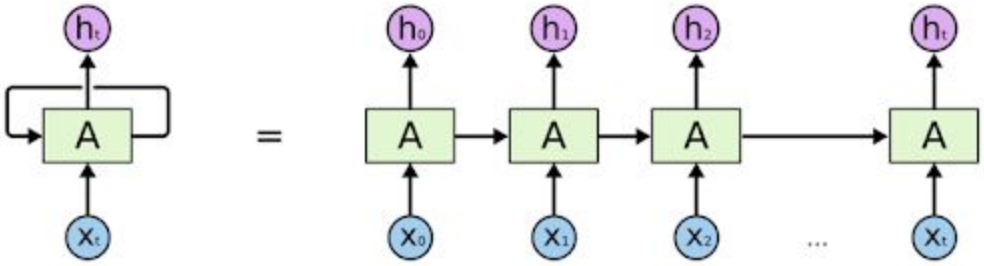
- CNN(Convolutional Neural Networks)

- 최근 이미지 인식 분야에서 많이 사용되고 있는 알고리즘으로, 인간의 시신경 구조를 모방하였다. Input 데이터의 일부 특징(feature)을 모아 패턴 발견을 통해 output을 구현하는 것이 특징이다. 전체 input을 연결해서 다 입력할 경우, 이 패턴 발견이 오히려 더 어려워 지기 때문에 띄어서 input을 입력하는 방법을 이용한다. 레이어를 추가해 갈 때마다 추상화의 정도 높아지며, 최종적으로 타겟으로 하는 예측을 진행할 수 있다.



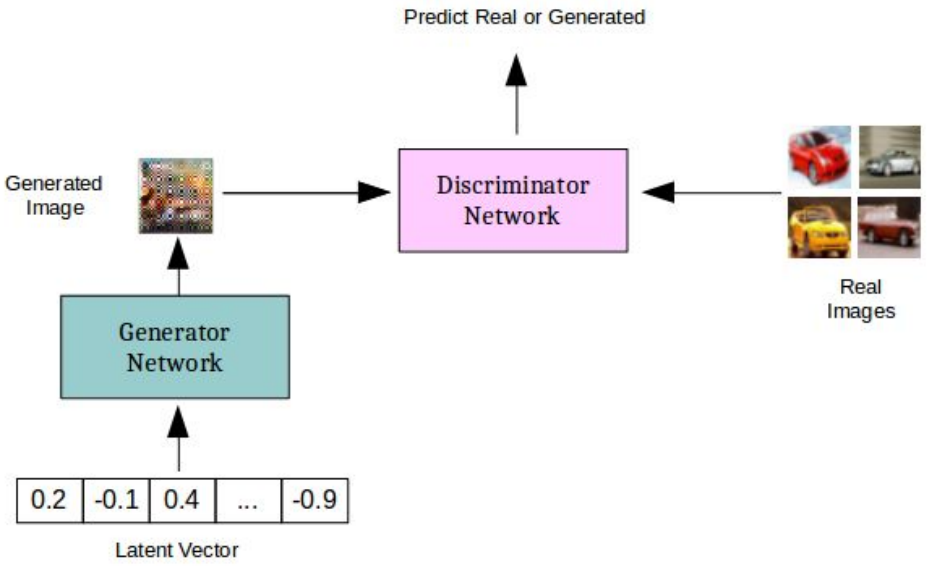
- RNN(Recurrent Neural Networks)

- 순환신경망으로 번역할 수 있으며, 바로 전의 데이터를 순차적으로 기억하는 알고리즘을 뜻한다. 보통 시계열 데이터 분석에 사용한다.
- 악보/음악 만드는 인공지능 개발에도 사용된다.



- GAN(Generative Adversarial Networks)

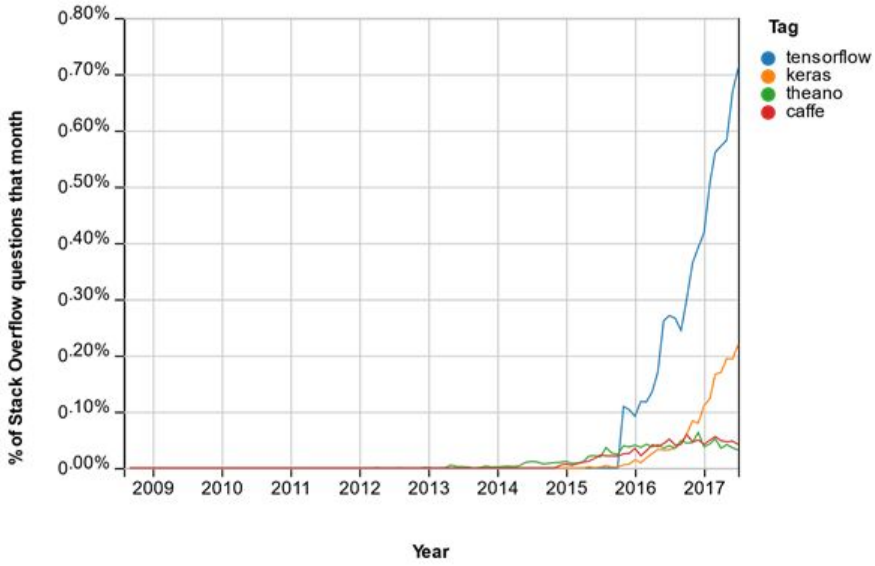
- GAN은 '생성적 적대 신경망'의 뜻으로, 판별자(Discriminator)와 생성자(Generator)가 경쟁하며 결과물을 만들어가는 알고리즘이다.
- 이미지 분석/생성에 많이 사용된다.



출처:https://medium.com/@utk.is.here/keep-calm-and-train-a-gan-pitfalls-and-tips-on-training-generative-adversarial-networks-edd529764aa9

(2) 대표적인 deep learning platform

그렇다면 위와 같은 다양한 딥러닝 알고리즘을 어떻게 구현할 수 있을까? 딥러닝을 구현할 수 있는 플랫폼은 다양하게 발전해 왔으며, 현재는 TensorFlow와 Keras를 가장 많이 사용하는 추세이다.



출처: https://www.guru99.com/deep-learning-libraries.html

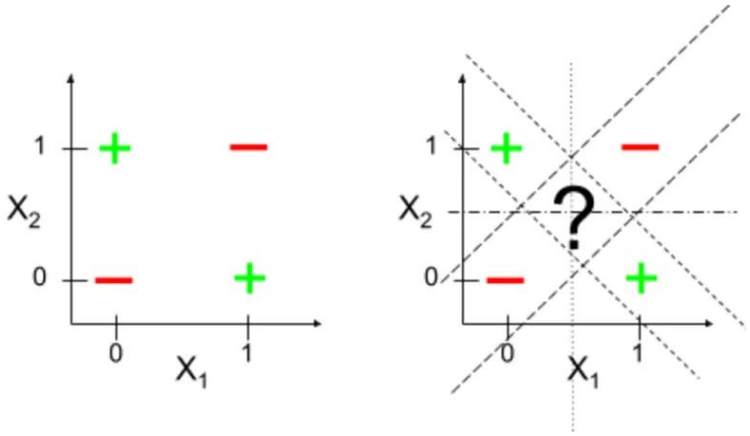
- 2004: Torch
- 2008: theano
- 2013: decaf/caffe(a berkeley vision project)
- 2014: Microsoft Azure, cuDNN
- 2015~: **TensorFlow(가장 많이 씀)**, Keras, CNTK, Lasagne, the veles

TensorFlow와 Keras는 딥러닝 과제를 하며 사용해 볼 기회가 있을 것이다.

(3) 딥러닝의 역사

최근 딥러닝이 비약적으로 발전하며, 딥러닝을 '신기술','신개념'으로 인식하는 경향이 많다. 하지만 딥러닝은 1940년대에 처음 고안된, 역사가 깊은 개념이다. 딥러닝은 인간의 신경 구조를 컴퓨터에 적용할 수 있을 것이라는 아이디어에서 출발했다. 이는 Perceptron의 개념이 인간의 뉴런에서 영향을 받은 점에서 알 수 있다. 하지만 단순한 perceptron의 개념은 XOR 문제와 같은 간단한 문제도 해결하지 못했고, 이에 제 1차 AI 빙하기가 오게 된다.

(* XOR 문제: eXclusive OR 의 줄임말로, 아래 그림에서 같은것끼리 분류하려 할 때 하나의 선으로는 분류가 어렵다. 따라서 레이어를 추가하여 이를 명확히 분류(classification)하고자 했다. 아래에서 더 자세한 설명이 이어진다.)

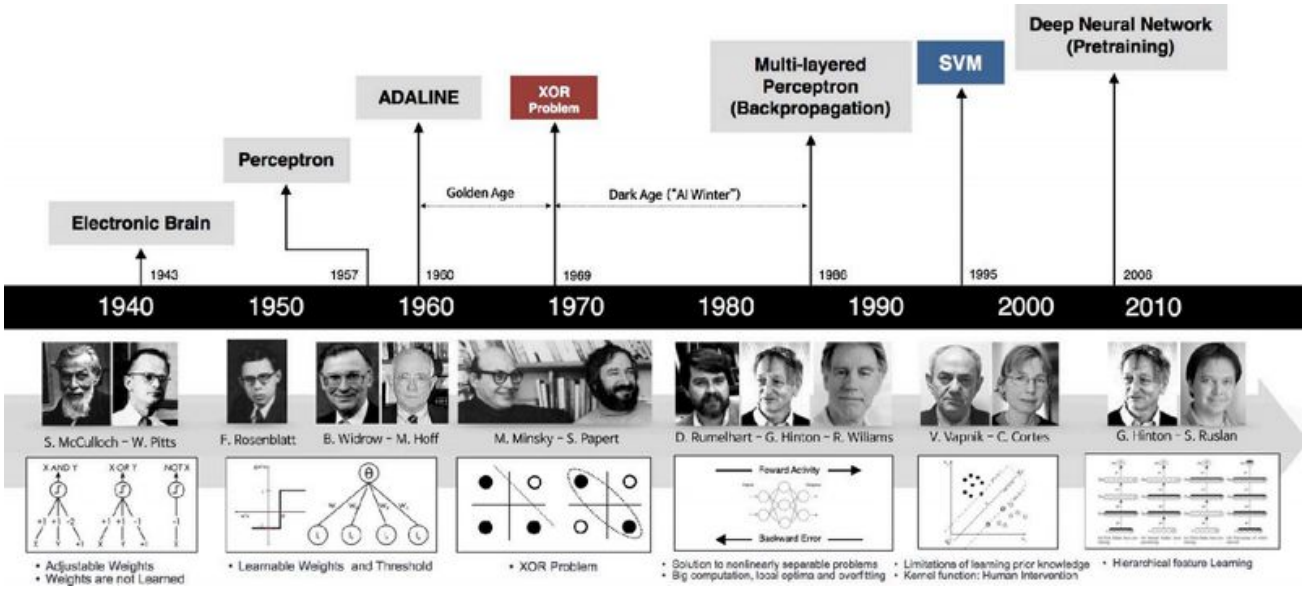


출처: https://www.tech-quantum.com/solving-xor-problem-using-neural-network-c/

이와 같은 문제를 해결하기 위해 Multilayer Perceptron(MLP)이 고안된다. Input과 output 사이 1개 이상의 hidden layer를 추가하여 분류 능력을 향상하는 방법으로 이해할 수 있다. 하지만 hidden layer가 늘어날수록 weight의 개수도 증가하게 되며, 학습을 시키기가 어려워진다.

Gradient descent 방법을 활용하여 학습을 진행하지만 이는 학습을 하면 할 수록 성능이 떨어지는 vanishing gradient problem을 유발한다. Vanishing gradient problem은 급속도로 weight를 0에 가까이 만들어 학습을 어렵게 만든다. 이를 해결하지 못해, AI의 2차 빙하기가 오게 된다.

이 문제는 Sigmoid 함수를 ReLU 함수로 대체하고, dropout 기법을 사용하며 해결하게 된다. 두 개념에 대해서는 추후 더 자세히 살펴보도록 하겠다.



2. DNN(Deep Neural Network)

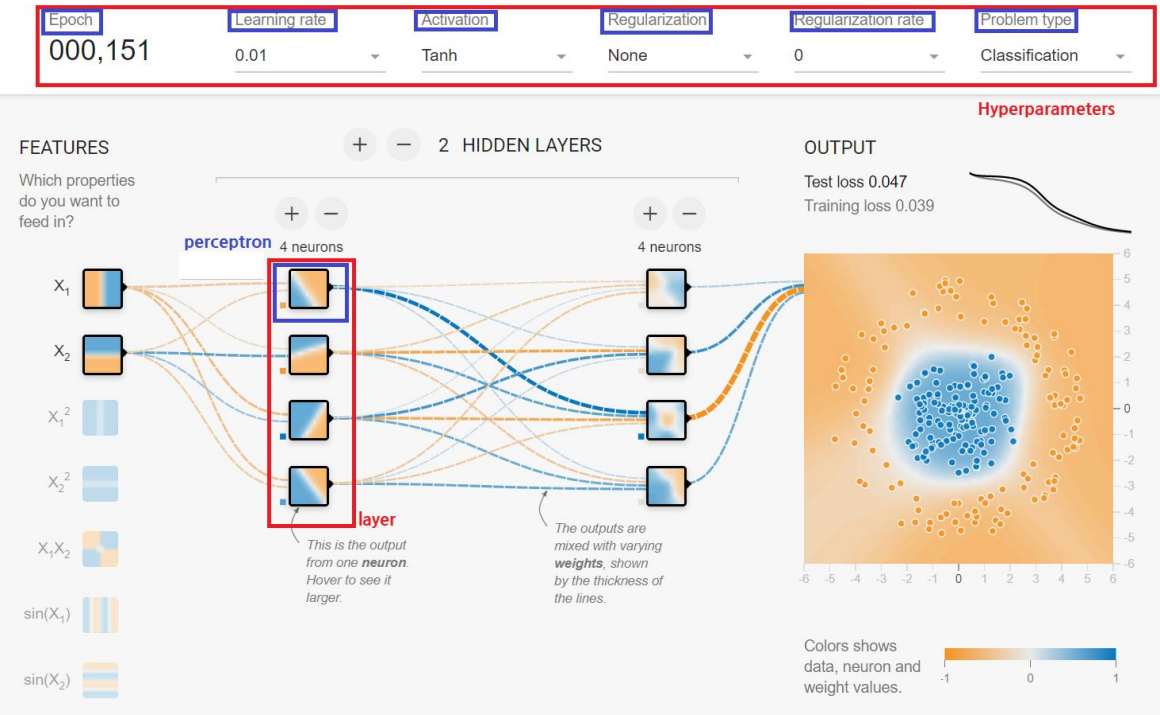
(1) DNN의 뜻과 의의

사진으로 글자를 찍으면 이를 텍스트로 변환하는 서비스를 경험해 본 적이 있을 것이다. 과연 기계가 어떤 방식으로 이러한 글자들을 인식하여 텍스트로 변환해 줄 수 있는 것일까? 그것은 기계에 글자를 인식하는 신경망을 달아주었기 때문이다.

쉽게 말해 이를 기계의 뇌라고 생각할 수 있다. 여기서, 더 훌륭한 성능을 내기 위해 여러 연산 단계를 거치게끔 한 것이 Deep Neural Network, 말 그대로 깊은, 심층신경망이다. 결국 DNN은 Neural Network의 한 종류인 셈이다. '신경망' 이라는 단어에서도 얼추 생각할 수 있듯이, 이는

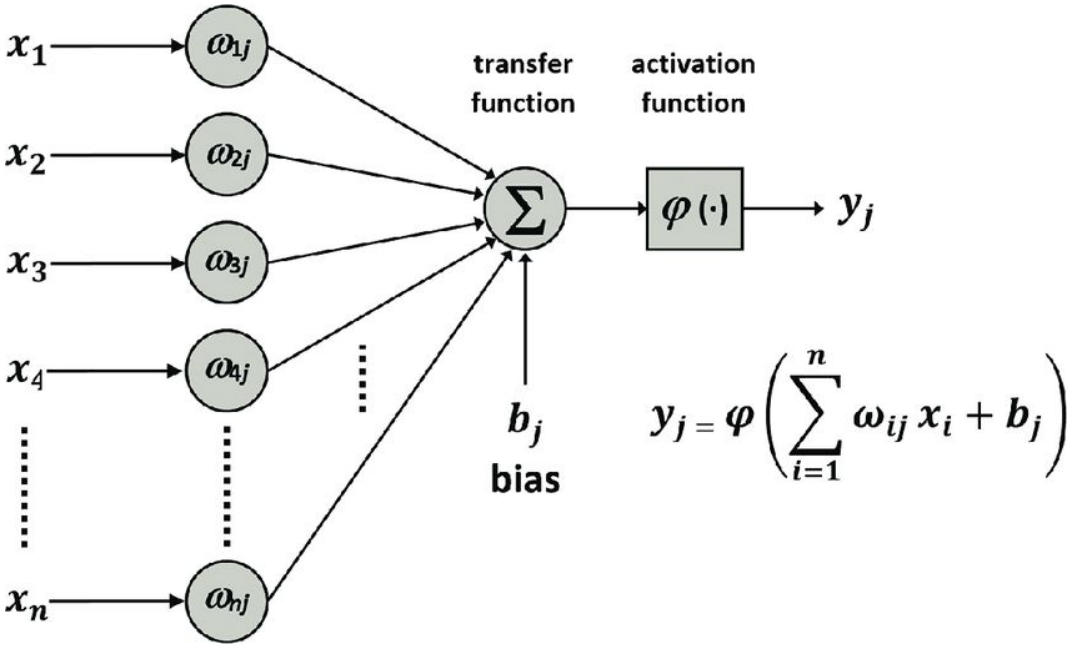
뇌의 구조를 모방하여 뇌와 비슷한 성능을 기계가 구현할 수 있게 해 주는 역할을 한다. 그렇다면 당연히 정확하게 신경망을 구축할 수록 높은 성능을 뽑아낼 수 있을 것이라 예측할 수 있다.

(2) Neural Network의 구조 및 구성요소



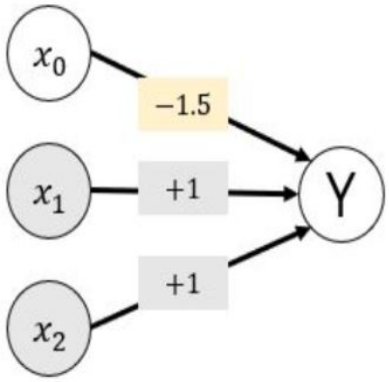
Neural Network는 기본적으로 위와 같은 구조로 이루어져 있고, 위의 구조에서 Layer 층이 두 개 이상인 경우를 통틀어 Deep Learning 이라고 한다. 또한, Layer층이 다수인 Neural Network를 DNN 이라고 한다. 기계학습에 앞서 설정해주어야 할 여러 요소들이 있는데, 이런 요소들을 Hyperparameter 라고 하며, 위 빨간 박스의 Epoch, Activation(Activation Function), Learning rate 등이 이에 속한다. 이를 하나 하나 풀어가 보기 전에, 먼저 가장 기본 단위인 Perceptron에 대하여 알아보자.

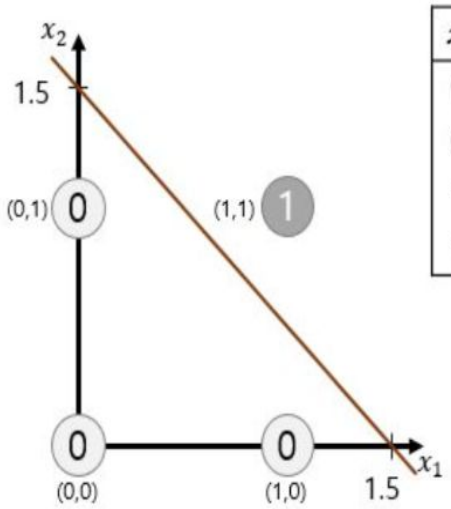
(3) Perceptron



인공지능의 기본 단위는 Perceptron 이다. 그런 만큼, 퍼셉트론에 대한 이해는 딥러닝에 있어서 첫 단추라고도 볼 수 있을 정도로 중요하다. 퍼셉트론은 뇌로 따지면 뇌세포 하나와 같은데, 위 그림과 같이 Input node(동그라미 하나) 와 Output node(사각형) 그리고 Input node와 output node를 이어주는 arc의 세 가지로 이루어져 있다.
Input node와 output node는 각각의 weight(가중치)를 가지고 연결되어 있으며, 이 가중치는 곧 노드 하나의 중요도와 같다. 최종적으로, 퍼셉트론은 인간의 뉴런이 역치 이상의 자극을 받으면 활성화되듯이, Input node의 가중치와 입력치를 모두 곱한 값이 임계치보다 크면 1을 출력하고, 그렇지 않다면 0을 출력한다. 여기에, bias라는 개발자 재량의 값(Default는 0)을 추가하여 (입력치 * 가중치 + bias) > 임계치 일 때, 1을 뱉어내게 된다.

다음 그림은 간단한 분류 과정의 예를 나타낸 것이다.





x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

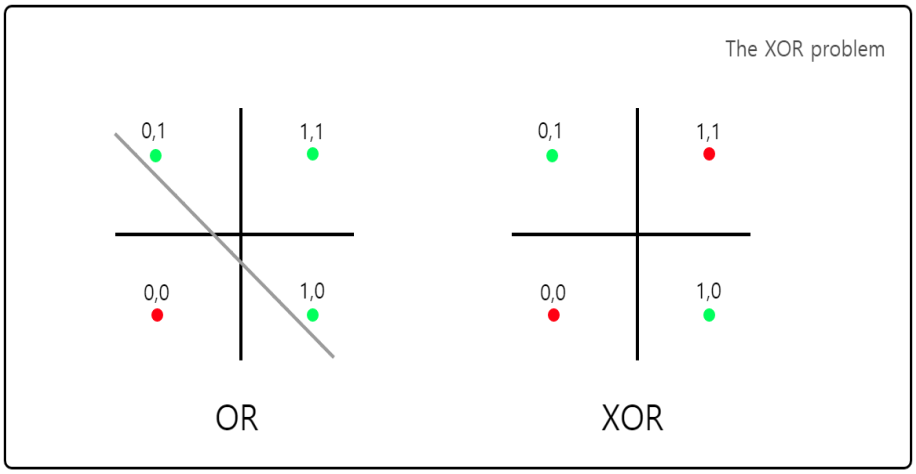
이와 같이, 최적의 퍼셉트론 모델을 만들기 위해서는 올바른 Weight와 bias를 설정해야 한다. 이 때 사용하는 것이 경사 하강법(Gradient Descent) 이며, 이는 y의 예측값과 실제 y값의 차이를 최소화하는 모델이다.

$$w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}^2, \ \lambda : \text{learning rate (학습 속도)}$$

$$\text{Gradient descent} : w_j^{(k+1)} = w_j^{(k)} + \lambda \frac{\partial}{\partial w_j} \|y_i - \hat{y}_i^{(k)}\|_2^2$$

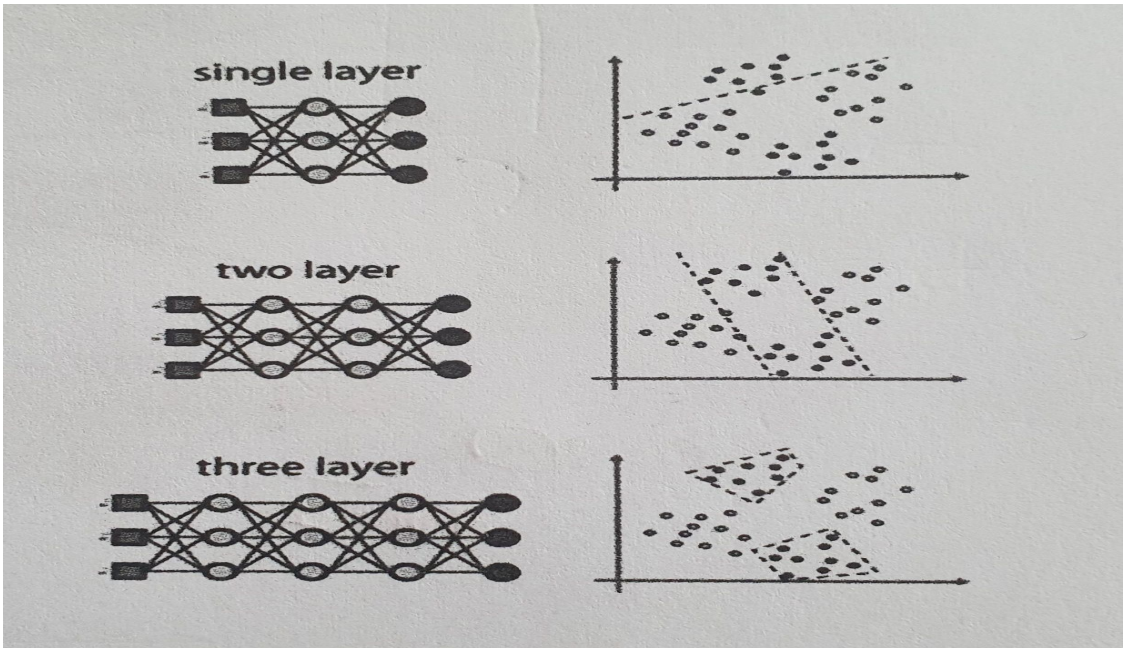
- **MLP(MultiLayer Perceptron)**

이러한 Perceptron에는 치명적인 단점이 있었으니, 바로 모델의 형태가 너무나도 단순하여 간단한 Decision Boundary(Class를 나누는 기준 선)생성 정도의 문제 외에는 해결을 할 수가 없다는 것이다. 하지만 대부분의 경우는 기준선 하나만으로 분류를 완벽하게 할 수 없는 경우가 많은데, 대표적으로 XOR Problem이라 불리는 문제가 있다.



위 문제는 Minskey & Papert(the book 'Perceptrons' 1969) 에서 소개된 XOR Problem이다. OR의 경우는 기준선 한 개로 Classification이 가능하나, XOR의 경우 어떤 방법을 써도 기준선 한 개로 Classification을 수행할 수 없다. 이러한 상황은, Decision Boundary가 여러개여야만 해결이 된다.

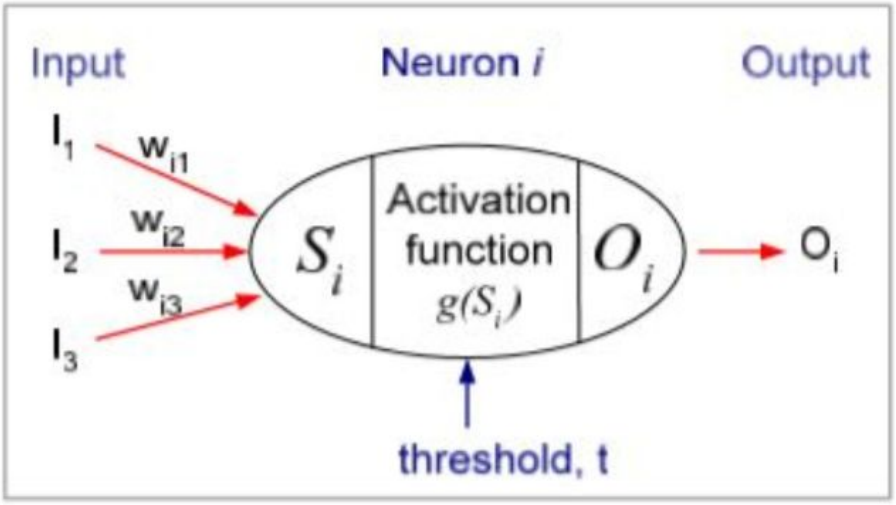
그래서 나온 개념이 MLP 인데, 이는 기존의 Perceptron이 Input layer와 Output layer만으로 이루어진 구조와는 달리, Input layer와 Output layer사이에 여러 개의 Hidden Layer를 추가한 Perceptron 구조이다. 여기서 Hidden Layer의 층 수가 곧 Decision Boundary의 개수가 된다. 이론상으로는 Hidden Layer의 수가 많으면 많을수록 더 정교한 분류를 할 수 있다. 다만, 이 경우에는 사실 '과적합(Overfitting)' 이라는 장애물이 존재한다. 이에 대해서는 나중에 알아보도록 하겠다.



위의 그림은, 최초의 Perceptron구조로는 해결할 수 없는 분류를 MLP모델로 해결한 경우이다. 여기서 한 가지 의문이 들 수 있다. 아무리 Decision Boundary가 많다고 해도, 결국은 선형인데 그렇다면 어느정도는 한계가 있지 않을까? 그러한 문제를 해결하기 위해 나온 개념이 Activation Function이다.

- **Activation Function(활성화 함수)**

선형 뿐 아닌 좀 더 유연한 Boundary를 얻고 싶다면 Activation Function을 이용하면 된다. 각 Node에서 선형 결합의 결과를 Output으로 가지는 것이 MLP인데, 이 Output의 값을 그대로 비선형적인 활성화 함수에 넣어주면 비선형 Boundary가 탄생한다. 이를 통해 선형의 Boundary를 찌그러뜨릴 수 있다. 하지만 약간 연산량이 늘어난다는 단점이 있다.



위는 Perceptron에 Activation Function을 적용한 예이다. 이와 같이, Activation Function은 각 Perceptron마다 적용되어 있다. 학습된 Weight을 토대로 선형결합한 결과인 Si를 Activation Function에 적용하여 Output인 Oi를 뽑아 내게 된다. Activation Function에도 여러 종류가 있고, 또 그 장단점이 다 다르다.

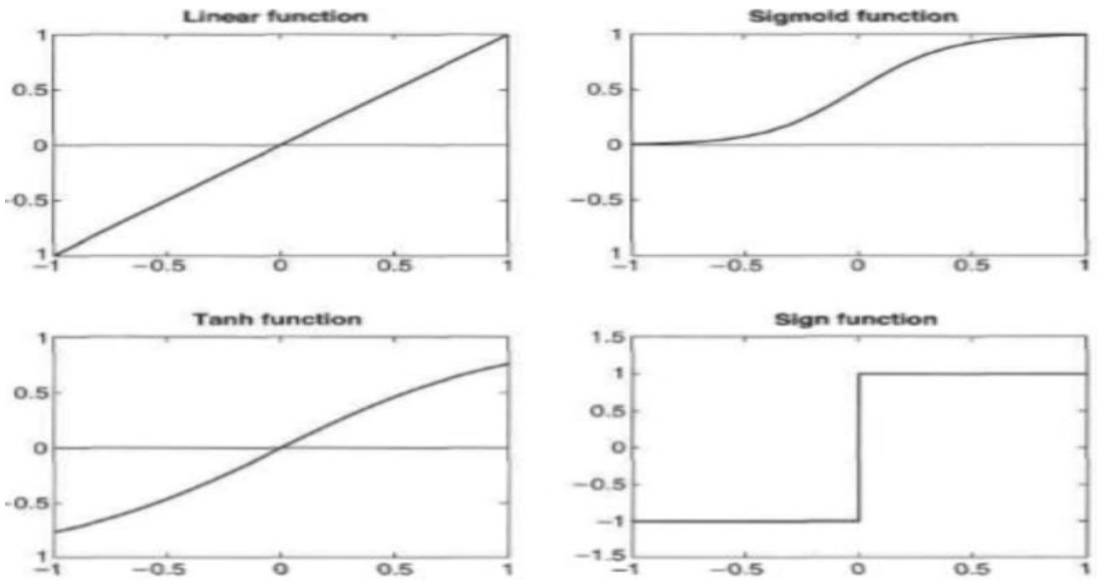
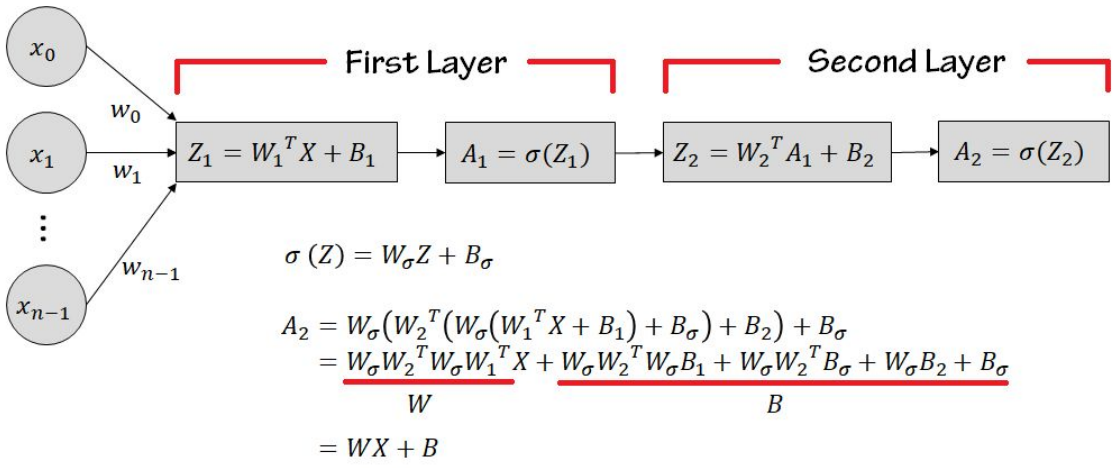


Figure 5.18. Types of activation functions in artificial neural networks.

먼저, Linear Function이 있으나, 이 경우는 크게 의미가 없다. 그 이유는 다음과 같다.



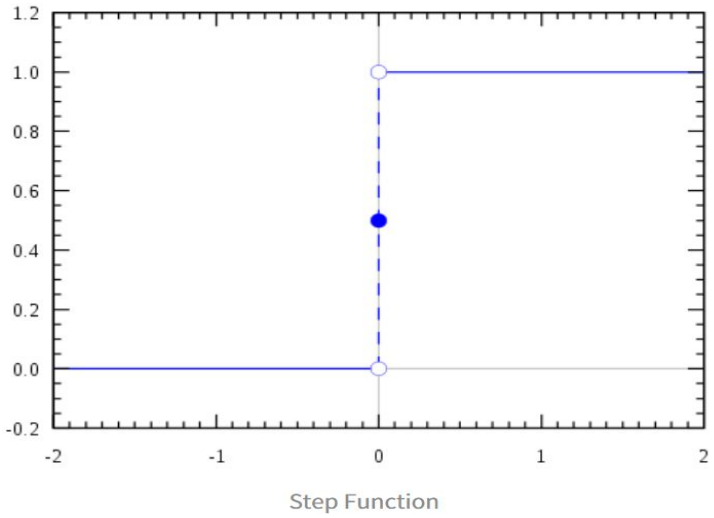
$$\sigma(Z) = W_\sigma Z + B_\sigma$$

$$\begin{aligned} A_2 &= W_\sigma\left(W_2{}^T\left(W_\sigma\left(W_1{}^T X + B_1\right) + B_\sigma\right) + B_2\right) + B_\sigma \\ &= \underbrace{W_\sigma W_2{}^T W_\sigma W_1{}^T}_{W} X + \underbrace{W_\sigma W_2{}^T W_\sigma B_1 + W_\sigma W_2{}^T B_\sigma + W_\sigma B_2 + B_\sigma}_{B} \\ &= WX + B \end{aligned}$$

출처 : https://bbangko.tistory.com/5

어차피 X에 곱해질 항들은 W로 치환이 가능하고, 입력과 무관한 함수들은 B로 치환이 가능하여, 몇 겹을 쌓든지에 관계 없이 Single Layer Perceptron과 같은 결과를 내게 된다.

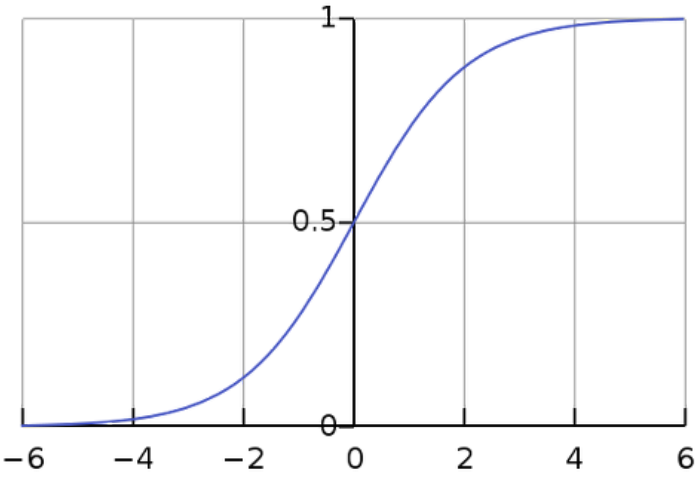
Sign(Step) Function은, 주어진 Input을 0과 1로 분류하는데, 입력이 음수이면 0, 양수이면 1을 출력한다.



(출처 : https://bbangko.tistory.com/5)

매우 간단하게, Output이 1이면 출력, 0이면 뉴런을 죽인다는 이분법적 구조이다. 다만, 모델의 Optimization 과정에서 미분이 필요하기 때문에 이는 사용할 수 없다.
Optimization은 추후에 알아보기로 하고, 그런 문제점이 있다 정도만 알아두도록 하자.

다음으로, Sigmoid 함수를 보자.



(출처 : https://bbangko.tistory.com/5)

sigmoid function은 입력값을 (0,1)의 범위로 normalize 해 주는 함수이다. 과거에 상당히 인기가 많았던 함수로, 수식은 다음과 같다.

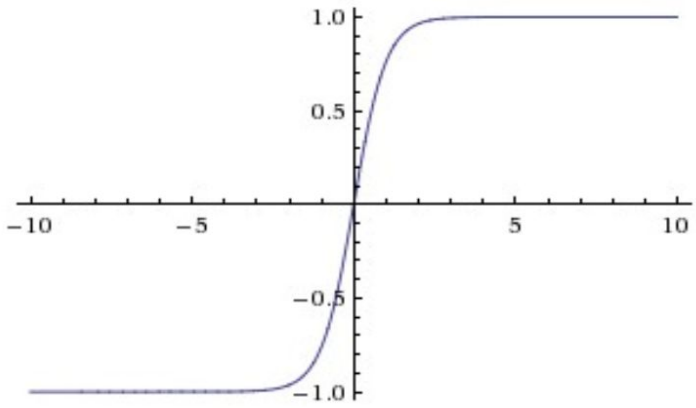
$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{\partial x} sigmoid(x) = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right)$$

(출처 : https://bbangko.tistory.com/5)

이와 같이 간단한 수식으로 코드로 구현하기도 쉽고, 무엇보다 미분값이 간단하다 (A'=A(1 - A)).
다만 이 함수는 Layer가 깊어질수록 문제가 생기는데, 이는 Gradient Vanishing 이라는 문제로, 이 또한 추후에 설명하기로 한다. 간단하게 말하면, 학습이 진행될수록 미분값이 0또는 1에 가까워져 문제가 되는 것이다.

이런 문제를 해결하기 위해, tanh function이라는 활성함수가 도입되었다.



Tanh Function

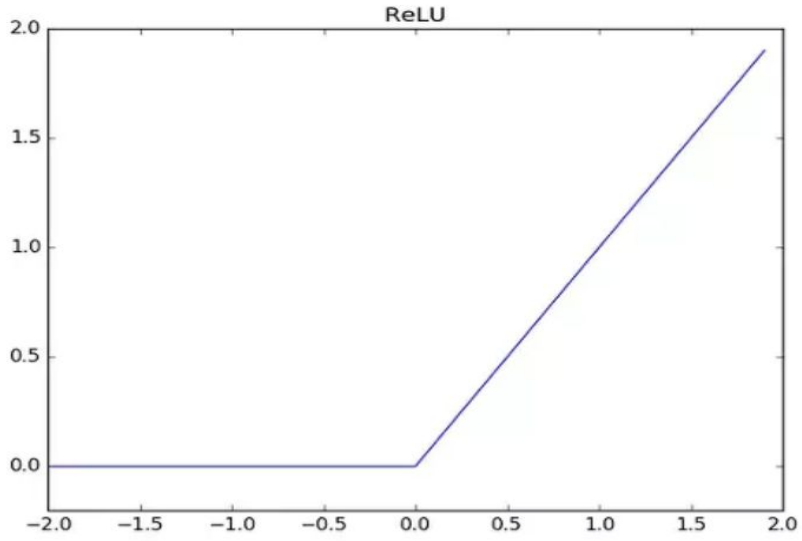
$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$\frac{d}{dx} tanh(x) = 1 - tanh(x)^2$$

(출처 : https://bbangko.tistory.com/5)

tanh function의 수식은 다음과 같다. tanh 함수는 입력신호를 (-1, 1) 사이의 값으로 Normalize해주는 함수인데, Sigmoid 함수에 비하여 거의 모든 면에서 성능이 좋다.
그러나 Gradient Vanishing 문제를 여전히 해결할 수 없다는 문제가 있다(Sigmoid 함수보다는 덜하지만).

이러한 Gradient Vanishing 문제를 해결하기 위하여 제안된 함수가 ReLU Function 이며, 현재로선 가장 인기가 많은 Activation Function이라고 할 수 있겠다.



Relu Function

(출처 : https://bbangko.tistory.com/5)

위와 같이 ReLU Functio은 음수 Input에서는 0, 양수에서는 Linear한 구조를 가진다(Relu(x) = max(0,x)). 우선 기울기가 0 또는 1의 값을 가지기 때문에 Gradient Vanishing 문제가 나타나지 않는다는 장점이 있다. 또한 Linear Function과 비슷한 구조를 가지지만, 엄연히 Non-Linear Function이기 때문에 Layer를 깊게 쌓을 수 있다는 장점이 있다. 동시에 수식이 간단(Exp()함수를 사용하지 않음)하므로, 연산능력 또한 Sigmoid나 tanh 함수보다 빠르다(6배 이상 빠른 것으로 알려져 있다).

마지막으로, Softmax 함수가 있는데, 이는 출력층(Output layer)에 적용되는 Activation Function이다. Softmax 함수는 Classification에 유용하게 사용되는데, 예를 들어 어떤 사람의 표정을 분석할 때, 여러 결과에 대하여 화남(98%), 기쁨(1%), 무표정(1%) 등으로 확률적인 분석을 해야 하는 경우 등이 있다.

$$p_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

$$= \frac{e^{x_j}}{e^{x_1}+e^{x_2}+\cdots+e^{x_K}} \quad for\ j\ =\ 1,\ldots,K$$

이제 Perceptron의 구성요소와, 그 용도에 대하여 간략하게 살펴 보았고, 그러면 어떻게 Neural Network를 학습시킬 것인가에 대하여 알아보도록 한다.

(4) Training Neural Network

(출처 : https://tykimos.github.io/2017/03/25/Fit_Talk/, 김태영의 케라스 이야기 를 참고하였습니다.)

- Intro

케라스를 기준으로, 학습은 model.fit() 함수를 통하여 이루어진다. 이는 수많은 인자들(Hyperparameters)에 의하여 영향을 받는데, Epoch(반복횟수), Batch(학습 데이터 나누는 방식)등이 그렇다. 먼저, Keras 내에서 Neural Network를 학습시키는 코드를 보자.

```
model.fit(x, y, batch_size=32, epochs=10)
```

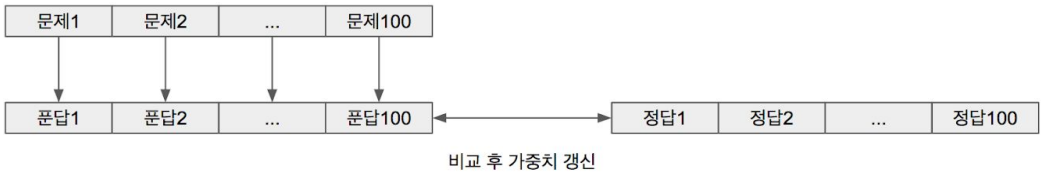
- x = 입력 데이터
- y = 라벨 값
- batch_size = 몇 개의 샘플로 가중치를 갱신할 것인지 설정
- epochs = 학습 반복 횟수

이를 모의고사 1회분의 시험지를 가지고 공부를 하는 상황에 빗대어 보자. 이 시험지는 100문항이 있고, 해답이 제공된다.

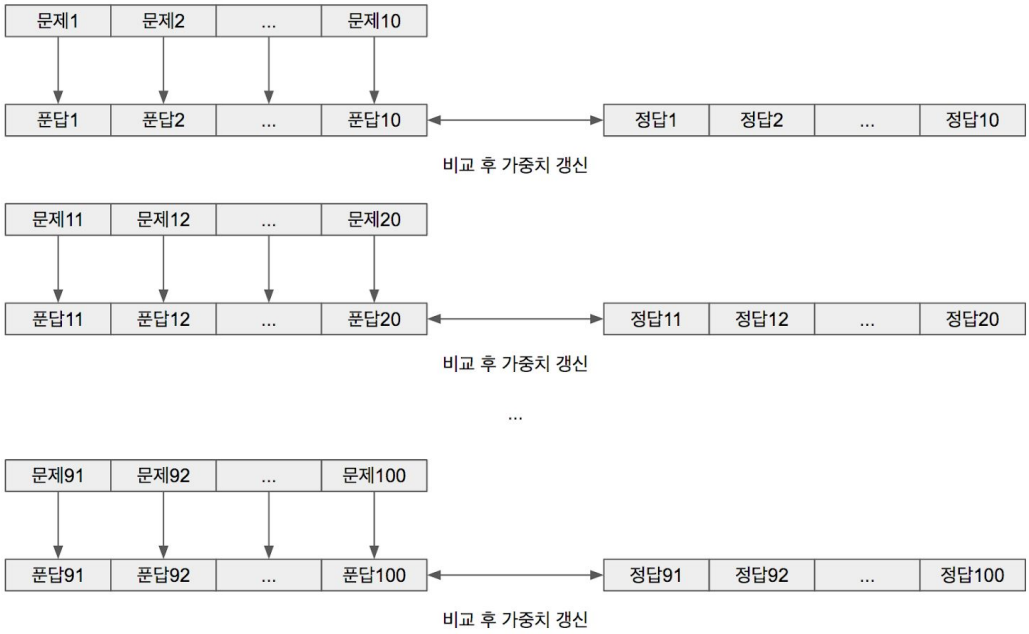
문제지	문제1	문제2	문제3	문제4	문제5	문제6	...	문제100
해답지	정답1	정답2	정답3	정답4	정답5	정답6	...	정답100

여기에 주요 인자들을 비유하면,

- x : 100문항의 문제들
- y : 100문항의 정답들
- batch_size : 몇 문항마다 해답을 맞춰볼 것인지, 사이클을 정해주는 것. 예를 들어, 배치사이즈가 100이라면 100문제를 풀 때마다 답을 한 번 맞춰보는 것이다. 우리가 공부를 할 때, 정답지를 보고 “아, 이렇게 하는 거구나.”하고 학습이 되는 것처럼, 신경망의 학습 또한 정답과 비교를 하며 가중치가 갱신되는 것이다(Backpropagation algorithm).

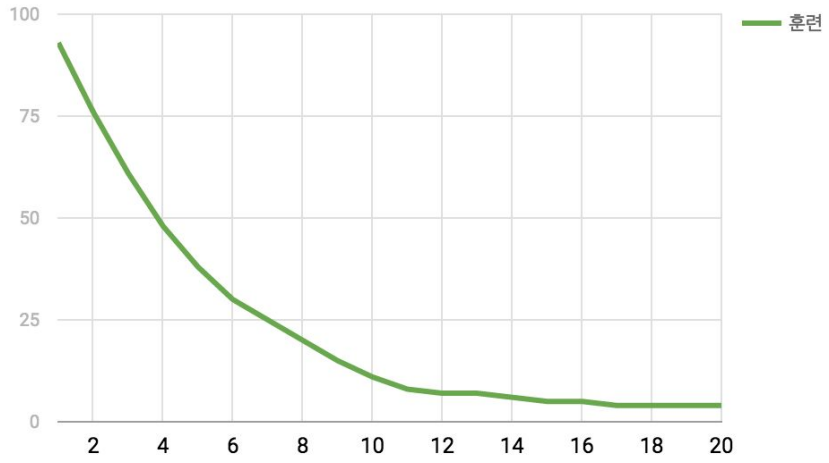


Batch size가 100인 상황을 도식화하면 다음과 같다. 그렇다면, Batch size가 10인 상황은 어떻게 생각할 수 있을까? 모의고사 1회분을 10문제씩 나누어, 10문제를 풀고 정답을 맞춰보고, 또 10문제를 풀고 맞춰 보고.. 하는 것을 반복하는 것이 된다.
이는 앞선 학습이 뒤의 문제풀이에 영향을 주기 때문에, 좀 더 정밀하게 fitting을 할 수 있다고 직관적으로 생각할 수 있다.



그렇다면, batch size가 1일 때가 가장 좋다고 생각이 드는 것이 정상이다. 당연히 한 문제를 풀고, 한 문제를 맞춰보고 하는 것이 주어진 학습 데이터를 효율적으로 사용할 수 있기 때문이다. 하지만 늘 그렇지는 않다. 앞서도 언급한 바와 같이, Gradient Vanishing 등의 이슈가 있기 때문에, Backpropagation이 너무 자주 일어나도 좋지 않기 때문이다.

- Epochs : 에포크는 모의고사 1회분을 몇 번 풀어볼지를 조정하는 것이다. 주어진 예시로 치면, 100문항을 몇 번이나 반복해서 풀어보는 지 정하는 것인데, 만약 에포크가 20이라면 모의고사 1회분을 20번 푸는 셈이 된다. 같은 문제를 반복해서 푸는 것이 무슨 효과가 있느냐는 생각을 할 수 있겠지만, 이전에 학습을 시행했을 때와 또 다시 학습을 시행했을 때 가중치가 달라지기 때문에, 다시 학습이 일어나는 것이다.



여기에서 세로축은 100문항 중 틀린 개수, 가로축은 모의고사의 풀이 반복 회수, 즉 에포크이다. 처음에는 틀린 개수가 확 적어지지만(즉 학습 효율이 좋지만), 갈수록 완만하게 틀린 개수가 줄어든다는 것을 알 수 있다. 이는 시험 성적을 생각하면 쉽게 이해할 수 있다. 열심히 노력하여 30점에서 70점이 되기는 비교적 쉽지만, 98점에서 100점이 되기는 어려운 것과 같이 말이다.

만약 주어진 양질의 데이터가 많다면(예를 들면, 다른 모의고사가 20회분 있는 상황) 굳이 에포크의 수가 많지 않아도 되는 것일까? 이 문제도 우리가 공부할 때와 같다. 일반적으로 양질의 문제를 여러번 푸는 것이 잡다한 문제를 많이 푸는 것보다 성적을 올리는데 도움이 될 가능성이 높다.

이를 앞서 Perceptron과 접목시키면, 적절한 Neural Network를 설계하고, 주어진 데이터를 어떻게 나눌 것이며, 그것을 몇 번 반복시킬 것인지 정하는 것이 Neural Network를 학습시키는 것이라고 할 수 있겠다.

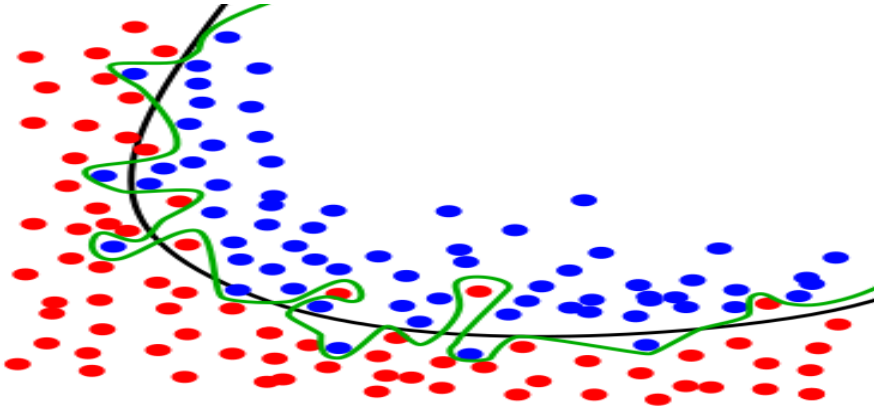
그렇다면, 주어진 데이터에 대하여 에포크를 무한으로 하면 완벽한 딥러닝이 되는 것일까?

- 각종 문제와 해결법

Neural Network의 학습 과정에서 대두되는 문제점은 크게 세 가지로 분류되는데, 이는 Overfitting, Gradient Vanishing, Time complexity이다. 여기서는 각 문제점과 그 해결방안을 간략하게 살펴보기로 한다.

- Overfitting(과적합)

앞서 몇번 언급된 바와 같이, 분명히 Layer의 수가 많을수록, 그리고 Epochs의 수가 많을수록 완벽하게 Training이 되어야 합리적이다. 실제로 완벽하게 되기는 한다. 다만 그 완벽한 것이 Train Data안에서만 완벽한 것이 문제가 되는 것이다.



(출처 : https://ko.wikipedia.org/wiki/%EA%B3%BC%EC%A0%81%ED%95%A9)

위 그림에서 검은 선이 일반적인 모델을, 초록색 선이 과적합을 나타낸다. 초록색 선은 Train Set은 완벽하게 분류할 수 있지만, 그 결과를 다른 데이터셋에 적용하면 전혀 유용하지 않게 된다. 정답을 이미 알고 있는 Dataset에만 정확한 모델이 무슨 소용이 있겠는가?

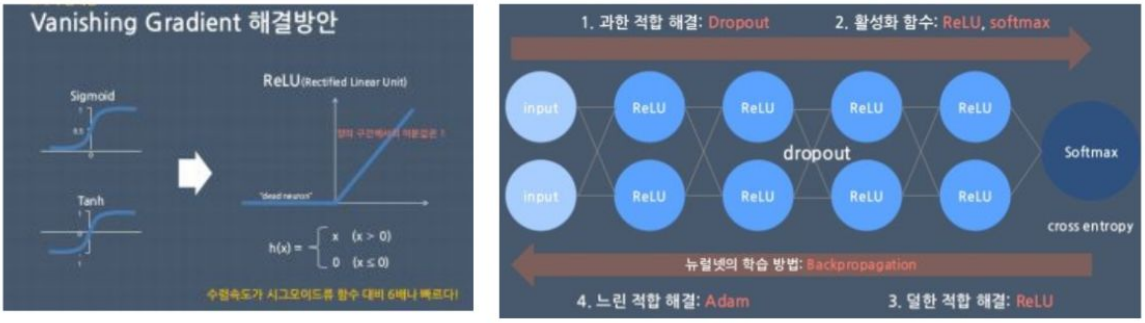
이러한 Overfitting은 결국 데이터셋을 너무 정확하게 분류하려다가 생기는 문제이다. 그 해결책은, 간단하게 학습을 '덜' 하는 것이 된다. 이를 Dropout 이라고 한다.

https://www.slideshare.net/HeeWonPark11/ss-80653977

Dropout은 일부러 몇 개의 Neuron을 학습 과정에서 빼는 것이다. 빠지는 Neuron은 무작위로 정해지게 된다. 결과적으로 Train Set에 대한 설명력은 약간 떨어지지만 전체적인 성능은 크게 개선이 된다.

- Gradient Vanishing(기울기 소멸 문제)

Gradient Vanishing 문제는 학습이 진행될수록 Gradient가 0또는 1에 가까워 지면서 생기는 문제이다. 그 반대의 경우로는 Gradient Exploding이 있다.
이는 ReLU Function을 통해 해결할 수 있다. ReLU Function은 앞서 보았듯이, 음수에서는 기울기가 0이고, 양수에서는 y=x 형태의 그래프를 가지므로 기울기가 0 또는 1이 되어 Gradient Vanishing, Exploding 둘 다 발생하지 않는다.
그 대신, 음수에서의 기울기가 항상 0이므로, 음수값의 Input에 대해서는 학습 자체가 이루어지지 않는다는 단점이 있다.

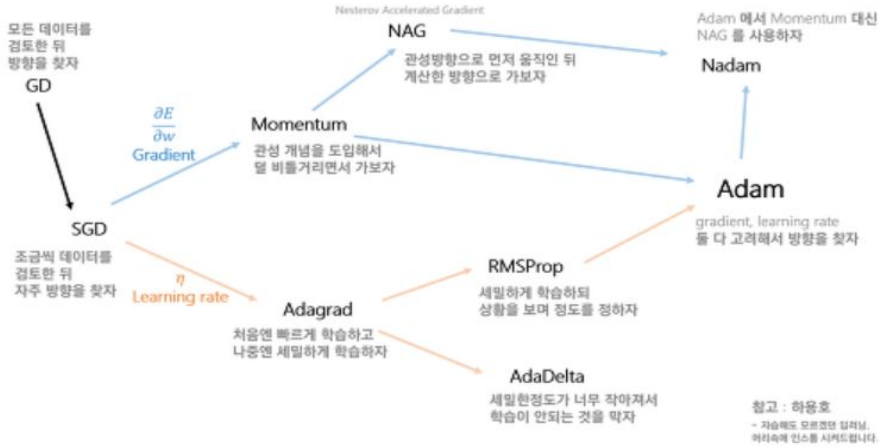


https://www.slideshare.net/HeeWonPark11/ss-80653977

- Time Complexity

기계학습의 목적은 Neural Network에 대하여 최적의 bias값과 weight값을 찾는 것이다. 이 과정은 Gradient Descent(Cost Function의 최솟값을 찾기 위한 방법)를 이용하여 진행된다. 그러나, 이 방법은 딥러닝이 요구하는 연산량을 감당하기에는 너무 오래 시간이 걸린다. 결론적으로, Time Complexity는 '속도가 너무 느림' 의 문제인 셈이다.

딥러닝에서는 이를 해결하기 위해 Optimizer를 이용한다.



$$w^{+} = w - \boxed{\eta} * \boxed{\frac{\partial E}{\partial w}}$$

learning rate : 한번에 얼마나 학습할지

gradient : 어떤 방향으로 학습할지

(출처 : https://gomguard.tistory.com/187, Optimizer별 간략한 정리)

예를 들어 10억건의 데이터가 있다고 하는 상황에서, 전체 데이터를 고려해서 한 번 가중치를 수정하려면 10억번의 연산을 진행해야 한다. 이는 당연히 속도상의 문제가 있다. 그래서 최초로 나온 Optimizer가 SGD(Stochastic Gradient Descent, 확률적 경사하강법) 이다.

이는 주어진 데이터를 랜덤으로 분할해서, 각 set에 대해 경사하강법을 시행하는 것이다. 속도 자체는 개선이 되었지만, 우선 만족할 만한 수준도 아니었고 랜덤으로 데이터셋을 분할하다 보니 수렴안정성이 낮았으며 Cost Function의 최소점이 아닌 극소점을 찾으면 학습이 중단되는 현상이 발생하는 문제가 있었다.

그 외 지속적인 발전을 거듭하며 여러 Optimizer가 등장하였는데, 간략하게 정리하면 다음과 같다.

- 관성 중심 : Momentum, NAG
- 속도 중심 : Adagrad, RMSProp, AdaDelta
- 두 장점을 합친 것 : Adam, Nadam

현재 가장 보편적으로 우수한 Optimizer는 Adam이지만, 상황에 따라 다른 Optimizer가 더 좋은 성능을 발휘하는 경우도 많아, 적절한 Optimizer를 적용하는 것이 중요하다.

- 학습을 위한 기술

- 미니배치 학습

딥러닝에서는 train data에 대한 손실함수의 값을 구하고, 그 값을 최대한 줄여주는 매개변수를 찾아낸다. 다만, 이렇게 하려면 모든 데이터에 대하여 손실함수의 값을 찾아야 한다. 예를 들어, 데이터가 1000개 있다고 치면, 1000번의 손실함수 값 계산을 수행해야 하는 것이다.
그러나 빅데이터를 다루게 되면, train data가 1000개가 아닌 1만개, 1억개가 될 수도 있다. 그렇게 된다면, 일일히 계산하는데 굉장히 오랜 시간이 걸리게 될 것이다.
따라서 한 번에 한 개만 계산하는 것이 아닌, 일부 sample을 가져와서 이를 전체의 '근사치'로 놓고 학습을 진행한다. 이 때의 sample을 미니배치라고 하고, 이런 학습방법을 미니배치 학습이라고 한다.

- 가중치의 초기값 설정

신경망의 학습에 있어 가중치의 초기값 설정은 매우 중요한 부분이다. 그러나 덮어놓고 아무 값이나 설정해 놓을 수는 없는 노릇이다. 그렇다면 이는 어떻게 설정해야 하는가? 다행스럽게도 이미 설정된 초기값이 몇 개 있다.

1) 가중치 초기값이 0이거나, 동일한 경우

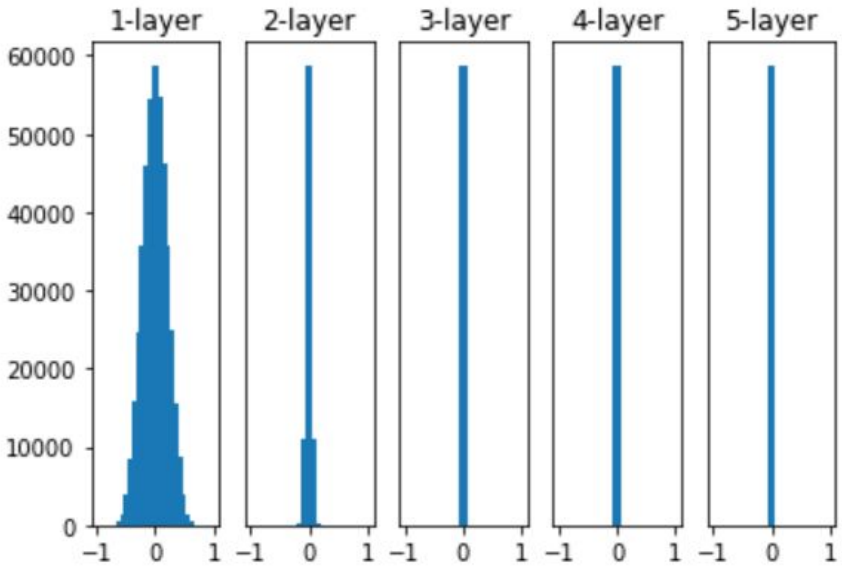
가중치의 초기값을 모두 0으로 하거나, 동일한 값으로 초기화 할 경우 모든 뉴런이 동일한 출력값을 내보낼 것이다. 학습이 잘 되려면, 뉴런의 가중치가 비대칭적(어떤 뉴런을 가중치가 크고, 어떤 뉴런은 작고)이어야 하는데, 이 경우는 모든 뉴런이 동일한 그래디언트 값을 가지므로 뉴런의 개수가 아무리 많아도 뉴런이 단 한 개 인것처럼 작동하게 되므로 학습이 제대로 이루어지지 않는다. 따라서, 가중치 값을 동일한 값으로 초기화해서는 안 된다.

2) 작은 난수로 초기화 하는 경우

가중치 초기값은 작은 수로 초기화해야 하는데, 그 이유는 Activation Function이 Sigmoid인 경우 만약 가중치 초기값을 큰 값으로 한다면 Gradient가 0 또는 1로 수렴하여 Gradient Vanishing 문제가 발생하게 되기 때문이다.

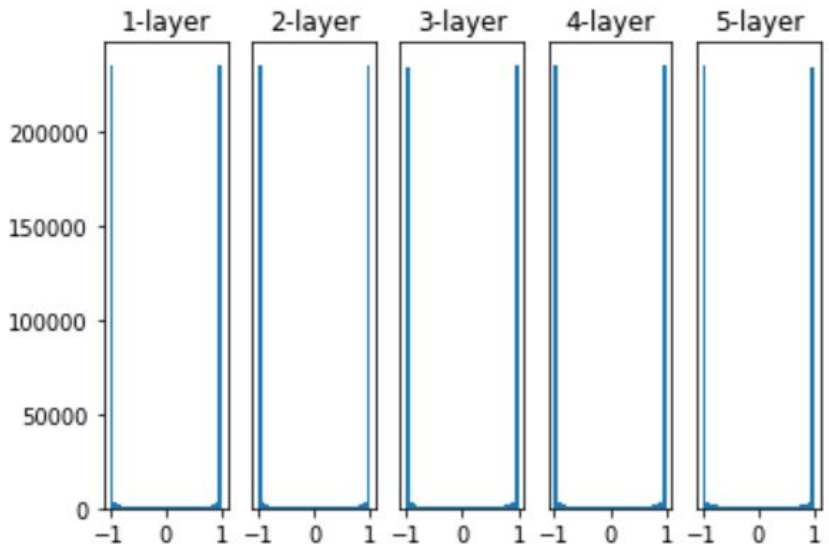
따라서, 가중치 초기값은 작은 수로 초기화하되 같은 값을 가지지 않도록 랜덤으로 초기화해야 한다. 일반적으로, 가중치 초기값은 평균 0, 표준편차 0.01의 정규분포(가우시안 분포)를 따르는 값으로 랜덤하게 초기화한다.

하지만 이런 방식에도 문제가 있다. 이는 다음 그래프를 보면 알 수 있다.



(출처 : https://excelsior-cjh.tistory.com/177)

위 그림은 Activation Function 을 tanh 함수로 하고, 평균이 0, 표준편차가 0.01인 가우시안 분포를 따르는 가중치로 초기화를 수행하였을때의 레이어에 따른 가중치 분포이다. 이 때, 위 그림처럼 첫 번째 레이어만 가중치가 그나마 고르게 분포하고 나머지는 0에 수렴함을 알 수 있다. 즉, Gradient Vanishing이 발생하는 것이다. 여기서 알 수 있는 것은, 가우시안 분포를 이용한 가중치 초기화 방법은 얕은 신경망에는 적합할지 모르나, DNN에는 적합하지 못하다는 것이다.
평균을 0으로, 표준편차를 1로 하는 랜덤한 난수로 가중치를 초기화하고, Activation Function을 Tanh 함수로 사용한 경우, Gradient의 분포는 다음과 같다.

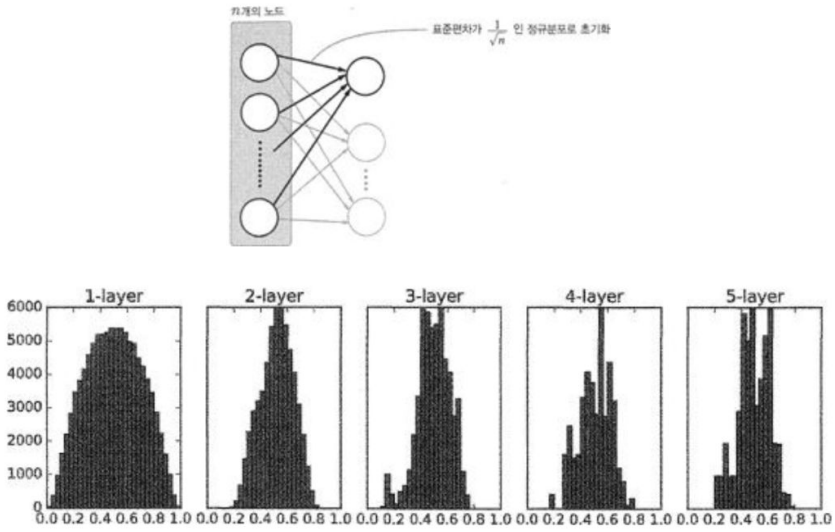


(출처 : https://excelsior-cjh.tistory.com/177)

이 경우에는 -1 과 1로 Gradient가 치중하게 되어, 학습이 일어나지 않을 것을 알 수 있다.

3) Xavier 초기값과 He 초기값

이러한 Gradient Vanishing 등의 문제를 덜 일으키기 위해, 권장되는 초기값들이 몇 개 있다. 그 중 Xavier 초기값부터 알아 보겠다.
Xavier 초기값은 간략히 말하면 앞 계층의 노드가 n개라면, 표준편차가 $1/\sqrt{n}$ 인 분포를 사용해서 가중치를 초기화하면 된다는 원리이다.



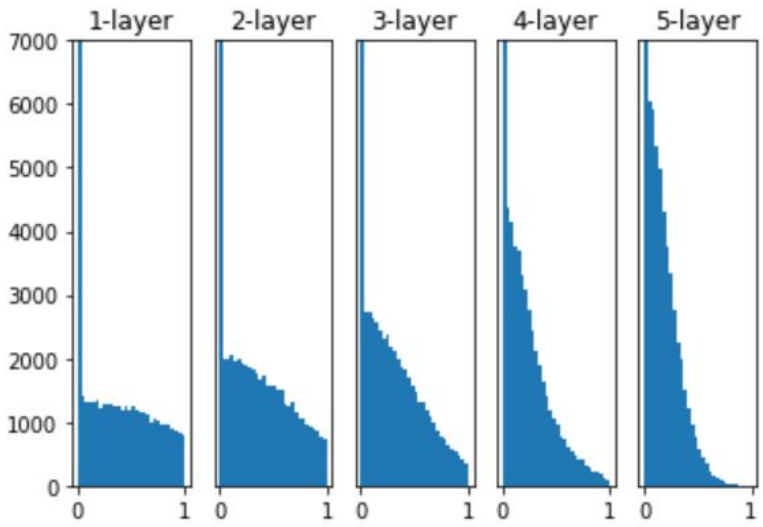
이는 Xavier 초기값을 이용한 경우의 가중치 분포이다. 위와 비교해 보면, 그 분포가 완벽히 고르지는 않으나 확실히 더 다양하게 분포되어 있음을 알 수 있다. 더 자세한 수식은 다음과 같다.

- 평균이 0이고 표준편차 $\sigma = \sqrt{\frac{2}{n_{\text{inputs}}+n_{\text{outputs}}}}$ 인 정규분포
- 또는 $r = \sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}$ 일 때 $-r$과 $+r$ 사이의 균등분포
- 입력의 연결 개수와 출력의 연결 개수가 비슷할 경우 $\sigma = 1/\sqrt{n_{\text{inputs}}}$ 또는 $r = \sqrt{3}/\sqrt{n_{\text{inputs}}}$ 를 사용

(출처 : https://excelsior-cjh.tistory.com/177)

이는 텐서플로우, 케라스 등에 내장되어 있으므로 수식으로 복잡한 과정을 거쳐 적용할 필요는 없다.

그러나 이런 Xavier 초기값에도 문제가 있으니, 바로 ReLU 함수에 적용하는 경우이다.



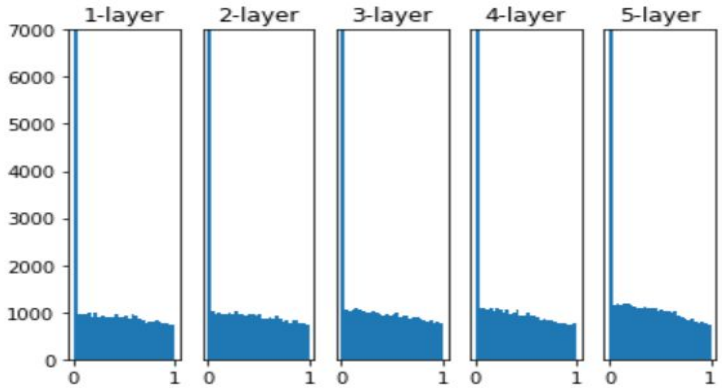
(출처 : https://excelsior-cjh.tistory.com/177)

보다시피 Xavier 초기값을 ReLU 함수에 적용하면, 레이어가 깊어질 수록 출력값이 0으로 치우치는 문제가 발생한다. 이는 Xavier 초기값이 tanh 함수에 적합하게 설정되었기 때문에 발생하는 문제이다.
따라서 ReLU 함수에 적합한 가중치를 따로 설정해야 하는데, 이 또한 권장되는 값이 이미 있다. 이를 He 초기값이라 한다.

- 평균이 0이고 표준편차 $\sigma = \sqrt{2} \cdot \sqrt{\frac{2}{n_{\text{inputs}}+n_{\text{outputs}}}}$인 정규분포
- 또는 $r = \sqrt{2} \cdot \sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}$ 일 때 $-r$과 $+r$ 사이의 균등분포
- 입력의 연결 개수와 출력의 연결 개수가 비슷할 경우 $\sigma = \sqrt{2}/\sqrt{n_{\text{inputs}}}$ 또는 $r = \sqrt{2} \cdot \sqrt{3}/\sqrt{n_{\text{inputs}}}$ 를 사용

(출처 : https://excelsior-cjh.tistory.com/177)

이는 제법 간단한데, Xavier 초기값에 √2 를 곱해준 것이다. 그 이유는, ReLU 함수의 경우 입력이 음수일 때 출력이 전부 0 이므로, 가중치를 더 넓게 분포시킬 필요가 있기 때문이다.



(출처 : https://excelsior-cjh.tistory.com/177)

He 초기값을 ReLU 함수에 적용시킨 가중치 분포는 다음과 같다.
이 또한 keras 또는 tensorflow 내에 내장이 되어 있으므로, 복잡한 과정을 거쳐 구현을 할 필요는 없다.

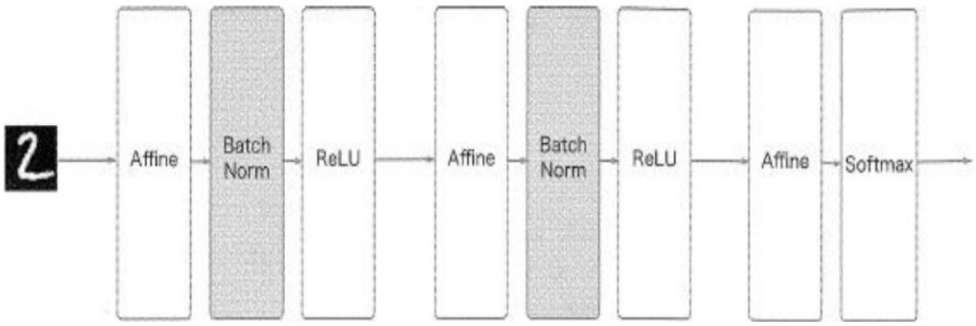
- 배치 정규화

앞에서 가중치의 초기값을 적절하게 퍼트리면 학습이 더 적절하게 이루어진다는 것을 보았다. 그렇다면, 각 층이 활성화값을 강제로 퍼뜨린다면 어떨까? 이런 아이디어에서 착안한 것이 배치 정규화이다.

배치 정규화의 장점으로는,

1. 빠른 학습 진행
2. 초기값에 크게 의존하지 않는다
3. Overfitting을 억제한다.

가 있다. 배치 정규화에서는, 각 층에서의 활성화값을 고르게 분포시키기 위해 다음과 같이 데이터 분포를 정규화하는 '배치 정규화 계층'을 신경망에 삽입한다.



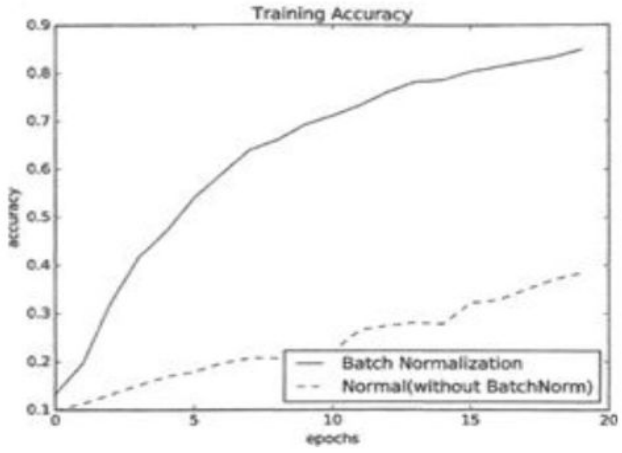
배치 정규화는 그 이름과 같이, 학습 시 미니배치를 단위로 정규화한다. 구체적으로는 데이터 분포가 평균이0, 분산이 1이 되도록 정규화한다.

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

각각의 미니배치의 m개의 데이터 입력에 대해, 평균과 분산을 구하고, 입력 데이터를 평균이 0, 분산이 1이 되게 정규화한다. 단순히 미니배치의 입력 데이터를 평균이 0, 분산이 1이 되게 변환하는 일을 활성화 함수 앞에서 수행함으로써, 데이터 분포가 덜 치우치게 할 수 있는 것이다.



이는 미니배치를 수행하였을 때, MNIST 데이터셋의 학습 능률을 그래프로 표현한 것이다.