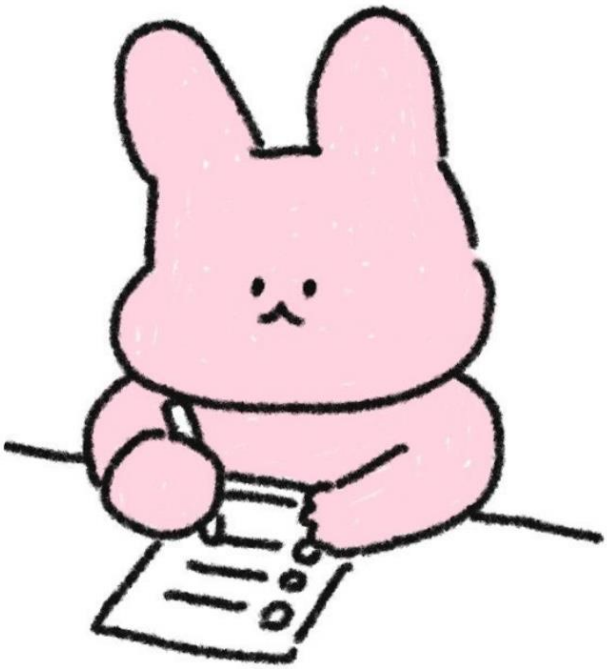
# Deep Learning Basic Instructions



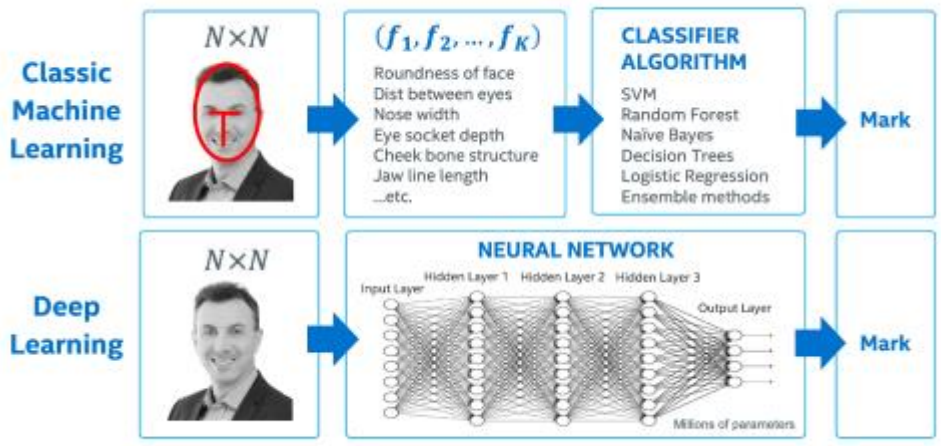
만든이: 수경

# Contents

# Ⅰ. 딥러닝 배경지식 쌓기

## • 0. Why Deep Learning?

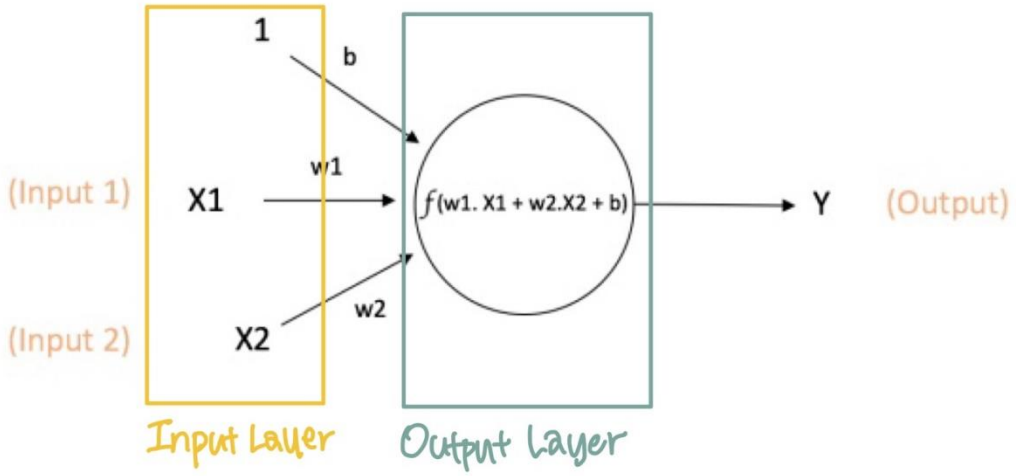
딥러닝은 인공지능이라는 커다란 개념 안에 있는 머신러닝의 한 분야입니다. 딥러닝은 놀라운 속도로 발전하며 불과 5 년 사이에 사람을 능가하는 수준까지 도달했습니다. 사람들은 도대체 왜 딥러닝에 열광하며, 딥러닝을 연구하는 걸까요? 딥러닝에 대해 본격적으로 알아보기 전에, 딥러닝과 머신러닝을 포괄하고 있는 개념인 인공지능에 대해 잠깐 이야기하고자 합니다. 인공지능, 말 그대로 인공적으로 만들어진 지능이라는 뜻일 텐데, '지능(知能)'란 무엇일까요? 분야에 따라 그 정의는 다르겠지만 심리학에서는 지능을 "새로운 대상이나 상황에 부딪혀 그 의미를 이해하고 합리적인 적응 방법을 알아내는 지적 활동의 능력"이라고 정의합니다. 지능은 인간이 지구 먹이 사슬 최상층에 앉는데 큰 공헌을 했습니다. 인간은 그 자리에 만족하지 않고 인간과 비슷하게 생각할 수 있는, 인간과 비슷한 지능을 가진 존재를 만나고 싶어했던 것 같습니다. (온 우주에 일명 '고등 지능 생명체'가 인간 밖에 없을까봐 외계인을 찾는 모습을 보면 말이죠…?) 하지만 현재 인간이 구현한 인공지능은 '스스로 해석하는' 존재와는 조금 거리가 있습니다. 인간의 도움이 필요하기 때문입니다.




사람들이 딥러닝에 열광하는 큰 이유는 바로 여기에 있습니다. 인간의 개입이 다른 인공지능 기술들에 비해 현저히 적습니다. 머신러닝의 경우, 들어온 입력 데이터의 특징을 추출하는 전처리 작업인 'feature extraction' 작업을 인간이 해주어야 합니다. 그림에서 보면 어디가 눈이고 어디가 코인지를 인간이 알려주어야 한다는 뜻입니다. 코 길이, 미간 길이 등등을 얼굴의 형태 등을 하나씩 따서 숫자로 입력해줘야 합니다. 반면 딥러닝의 경우, 그런 작업 없이도 인공신경망을 통해 알아서 인식할 수 있습니다. 이 외에도 딥러닝은 기존의 모델들이 어려워했던 비정형 데이터인 음성 데이터나 이미지 데이터 등을 잘 다룬다는 장점이 있습니다. 다만 어떤 방식으로 눈과 코를 분리해서 인식하는지를 알 수 없어서 해석에 어려움이 있습니다. 많은 데이터를 필요로 하기 때문에 데이터 확보에 시간과 비용이 많이 들고 학습 시간도 오래 걸린다는 단점이 있습니다.

개인적으로 딥러닝이 앞서 언급했던 '스스로 해석하는 존재'를 만드는 데 가장 가까운 모델이 아닐까 생각합니다. 스스로 학습하고, 학습에 대한 나름의 답을 내놓는 것까지 인간과 꽤 유사하지 않나요? 실제로 딥러닝의 기본 아이디어인 인공신경망은 생물의 신경망, 특히나 뇌에서 영감을 얻었다고 합니다. 때문에 뇌의 신경세포인 뉴런과 딥러닝의 근간인 퍼셉트론을 빗대어 설명한 내용을 많이 찾아볼 수 있습니다. 하지만 요 근래 이 주장은 힘을 잃고 있고 있습니다. 아직 뇌의 뉴런이 무슨 일을 하는지 정확히 알지 못하는데 둘을 엮는 것이 합리적이지 않다는 논리입니다. 그래서 그냥… 그런 예시 없이 바로 딥러닝의 기본 단위인 퍼셉트론을 들여다보겠습니다!

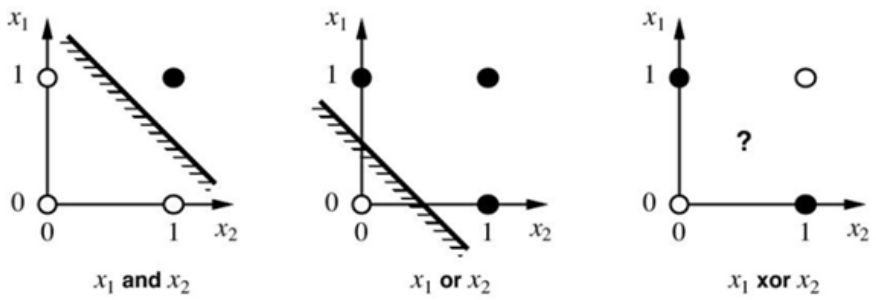
## • 1. Perceptron




Output of neuron = Y= $f$(w1. X1 + w2.X2 + b)

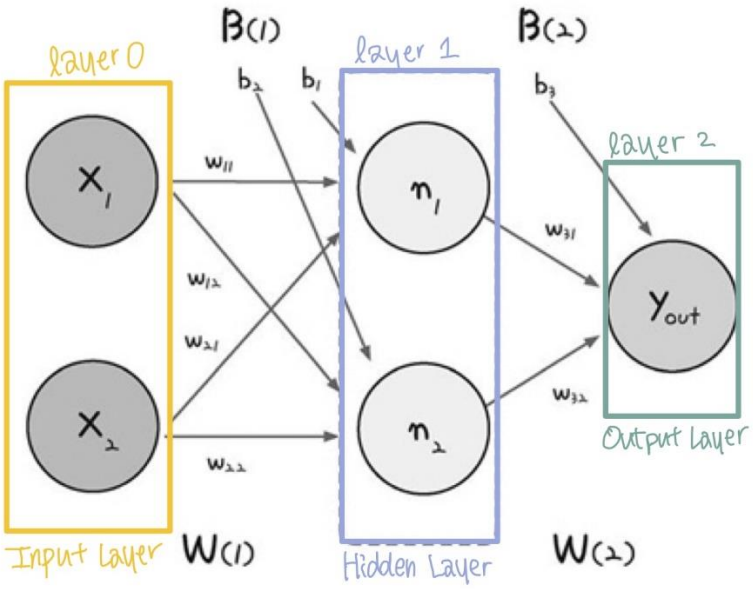
퍼셉트론(perceptron)은 약 60 년 전에 고안된 단순한 알고리즘으로 인공신경망 모형 중 하나입니다. 오래된 알고리즘이지만, 퍼셉트론은 여전히 딥러닝의 근간이 되고 있습니다. 퍼셉트론은 여러 개의 입력이 들어가서 하나의 결과를 출력하는 구조를 가진 일종의 '함수'입니다. 그림에서 $input\ layer$로 묶여 있는 $X = [x_1, x_2]$는 입력 데이터입니다. 입력 데이터들은 각각 가중치 $W = [w_0, w_1, w_2]$가 곱해진 채로 동그라미에서 만나게 됩니다. 앞으로 동그라미들은 '노드'로 부를 것입니다. $output\ layer$로 묶여 있는 노드에서는 두가지 일이 일어납니다. 첫번째, $Z = \sum_{k=1}^{3} X * W + b$: 노드로 들어온 입력 값과 가중치의 곱, 그리고 b 로 표시되어 있는 편향을 전부 합합니다. 두번째, $f(z)$ : 활성화 함수에 첫번째 계산의 결과 값을 넣어 계산합니다. 여기서는 계단함수(step function)라 불리는 활성화 함수를 사용합니다. 활성화 함수에 대해서는 밑에서 자세히 설명할테니, 여기에서는 $y = \begin{cases} 0\ (x \leq 0) \\ 1\ (x \geq 0) \end{cases}$의 함수로 생각해주시면 됩니다. 이렇게 두번의 연산을 거치고 scalar 가 된 값은 그대로 $output$이 됩니다.

이처럼 간단한 구조를 가진 퍼셉트론은 꽤 괜찮은 성능을 냈고, 인공지능 분야에 뜨거운 반응을 불러일으켰습니다. XOR problem 을 만나기 전까지는요.




XOR problem 은 그림에 나타나 있듯이 (0,1), (1,0)을 같은 것으로 묶고 (0,0), (1,1)을 같은 것으로 묶는 문제입니다. 퍼셉트론은 XOR 게이트를 구현할 수 없습니다. 퍼셉트론은 직선 하나로 나눈 영역만 표현할 수 있기 때문입니다. 즉, 선형 문제는 잘 풀지만 비선형 문제는 해결하지 못하는 한계를 가지고 있습니다.

## • 2. Multi-layered Perceptron




다층 퍼셉트론은 퍼셉트론의 두 층, 입력층(input layer)와 출력층(output layer)의 사이에 은닉층(hidden layer)이라 불리는 층 하나를 끼워 넣은 것입니다. 파란색으로 묶인 은닉층 하나가 추가되어 다층 퍼셉트론이 되면서, 퍼셉트론은 XOR problem 을 해결할 수 있게 되었습니다. 시그모이드라는 활성화 함수를 두 번 사용하게 되면서 비선형성이 추가되었기 때문인데요. 다층 퍼셉트론에서 일어나는 연산을 자세히 살펴보도록 하겠습니다.

은닉층에서는 앞서 퍼셉트론의 출력층에서 했던 연산이 노드 마다 발생합니다. 때문에 $n_1, n_2$는 각각 아래와 같습니다.

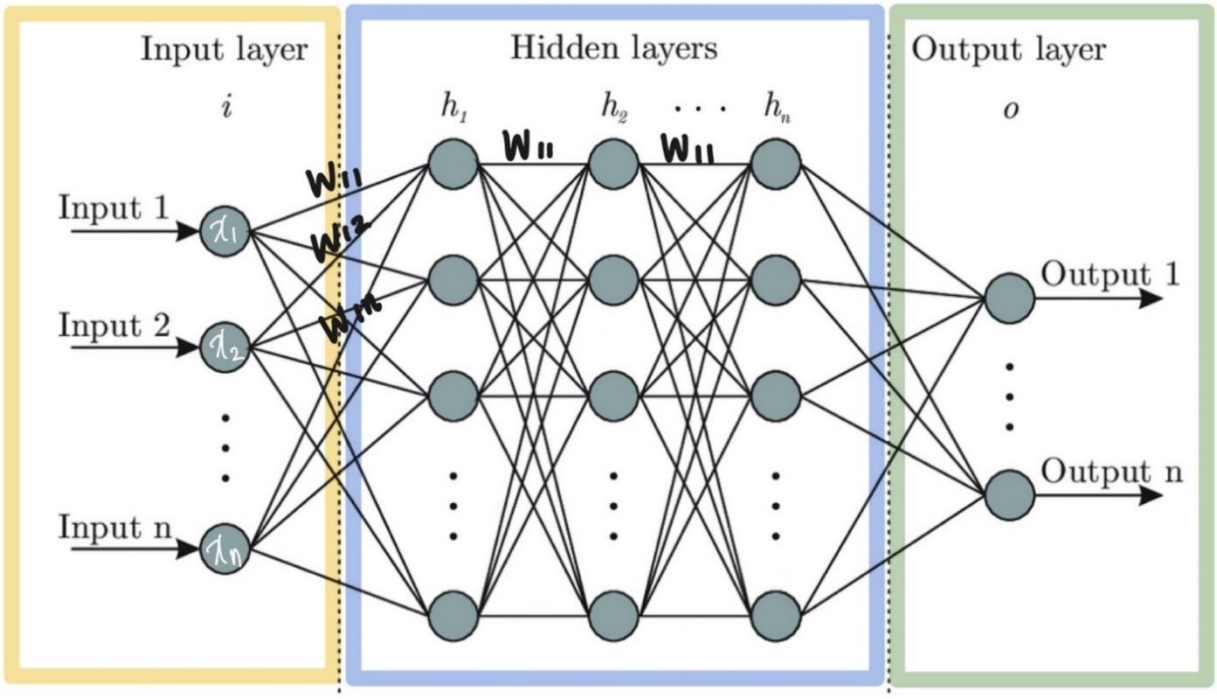
$$n_1 = \sigma(x_1 w_{11} + x_2 w_{11} + b_1)$$
$$n_2 = \sigma(x_1 w_{21} + x_2 w_{22} + b_2)$$

이 $n_1, n_2$ 값은 출력층에서 모여 또다시 $y_{out} = \sigma(n_1 w_{21} + n_2 w_{22} + b_2)$의 연산이 발생합니다(여기서 사용된 활성화 함수는 계단 함수가 아니라 시그모이드 함수라는 것인데, 이후에 설명하겠습니다.)이처럼 입력층에서 은닉층을 지나, 출력층으로 나오는 연산을 '순전파(Feed Forward)'라고 부릅니다. 반대로 출력층에서부터 은닉층, 입력층을 지나면서 일어나는 연산을 '역전파(Backward)'라고 하는데, 이 둘에 대해서는 뒤에서 더욱 자세하게 배워보도록 하겠습니다. 위의 다층 퍼셉트론처럼, 이전 계층의 모든 노드와 현재 계층의 모든 노드가 연결된 경우를 '전결합 계층(Fully-Connected Layer)'라고 부르는데, 우리가 앞으로 만나볼 대부분의 신경망들은 전결합 계층의 형태를 띠고 있습니다.

이렇게 퍼셉트론 하나로는 해결되지 않는 문제를 은닉층을 추가하면 해결할 수 있습니다. 이제부터는 은닉층을 많이 많이 쌓아보겠습니다.

# II. 신경망과 딥러닝

## • 1. 신경망




우리가 앞으로 계속 볼 신경망은 이런 구조로, 다층 퍼셉트론과의 차이라고는 은닉층이 여러 개인 것 밖에 없지요? 사실은 앞서 봤던 다층 퍼셉트론 또한 신경망의 한 종류입니다. 다만, 다층 퍼셉트론처럼 은닉층이 하나인 신경망을 '얕은 신경망'이라고 부르는데 중요하지는 않습니다. 여러 층을 쌓는 이유는, 같은 작업을 시행했을 때 (극단적인 예를 들자면) 은닉층 한 개에 100 개의 노드를 둔 신경망보다, 은닉층 50 층에 2 개의 노드를 둔 신경망이 더 빠르게 학습할 수 있기 때문입니다. 신경망 내부에서 일어나는 일은 다층 퍼셉트론에서 전부 설명했습니다! 드디어 지금까지 저게 뭘까 싶었던 여러 용어들을 정리하는 시간을 가져 보겠습니다.

## • 2. 용어정리

**a ) 가중치** ($w_{ij}$ : 출발하는 layer 의 $i$번째 노드, 도착하는 layer 의 $i+1$번째 노드)

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix}$$

가중치는 이전 층과 이후 층의 결합 강도를 담당하는 값으로, 출발 노드의 영향력을 결정합니다. 가중치는 행렬로 표현할 수 있는데요. 왼쪽 그림과 같은 형태입니다. 아까 전에 봤던 다층 퍼셉트론을 가지고 예를 들어보겠습니다. 입력층의 첫번째 노드와 은닉층의 두번째 노드를 연결하는 가중치는 $w_{12}$임을 볼 수 있습니다.

## b ) 편향

$$\overrightarrow{b_j} = (b_1, b_2, \dots, b_n)$$

편향은 해당 노드가 얼마나 쉽게 활성화되는지를 결정하는 값입니다. 편향은 도착 layer 의 각 노드에 하나씩 있어서 $w_{1j}$이나 $w_{2j}, w_{3j}, \dots, w_{mj}$ 모두 $b_j$의 영향을 받습니다. 때문에 다음과 같이 벡터로 표현됩니다. ($j$는 $i+1$이라고 할게요.)

## c ) 입력층

입력층은 이름 그대로, 데이터의 값들이 입력되는 층으로, 위의 신경망에서 $x_1, x_2$가 입력 값의 역할을 합니다. $x_1, x_2$는 데이터의 feature 입니다. 앞으로 배울 신경망에서 layer 의 개수를 셀 때 이 입력층의 개수는 세지 않는데요. 그 이유는 입력층에서는 연산이 일어나지 않고 값을 받아서 은닉층으로 넘겨주는 역할만 하기 때문입니다. 따라서 위의 신경망은 $n=3$이라 했을 때, 4 층짜리 신경망이라고 할 수 있겠습니다.

## e ) 은닉층

은닉층은 입력층과 출력층 사이의 모든 layer 를 뜻합니다. 신경망이 복잡한 문제를 해결할 수 있도록 도와주는 핵심적인 계층입니다. 이 은닉층에서는 크게 세 가지 일이 일어납니다. ① 합계값 z 의 선형 연산, $z_j = \sum_{i=1}^{m} X * W + b$. ② z 의 활성화 함수 통과, $\sigma(z_{i+1})$. (이 둘은 아까 설명했죵?) ③ 다음 은닉층으로의 전달 $\overrightarrow{y_i} = \overrightarrow{x_{i+1}}$. 따라서 $i$번째 은닉층의 출력 값은 $i+1$번째 은닉층의 입력 값이 됩니다.

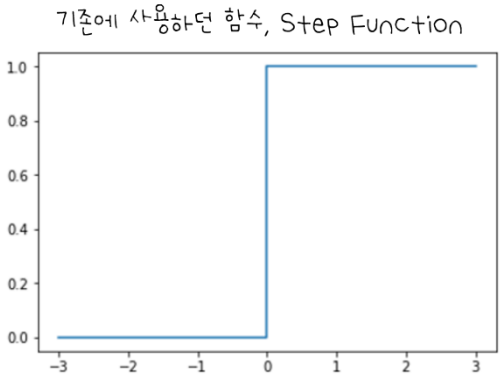
## f ) 출력층

출력층은 마지막 은닉층 다음에 오는 계층으로, 신경망의 외부로 출력 신호를 전달하는 층입니다. 출력 계층 노드의 개수가 출력 vector 의 길이와 같게 됩니다. 출력층 활성화 함수에 의해 전체 신경망이 어떤 역할을 하는 신경망인지가 결정됩니다. 예를 들어 출력층 활성화함수로 $y = x$를 사용한다면 회귀 문제를 다루는 신경망, $sigmoid(x)$를 사용한다면 이진 분류 문제를 다루는 신경망입니다.

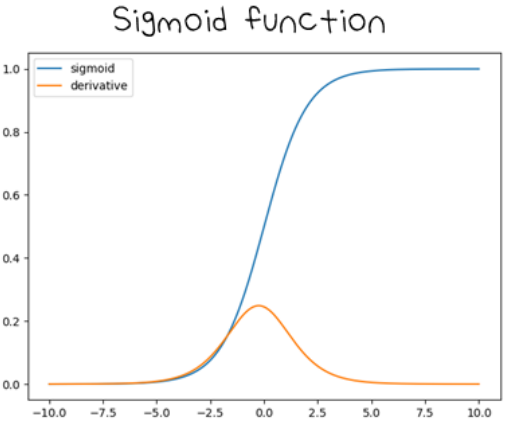
## • 3. 활성화함수

활성화함수의 사용처는 두 곳입니다. 은닉층과 출력층 두 곳입니다. 은닉층에서 사용되는 활성화 함수는 앞서 설명했듯, 선형결합을 거친 값을 활성화 함수에 통과시켜 비선형성을 더합니다. 출력층에서 사용되는 활성화 함수는 입력 신호의 총합. 즉 입력층에서 은닉층들을 거쳐 출력층에 온 값이 어떤 모양으로 출력될지를 결정합니다. 활성화를 일으키는 방법을 정하는 역할을 한다고 보시면 될 것 같습니다. 활성화 함수에는 여러 종류가 있습니다. 어느 한 가지가 늘 좋은 성능을 내는 것이 아니라, 모델의 목적과 성능에 따라 적합한 활성화 함수가 달라지는데요. 먼저, 은닉층에서 주로 사용되는 활성화함수를 살펴보겠습니다.

### a ) Sigmoid Function




Step Function, 즉 계단 함수는 가장 먼저 살펴보았던 퍼셉트론과 같은 간단한 모델에서 사용되는 활성화 함수입니다. 요즘 어떤 가게를 들어가도 체온을 측정하고 있는데요. 그 기계는 37.5℃ 이상이면 출입 금지(y=1), 그 이하면 출입 가능 (y=0)의 결과를 내는 모델입니다. 이와 같은 모델은 37.4℃, 혹은 37.49999999℃ 처럼 37.5℃ 에 아주 가까운 값이더라도 37.5 만 넘지 않으면 모두 0 을 출력한다는 한계점이 있습니다. 통계적으로 표현하자면, 확률이 표현되지 않습니다. 따라서 다양한 표현이 불가능하기 때문에 신경망에서는 기본적으로 아래의 시그모이드 함수를 활성화 함수로 사용합니다.

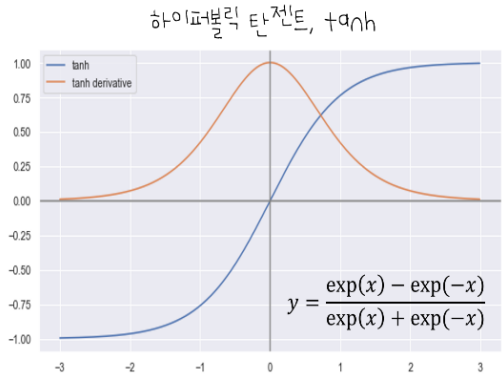



$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}.$$

시그모이드 함수는 0 과 1, 두 가지 값만 내는 계단 함수와는 달리 0 에서 1 사이의 모든 실수를 출력합니다. 입력값 $x$가 작을수록 출력값 $y$는 0 에 가까워지고, $x$가 커지면 $y$는 1 에 근접합니다. 이는 두 가지 장점이 있습니다. 첫번째로 미분이 불가능한 계단 함수와는 달리, 미분이 가능합니다. 미분이 가능하다는 것은 역전파가 가능하다는 뜻인데요. 역전파는 간단하게 말해 모델이 더 좋은 결과를 낼 수 있도록 하는 기능(?)으로, 신경망 학습에 사용됩니다. 이때 미분이 사용되기 때문에 미분이 가능하다는 것이 장점이 됩니다. 자세한 이야기는 나중에 할 것이니 이정도로만 이해하고 넘어가도 됩니다! 두번째로는 비선형 함수이기 때문에 변수 간의 좀 더 복잡한 관계를 설명할 수 있습니다.

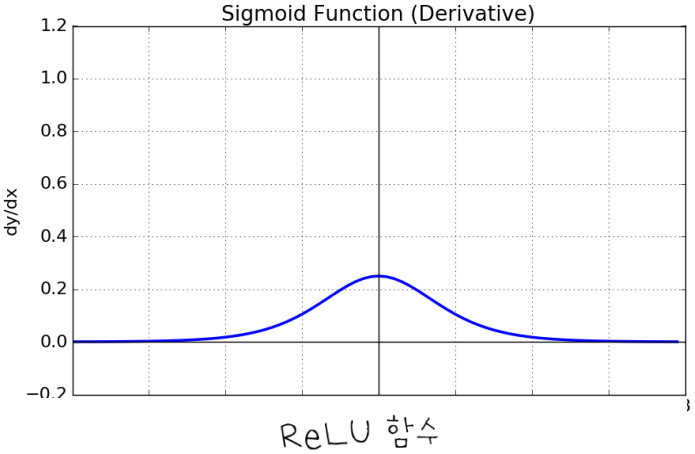
하지만 최근 시그모이드 함수는 잘 쓰이지 않고 있습니다. 대표적으로 두 가지 이유 때문인데요. 첫번째 이유는 $x$의 중심값이 0 이 아니라 1/2 이기 때문입니다. 이는 학습을 느려지게 하는 원인이 됩니다. 두번째 이유는 기울기 소실 문제를 야기합니다. 차례로 두 문제를 해결한 활성화 함수를 알아보도록 하겠습니다.

## b ) tanh

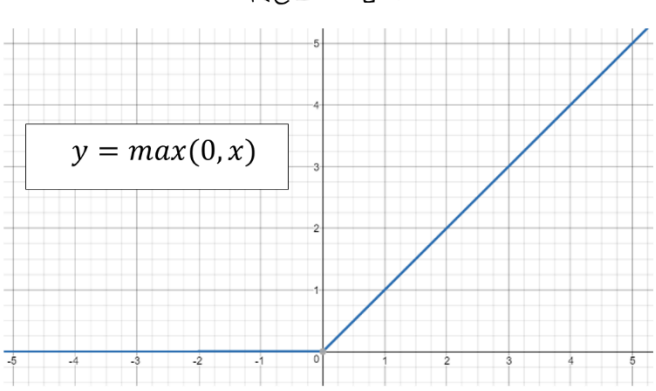



하이퍼볼릭 탄젠트 hyperbolic tangent 라고 읽는 tanh 는 -1 과 1 사이에서 매끄러운 곡선으로 변화하는 함수입니다. 쌍곡선 함수[1]중 하나로, 시그모이드 함수를 transformation 해서 얻은 함수라서 시그모이드와 아주 유사한 형태를 띠고 있습니다. Tanh 는 시그모이드 함수의 첫번째 단점을 해결한 함수로, 0 을 기준으로 대칭입니다. 즉, 중심값이 0 이기 때문에 시그모이드 함수보다 학습 속도가 빠른데요. 여전히 시그모이드 함수의 두번째 단점인 기울기 소실을 완벽히 해결하지 못했습니다.

## c ) ReLU Rectified Linear Unit




기울기 소실 (Gradient Vanishing) 문제는 역전파 시에 발생하는 문제입니다. 왼쪽의 그래프는 시그모이드 함수의 도함수 그래프인데요. 최대값이 0.25 로 아주 작습니다. 역전파 시에는 이 기울기 값을 계속 곱해주기 때문에 계속해서 최대 0.25 의 값을 곱 하다 보면 결국 0 이 되어버립니다. 이는 모델 성능에 악영향을 끼칩니다. 때문에 현재는 시그모이드 함수의 대표적인 문제 두 가지를 모두 해결한 ReLU 함수를 은닉층의 활성화 함수로 주로 사용하고 있습니다.

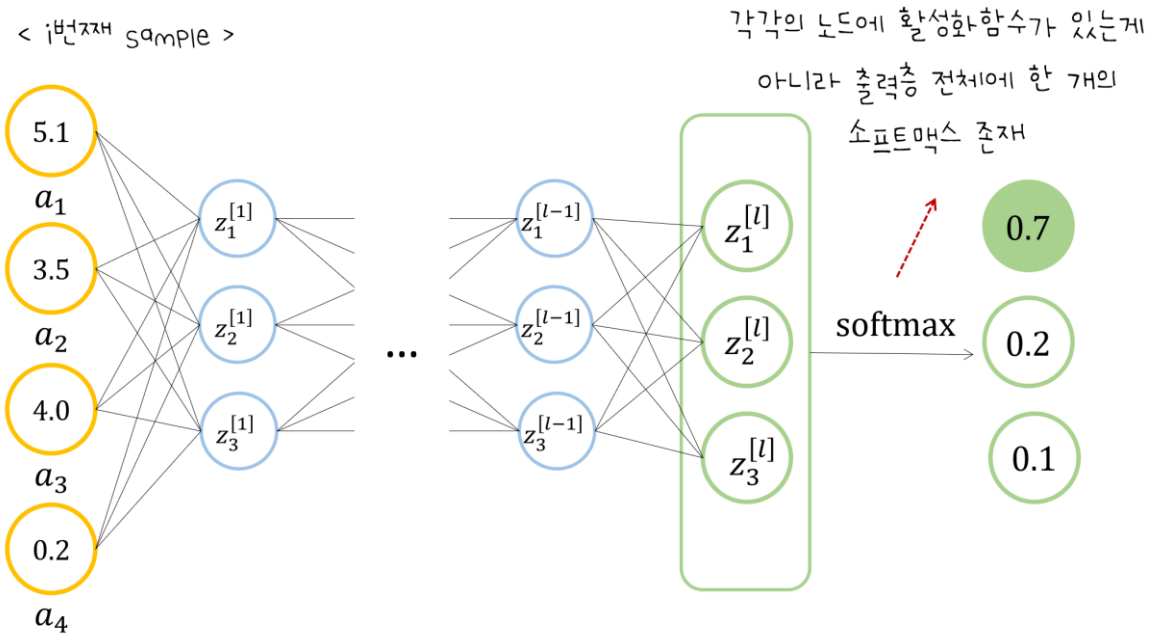



ReLU 는 램프함수라고도 불리는 함수입니다. $x$가 0 이하일 경우 0 을, 0 이상일 경우 그대로 $x$를 출력하는 단순한 함수입니다. 이처럼 간단하게 구현할 수 있으면서도, 층의 수가 많아져도 안정적으로 학습할 수 있기 때문에 자주 사용됩니다. ReLU 함수의 도함수는 $y = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$ 입니다. 도함수를 살펴보면 입력값 $x$에 상관 없이 안정적인 미분 값을 얻을 수 있어 안정적인 학습이 가능합니다. 기울기 소실 문제도 어느정도 해결할 수 있구요. 현재는 Leaky ReLU, PReLU 처럼 ReLU 함수에서 출발한 다양한 활성화 함수들도 만들어졌습니다.

지금부터는 출력층에 주로 사용되는 활성화 함수 두 가지를 살펴보겠습니다. 다시 한 번 리마인드 해보자면, 모델의 목적에 따라 출력층에 사용되는 활성화 함수가 달라집니다. 모델의 목적은 크게 분류와 회귀로 나뉘는데, 먼저 분류 문제에 사용되는 활성화 함수를 살펴보겠습니다.

[1] 쌍곡선 함수: 삼각함수와 유사한 성질을 가지고, 표준 쌍곡선을 매개변수로 표시할 때 나오는 함수

## d ) Softmax




Softmax 함수는 분류 문제 중에서도 다중 분류 문제 해결에 주로 사용됩니다. (이진 분류는 sigmoid 를 사용합니다.) 다중 분류는 여러 개의 답 중 하나를 고르는 분류 문제인데요. 수식보다도 먼저 어떤 식으로 동작하는지를 예시를 통해 살펴보려고 합니다. (제가 그림 만들었어요^^!!!) 아이리스 품종 예측 데이터를 활용해 3 가지의 품종 중 하나로 분류하는 모델을 만든다고 생각해볼게요. 다중 분류 문제니까 출력층 활성화 함수로 소프트맥스를 사용한 모델에 데이터의 i 번째 sample 을 Input 으로 넣어보겠습니다. 은닉층을 거쳐 출력층에 도달했을 때 $z_1^{[l]} = h_1^{[l]} \cdot w_{11} + h_2^{[l]} \cdot w_{21} + h_3^{[l]} \cdot w_{31} + b$로, 선형결합까지 마친 상태입니다. 출력층의 모든 노드를 소프트맥스에 넣으면 한 번에 각 노드의 확률 값이 0 에서 1 사이의 실수로 계산되어 나옵니다. 이 출력층 노드들의 총합은 1 으로, 각각은 특정한 범주로 분류될 확률을 나타내고 있습니다. 엄밀히 말하면 확률은 아니고, 소프트맥스를 통과함으로써 값이 0~1 사이로 모이는 것이지만, 확률이라고 이해해도 큰 문제는 없습니다. 신경망을 이용한 분류 문제에서는 일반적으로 가장 큰 출력을 내는 노드에 해당하는 클래스로만 인식합니다. 품종의 클래스가 순서대로 virginica, setosa, versicolor 라면 이 모델이 최종적으로 내는 예측은 'virginica'일 것입니다. 확률적으로는 'i 번째 sample 은 70%의 확률로 virginica 다'라고 말할 수 있겠지요? 수식을 통해 한 번 더 알아보겠습니다.

$$y_i = \frac{exp(a_i)}{\sum_{k=1}^{n} exp(a_k)}$$

왼쪽 식은 위 그림에서 '확률'이라고 적힌 부분 노드 하나의 값을 나타낸 것입니다. 그림에서는 $z_i^{[l]}$로 표현된 $a_i$를 지수함수에 넣은 것입니다. 이들의 총합은 아래의 식으로 나타냅니다. 복잡해 보이지만 모든 노드의 값을 다 더하면 1 임을 기억해주시면 될 것 같습니다. 지금까지 한 이야기를 요약하자면, 소프트맥스 함수는 다중 클래스 분류에 사용되며 확률값을 반환합니다.

$$y = \sum_{l=1}^{n} \frac{\exp(x_l)}{\sum_{k=1}^{n} \exp(x_k)} = \frac{\sum_{l=1}^{n} \exp(x_l)}{\sum_{k=1}^{n} \exp(x_k)} = 1$$

### e ) Identity Function

identity function 은 여러분 모두가 알고 계실, 항등 함수입니다. 범위에 제한이 없고 연속적이기 때문에, 연속적인 수치를 예측하는 회귀 문제를 다룰 때 자주 사용됩니다. 입력값을 그대로 출력값으로 출력합니다. 위의 신경망 그림을 예로 들어보겠습니다. 소프트맥스를 출력층 활성화 함수로 사용했을 때는 첫번째 노드의 예측값, 즉 $\widehat{y_1} = 0.7 = \sigma\left(z_1^{[l-1]} \cdot w_{11} + z_2^{[l-1]} \cdot w_{21} + z_3^{[l-1]} \cdot w_{31} + b\right)$ 이었습니다. 하지만 소프트맥스가 아니라 항등함수를 사용한다면 $\widehat{y_1} = z_1^{[l-1]} \cdot w_{11} + z_2^{[l-1]} \cdot w_{21} + z_3^{[l-1]} \cdot w_{31} + b$ 그대로 출력됩니다.

## • 4. 손실함수

지금까지 우리가 배운 내용을 한 번 정리해야 할 때가 온 것 같습니다. 신경망의 흐름을 한번 읊어보겠습니다.

**<순전파 Feed Forward Propagation> :** 우리가 지금까지 배운 과정을 '순전파'라고 합니다.

1. 입력층에서 입력값을 받습니다. 이때, sample 이 가진 feature 의 개수만큼 노드가 생성됩니다.

2. 입력된 값들은 은닉층으로 전달됩니다.
   은닉층에서는 합계값 $z_j = \sum_{i=1}^{m} X * W + b$의 선형결합과 은닉층 활성화 함수의 연산이 여러 번 발생합니다.

3. 모든 은닉층을 거친 값은 출력층으로 가게 됩니다. 최종적으로 선형결합과 출력층 활성화 함수의 연산이 발생합니다.

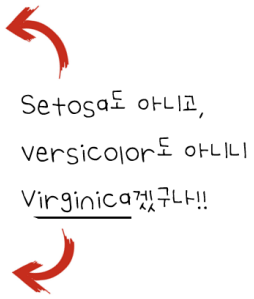
4. 그 값은 출력값, $\hat{y}$이 됩니다.

그러면 이 다음엔 뭘 해야 할까요? 바로 정답, y 와의 비교입니다. **$\hat{y}$와 y 의 비교를 통해 둘 사이의 차이를 점점 줄여 나가는 것을 '학습'이라고 합니다.** 이때 둘 사이의 차이를 계산해주는 함수를 바로 '손실함수'라 부릅니다. (학습의 목적이 되는 함수라는 뜻에서 목적 함수 objective function 라고도 불립니다. 학습하는 동안 이 값이 최소화되는 방향으로 학습이 진행되기 때문입니다.) 우리가 모델의 목적에 따라 다른 활성화 함수를 사용했듯, 손실함수 또한 모델의 목적, 즉 문제에 따라 다른 손실함수를 사용합니다. 분류 문제일 때, class1 이 99% class2 가 1%를 차지하는 데이터가 있다고 했을 때, 정확도를 손실함수로 쓴다면 모든 데이터를 class1 이라고 예측하는 편법을 모델이 사용할 수 있습니다. 이렇게 해도 정확도는 99%가 나올 테니 괜찮은 모델인 것처럼 보일 것입니다. 따라서 적절한 손실함수를 선택하는 일은 아주 중요합니다.

### a ) 교차 엔트로피 오차 Cross Entropy Error, CEE

교차 엔트로피 오차는 두 분포, 예측 값과 실체 값 분포 간의 차이를 나타내는 척도입니다. 분류 문제에서 손실함수로 많이 사용됩니다. 설명을 위해 다시 한번 아이리스 품종 예측 데이터를 생각해볼까요? 아이리스의 품종은 ['setosa', 'versicolor', 'virginica']의 세 가지입니다. 이들은 모두 0 과 1 로 구성되어 있습니다. 무슨 뜻인지는 아래 표를 통해 이해해 보도록 하겠습니다.

| | Setosa | versicolor |
|---|---|---|
| Sample 1 | 0 | 0 |
| Sample 2 | 0 | 1 |
| Sample 3 | 1 | 0 |
| Sample 4 | 1 | 0 |
| Sample 5 | 0 | 0 |
| Sample 6 | 0 | 1 |

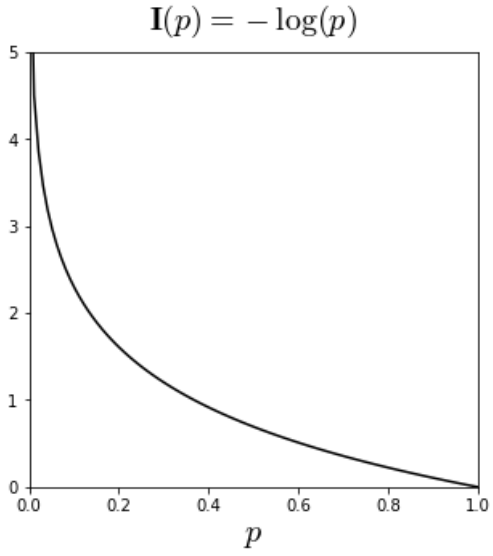



위의 표는 iris 데이터의 실제 값을 나타낸 것입니다. 실제 값의 이러한 형태를 '**원 핫 인코딩** One-hot Encoding'했다고 하는데요. 원래는 label ['setosa', 'versicolor', 'virginica']의 형태로 되어 있던 것을 각각 0 과 1 로 구성된 열로 만들어준 것입니다. Setosa [1,0], versicolor [0,1] 이렇게 말이예요. 컴퓨터는 문자를 인식하지 못하기도 하고, 활성화 함수를 사용하려면 Y 값이 0 과 1 로 이루어져 있어야 하기 때문에 원 핫 인코딩 단계가 필요합니다. 변환하고 싶은 범주는 ['setosa', 'versicolor', 'virginica'] 세개인데 왜 두개의 열만 생겼을까요? 그 이유는 다중공선성 문제를 예방하기 위해서입니다. Vertisa [0, 0, 1]이라는 열을 만들게 되면 Full rank 가 성립되지 않아서 모델 성능에 악영향을 미칩니다. 사실 이렇게 저차원일 때는 setosa[1,0,0], versicolor[0,1,0], virginica[0,0,1]로 원 핫 인코딩 해도 상관 없지만 고차원일 때는 문제를 일으키니 이렇게 알아둡시다!

$$E = \sum_{k=1}^{N} y_k(-log(\widehat{y_k}))$$

$$-\sum_{i=1}^{n}\left(y^{(i)}\log\left(a^{(i)}\right)+\left(1-y^{(i)}\right)\log\left(1-a^{(i)}\right)\right)$$

원 핫 인코딩은 크로스 엔트로피의 식을 설명하기 위한 발판이었습니다.. 위의 두 식은 크로스 엔트로피의 식입니다. 좀 달라보이지만 본질적으로는 같습니다. 위의 식으로 살펴볼까요? 실제 값$(y_k)$은 앞서 살펴봤듯이 0 과 1 로 이루어져 있고, 예측 값$(\widehat{y_k})$은 0 과 1 사이의 소수입니다. 이항 분류일 때는 시그모이드 함수를, 다항 분류일 때는 소프트맥스 함수를 통과하여 나온 결과 값이지요. 위의 식을 보시면 $y_k$를 $log\,\widehat{y_k}$와 곱해줌으로써, $y_k$가 1일 때만 그 값을 계산하게 됩니다. 따라서 실제 값이 1인 항의 예측 값만 오차에 영향을 주고, 실제 값이 0 인 항의 예측 값은 오차에 영향을 주지 않습니다. 즉, 실제 값이 1인 하나의 항에 대해서만 오차가 계산됩니다.




마이너스 로그를 취하는 이유는 양수 값을 내기 위해서입니다. $-log\,x$는 $x$가 1 일 때는 0 이고, $x$가 0 에 가까울수록 무한대로 커집니다. 이러한 성질에 따라서 예측 값이 1, 정답에 가까울수록 작은 오차를, 예측 값이 0 에 가까울수록 한없이 큰 오차를 냅니다. 따라서 학습 속도가 빨라집니다

## b ) 오차 제곱합 Sum of Squares for Error, SSE

$$E = \frac{1}{2}\sum_{k=1}^{N}(\widehat{y_k} - y_k)^2$$

오차 제곱합은 회귀 문제에서 자주 사용됩니다. 앞에 1/2 이 붙어 있는 이유는 역전파 시 미분을 하는데, 이 미분 값을 깔끔하게 하기 위해서입니다.

# Ⅲ. 딥러닝의 학습

## • 1. 역전파


우리가 지금까지 했던 걸 다시 한 번 정리해볼까요? 정리에 집착하는 것 같지만, 앞의 단계를 제대로 이해하지 못했다면 역전파를 이해할 수 없기 때문에 중요합니다. 우리는 지금까지 크게 순전파와 손실함수를 배웠습니다.

<순전파>

① 입력값을 입력 받는다. (이때, 각 노드는 특정 sample 의 feature)

② 가중치를 곱하고 편향 더하기 … $(z_j = \sum_{i=1}^{m} X * W + b)$, $(j = i+1)$

③ $z_j$를 활성화 함수에 넣기 … $a_j = \sigma(z_j)$

④ 다음 층의 입력 값이 되기 (②~④의 과정을 출력층 직전까지 반복)

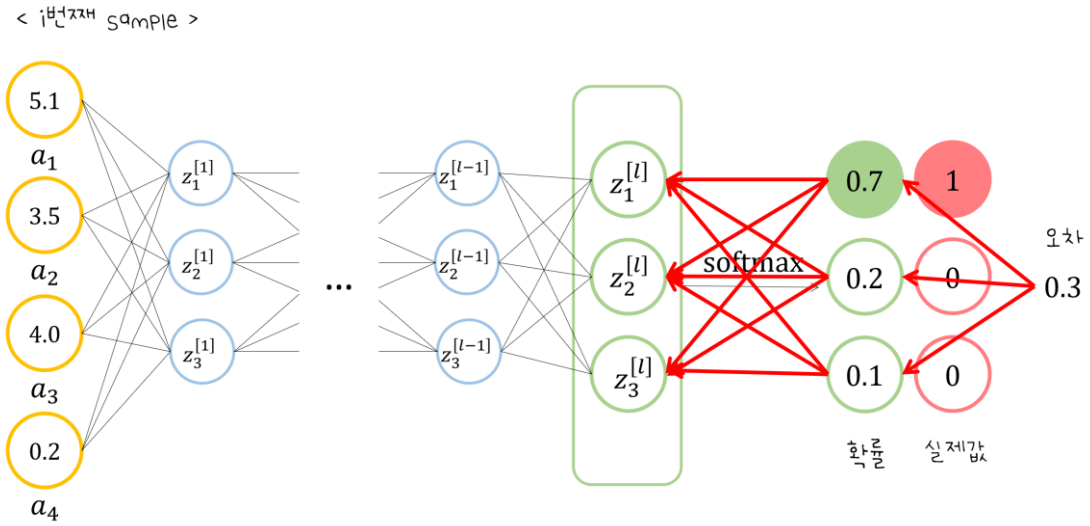
⑤ 출력층의 활성화 함수 통과 후 ➔ 출력값! (=예측 값)

**<손실함수>**

**⑥ 손실함수를 통해 출력 값과 실제 값을 비교한다.**

이 여섯 단계가 우리가 지금까지 배운 딥러닝의 학습 단계입니다.

학습이란, **$\hat{y}$와 y의 비교를 통해 둘 사이의 차이를 점점 줄여 나가는 것**이라고 이야기했습니다. 차이를 줄여서 예측 값을 실제값과 비슷하게 만드는 것이 목표인데. 어떻게 차이를 줄일 수 있을까요? 바로 모델의 파라미터를 조정하는 방식으로 줄여나갑니다. 여기서 파라미터는 가중치 $W$와 편향 $b$를 의미합니다. 인간이 가중치와 편향의 값을 정하는 것이 아니라 모델이 학습을 통해 알아서 가중치와 편향을 업데이트 하는 방식으로 학습이 진행됩니다.

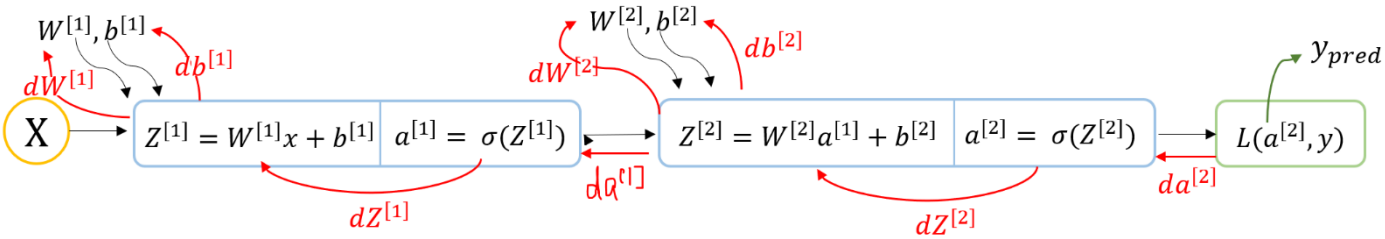



이와 같은 방식을 **역전파** Back Propagation 라고 합니다. 역전파는 일단 순전파를 한 번 진행해야 시행할 수 있습니다. 따라서 순전파부터 다시 살펴보겠습니다. 우리가 지금까지 순전파에서 곱해줬던 가중치 $W$와 편향 $b$는 무작위로 정해진

값입니다. (이를 Random Initialize 라고 합니다.) 단, 0 과 1 사이의 무작위 값을 정규분포로 뽑아 사용합니다. 랜덤하게 정한 가중치와 편향을 가지고 순전파를 한 번 진행하고 손실함수로 실제 값과 예측 값을 비교하여 오차를 구합니다. 이 오차를 미분을 활용하여 앞으로 앞으로 전달하면서 각 가중치와 편향을 업데이트 하는 과정이 역전파입니다. 그림의 빨간 화살표처럼 오차에서부터 시작해서 점점 역으로 나아가는 것입니다. (설명의 ez 함을 위해 오차는 0.3 이라고 생각할게요.) 아래와 같은 식을 통해 $W$와 $b$를 업데이트 합니다.

$$W \leftarrow W - \eta\left(\frac{\partial E}{\partial w}\right),\ b \leftarrow \eta\left(\frac{\partial E}{\partial b}\right)$$

여기에서 E 는 손실함수입니다. 따라서, $\frac{\partial E}{\partial W}$는 손실함수를 W 로 편미분한 것입니다. 오차에 대해 가중치가 가지고 있는 비중을 계산하는 것이라고 해석할 수 있겠습니다. 따라서 저 식은 비중이 큰 가중치, 즉 예측 값이 실제 값에서 많이 벗어나도록 하는 가중치를 크게 업데이트 합니다. 반대로 예측 값과 실제 값을 비슷하게 만드는 가중치는 작게 업데이트 되겠지요? 미분을 사용하는 것을 보니, 함수의 기울기를 지표로 하여 가중치가 전자와 후자 중 어떤 역할을 하는지를 결정하는 것임을 알 수 있습니다. 이처럼 **기울기** Gradient **를 활용**하여 함수의 값을 점차 줄여 나가는 방법을 **경사 하강법** Gradient Descent 이라고 합니다. 참고로, 저 편미분 값에 곱해지는 η (에타, 학습률)은 한 번의 학습으로 비중을 학습해야 할지, 다른 말로 W 또는 b 값을 얼마나 갱신할지를 정하는 값입니다. 학습률은 0.01 이나 0.001 등 미리 특정 값으로 정해두어야 합니다.




뭔가 삐뚤한게 하나 있지만.. 사랑으로 넘어가주세요^^..

설명을 위해 출력층까지 3 층으로 구성된 신경망을 데려왔습니다. 각 노드들은 vectorize 해서 한 개로 표현했구요. 역으로 오차가 전달된다는 흐름이 눈에 보이시면 좋을 것 같습니다. 각 값들은 연쇄법칙을 통해 구할 수 있습니다. 제가 구구절절 말로 설명하는 것 보다는 수식으로 보여드리는게 더 이해가 빠를 것 같아서 수식으로 보여드리겠습니다!

$$da^{[2]} = \left(\frac{dL}{da}\right)$$

$$dZ^{[2]} = \left(\frac{dL}{da}\right)\left(\frac{da}{dz}\right)$$

$$db^{[2]} = \left(\frac{dL}{da}\right)\left(\frac{da}{dz}\right)\left(\frac{dz}{db}\right)$$

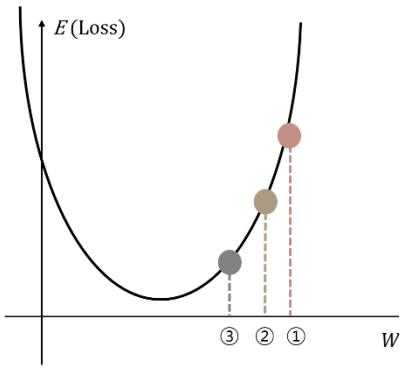
$$dW^{[2]} = \left(\frac{dL}{da}\right)\left(\frac{da}{dz}\right)\left(\frac{dz}{dW}\right)$$

뭔지 이해가 가시나요? 이해가 안 가신다면 물어봐주시고, $da^{[1]}, dZ^{[1]}, db^{[1]}, dW^{[1]}$은 직접 식을 써봅시다! 이렇게 단순화된 식 말고 실제 수식으로 계산해보는건 마지막에.. 해봅시다!!><

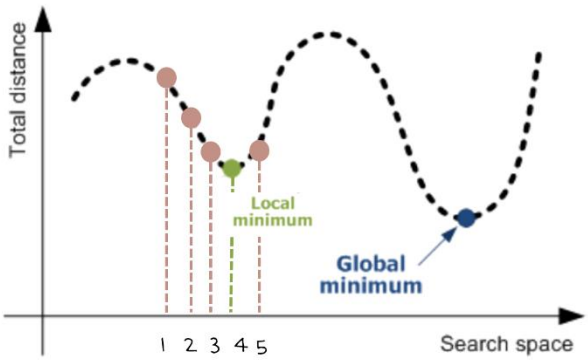
## • 2. 최적화 Optimizer

### a ) 기본적인 최적화 알고리즘

역전파에 대해서 조금 감을 잡으셨나요? 위에서 잠깐 설명했던 경사 하강법은 최적화 알고리즘의 한 종류입니다. 역전파가 순전파와는 반대 방향으로 값(오차)이 전달되는 것이고, 이를 통해 W 와 b 를 업데이트 하고, 이때 연쇄법칙이 사용된다는 큰 그림만 이해해주시면 좋을 것 같습니다. 아마 여기에서 역전파를 더 잘 이해하실 수 있을겁니다. 최적화는 역전파를 이용해 손실 함수의 값을 가능한 한 낮추는 파라미터(W 와 b)들의 최적 값을 찾아가는 과정으로, 역전파에서의 **업데이트 방법**을 담당합니다. 정리하면 역전파는 값을 역으로 전달한다는 의미이고 그 전달된 값들을 통해 어떻게 업데이트 하느냐는 최적화 알고리즘에 따라 달라집니다.




최적화는 최적 값을 찾아가는 과정이라고 말씀드렸는데, 우리는 최적 값을 찾을 수 없습니다. 무슨 소리냐면..? 파라미터의 개수도 엄청 많고, 값의 경우의 수 또한 무한합니다. 이 둘의 조합을 생각해보면 불가능하다고 생각되지 않나여? 지금까지 편의상 W 라고 해왔지만 레이어의 개수에 따라서 $W^{[1]}, W^{[2]}, W^{[3]}, \dots, W^{[L]}$개가 있고, 각 $W^{[i]}$는 $W_{11}, W_{12}, W_{13}, \dots, Wmn$의 행렬이니까요. 편의상 b 를 생략해도 이렇게 많습니다. 그래서 어떻게 하냐면 최적 값에 가장 가까워 보이는 방향으로 값을 쪼끔씩 이동하는 것입니다. 왼쪽의 그림에서도 첫번째에는 최솟값과 아주 멀었지만 세번째에는 첫번째 보다 최솟값에 조금 더 가깝다는 것을 볼 수 있습니다.



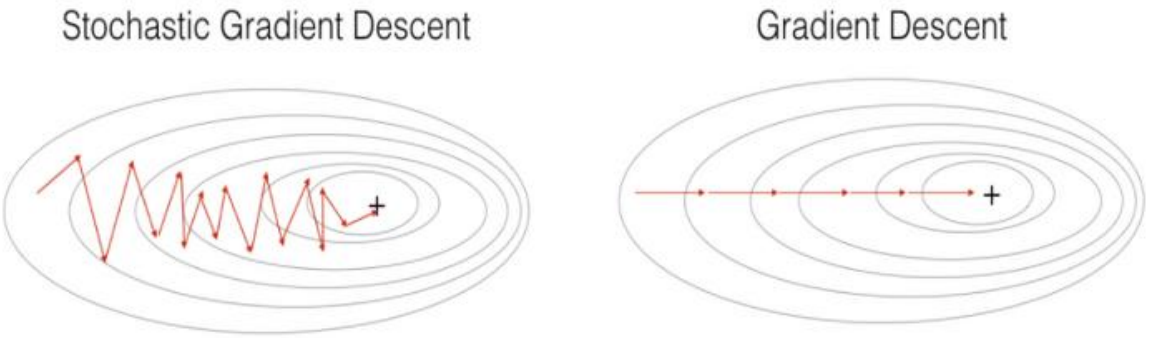

아까 봤던 경사 하강법은 한 번의 업데이트에 전체 데이터를 사용하는데요. 이는 국소 최적해, local optima 의 문제에서 빠져나오기 어렵습니다. 최초의 가중치가 1 번 자리에서 시작한다면, 가중치는 1~5 번을 지나 최종적으로 4 번의 값을 손실함수를 최소화하는 가중치라고 여길 것입니다. 4 번을 지나 5 번까지 갔는데, 5 번에서는 오히려 손실함수 값이 올라가니 4 번이 최솟값, 즉 최적해라고 판단하는 것입니다. 저 언덕을 넘으면 global minimum 에 갈 수 있지만 언덕 너머 무엇이 있는지를 알 수 없으니까 4 번을 최솟값으로 하고 최적화를 끝낼 것입니다.

이런 문제를 해결하고자 하는 가장 기본적인 최적화 알고리즘 두 가지를 알아보겠습니다.

#### 1. 확률적 경사 하강법 Stochastic Gradient Descent, SGD

확률적 경사 하강법은 가장 기본적인 최적화 기법입니다. 배치 사이즈가 1 인 경사하강법 알고리즘이지요. 배치 사이즈가 뭐냐면, 한 번 업데이트에 사용할 데이터의 개수를 뜻합니다. 앞서 살펴봤던 경사하강법은 전체 데이터셋을 한 바퀴 돌았을 때 한 번 업데이트가 발생한다고 했으니, 배치 사이즈 = number of rows 입니다. 이처럼 전체 데이터 셋을 한 번 학습에 사용하는 학습 방법을 배치학습이라고 합니다. 이렇게 계산하는 경우, 아까 말했던 국소최적해 문제도 있지만 무엇보다 학습 시간이 너무 오래 걸립니다. 데이터가 10 만 개 있다고 했을 때, 한 번 업데이트를 위해서 10 만 번의 순전파와 각각의

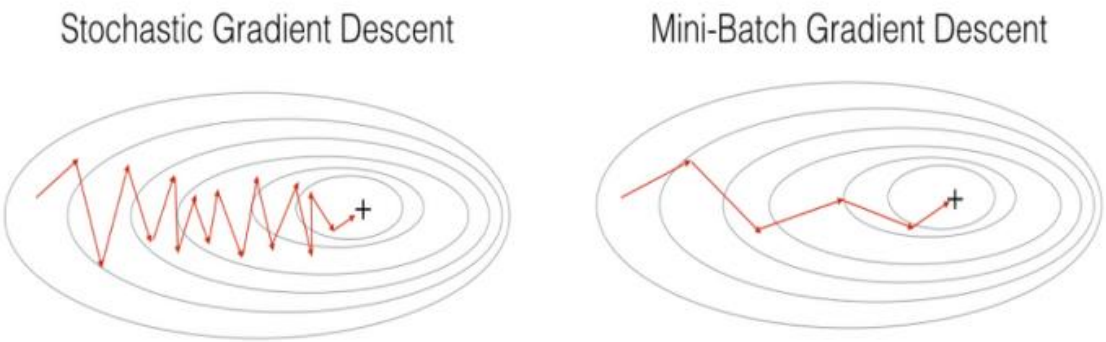
손실함수 값을 구해 평균 내고, 그 평균 값으로 10 만 번의 역전파를 진행 해야 합니다. 쪼끔 무리일 것 같죠? 그래서 확률적 경사 하강법은 샘플 한 개 마다, 한 번의 업데이트를 진행합니다.




확률적 경사 하강법은 한 개 데이터의 손실함수 값을 가지고 업데이트를 진행하기 때문에 각 데이터의 특성에 따라 이리저리 튀어다닙니다. 이를 Shooting 한다고 합니다. 반면 경사 하강법은 느리지만 안정적으로 최적값을 향해서 갑니다. 마찬가지로 데이터가 10 만개 있다고 할 때, 확률적 경사하강법은 1 에포크 당 10 만 번의 업데이트가 일어납니다. 따라서, 배치학습보다 학습 속도는 빠르지만 여전히 국소 최적해 문제는 해결하지 못했습니다. (후술하겠지만 현실적으로 국소 최적해 문제에 빠지는 상황은 드뭅니다.)

## 2. 미니배치 경사 하강법 Mini-batch gradient descent, MSGD

전체를 다 쓰거나, 하나만 쓰거나. 이렇게 극단적인 방법 말고 중간의 m 개만 선택하면 안 되나? 하고 생각했던 사람들이 있었나봅니다. 미니배치 경사 하강법은 m 개의 샘플을 random 하게 뽑아서 순전파 시킨 후, 손실함수 값을 평균 내고, 그 값으로 역전파를 진행합니다. 배치 경사하강법과 다른 점은 전체 샘플이 아니라 m 개의 샘플만 사용한다는 점입니다. 이쯤되면 미니배치가 무엇인지 감이 오실 것입니다. 미니배치는 전체 샘플 중 m 개의 샘플만 사용하여 학습하는 방법입니다. m 개의 샘플만 사용한다는 건, 다시 한 번 말하자면 m 개의 손실함수 값 평균만을 이용해 파라미터를 업데이트 한다는 것입니다. 1epoch 당 $number\ of\ rows\ /\ m$번의 업데이트가 일어나는 것입니다. 사실 요즘엔 미니배치 경사 하강법을 확률적 경사 하강법과 혼용하여 사용하지만, 저는 그냥 명확하게 설명하고자 둘을 나눠보았습니다.

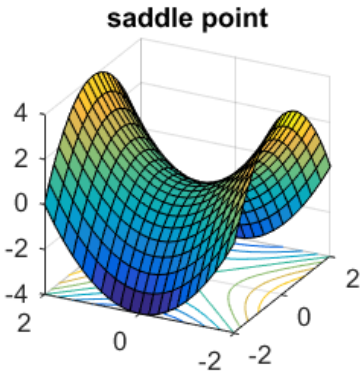



확률적 경사 하강법과 미니배치 경사 하강법의 비교입니다. 확실히 확률적 경사 하강법 보다는 미니 배치 경사하강법이 노이즈가 더 적습니다. 배치 사이즈는 하이퍼 파라미터로, 사람이 직접 설정해야 하는 값인데. 주로 2 의 제곱수를 선택합니다. 미니배치에 대해서는 아래에서 더 자세하게 알아볼 것입니다.

## b ) 현실적인 문제

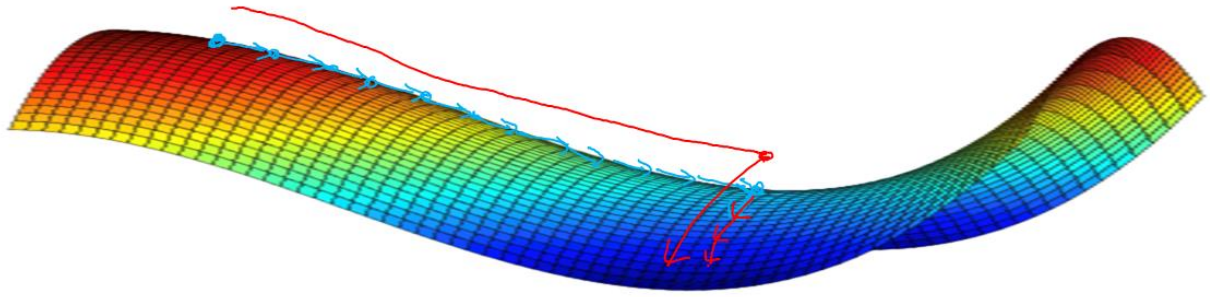
그러나 현실적으로 국소 최적해 문제는 빠지기 쉽지 않은 문제입니다. 오히려 다른 것들이 더 문제입니다. 그 이유는 실제 데이터는 훨씬 고차원의 데이터이기 때문입니다. 위에서 국소최적해를 설명하며 보여드렸던 그래프는 2 차원이기 때문에 국소 최적해 문제에 빠지기 쉽습니다. 그렇지만 고차원에서는 수많은 w 들이 모두 국소최적해에 **동시에** 빠져 있어야 더 이상 업데이트를 진행하지 않기 때문에 이러한 경우는 거의 발생하지 않습니다. 현실적인 문제 두 가지와, 이를 해결하기 위한 여러가지 최적화 알고리즘들을 맛만 보겠습니다.

### 1. 안장점 Saddle point



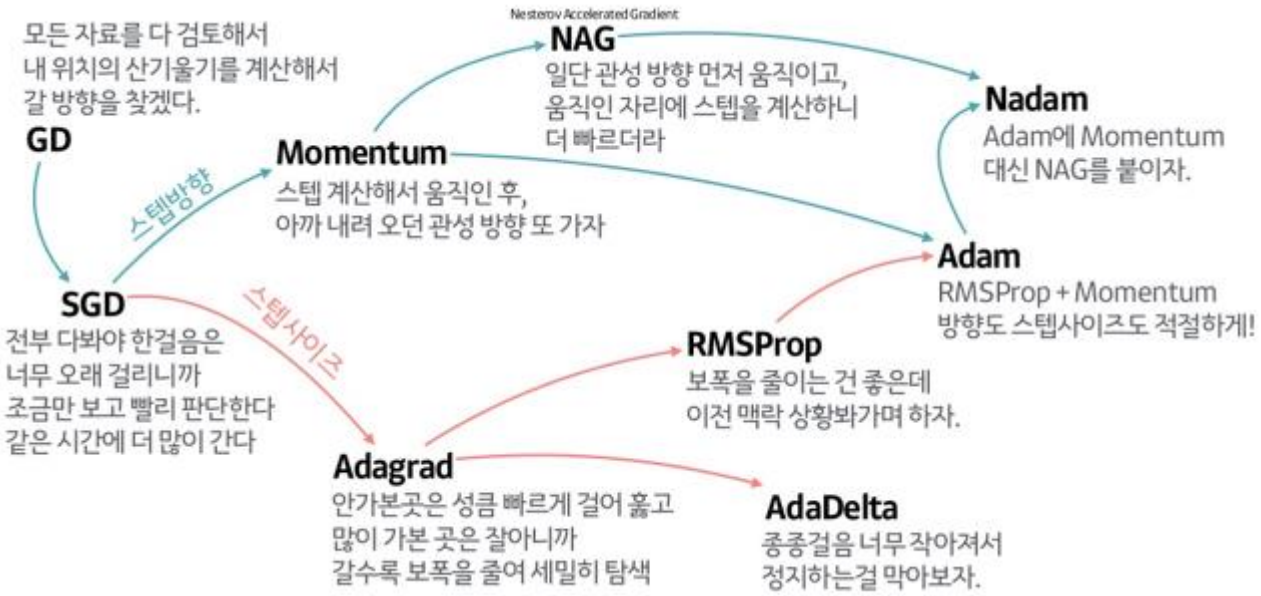

안장점은 왼쪽 그림과 같은 문제입니다. 고차원 함수에서 기울기 gradient 가 0 이 되는 지점은 방향에 따라 아래로 볼록이거나, 위로 볼록일 수 있습니다. 즉, 최대값과 최소값이 공존하는 상황입니다. 함수의 차원이 높을수록, 국소 최적해보다는 안장점이 많을 확률이 훨씬 큽니다. 그 이유는 앞서 언급했듯이, 국소 최적해가 되기 위해서는 수많은 w 들이 모두 동시에 국소 최적해에 빠져 있어야 하기 때문입니다.

### 2. Plateau



plateau 는 한국어로 '평지'입니다. 그림을 보시면 왜 문제인지 느낌이 오실 것 같습니다. Problem of plateau 는 그래프에서 기울기가 0 에 가까운 지점들이 길게 늘어져 있어 발생하는 문제입니다. 기울기가 0 이기 때문에 도함수 값이 작아서 파라미터 업데이트가 거의 되지 않습니다. 따라서 학습 시간이 느려지게 만드는 원인입니다.

## 3. 최적화 알고리즘들



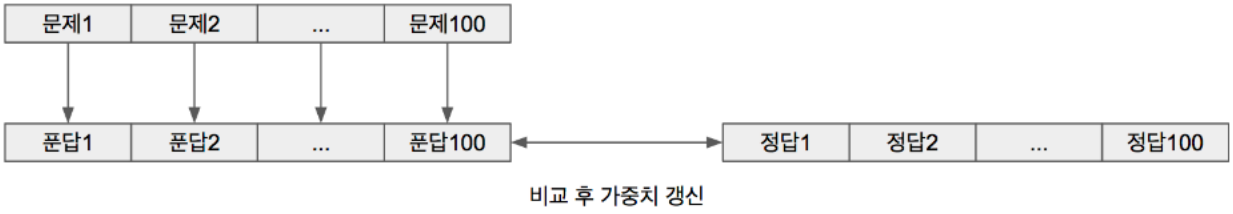

현재는 이처럼 다양한 최적화 알고리즘들이 있습니다. 각각을 살펴보기에는 시간도 부족하고 상당히 복잡한 수식들을 많이 소개해야 해서.. 이렇게만 소개해드리려고 합니다. 참고로 최근 가장 널리 사용되는 최적화 알고리즘은 Adam 입니다. RMS Prop 과 Momentum 의 장점만 뽑아온 알고리즘입니다.

더 알아보고 싶은 분은 아래 링크로!

https://velog.io/@minjung-s/Optimization-Algorithm/

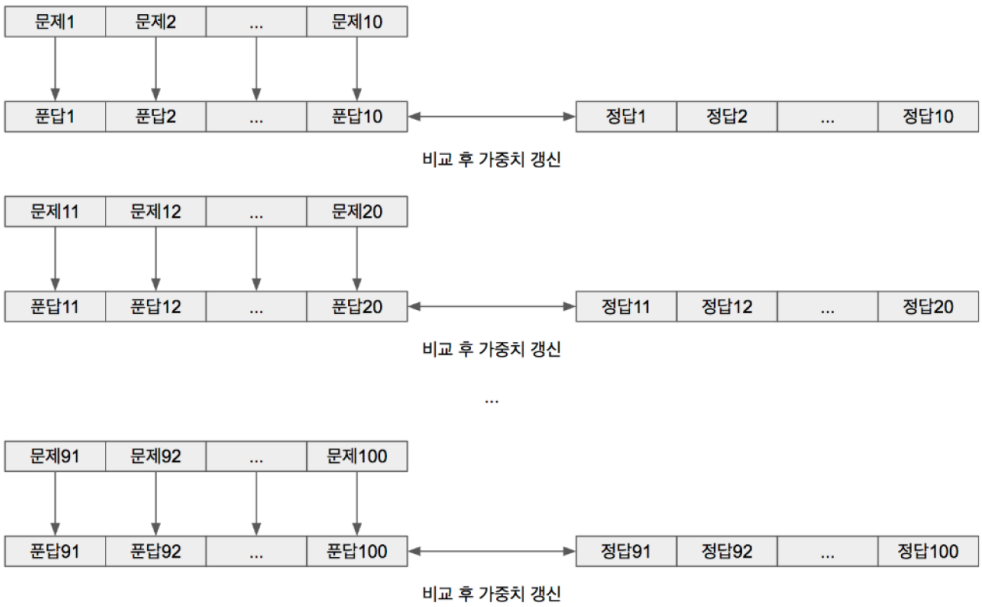
## • 3. 미니배치

앞서 미니배치 경사 하강법을 설명하며 배치와 미니배치에 대해 설명했습니다. 다시 정리하자면, 가중치를 한 번 업데이트 할 때, 배치학습은 전체 데이터셋을 사용하고 미니 배치는 전체 데이터 셋 중 m 개의 데이터셋만 사용하는 방법입니다.
역전파 시, 배치학습은 전체 데이터셋 손실함수의 평균을, 미니 배치 학습은 m 개 데이터 셋 손실함수의 평균을 사용합니다.




배치학습

배치학습과 미니 배치 학습은 모두 장단점을 가지고 있습니다. 배치 학습은 모든 데이터를 가지고 학습하기 때문에, 모든 경우에 대응하는 학습이 가능합니다. 주어진 데이터에서 만들 수 있는 가장 일반화된 모델이 되는 것입니다. 하지만 이를 위해서는 모든 데이터의 손실값을 저장해야 합니다. 이는 컴퓨터 자원의 한계로 인해 제한이 있습니다. 현실적으로 10 만개, 1000 만개의 손실값을 저장하는 컴퓨터는 우리가 가질 수도 없겠죠?




미니 배치는 m 개만 사용하는 만큼, 현실적이고 빠른 학습이 가능합니다. m 개를 사용한 후, 다음 m 개를 사용할 때 이전에 저장해놨던 손실 값은 모두 지웁니다. 따라서 저장이 누적되지 않아 상대적으로 빠릅니다. 다만, 미니 배치 각각이 가지는 데이터의 편향이 존재합니다. 아까 확률적 경사 하강법을 보면서 데이터 한 개 마다 영향을 받기 때문에 노이즈가 심한 것을 보셨을 것입니다. 미니 배치 또한 데이터를 한 개만 쓸 때 보다는 적지만, 여전히 노이즈가 있습니다. 이로 인해 일반화된 모델 보다 편향된 모델로 학습될 가능성이 높습니다. **정리하자면, 배치 사이즈가 커질수록 일반화된 모델이 탄생하지만 시간이 오래 걸립니다. 배치 사이즈가 작아질수록 빠르게 학습이 되지만 편향된 모델이 탄생합니다.**

아까 전에 1 에포크(epoch)는 전체 데이터 셋을 한 번 학습시킨 것이라고 말씀 드렸습니다. 우리는 에포크의 숫자 만큼, 즉 1000 에포크라면 전체 데이터 셋을 1000 번, 학습시킵니다. 이 에포크의 숫자를 정해주어야 합니다. 학습 횟수가 많아질수록 시간이 오래걸리고 과적합의 위험이 커집니다. 과적합에 대해서는 밑에서 알아볼 것입니다.

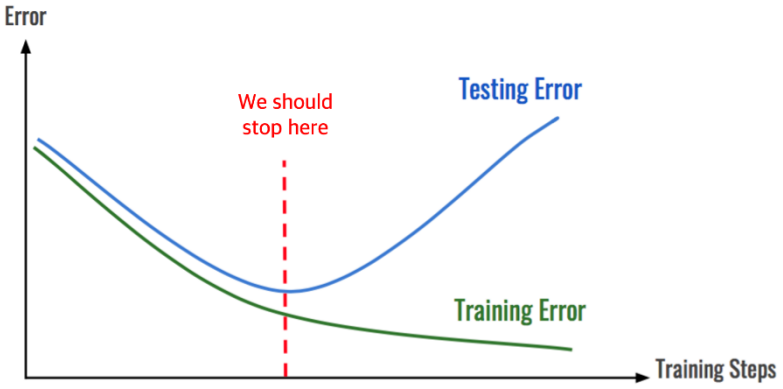
# IV. 모델 성능 향상 시키기

## • 1. 오버 피팅 피하기

### a ) Over fitting?


오버 피팅은 모든 머신러닝과 딥러닝이 고민하는 문제입니다. 오버 피팅이란 그림에서 볼 수 있듯이, 모델이 train set 에 과적합된 상태를 말합니다. 초등학교 때, 구구단은 줄줄 외워서 9*8, 7*6, 5*4, 어떤 숫자를 물어도 1 초만에 답을 낼 수 있는 친구들이 있었을 것입니다. 하지만 공식을 이해하지 못하고 답만 외웠다면 구구단에서 벗어난, 3*12, 8*15 와 같은 값을 물어보면 답을 하지 못하죠. 과적합은 이런 상태입니다. 문제를 이해하고 푸는 것이 아니라, 문제에 대한 답을 외운 것입니다. 곱셈 공식을 이해하지 못했다면 그 친구는 곱셈을 할 수 있는 것이 아니겠죠?

오른쪽의 그림을 보면 학습 횟수에 따른 Training error 와 testing error 입니다. Training error 은 학습을 하면 할수록 줄어드는데, testing error 은 오히려 어느 시점부터는 올라가는 것을 볼 수 있습니다. 이럴 때 우리는 과적합이 되었다고 합니다. 오버피팅을 피해야 일반화된 모델, 우리가 풀고 싶은 문제를 잘 푸는 모델을 만들 수 있습니다. 어느 곱셈 문제를 내도 잘 풀어야 좋은 모델이겠죠? 구구단만 답할 줄 알면 어디에 써먹겠어요. 그럼 이제 어떻게 하면 오버피팅을 피할 수 있을지 알아봅시다.



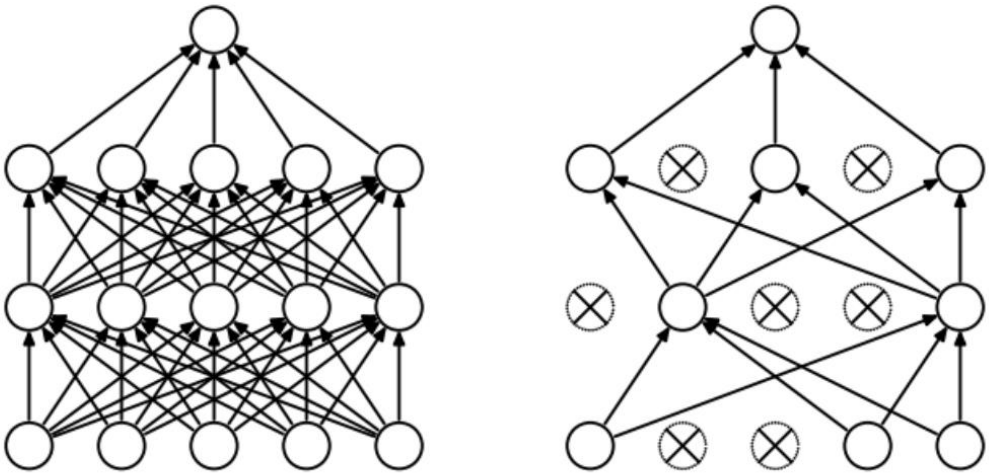

오버 피팅을 피하는 방법에 대한 자세한 이야기는 데이터마이닝 팀을 참고하면 좋을 것 같습니다. 여기서 다루기엔 너무 많은 양이거든요.

### b ) 데이터 수, 모델 구조

결론부터 말하자면 **데이터 수가 많을수록, 모델 구조가 단순할수록 과적합 우려가 적어집니다**. 말처럼 간단한 이야기는 아닙니다. 데이터 수는 현실적으로 제한되어 있기 때문에 우리가 조절하기 어렵습니다. 따라서 우리가 조절할 수 있는 부분은 모델 구조입니다. 모델 구조를 조절할 수 있는 대표적인 방법은 파라미터의 개수입니다. 파라미터의 개수를 줄이기 위해서는 layer 내부의 노드 개수를 줄이거나, layer 개수 자체를 줄여야 합니다. 하지만 복잡한 문제일수록(변수의 개수가 많고, 데이터의 형태가 비정형에 가까울수록 복잡한 문제인 경향이 있습니다.) 모델이 적절히 복잡한 모델을 사용해야 일정 수준

이상의 성능을 달성할 수 있습니다. 즉, 적당히 복잡해야 좋은 성능을 내면서 과적합을 피할 수 있습니다. 이는 모델 구조를 바꿔가며 비교를 해보아야 알 수 있습니다.

## c ) Dropout



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

위에서 말한 복잡하면서 단순한 모델은 사실 한계가 있습니다. 두 개념 사이의 타협을 통해 만들어졌기 때문입니다. 그래서 등장한 것이 드롭아웃 dropout 입니다. 드롭아웃은 출력층 이외의 뉴런을 일정한 확률로 무작위로 제거하는 방법입니다. 이론적으로 입력층의 뉴런도 드롭아웃 할 수는 있지만 안 하는 것이 일반적이라고 합니다. 한 번 학습 시 마다 일정 비율의 노드를 사용하지 않는 방식입니다. 이를 통해 모델 학습 시에는 모델 구조가 단순해져 과적합의 우려가 적어지고, 전체 모델은 문제에 적합할 정도로 복잡해 성능은 좋아지게 됩니다. 레이어마다 삭제할 노드의 비율을 다르게 설정할 수도 있고, 비교적 손쉽게 구현할 수 있어 자주 사용합니다.

## d ) Regularization

모델 복잡도에 대한 패널티로, 정규화는 과적합을 예방합니다. Regularization 은 특정 가중치가 너무 과도하게 커지지 않도록 만드는 기능을 합니다. 다르게 말하자면 모델이 복잡해질수록 손실함수의 값이 커지도록 만드는 것입니다. 모델의 설명도는 유지하면서 복잡도는 줄이는 것입니다. 종류는 L1 regularization 과 L2 regularization 의 두 가지가 있습니다. 손실함수 식을 수정함으로써 사용됩니다.

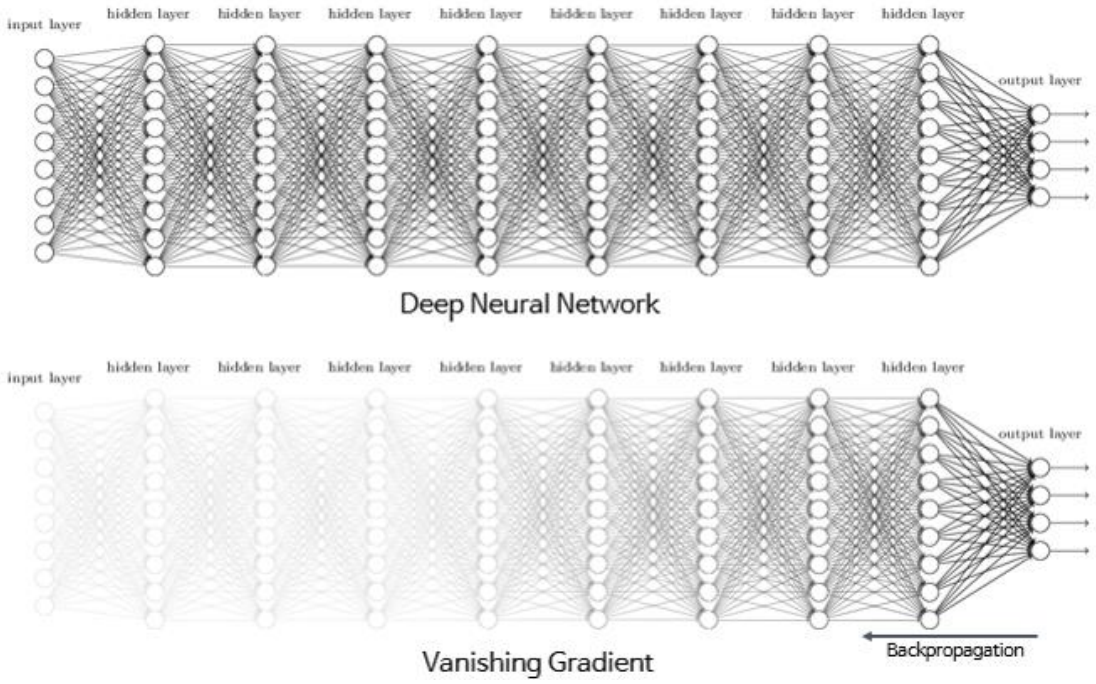
$$cost = \frac{1}{n}\sum_{i=1}^{n} L(\hat{y}_i, y_i) + \frac{\lambda}{2}|w|(1) \qquad (1)\ L1\ Regularization$$

$$cost = \frac{1}{n} i = 1nL(widehaty_i, y_i) + \frac{\lambda}{2}|w|^2 \qquad (2)\ L2\ Regularization$$

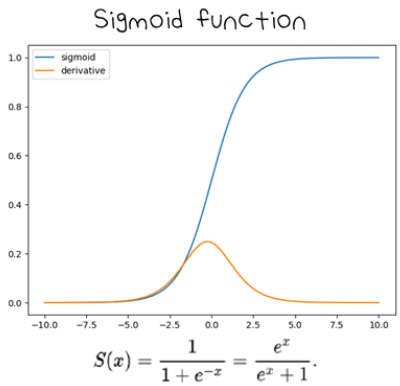
# • 2. 기타 등등

여기에는 앞의 내용들을 모두 알고 있어야만 이해할 수 있는 개념들을 모아봤습니다. 중요한 내용이지만 하나로 묶기에는 공통으로 묶을 수 있는게 없어서 기타 등등이라구 적어보았어요..><..!!!!@@ 앞에서 “나중에 이야기하겠습니다.” 해놓고 아직까지 이야기하지 않은 내용들 모음집입니다.

## a ) 기울기 소실 문제 Vanishing Gradient Problem




기울기 소실 문제는 역전파 시에 발생합니다. 층을 거슬러 올라감에 따라 기울기가 0 에 가까워지는 현상을 의미합니다. 기울기는 역전파 시 계산되는 미분 값이고, 가중치를 얼마나 업데이트 할 지 정하는 양입니다. 기울기가 크면 가중치를 많이 업데이트 하고, 기울기가 작으면 가중치를 적게 업데이트 합니다. 기울기가 작다는 것은 실제 값과 예측 값의 차이가 작거나, 해당 가중치가 오차에 미친 영향이 작다는 의미입니다.




기울기 소실 문제는 신경망의 층이 깊어질수록 많이 발생합니다. 그 이유는 활성화 함수로 사용하던 시그모이드 함수 때문인데요. 왼쪽의 주황색 그래프는 시그모이드의 미분값(기울기)그래프입니다. 최대값이 0.25 이므로 층을 거슬러 올라갈 때마다 각 기울기가 작아집니다. 왜 그런지 이해가 잘 안 간다면 역전파 식을 다시 한 번 잘 살펴보면 이해가 가실 겁니다. 활성화함수의 미분값을 계속해서 곱해주거든요. 0.25 를 수십번, 수백번 곱해지자 입력층과 가까운 laeyr 의 가중치들은 실제값과 예측값 사이의 오차나, 해당 가중치가 오차에 미친 영향력과 관계 없이 작은 미분값을 받게 되어, 가중치가 변하지 않게 되었습니다. 즉, 출력층에서 멀어질수록 미분값 자체가 작아져 결국 사라지는 현상이 발생하는 것입니다.

이 현상은 모델이 학습을 하지 않는 문제를 초래했습니다. 하지만 우리 모두 알다시피 지금은 다른 좋은 활성화 함수들이 많이 개발되면서 이 문제에서 많이 자유로워졌습니다.

## b ) 가중치 초기화 weight initialize

딥러닝에서의 학습은 최적의 파라미터를 탐색하는 것이라고 배웠습니다. 따라서 가중치의 초깃값은 때때로 학습의 성패를 좌우합니다. 가중치와 편향의 초기값은 무작위로 설정하는 것이 좋습니다. 가중치를 0 으로 설정할 경우, 모든 값이 0 이 되면서 아무리 학습을 해도 학습이 되지 않습니다. 다만, 편향은 0 으로 initialize 해도 큰 상관 없습니다. 가중치의 초기값이 0 이 아니더라도, 0 에 너무 가까운 값이면 학습이 거의 되지 않습니다. 앞에서 말했던 plateau 와 같은 상황에 놓이는 것입니다. 학습 속도가 무척 느려집니다. 반대로 가중치의 초기값이 너무 커도 문제입니다. 모델이 학습 데이터를 과잉학습하여 오버피팅 문제가 발생할 수 있기 때문입니다. 이를 고려하여 무작위로 선정해봅시다

$$r = \sqrt{\frac{3}{n_{in}}}[LeCun1998]$$
$$r = \sqrt{\frac{6}{n_{in}+n_{out}}}[Glorot2010]$$
$$r = \sqrt{\frac{6}{n_{in}}}[KaimingHe2015]$$

Uniform distribution

$$r = \sqrt{\frac{1}{n_{in}}}[LeCun1998]$$
$$r = \sqrt{\frac{2}{n_{in}+n_{out}}}[Glorot2010]$$
$$r = \sqrt{\frac{2}{n_{in}}}[KaimingHe2015]$$

gaussian distribution

위의 식들은 가중치의 초기값 범위를 최적의 가중치 범위에 근사하게 정할 수 있도록 하는 식입니다. 즉, 최대한 정답에 가까운 곳에서부터 시작한다는 의미입니다. 이렇게 되면 학습 속도가 더욱 빨라지겠지요? 둘 중에 하나를 사용하시면 됩니다. 사용하고 싶은 분포에 따라 왼쪽 또는 오른쪽의 r 을 이용해 (-r, r)의 범위에서 무작위로 추출하면 됩니다. 얘네가 완전히 들어맞는다기 보다는 경험상 얘네들로 하는게 그나마 낫더라 하는 정도이니 참고해두세요!

## c ) Normalization

딥러닝 모델은 모두 비선형적 결합의 반복입니다. 노드 간의 비선형적 결합을 통해 최적 가중치를 찾는 것이 공통적인 목표입니다. 이때, 변수 간 범위가 많이 차이가 날 경우 모든 변수들의 중요도가 동일하게 취급되지 않습니다. 때문에 전처리를 통해 신경망의 성능을 향상시키고 학습 속도를 향상시킬 수 있습니다.

Normalization 은 feature 간의 스케일을 조정한다는 의미로, feature scaling 이라고도 불립니다. Normalization 은 데이터가 특정한 범위 안에 들어가도록 변환하는 역할을 합니다. 데이터의 분포는 유지한 채, 범위를 줄여주는 것이라고 이해하시면 됩니다. 다양한 normalization 기법이 있지만 가장 대표적인 기법은 최대값과 최소값을 이용한 Min-Max scaling 입니다. Min-Max Scaler 를 이용하면 모든 feature 가 최소값이 0 이고 최대값은 1 으로 변환됩니다.

$$X_{new} = \frac{(X - X_{min})}{(X_{max} - X_{min})}$$

# CODE!

딥러닝 처음인 분도 계시고 파이썬에 익숙하지 않은 분도 계시고 하니까 1 주차는 쪼끔 ez 하게 만들어보았습니다! 스터디 코드 과제는 쉽게 쉽게 나갈거니까 부담 갖지 않으셔도 되어용~~ 오늘은 아까 softmax 설명하면서 들었던 예시인 iris 품종 예측 알고리즘을 만들어보려고 합니다.

0. 텐서플로, 케라스 설치하기

1. 아래 코드 실행

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import tensorflow as tf
import seaborn as sns
iris = sns.load_dataset('iris')
```

2. X 와 target 으로 데이터 분리

3. 문자열을 숫자로 변환 (원 핫 인코딩) → hint : LabelEncoder() 사용

4. Train-test set 분리하기 → hint: train_test_split , random_state = 417

5. Data EDA → 자유롭게 해주세요!

6. 모델 설정 → hint: import 한 모듈들 중 아직까지 사용 안 한 것들을 찾아볼까요?

은닉층 2 개짜리 모델을 만들어 주세요. 은닉층의 activation 함수는 relu, 출력층의 actvation 함수는 softmax 사용해주세요.

입력층 노드는 4 개, 첫번째 은닉층에서는 30 개의 노드를, 두번째 은닉층에서는 16 개의 노드를 사용해주세요.

7. 모델 컴파일

Loss 는 적절한 걸로 설정해주세요. Optimizer 는 adam, metrics 는 정확도로 해주세요.

8. 모델을 실행

모델을 실행해주세요. Epcohs = 100, 배치 사이즈는 1 로 해주세요.

9. 테스트셋에 모델 예측

10. 해석 (아주!! 간단하게 해주세요.)

# IV. 부록

은닉층에서는 Relu 를, 출력층에서는 sigmoid 를 사용하는 신경망 모델을 하나 만들려고 합니다. 은닉층의 개수는 L 개입니다.

## (1) 핵심 함수 구현

```
Import numpy as np
def sigmoid(Z):
    A = 1/(1+np.exp(-Z))
    cache = Z
    return A, cache
```

```
Import numpy as np
def relu(Z):
    A = np.maximum(0,Z)
    cache = Z
    return A, cache
```

```
Import numpy as np
def softmax_function(Z):
    A = np.exp(Z)/np.sum(np.exp(Z))
    cache = Z
    return A, cache
```

Cache 를 따로 저장하는 목적을 미리 지금 간단히 말해두자면, 역전파 시 동적 계획법 Dynamic Programming 을 하기 위해서입니다. 동적 계획법을 그냥 간단히 제 맘대로 설명하자면?! 나중에 또 쓸 값인데 미리 저장 안 해 놓으면 나중에 또 계산해야 하는 번거로운 일이 생기니까 한 번 계산했을 때 저장해두는 방법입니다. 아직은 cache 를 왜 저장하는지 이해가 되지 않아도 괜찮습니다! 밑에 내려가면서 이해가 될 거예요. Softmax 는 예제 파일엔 안 쓰이지만 자주 쓰이는거라서 한 번 넣어봤습니다.

```
def sigmoid_backward(dA, cache):
    Z = cache
    s = 1/(1+np.exp(-Z))
    dZ = da * s * (1-s)
    return dZ
```

```
def relu_backward(dA, cache):
    Z = cache
    dZ = np.array(dA, copy = True)
    dZ [ Z <=0 ] = 0
    return dZ
```

역전파에서 쓰일 함수들을 구현해보았습니다. 아래의 식을 그냥 코드로 바꿔 놓은 것입니다.

시그모이드 함수의 도함수: $\frac{d}{dx} sigmoid(x) = sigmoid(x)\big(1 - sigmoid(x)\big)$

ReLU 함수의 도함수: $\frac{d}{dx}\, relu(x) = \begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases}$

## (2) Initialization

가중치와 편향을 initializing 하는 함수입니다. random 하게 initialize 하겠습니다.

```
def initialize_parameters(layer_dims):
    np.random.seed(3)      # random seed 설정해줘야 random.randn 했을 때 같은 값 나옴
    parameters={}
    L=len(layer_dims)

  for l in range(1,L):
      parameters['W'+str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1])*0.01
      parameters['b'+str(l)] = np.zeros((layer_dims[l],1))

  return parameters
```

Np.random.randn :평균 0, 분산 1의 가우시안 분포의 random 값들로 W를 initialize

0.01을 곱하는 이유: 작은 값에서부터 가중치를 시작하기 위해서! 하지만 너무 작으면 안 되니까 0.01만 곱해준다.

이를 통해 가중치의 표준편차를 0.01로 바꿔주어 기울기 소실 문제 완화

이 외에도 가중치의 초깃값을 위한 방법으로는 Xavier 초깃값과 He 초깃값이 있으나 여기에서는 다루지 않겠습니다.

## (3) Forward propagation

### (1) 선형결합

```
def linear_forward (A,W,b):
    Z = np.dot(W,A) + b
    cache = (A, W, b)
    return Z, cache
```

선형 결합을 해주는 함수입니다. 아까 위에서 봤던 $z_j = \sum_{i=1}^{m} X * W + b$ 식을 코드로 구현한 것이지요. 여기에서도 cache 를 저장해뒀는데 위와 마찬가지의 이유입니다. X 가 아니고 $A$인 이유는 은닉층의 중간에서 일반화했다고 생각해보면 됩니다. 전 층의 출력값($A_i$)이 이번 층의 입력값($X_j$)이라서 $A_i = X_j$입니다.

### (2) Activation function

```
def linear_activation (A_prev,W,b, activation):
    if activation == 'sigmoid':
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
    elif activation == 'relu':
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
    cache = (linear_cache, activation_cache)

    return A, cache
```

활성화 함수입니다. 은닉층의 활성화 함수는 sigmoid 를, 출력층의 활성화 함수는 relu 를 쓸 것이기 때문에 if 문을 사용하여 둘의 경우를 나누어 주었습니다. 위에 정의해뒀던 선형결합 함수를 이용하여 Z 와 linear_cache 를 구합니다. 그 이후 Z 를 sigmoid 함수 (위에서 정의해뒀지요?) 에 집어넣어 A 와 activation_cache 를 구합니다. 여기에서도 마찬가지로 cache 를 저장하는데, 이때 선형결합에서의 cache 와 활성화함수를 통과하며 생긴 cache 둘을 tuple 로 저장합니다.

### (3) 여러 layer 로 확장

```
def forward_model(X, parameters):
    caches = []
    A = X
    L = len(parameters)//2

    for l in range(1,L):
        A_prev = A
        A, cache = linear_activation(A_prev, parameters['W'+str(l)],
                                     parameters['b', str(l)], activation='relu')
        caches.append(cache)
    AL, cache = linear_activation = (A, parameters['W' str(L)], parameters['b',str(l)],
                                     activation = 'sigmoid')
    caches.append(cache)
    return AL, caches
```

위의 두 함수를 활용하여 여러 레이어에 적용할 수 있도록 확장시킨 함수입니다. Parameters//2 해주는 이유는 parameters 는 각 층 마다 W 행렬과 b 벡터가 있어서 개수가 레이어 개수의 두배이기 때문에 나눠줍니다.

## (4) Cost Function

```
def compute_cost (AL, Y):
    m=Y.shape[1]
    cost = (-1/m) * np.sum(np.multiply(Y, np.log(AL)) + np.multiply ( 1 - Y, np.log(1 - AL))
    return cost
```

Cost function 입니다. CEE 를 사용했습니다. 아래 식을 구현한 것입니다. 여기에서는 미니배치를 사용할 것이라서 m 으로 나누어줬습니다.

$$-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right)+\left(1-y^{(i)}\right)\log\left(1-a^{[L](i)}\right)\right)$$

## (5) Backward prolagation

```
def linear_backward (dZ,cache):
    A_prev, W, b = cache

    dW = np.dot(dZ, cache[0].T)
    db = np.squeeze(np.sum(dZ, axis=1, keepdims=True))
    dA_prev = np.dot(cache[1].T, dZ)

    return dA_prev, dW, db
```

### (1) Linear Backward

Forward 할 때와 마찬가지로, 먼저 linear 연산을 하는 함수부터 만듭니다. 여기에서 지금껏 저장해 온 cache 를 사용합니다. cache 값은 위에서부터 저장해왔던 선형결합한 값$z_i$와 활성화함수를 통과한 값 $a_i = \sigma(z_i)$입니다. 여기서는 cache[0]만 쓰니까 linear_cache 만 사용하는 것입니다. Cache 를 저장해두지 않았더라면 여기서 또 저 값들을 계산해야 했을 것입니다. cache 덕분에 편하게 된 것이죠!

```
def linear_activation_backward (dA, cache, activation):
    linear_cache, activation_cache = cache
    if activation == 'relu':
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    elif activation == 'sigmoid':
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    return dA_prev, dW, db
```

### (2) activation backward

얘도 activation 을 역전파 하는 함수를 만들어줬습니다. Activation function 이 뭐냐에 따라 activation function 의 도함수가 달라지기 때문에 activation function 도 따로 입력을 받았습니다. Relu_backwawrd 와 sigmoid_backward 는 위에서 정의해두었죠?

### (3) 여러 layer 로 확장

```
def backward_model (AL, Y, caches):
    grads = {}          # gradient 들을 넣을 딕셔너리
    L = len(caches)   # the number of layers
    m = AL.shape[1]   # m 의 개수 (mini-batch size)
    Y = Y.reshape(AL.shape)   # Y 와 AL 의 shape 를 같게 만들어줌
    dAL = -(np.divide(Y, AL) - np.divide(1-Y, 1-AL))   # 역전파를 initialize
    # L 번째 layer 의 Sigmoid -> Linear gradients 를 계산해줌 (출력층을 위해서)
    current_cache = caches[-1]
    grads['da'+strL]], grads['dw'+str(L)], grads['db'+str(L)] = linear_backward(sigmoid_backward(dAL, current_cache[1]),
                                                                                current_cache[0])
    for l in reversed(range(L-1)): # L 번째 layer 의 Relu -> Linear Gradients 를 계산해줌
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_backward(sigmoid_backward(dAL, current_cache[1]), current_cache[0])
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
    return grads
```

상당히 복잡해 보이는 함수인데. Line by line 으로 차근히 살펴보겠습니다. 먼저, L=len(caches)는 layer 의 개수를 의미합니다. Caches 는 아까 forward_model 에서 return 한 값으로, 순전파를 한 층 할 때 마다 계산한 값들을 (linear_cache, activation_cache)의 tuple 형태로 묶어 리스트로 저장한 것입니다. 따라서 len(caches)는 layer 의 층 개수가 나옵니다.

m 은 mini-batch size 입니다. AL 로 들어갈 값은, cache 와 마찬가지로 forward_model 의 return 값 중 하나입니다. AL 이 의미하는 바는 Linear 을 거친 activation 계산 값입니다. 즉, $a_i = \sigma(z_i)$를 의미합니다.

## (6) Update Parameters

$$W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$$

미니배치 경사하강법을 활용하여 parameter 들을 update 할 계획입니다. 여기에서 $\alpha$는 앞전에 봤던 $\eta$와 같은 기능을 하는 learning_rate, 학습률입니다. Update 한 후에는 parameters dictionary 에 저장할 생각입니다.

```
def update_parameters(parameters, grads, learning_rate):
    L = len(parameters) // 2 # number of layers in the neural network
    for l in range(L):
            parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] - learning_rate * grads["dW" + str(l + 1)]
            parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] - learning_rate * grads["db" + str(l + 1)]
    return parameters
```

함수의 구조를 보여드리기 위해서 이렇게 직접 코드들을 구현해 보았어요! 앞으로는 이런식으로 일일히 정의하지 않고 모듈을 불러와서 활용할 것이기 때문에 너무 신경쓰지는 않으셔도 될 것 같습니다. 그냥 우리가 쓸 함수/코드가 어떤 원리에서 작동한건지, 어떤 구조로 돌아 가는지를 이해하고 쓰면 좋으니까 같이 첨부해두었습니다. ㅎㅎ 코드에서 이해가 안 가는 부분은 질문해주시면 언제든지 답변 드릴게용~~

# P.S.

여러분 안녕하세요. 여러분과 한 학기 동안 함께 할 딥러닝팀..ㅌ ..팀..팀장….이수경입니다..ㅜ 딥러닝팀의 막내딸로 어버버 하던게 엊그제인데 제가 갑자기 어떻게 이렇게 갑자기 팀장이 됐는지 잘 모르겠어요… 아직도 부끄러워요. 그래서 아직 많이 부족한 초짜지만 여러분이 사랑과 관심으로 잘 보듬어주시면 어떻게 저떻게 성장해보겠습니다.. 궁금한게 있으면 절대 주저 말고 물어봐주세요. 저도 모를테니까요 ^^.. 같이 알아가는 좋은 시간을 보낼 수 있을거예요. 만나게 되어서 정말 반갑고 잘 부탁드려요!!! 1주차는 어떠셨나요? 어렵지 않게 잘 설명하려고 많이 노력했는데 잘 전달 됐는지 모르겠네요. ㅜㅜ …흑흑 여러분이 한 학기 동안 즐거운 딥러닝 팀 생활 할 수 있도록 정말 열심히 노력할게요! 스터디 들어주셔서 감사합니다~~~~ <3

덧붙여 교안 쓰는 내내 저의 멘탈을 다독여주신 저의 딥러닝 아버지.. 김재희님께 감사 인사 올립니다.. (--) (__)