

CHAPTER 3

Table of contents

1. Introduction to deep learning

- 1) A brief history of deep learning
- 2) Artificial intelligence, Machine learning and deep learning

2. DNN

- 1) Layers : the building blocks of deep learning
- 2) Activation function
- 3) initialization
- 4) Loss function
- 5) Optimizer
- 6) Forward Propagation and Back Propagation
- 7) Iteration
- 8) Gradient Vanishing and Exploding
- 9) deep learning model
- 10) feature scaling

3. regression and classification

- 1) Deep learning model
- 2) regression with deep learning
 - example 1: diabetes
 - example 2: boston housing
 - example 3: MPG
- 3) classification with deep learning
 - example 1: Wine (Binary class)
 - example 2: Wine (Multiple classes)
 - example 3: Dry beans (Multiple classes)

1. Introduction to deep learning

Reference

Deep learning with python, François Chollet
 Hands-On Machine Learning with Scikit-Learn, Keras&TensorFlow, Aurélien Géron

1) A brief history of deep learning

사람이 새를 보고 비행기에 대한 영감을 얻었듯 많은 발명품은 자연에서 영감을 받아 개발되었다. 이처럼 지능적인 기계를 만들기 위해 뇌 구조를 관찰하는 것은 당연할 것이다. 이는 최초의 인공 신경망(ANN, Artificial neural network)이 제안된 근원이다. ANN은 1943년 신경생리학자 Warren McCulloch와 수학자 Walter Pitts의 논문¹에서 처음으로 등장하였다. 초기의 인공신경망은 곧 지능을 가진 기계와 대화를 나누게 될 것이란 믿음을 널리 퍼트렸다. 그러나 1960년대에 인공 신경망의 침체기에 들어갔으며 1980년대 초에 새로운 네트워크 구조가 발명되고 더 나은 훈련 기법이 개발되면서 다시 관심을 받기 시작했다. 비록 발전은 더뎠지만 1990년대 이후 컴퓨터 하드웨어의 발전²으로 납득할 만한 시간 안에 대규모 신경망을 훈련할 수 있었다.

현재에는 인공 신경망이라는 용어보다 AI, machine learning, deep learning 이라는 용어가 더 익숙하게 사용되고 있다. 인공 신경망으로부터 파생된 세가지 용어의 정확한 정의는 다음에서 다루도록 한다. 최근 개발되고 있는 인공 신경망은 여러분야에서 유용하게 이용되고 있다. 수백만 개의 이미지를 분류하거나(구글 이미지), 음성 인식 서비스의 성능을 높이거나(애플의 시리), 매일 수억 명에 이르는 사용자에게 가장 좋은 비디오를 추천하거나(유튜브), 바둑 세계 챔피언을 이기기 위해 수백만 개의 기보를 익히고 자기 자신과 게임하면서 학습하는(딥마인드의 알파고) 등 복잡한 대규모 머신러닝 문제를 다루는데 적합하다.

인공지능 알고리즘을 개발하기 위해선 머신러닝 혹은 딥러닝의 초창기 구조인 다층 퍼셉트론(MLP, multi-layer perceptron)의 구조와 구동 원리에 대한 이해가 필요하다. Chapter 3에서는 이를 심층신경망(DNN, deep neural network)이라고 정의하며 이 신경망의 구조를 단계별로 소개하며 간단한 통계적 문제를 DNN 모델을 이용하여 접근해볼 것이다.

인공 신경망은 다음과 같은 목표를 해결하기 위해 발전되고 있다.

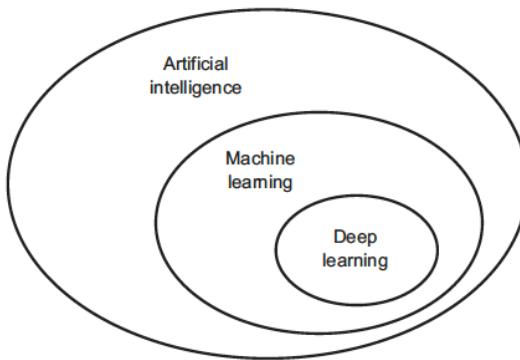
- To solve overfitting
- Improve learning speed and accuracy

¹ <https://link.springer.com/article/10.1007/BF02478259>

² 무어의 법칙(Moore's Law, 최근 50년간 집적 회로의 성능이 2년마다 두 배 증가했다.)

2) Artificial intelligence, Machine learning and Deep learning

최근 인공지능, 머신러닝, 딥러닝 이 세가지 용어에 대한 구분이 모호해지고 있다. 정확한 정의는 아래의 그림을 통해 이해하도록 한다.



- 인공지능(AI, Artificial intelligence)은 “think”, 생각하는 것이 가능한 컴퓨터를 만들기 위한 시도로 부터 출발이 되었다. 이런 시도는 현재에도 지속되고 있으므로 computer science 영역에서 인공지능에 대한 정확한 정의는 다음과 같다.

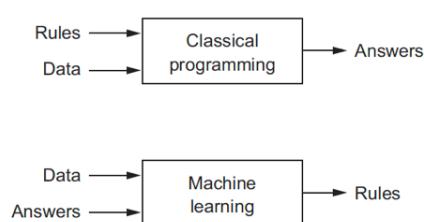
“The effort to automate intellectual tasks normally performed by humans”

초창기의 인공지능은 체스처럼 정확한 규칙이 존재하는 문제만을 프로그래밍 할 수 있었다. 이후 발전을 거듭하면서 이미지 분류(image classification), 음성인식(speech recognition) 등 더 복잡한 문제를 해결할 수 있게되었고 이런 문제를 해결하는 방법론을 일컬어 “machine learning”이라 부르기 시작하였다.

- 머신러닝(ML, machine learning)은 다음과 같은 질문으로 출발하게 되었다.

“what we know how to order it to perform?”

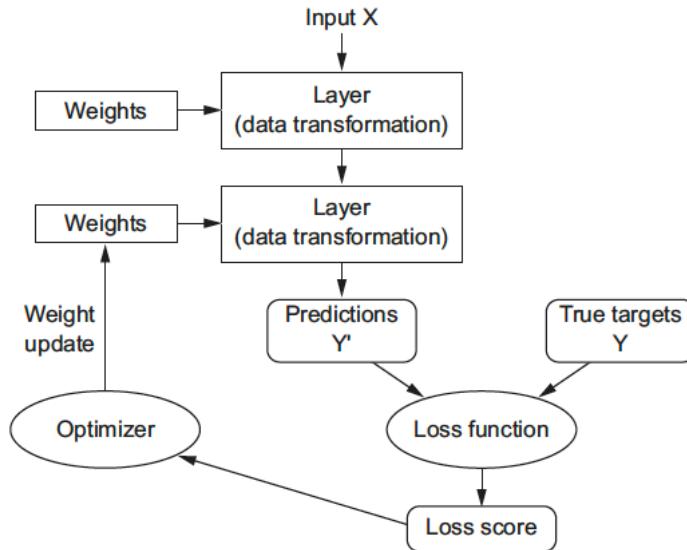
이 질문은 프로그래밍 분야의 새로움 패러다임을 열었다. 따라서, 프로그램의 목적의 목적이 “learning rules”, 규칙을 학습하는 것에 초점이 맞춰지면서 서포트 벡터 머신(SVM) 등 여러 방법론이 만들어지게 되었고 이런 방법론들은 기계학습, 즉 머신러닝이라 부르게 되었다.



- 딥러닝(DL, deep learning)의 “deep”은 신경망(neural network)의 층(layer)이 깊다는 것을 의미한다. GPU의 발달로 학습속도가 증가함에 따라 신경망의 층을 깊게 쌓는 것이 가능해졌기 때문이다. 따라서, 딥러닝 모델은 neural network 구조를 기반으로 만드는 모델을 통칭한다.

2. DNN

딥러닝 모델에 대한 이해를 하기 위해서는 Neural Network에 대한 이해가 필요하다. 아래의 그림은 신경망(Neural Network)의 학습 절차이다.

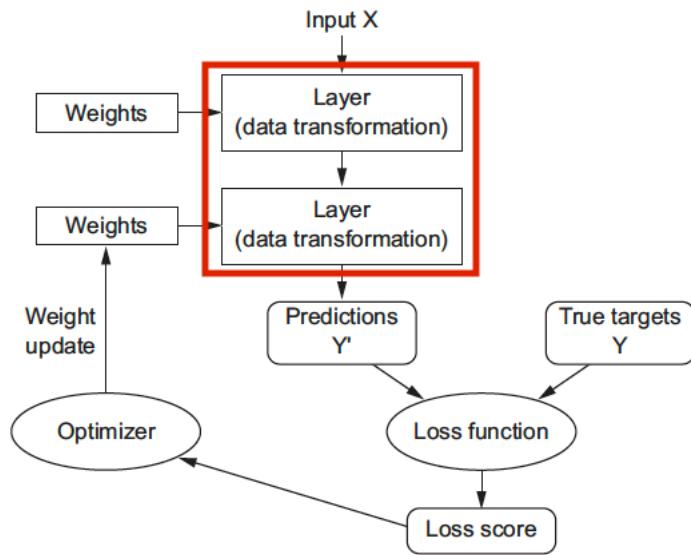


심층신경망(deep neural network)은 입력층과 출력층 사이에 다수의 은닉층을 포함하여 다양한 비선형적 관계를 학습할 수 있다. 알고리즘에 따라 input layer와 output layer 사이에 여러개의 layer를 쌓아 network를 형성하며 이 network를 거치면서 입력되었던 x 값은 예측값인 y' 로 계산되어 출력된다. 심층신경망(deep learning model)을 만들기 위해선 이 과정에서 일어나는 절차에 대한 이해가 필요하다. 신경망 학습의 목적은 전체 네트워크의 층과 층 사이에서의 연산에 사용되는 모든 가중치들의 최적값을 찾는 것이며 모델에 따라 정의한 LOSS가 충분히 줄어들 때까지 가중치의 업데이트를 반복하는 것이다.

학습의 과정은 다음의 절차를 정해진 횟수만큼 반복함으로써 이루어진다.

- ① 배치 사이즈만큼의 훈련 샘플 (x, y)를 가져와 입력층에 데이터 x 를 입력한다.
- ② 전체 네트워크를 따라 가중치(weight)를 곱하고 활성함수를 거쳐 입력 데이터를 변환하고 출력값 \hat{y} 를 계산한다.
- ③ 출력된 \hat{y} 와 실제 y 값의 오차를 사용해 현재 배치 안에서 평균 LOSS를 계산한다. (회귀문제의 경우 MSE, 분류문제의 경우 Cross Entropy)
- ④ 출력층부터 입력층까지 역방향으로 되돌아가며 가중치 파라미터에 대한 LOSS의 그래디언트를 계산하고, LOSS가 줄어드는 방향으로 가중치를 업데이트 한다.
- ⑤ 다음 배치 사이즈만큼의 훈련 샘플을 가져와 반복한다.

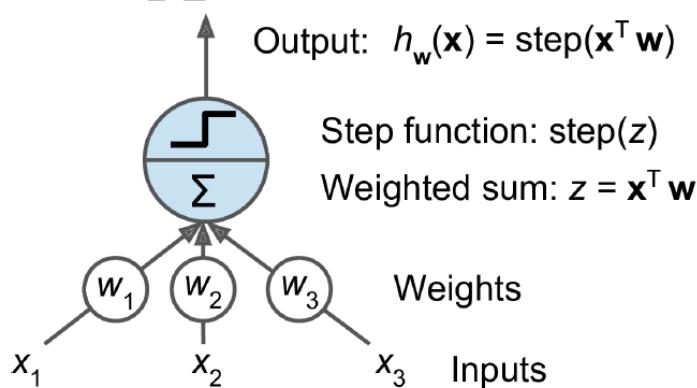
1) Layers : the building blocks of deep learning



신경망(neural network)의 기본은 층(layer)이다. 먼저, 각 층(layer)안에서 어떤 계산과정이 일어나는지와 그 원리를 이해하기 위해선 퍼셉트론에 대한 이해가 필요하다.

a) 퍼셉트론(Perceptron)

퍼셉트론(perceptron)은 가장 간단한 인공 신경망 구조 중 하나로 1957년에 프랑크 로젠틀라트가 처음으로 제안하였다. 퍼셉트론은 TLU(threshold logic unit) 또는 LTU(linear threshold unit)이라고 불리는 인공 뉴런이다. 입력과 출력은 숫자이며, 각각의 입력은 가중치와 연관된다. 퍼셉트론은 입력(inputs)의 가중치 합(weights sum)을 계산한 뒤 계산된 합(weights sum)에 활성화 함수(activation function)를 적용하여 결과(output)를 출력한다.



위의 그림은 하나의 퍼셉트론의 구조이다. 퍼셉트론은 입력 벡터 $\mathbf{x} = (x_1, x_2, x_3)$ 를 받아들인 뒤 각 성분의 가중치(Weight)를 곱하여 합 ($z = w_1x_1 + w_2x_2 + w_3x_3 = \mathbf{x}^T \mathbf{w}$) 을 계산한다. 다음으로, 계산된 합에 편향 벡터(w_0)를 더한뒤($\mathbf{x}^T \mathbf{w} + w_0$) 활성화 함수를 적용하여 ($h_w(\mathbf{x}^T \mathbf{w} + w_0)$) 결과를 출력한다.

NOTE)

노드는 하나의 퍼셉트론이다.

하나의 층(layer)에 존재하는 퍼셉트론들이 이전 층의 모든 퍼셉트론들과 연결되어 있을 때 이를 완전 연결

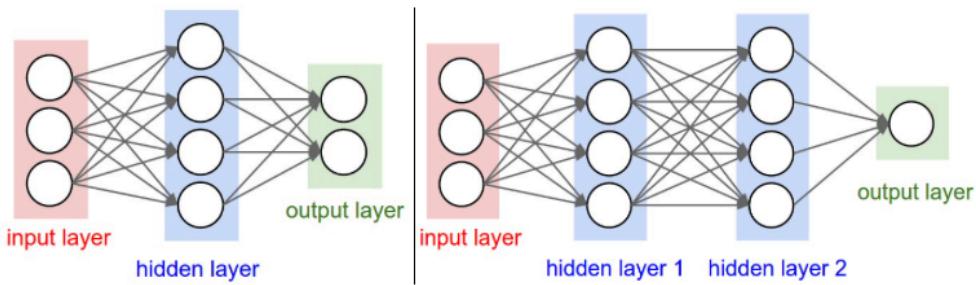
층(fully-connected layer) 또는 밀집 층(dense layer)라고 부른다.

다음은 완전 연결 층의 출력 계산과정이다.

$$h_{w,b}(X) = \Phi(XW + b)$$

- X 는 입력값의 행렬이다. 즉, 데이터로 행은 샘플(observations), 열은 특성(features)이다.
- 가중치 행렬 w 는 편향 벡터를 제외한 모든 연결 가중치를 포함한다. 이 행렬의 행은 입력 뉴런의 개수에 해당하고 열은 출력층의 뉴런에 해당한다.
- 편향 벡터 b 는 편향 뉴런과 인공 뉴런 사이의 모든 연결 가중치를 포함한다. 인공 뉴런마다 하나의 편향값이 있다.
- Φ 를 활성화 함수(activation function)라고 부른다. 인공 뉴런이 TLU일 경우 이 함수는 계단함수(step function)으로 활성화 함수에 적용시킨 후 결과값으로 0 또는 1 값만을 출력한다.

이런 퍼셉트론을 여러 개 쌓아 올린 것을 다층 퍼셉트론(MLP)라고 한다. 아래의 그림은 가장 기본적인 다층 퍼셉트론의 예시이다.



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs.
Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

다층 퍼셉트론은 입력층(input layer) 하나와 하나 이상의 은닉층(hidden layer)과 마지막 출력층(output layer)로 구성된다. 입력층과 가까운 층을 보통 하위 층(lower layer)이라 부르고 출력에 가까운 층을 상위 층(upper layer)하고 부른다. 출력층을 제외하고 모든 층은 다음 층과 완전히 연결(fully-connected)되어 있다.

NOTE)

신호는 입력에서 출력으로 한 방향으로만 흐르고 있으므로 위의 구조는 feedforward neural network(FNN)이다.

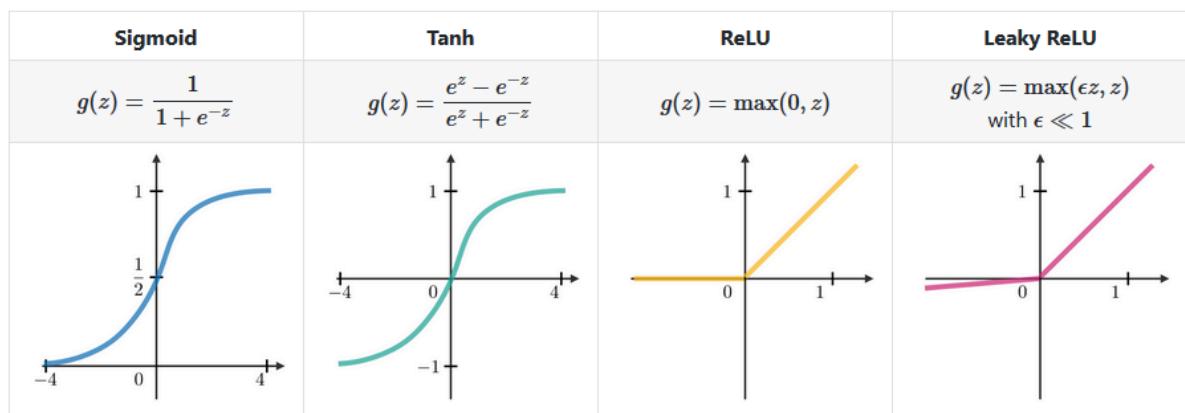
은닉층을 여러개 쌓아 올린 인공 신경망을 **심층 신경망(deep neural network, DNN)**이라고 한다. 딥러닝은 심층 연결망을 연구하는 분야이며 더 일반적으로는 연산이 길게 연결된 모델을 연구한다. 그러나 대부분 신경망이 사용되면 딥러닝이라 부른다.

2) Activation function(활성화 함수)

심층신경망(DNN)은 입력층, 은닉층, 출력층으로 구성되며 각각의 층은 여러 개의 노드(node, perceptron)들로 이루어져 있다. 하나의 층에 존재하는 노드들은 다음 층의 노드들과 연결되며 데이터가 입력된 후 연결강도의 가중치를 계산한다. 계산된 가중치는 각 입력값과 곱한 후 그 합이 활성화 함수를 통과하며 결과값이 출력된다.

심층신경망에서 입력 데이터를 받아 출력값을 생성해주는 함수를 활성화함수(activation function)라고 한다. 즉, 심층신경망에서 입력 받은 데이터를 어떤 값으로 출력할지를 결정하고 심층신경망을 통과해온 값을 최종적으로 어떤 값으로 만들지 결정한다.

활성화 함수에는 대표적으로 Sigmoid, ReLU, tanh 함수가 있다.



주로 이용되는 활성화 함수는 tanh와 ReLU이다. Tanh는 sigmoid function(logistic function)처럼 S자 모양이고 연속적이며 미분가능하다. 출력 범위는 -1에서 1 사이이며(sigmoid는 0-1 사이) 이 범위는 훈련 초기에 각 층의 출력을 원점 근처로 모으는 경향이 있다. 이런 특징은 종종 빠르게 수렴되도록 도와준다. 다음으로 ReLU함수는 연속적이지만 $z = 0$ 에서 미분가능하지 않다. (이 특징으로 기울기가 갑자기 변해서 경사 하강법이 엉뚱한 곳으로 틀 수 있다) $z < 0$ 인 경우 도함수는 0이지만 실제로는 잘 작동하고 계산 속도가 빠르다는 장점이 있어 기본 활성화 함수가 되었다. 중요한 점은 출력에 최댓값이 없다는 점에서 경사 하강법에 있는 일부 문제를 완화해준다.

출력층 직전의 은닉층에서 출력층으로 출력값을 내보내는 경우 활성화 함수는 알고리즘이 어떤 문제를 다루고 있느냐에 따라 달라진다.

- Regression 문제인 경우 identity function
- Classification 문제인 경우 softmax function

입력 데이터로부터 출력값이 생성되기 위해선 반드시 활성화 함수가 필요하다. 그 이유는 무엇일까? 앞서 심층 신경망은 비선형적 관계를 학습할 수 있는 모델이라고 하였다. 선형 변환을 여러 개 연결하는 경우 얻을 수 있는 것은 선형 변환뿐이다. 예를 들어, 두 선형 함수 $f(x) = 2x + 3$ 과 $g(x) = 5x - 1$ 을 연결하면 또 다른 선형 함수 $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$ 이 된다. 따라서 층 사이에 비선형성을 추가하지 않으면 아무리 층을 깊게 쌓아도 하나의 층과 동일해진다. (결론적으로 선형 모델을 만들었기 때문이다) 이런 구조로는 복잡한 문제를 풀 수 없다. 반면, 비선형 활성화 함수가 있는 충분히 큰 심층 신경망은 이론적으로 어떤 연속함수도 근사할 수 있다.

활성화 함수는 그래디언트 소실 문제의 해결과 함께 발전해왔다. 따라서, 활성화 함수의 목적에 대해 이해하기 위해서는 그래디언트 소실 문제에 대한 이해가 필요하다.

3) initialization(파라미터 초기화)

계속해서 하나의 노드 안에서 일어나는 계산에 대해 알아보도록 한다. 딥러닝을 정의하는 신경망의 원소는 노드이다. 각 층에 존재하는 노드들은 이웃한 층들의 노드들과 연결되고 이런 연결강도를 나타내는 가중치가 계산되는 과정을 “모델의 학습”이라고 한다. 각 가중치는 입력 데이터와 곱해진 후 그 합이 활성화 함수를 지나면서 출력값이 생성된다. 그렇다면 가중치는 어떻게 계산되는 것일까?

$$h_{w,b}(X) = \Phi(XW + b)$$

위의 식은 딥러닝 모델 안에서 하나의 밀집층에 대한 출력 계산 과정이다. 이때, X 는 입력된 데이터이며 Φ 는 활성화 함수로 두 값은 학습 전에 미리 정의된다. 출력값을 구하기 위해서는 나머지 두 값인 w (가중치)와 b (편향)이 필요한데 이 두 값은 모델 학습과정에서 구해지는 값이다. 따라서, 딥러닝 분야에서는 이 두 개의 값을 parameter(파라미터)라고 정의한다.

사전에 학습된 모형이 없을 경우 일반적으로 전체 네트워크의 파라미터(가중치 및 편향)에 랜덤한 초기값을 설정한다. 초기값에 따라 학습의 성능이 크게 좌우될 수 있으므로 주의가 필요하다.

a) 잘못된 초기화 예시

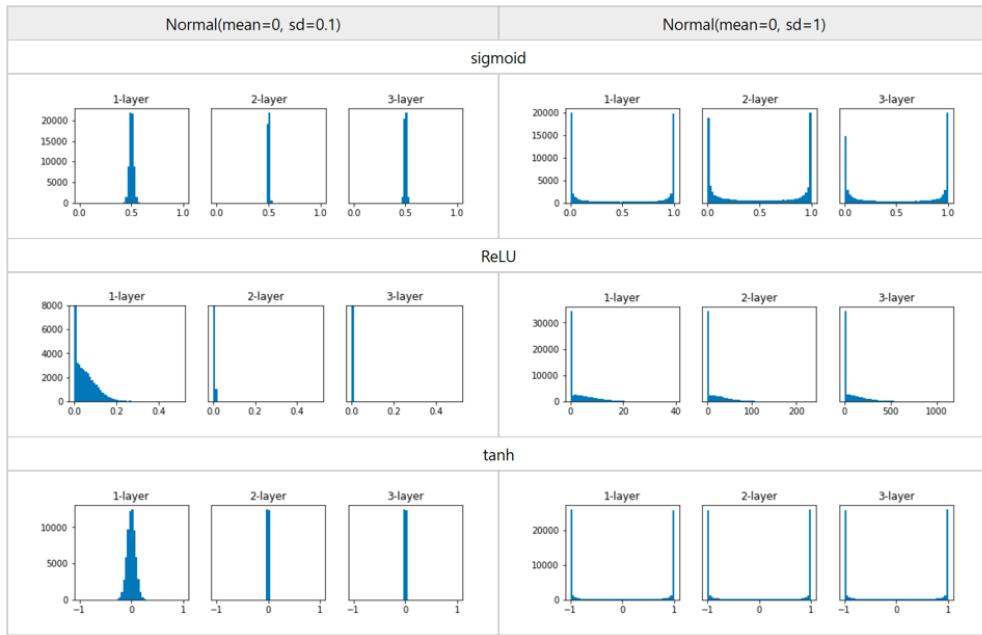
- 모두 0으로 놓는 경우 : 만약 가중치 행렬 W 의 초기값을 단순히 모두 0으로 놓는다면 W 의 원소들은 전방향 연산에서 출력 값 계산에 사용되고, 이 값을 이용해 역방향으로 그래디언트를 구하는 과정에서 그래디언트가 모두 0이 되어 사라진다. 따라서 이것은 네트워크의 학습이 이루어지지 않게 하므로 옳지 않은 방법이다.
- 모두 동일한 값(상수)로 지정하는 경우 : 초기값이 동일하면 모든 뉴런이 동일한 출력값을 내보낼 것이며 모두 동일한 그래디언트 값을 가지게 된다. 따라서 뉴런이 여러개라도 하나의 뉴런처럼 작동하므로 학습이 잘 되지 않는다.

b) Random Generator

가중치의 초기값을 서로 다른 랜덤한 값으로 구성하기 위해 일반적으로 uniform, 또는 Normal 분포에서 난수를 생성한다. 주의할 점은 적절한 분산을 설정하여 값의 크기를 적당하게 초기화해야 한다는 것이다. 초기화 분포의 선택은 활성화 함수의 영향을 받는다. 현재 많이 사용되고 있는 활성화 함수는 Sigmoid, ReLU, tanh이다.

만약 활성화함수가 Sigmoid일 경우 가중치 초기값의 크기(절댓값)이 너무 크다면 0과 1로 수렴하기 때문에 그래디언트 소실이 발생하게 된다. 활성화 함수가 ReLU일 경우, 절댓값이 크다면 음수일 경우 dead ReLU 문제가 발생하고 양수일 경우 그래디언트 폭주(exploding)가 일어난다.

크기가 너무 작은 값으로 초기값을 지정한다면 간단한 신경망에서는 잘 작동하는 편이나 신경망의 깊이가 깊어질수록 문제가 발생한다. 다음 그림에서 상위 신경망으로 갈수록 누적 곱해진 값이 점점 모두 동일한 값을 출력하는 것을 확인할 수 있다. 따라서 이 경우 역시 모든 뉴런의 그래디언트 값이 동일해지기 때문에 학습이 잘 이루어지지 않는다.



따라서 적절한 크기의 서로 다른 값으로 가중치의 초기값을 설정하는 것이 좋다. 가중치를 초기화하는 방법은 지속적으로 연구가 진행되고 있는 분야로, 현재는 랜덤한 값을 생성할 Uniform 또는 Normal 분포의 분산을 각 층의 입력과 출력 노드의 개수에 따라 결정하도록 하는 Xavier(2010)이나 He(2015) 방법을 주로 사용한다.

c) Xavier (Glorot)

$$W_i \sim \text{Uniform} \left[-\sqrt{\frac{2}{n_{in} + n_{out}}}, \sqrt{\frac{2}{n_{in} + n_{out}}} \right] \cdot \sqrt{3} \quad \text{or} \quad W_i \sim \text{Truncated Normal} \left(0, \sqrt{\frac{2}{n_{in} + n_{out}}} \right)$$

Xavier Glorot가 제안한 방법은 깊은 신경망 모형에서 그래디언트 소실 및 폭주 문제를 해결하기 위해 만들었다. Xavier는 기존의 초기값 생성 분포에서 난수를 생성할 경우 층이 깊어짐에 따라 sigmoid 또는 softsign 등의 비선형화 값의 분산이 점점 작아지는 것을 발견하였다. 또한 가중치 업데이트를 위해 순전파와 역전파를 반복하므로 각 층 사이의 가중치의 분산이 입력층의 노드의 개수뿐만 아니라 출력층의 노드의 개수에도 영향을 받음을 발견하였다. 따라서 순전파 시 가중치의 분산의 합과 역전파의 가중치의 분산의 합을 모두 1으로 하기 위해 i 번째 층과 $i+1$ 번째 층 사이의 가중치 분산 $\text{Var}(W_i) = 2/(n_i + n_{i+1})$ 이 되도록 하였다. 또한 uniform 분포에서 분산을 모두 1으로 맞추기 위한 정규화 상수 $\sqrt{3}$ 을 곱하는 방법을 도입하였다.

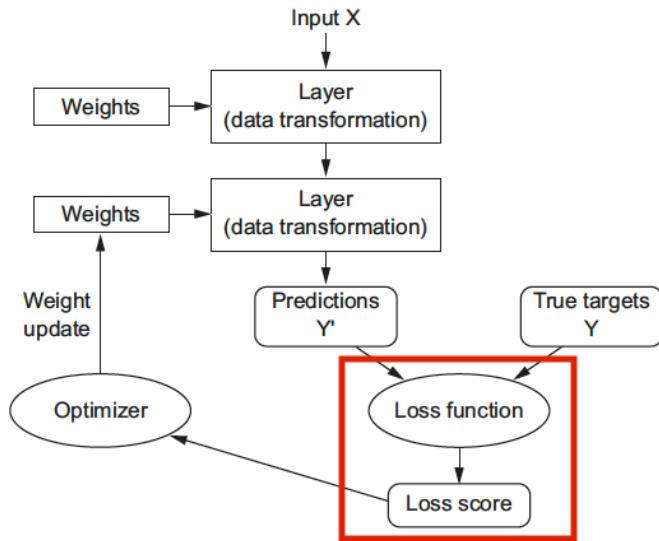
Xavier 방법은 이렇게 간단한 발견으로 sigmoid 또는 tanh, softsign 등의 비선형함수에서 효과적인 결과를 나타냈으나 ReLU함수에서 사용 시 출력 값이 0으로 수렴하게 되는 문제를 해결하지 못했다.

d) He

$$W_i \sim \text{Uniform} \left[-\sqrt{\frac{2}{n_{in}}}, \sqrt{\frac{2}{n_{in}}} \right] \cdot \sqrt{3} \quad \text{or} \quad W_i \sim \text{Truncated Normal} \left(0, \sqrt{\frac{2}{n_{in}}} \right)$$

ReLU를 사용한 모형에서의 추정이 잘 되지 않는 점을 해결하기 위해 Kaiming He가 제안한 좀 더 로버스트한 초기화 방법으로 Glorot과 유사하지만 뉴런의 출력 사이즈를 고려하지 않는다. ReLU가 0 이하의 신호를 제거하기 때문에 줄어드는 분산을 유지하기 위해 분산을 좀 더 크게 설정하는 방법이라고 할 수 있다.

4) Loss function (손실함수)



LOSS function은 모델을 통해 나온 예측값(prediction, \hat{Y})과 실제 데이터(True targets, Y)의 차이를 계산하는 함수이다. 다시 말해, 예측값과 실제값이 얼마나 유사한지 판단하는 기준이 필요한데 이를 계산하는 것이 손실 함수(LOSS function)이다. 따라서, 예측값과 실제값의 차이를 LOSS라고 하며 LOSS를 줄이는 방향으로 학습이 진행된다. 또한, 손실함수는 학습된 결과의 quality를 측정하는 역할을 있다고 할 수 있다. 모델에 어떤 손실 함수(LOSS function)를 이용하는지는 모델에서 다루는 문제의 종류에 따라 결정한다.

NOTE)

손실함수는 loss function, objective function, cost function으로 부른다.

통계학적 모델은 일반적으로 회귀(regression)와 분류(classification) 두 가지 종류로 나누어는데, 손실 함수는 이에 따라 두 가지로 나누어진다. 회귀 문제의 경우 대표적인 손실 함수는 MAE, MSE, RMSE가 있으며, 분류 문제의 경우 손실함수는 Cross-entropy가 있다.

a) MSE(평균 제곱 오차, Mean Square Error) & RMSE(평균 제곱근 오차, Root Mean Square Error)

$$MSE(X, h) = \frac{1}{N} \sum_{i=1}^N (h(X_i) - y_i)^2 \quad RMSE(X, h) = \sqrt{\frac{1}{N} \sum_{i=1}^N (h(X_i) - y_i)^2}$$

MSE와 RMSE는 회귀 문제에서 일반적으로 쓰이는 손실함수이다. 예측값과 실제값의 차이가 커지는 경우 제곱연산으로 인해 오차가 양수든 음수든 누적 값을 증가시킨다.

NOTE)

N 은 데이터셋에 있는 observation의 수이다.

b) MAE(평균 절대 오차, Mean Absolute Error)

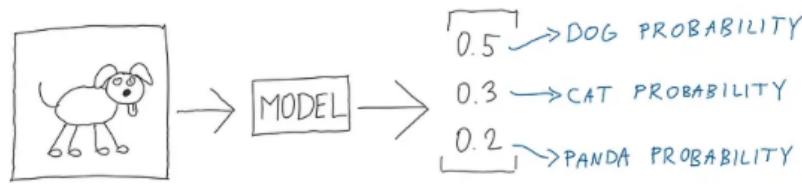
$$MAE(X, h) = \frac{1}{N} \sum_{i=1}^N |h(X_i) - y_i|$$

MAE는 예측값과 실제값의 차이의 절댓값을 평균 낸 값이다.

c) Cross-entropy

Cross entropy는 분류 문제에서 가장 많이 쓰이는 손실 함수이다. 회귀 문제에서 손실 함수를 이용하여 모델의 loss를 구하는 과정은 쉽게 이해할 수 있다. 회귀는 모든 예측값이 실수형으로 얻어지기 때문에 예측값과 실제값을 이용하여 직접적인 수학적 연산이 가능하다. 그러나, 분류 문제의 경우 예측값이 실수형이 아니기 때문에 loss를 구하기 위해 회귀와 다른 방식으로 접근해야 한다. 분류 문제에서 loss를 구하기 위해 cross-entropy의 개념이 어떻게 적용되는지 알아보도록 한다.

예를 들어, 이미지 분류 작업을 수행 중이라고 해보자. 입력된 이미지 데이터들은 dog, cat, panda 3가지 class로 분류된다. 분류 문제에서 딥러닝 모델의 예측값은 probability vector로 출력되는데 벡터의 각 원소의 값은 입력된 이미지가 각 클래스에 속할 확률이다.(각 클래스에 속할 확률을 더하면 1이다.)



신경망 마지막 층(output layer)의 활성화 함수를 softmax function으로 정의한 후 이 probability vector가 마지막 층을 지나게 되면 probability vector가 아래의 왼쪽 그림처럼 target vector로 출력된다. 이때, target은 true value이며 probability vector가 신경망의 output layer를 통과하면 prediction value가 구해진다.

TARGET	PREDICTION
1	0.5
0	0.3
0	0.2

dog에 속할 확률은 1이며, 나머지 클래스에 속할 확률은 0으로 최종 예측값이 출력된다. 이 결과를 이용하여 loss를 구한다. 이때, 각 클래스에 대한 loss를 구한 다음 각각의 값들을 합하여 최종 loss를 구한다.

$$\text{Loss for class } X = - \underbrace{p(X)}_{\substack{\text{probability} \\ \text{of class } X \\ \text{in TARGET}}} \cdot \log \underbrace{q(X)}_{\substack{\text{probability} \\ \text{of class } X \\ \text{in PREDICTION}}}$$

따라서, 클래스에 속할 확률이 0인 경우 해당 클래스의 loss 또한 0으로 구해진다.

$$\begin{aligned} \text{Loss for CAT} &= -p(\text{CAT}) \cdot \log q(\text{CAT}) \\ &= -0 \cdot \log q(\text{CAT}) \\ &= 0 \end{aligned}$$

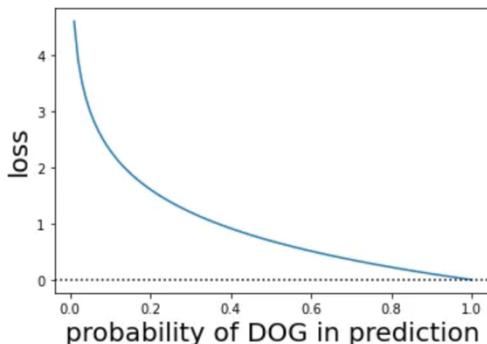
$$\begin{aligned} \text{Loss for PANDA} &= -p(\text{PANDA}) \cdot \log q(\text{PANDA}) \\ &= -0 \cdot \log q(\text{PANDA}) \\ &= 0 \end{aligned}$$

	TARGET	PREDICTION	
DOG	1	0.5	→ loss is ?
CAT	0	0.3	→ loss is 0
PANDA	0	0.2	→ loss is 0

마지막으로 class dog에 대한 loss를 구해본다.

$$\begin{aligned} \text{Loss for DOG} &= -P(\text{DOG}) \cdot \log q(\text{DOG}) \\ &= -1 \cdot \log 0.5 \\ &= 0.693... \end{aligned}$$

class dog에 속할 확률은 loss를 구하는데 이용된다. 따라서, 확률이 변함에 따라 loss에 영향을 준다.



그렇다면 loss를 구하는 과정에서 cat과 panda는 쓰이지 않는 것일까? 위의 예시에서 class cat의 확률이 0.8로 나왔다고 가정을 해보자. 그렇다면, target에 의해 이 확률은 버려지며 나머지 확률인 0.2를 dog와 panda가 나누게 된다. 다시 말해, class dog의 확률이 매우 낮은 상태에서 loss를 구하기 때문에 모델은 정답인 class를 얼마나 정확하게 식별하는지에만 관심이 있는 것이다.

NOTE)

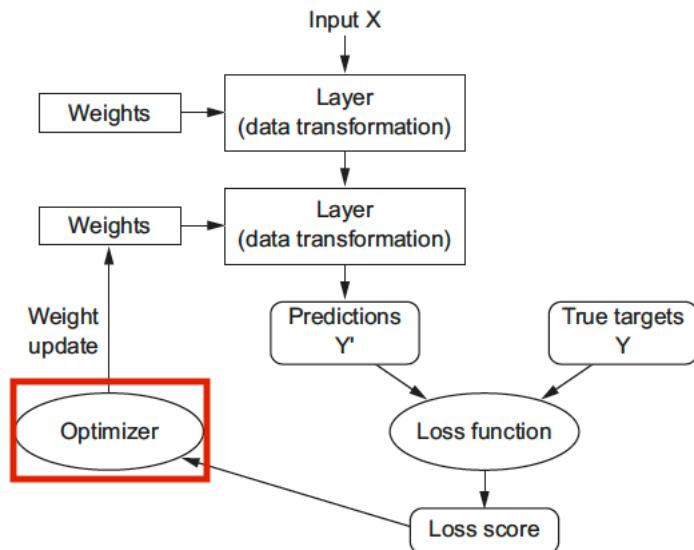
Reference : <https://towardsdatascience.com/cross-entropy-for-classification-d98e7f974451>

아래의 식은 분류 문제에서의 손실함수인 cross-entropy의 정의이다.

$$J = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^i \log (\hat{p}_k^i)$$

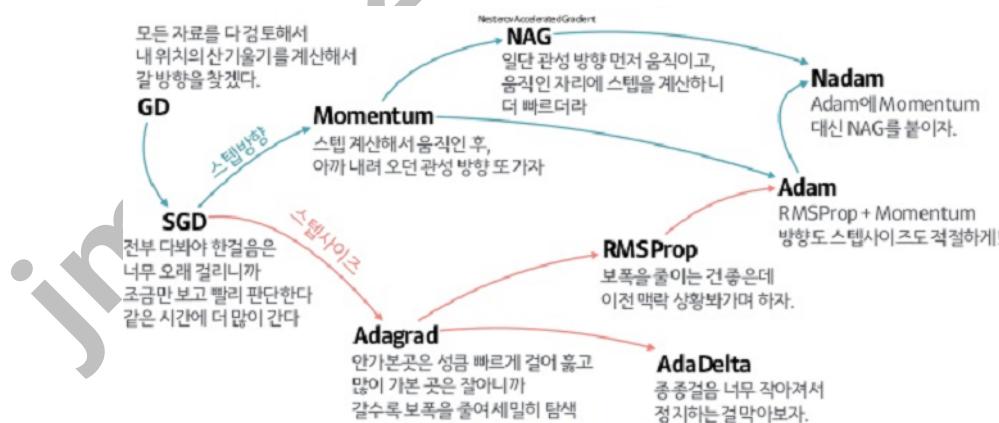
y_k^i 는 i 번째 샘플이 class k 에 속할 target 확률로 1 또는 0의 값을 가진다. \hat{p}_k^i 는 class k 에 속할 예측값으로 target class에 대하여 높은 예측 확률이 나오게 된다면 loss는 작아지고, target class에 대하여 낮은 예측 확률이 나오게 된다면 loss가 커진다.

5) Optimizer



모델이 학습한 파라미터의 성능을 판단하기 위해 손실함수를 이용하여 loss를 구하였다. 계산된 loss를 이용하여 모델의 quality를 평가한 다음의 단계는 가중치의 update이다. Loss가 큰 경우 모델의 quality를 높이기 위해 가중치를 변경해야 하는데 어떤 방향으로 얼마나 변경할지 알려주는 지표 역할을 하는 것이 optimizer이다.

Optimizer는 loss function을 기반으로 최적의 weight를 구한다. 다시 말해, optimizer는 입력된 데이터(train data set)으로 모델을 학습할 때 데이터의 실제값과 모델의 예측값을 기반으로 이 두 값의 차이가 줄어들어 예측값이 실제값에 가까워질 수 있도록 만드는 역할을 한다. Optimizer의 대표적인 방법으로 경사하강법(Gradient Descent), 확률적 경사 하강법(Stochastic Gradient Descent, SGD) 등이 있다. 딥러닝에서 일반적으로 이용되는 최적화 함수는 Adam, RMSProp가 있다.



NOTE)

reference : <http://www.slideshare.net/yongho/ss-79607172>

NOTE)

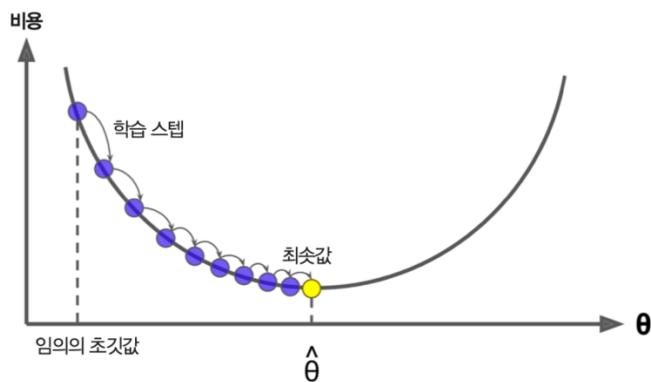
최적화(optimization)란 함수의 극대, 극소값을 찾는 것이다. 즉, 함수를 최소화하거나 최대화하는 지점을 찾는 과정을 의미하는데 이때 이용되는 함수를 목적함수(object function)이라고 한다. 딥러닝에서는 이 목적함수를 최적화시킴으로써 학습을 진행한다. 다시 말해, 최적화란 목적함수를 최대화, 최소화하는 파라미터 조합을 찾는 과정이다.

a) 경사 하강법(Gradient Descent)

이차함수를 생각해보자. 이 함수는 미분가능하며 최고차항의 차수에 따라 극소 혹은 극대값을 가진다. 미분가능한 함수에서는 극소 혹은 극대값에서의 미분계수는 0이므로 극값을 찾기 위해선 함수의 미분계수가 0이 되는 지점들을 바탕으로 찾으면 된다. 이 개념을 신경망에 적용시키면, 경사 하강법이란 N 차원의 가중치에서 loss를 최소로 만드는 조합을 찾는 과정이다. 다시 말해, 다음 방정식을 푸는 것이라고 생각할 수 있다.

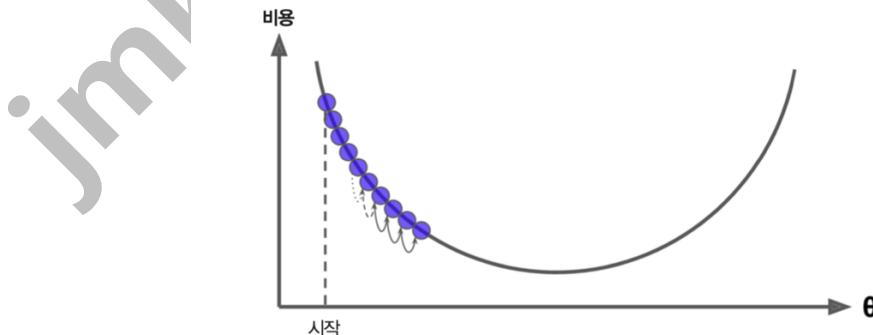
$$\text{gradient } f(\theta) = 0$$

경사 하강법은 여러 종류의 문제에서 최적의 해법을 찾을 수 있는 일반적인 최적화 알고리즘이다. 경사 하강법의 기본 아이디어는 손실 함수를 최소화하기 위해 반복해서 파라미터를 조정해가는 것이다. 예를 들어, 짙은 안개 때문에 산속에서 길을 잃었다고 해보자. 이 상황에서는 지면의 기울기에만 의존하여 길을 찾아야 할 것이다. 그렇다면 가장 빠르게 골짜기로 내려가는 방법은 가장 가파른 길을 따라 아래로 내려가는 것이다. 이것이 경사 하강법의 원리이다. 파라미터에 대해 손실 함수의 현재 그레디언트를 계산하고 이 값이 감소하는 방향으로 진행한다. 그레디언트가 0이 되면 최솟값에 도달하게 된다.

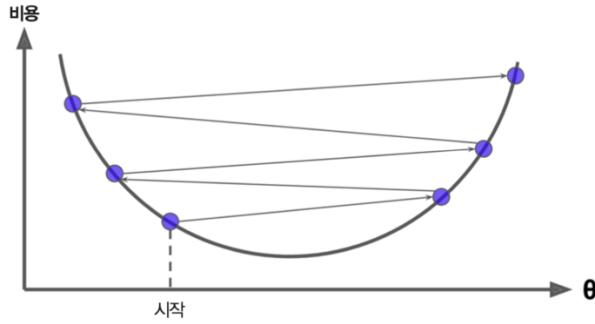


그림을 보면 파라미터 θ 가 랜덤하게 초기화된 후 한 번에 조금씩 손실 함수(MSE, cross-entropy 등)가 감소되는 방향으로 진행하여 알고리즘이 최솟값에 수렴할 때까지 점진적으로 향상된다.

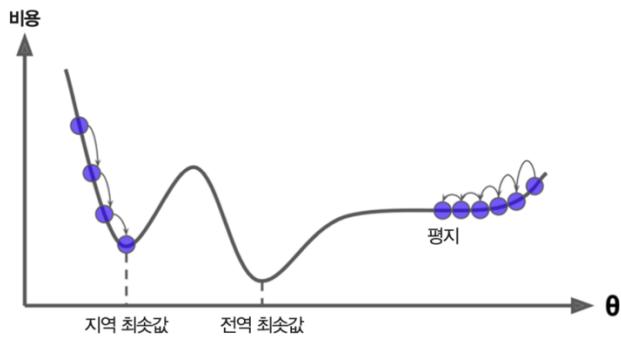
경사 하강법에서 중요한 하이퍼파라미터(hyper-parameter)는 학습률(learning rate)이다. 학습률은 그레디언트를 계산후 현재의 θ 로부터 다음 θ 까지 얼마나 걸어갈 것인지 스텝의 크기를 결정하는 값이다. 학습률이 너무 작으면 알고리즘이 수렴하기 위해 반복을 많이 진행해야 하므로 시간이 오래걸린다.



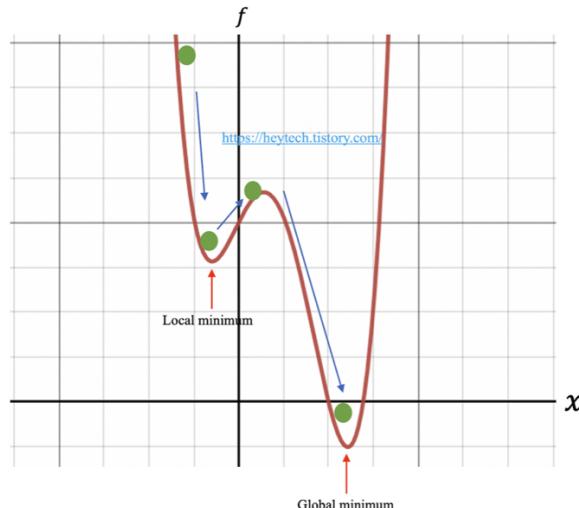
한편, 학습률이 너무 크면 골짜기를 가로질러 반대편으로 건너뛰게 되어 이전보다 더 높은 곳으로 올라가게 될 수도 있다. 이는 알고리즘을 더 큰 값으로 발산하게 만들어 적절한 방향성을 찾기 못하게 방해한다.



또 다른 문제점은 무작위 초기화로 알고리즘이 시작점에 따라 global minimum 보다 덜 좋은 local minimum에 수렴할 수도 있다. 아래의 그림에서 왼쪽의 시작점은 학습의 과정에서 local minimum에 도달한다.



일반적으로 파라미터 공간의 차원은 매우 크기 때문에 수많은 local minimum이 존재한다. 이 문제를 피하기 위한 방법이 momentum이다. Momentum은 물리학 용어로 관성의 법칙을 의미한다. 다시 말해, 경사진 곳에서 물체가 굴러가면 운동을 지속하려는 성질인데 이 성질을 활용하여 고안된 momentum은 경사 하강법으로 이동할 때 관성을 부여하는 최적화 기법이다.



즉, momentum은 이전에 이동했던 방향을 기억하면서 이전 그레디언트의 크기를 고려하여 정의된 학습률에서 추가로 이동을 한다. 이 방법으로 local minimum을 빠져나갈 수 있으며 파라미터 공간의 모든 minimum을 찾게 되므로 local minimum에 빠져 학습을 멈추지 않고 global minimum을 탐색할 수 있는 것이다.

딥러닝의 경사 하강법에 대해 정리하자면, 이 방법은 순실 함수를 최소화하는 방향으로 파라미터들을 반복적으로 업데이트한다. 심층 신경망에서 전체 네트워크의 순실 함수는 파라미터 θ (개별 가중치들의 집합)에 대해 미분 가능한 함수의 형태이다. 이를 이용해 각 층의 가중치들에 대한 loss의 미분, 즉

그레디언트를 계산할 수 있다. $\text{Loss}(E)$ 의 계산에 포함된 어떤 가중치 w 에 대해 그레디언트를 구하면 w 로부터 단위(Δw)당 변화량을 계산할 수 있고, 이것이 감소하는 방향(descent)으로 가중치를 업데이트한다.

$$\theta_{ij}^* = \theta_{ij} - \eta \cdot \frac{\delta E}{\delta \theta_{ij}}$$

여기서 η 는 하이퍼 파라미터인 학습률(learning rate)에 의해 결정되는 값이다. 학습률이 너무 크면 현재 위치에서 보폭을 매우 크게 하는 것으로 최적값으로 수렴하지 않고 더 큰 값으로 발산할 가능성이 있으며, 학습률이 매우 작을 경우 알고리즘이 수렴하기 위해 반복을 많이 진행해야 하므로 시간이 많이 걸린다.

b) 배치 경사 하강법(Batch Gradient Descent)

신경망에서 경사 하강법을 구현하기 위해선 모델의 파라미터 θ_j 에 대해 손실 함수의 그레디언트를 계산해야 한다. 다시 말해 θ_j 가 변할 때 손실 함수가 얼마나 바뀌는지 계산한다. X 가 N 개의 sample로 이루어진 전체 훈련 셋이라면 배치 경사 하강법은 훈련 데이터 전체를 사용해서 그레디언트 벡터를 계산한다. 또한 loss의 계산 시 전체 훈련 셋에 포함된 각각의 샘플에 대한 오차를 전체 훈련 셋에 걸쳐 평균을 계산한다. 일괄적으로 처리하는 계산적 속성때문에 이 알고리즘을 **배치 경사 하강법**이라고 한다. 만약 훈련 데이터 셋의 크기가 매우 크면 속도가 매우 떨어지며, 복잡한 모형일수록 심한 속도 저하가 발생한다. 그러나 점진적으로 최소값으로 수렴하고 알고리즘이 안정적이다.

c) 확률적 경사 하강법(Stochastic Gradient Descent)

배치 경사 하강법의 단점인 속도 저하 문제를 보완할 수 있는 방법이 확률적 경사 하강법이다. 이 알고리즘은 매 단계마다 하나의 샘플을 랜덤으로 선택하여 그 샘플에 대한 그레디언트를 계산한다. 매 반복에서 매우 적은 데이터만을 처리하기 때문에 알고리즘의 속도가 훨씬 빠르며 메모리 소모 비용 또한 적다. 그러나 이 알고리즘은 배치 경사 하강법보다 훨씬 불안정하며, loss가 최소에 도착할 때까지 부드럽게 감소하지 않고 위아래로 요동치면서 감소한다. 만약 loss가 전역과 지역 최솟값을 모두 가지는 형태라면 배치 방법은 로컬 값에 빠질 경우 빠져나오기 힘들지만 SGD는 확률적으로 로컬 값에서 벗어날 수 있도록 도와준다. 따라서 배치 방법보다 전역 최솟값을 빠르게 찾을 가능성이 높지만, 학습을 멈추기까지 전역 최솟값을 찾지 못하고 불안정하게 종료될 수 있다. 일반적으로 학습의 한 epoch에서 훈련 셋의 샘플의 개수인 n 번 되풀이되도록 한다. 샘플을 무작위로 선택하기 때문에 어떤 샘플은 한 epoch에서 여러번 선택되거나 아예 선택되지 않을 수 있다. 따라서 훈련 셋 안의 각각의 샘플들이 서로 동질적이라면 계산 비용을 효과적으로 절약하면서도 좋은 파라미터의 추정치를 찾을 수 있으나, 그렇지 않은 경우 노이즈가 발생한다. 각 epoch마다 모든 샘플을 사용하게 하려면 훈련 셋을 랜덤하게 섞은 후(shuffle) 차례대로 하나씩 선택하게 할 수 있다.

NOTE)

stochastic is a scientific synonym of random

d) 미니배치 경사 하강법(Mini-Batch Gradient Descent)

배치 경사 하강법의 단점인 속도 저하 문제를 보완할 수 있는 방법이 확률적 경사 하강법이다. 이 알고리즘은 매 단계마다 하나의 샘플을 랜덤으로 선택하여 그 샘플에 대한 그레디언트를 계산한다. 매 반복에서 매우 적은 데이터만을 처리하기 때문에 알고리즘의 속도가 훨씬 빠르며 메모리 소모 비용 또한 적다. 그러나 이 알고리즘은 배치 경사 하강법보다 훨씬 불안정하며, loss가 최소에 도착할 때까지 부드럽게 감소하지 않고 위아래로 요동치면서 감소한다. 만약 loss가 전역과 지역 최솟값을 모두 가지는 형태라면 배치 방법은 로컬 값에 빠질 경우 빠져나오기 힘들지만 SGD는 확률적으로 로컬 값에서 벗어날 수 있도록 도와준다. 따라서 배치 방법보다 전역 최솟값을 빠르게 찾을 가능성이 높지만, 학습을 멈추기까지 전역 최솟값을 찾지 못하고 불안정하게 종료될 수 있다. 일반적으로 학습의 한 epoch에서 훈련 셋의 샘플의 개수인 n 번 되풀이되도록 한다. 샘플을 무작위로 선택하기 때문에 어떤 샘플은 한 epoch에서 여러번 선택되거나 아예 선택되지 않을 수 있다. 따라서 훈련 셋 안의 각각의 샘플들이 서로 동질적이라면 계산 비용을 효과적으로 절약하면서도 좋은 파라미터의 추정치를 찾을 수 있으나, 그렇지 않은 경우 노이즈가 발생한다. 각 epoch마다 모든 샘플을 사용하게 하려면 훈련 셋을 랜덤하게 섞은 후(shuffle) 차례대로 하나씩 선택하게 할 수 있다.

도록 한다. 샘플을 무작위로 선택하기 때문에 어떤 샘플은 한 epoch에서 여러번 선택되거나 아예 선택되지 않을 수 있다. 따라서 훈련 셋 안의 각각의 샘플들이 서로 동질적이라면 계산 비용을 효과적으로 절약하면서도 좋은 파라미터의 추정치를 찾을 수 있으나, 그렇지 않은 경우 노이즈가 발생한다. 각 epoch마다 모든 샘플을 사용하게 하려면 훈련 셋을 랜덤하게 섞은 후(shuffle) 차례대로 하나씩 선택하게 할 수 있다.

미니배치는 배치와 SGD의 결합으로 각 스텝에서 전체 훈련 셋에서 미니배치라고 부르는 임의의 작은 샘플 셋에 대해 그라디언트를 계산한다. SGD에 비해 얻는 주요 장점은 행렬 연산에 최적화된 GPU에서 얻는 성능 향상이다. 미니배치 사이즈를 어느 정도 크게 하면 파라미터의 공간에서 SGD보다 덜 불규칙하게 움직이며 SGD보다 최솟값에 더 가까이 도달하게 될 것이다. 따라서 연산량을 줄이면서도 어느 정도 안정적인 학습을 위하여 일반적으로 16, 32, 64, 128 등의 배치 사이즈를 이용한 미니배치 학습을 주로 사용한다.

SGD와 미니배치 방법에서 가중치의 변동을 줄이기 위한 방법으로 학습률을 점진적으로 감소시키는 방법이 있다. 초기 학습 단계에서는 학습률을 크게 해 수렴을 빠르게 하고 지역 최솟값에 빠지지 않게 하며, 점차 작게 줄여서 전역 최솟값에 도달하도록 하는 것이다.

cf. 배치 정규화(Batch Normalization, BN) (2015)

학습 동안 이전 층의 파라미터가 변함에 따라 각 층에 들어오는 입력의 분포가 변화하는 문제(내부 공변량의 변화)를 발견하고 이를 해결하기 위한 기법으로 개발되었다. 입력의 분포가 학습할 때마다 변하면서 가중치가 영뚱한 방향으로 갱신될 문제가 발생할 수 있다. 일반적으로 DNN에서, 각 층의 노드 하나에서는 활성화 함수를 통과하기 전 계산된 $x^{(i)}$ 들을 미니배치 단위로 표준화한 후 각 층에서 두 개의 새로운 파라미터 γ (scale), β (bias)로 결과 값의 스케일을 조정하고 이동시킨다. 각 층의 연산 $x = Wu + b$ 에 배치 정규화를 적용하는 형태로 표현할 수 있다. 이는 곧 $Wu + b$ 를 정규화 하는 것이 되므로, b 의 영향이 사라지기 때문에 b 를 무시한다. 따라서 이 연산으로 네트워크는 층마다 현재의 미니배치에서 평균과 표준편차를 평가하며, 최적의 γ, β 를 학습한다. 이를 통해 학습률을 높게 설정해도 그라디언트가 폭주/소실되거나 나쁜 local minima에 빠지는 문제가 없어 학습 속도가 개선되며, 가중치 초기값 선택의 의존성이 적어진다. 또한 과대적합 위험을 줄이고 그라디언트 소실 문제를 해결할 수 있다.

$B = \{x^{(1)}, \dots, x^{(m_B)}\}$: 활성화 함수를 통과하기 전 현재 미니배치에서 계산된 모든 출력 값의 집합

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2$
3. $\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
4. $z^{(i)} = \gamma \hat{x}^{(i)} + \beta$

만약 현재 층에서 활성화 직전 계산된 (미니배치 샘플 수, 출력 채널의 수) 크기의 행렬이 있다면 배치 정규화에서 출력 채널 하나당 평균과 분산을 계산한다. 즉 배치 정규화는 각 층의 노드 하나하나에서 독립적으로 진행되는 것이다. 층에서 출력되는 채널마다 독립적으로 사용되는 γ 와 β 를 각각 학습한다.

합성곱 층에서는 어떤 위치에서 필터를 적용하든지 같은 파라미터 값을 공유하기 때문에 배치 정규화가 조금 다르게 이루어진다. 일반적인 배치정규화와 동일하다면 미니 배치의 크기를 m , 하나의 출력 특성맵의 크기를 (p, q) 라고 할 때, (p, q) 의 각 원소에 대해 각각 미니배치 정규화가 진행될 것이다. 그러나 합성곱 층에서의 (p, q) 를 하나의 단위로 보고 (m, p, q) 에 대해 각각 하나의 평균과 분산을 구하여 정규화 한다.

6) 순전파(Forward Propagation)와 역전파(Back Propagation)

지금까지 배운 딥러닝 모델의 과정을 정리하면 다음과 같다.

모델을 구성하는 기본단위는 하나의 층(layer)으로 input layer, hidden layer, output layer로 나눈다. 각 층의 역할과 그들의 연결이 어떤 의미인지 이해하기 위해서는 딥러닝 모델의 목적에 대해 알고 있어야 한다. 통계학에서 좋은 성능을 보이는 딥러닝 모델이란 실제값에 가까운 예측값을 출력하는 모델이다. 이때, 실제값과 예측값의 차이를 loss라고 정의하는데 딥러닝 모델의 학습은 입력된 훈련 데이터의 loss를 줄이는 방향으로 이루어진다. 계산된 loss를 기준으로 최적화(optimizer) 단계를 거쳐 가중치(Weight, W)가 업데이트 된다. 즉, 최적화 단계에서는 loss를 줄일 수 있는 방향으로 가중치를 계산한다. 마지막으로, 업데이트 된 가중치는 다음 과정의 가중치로 입력되며 학습이 반복적으로 이루어진다.

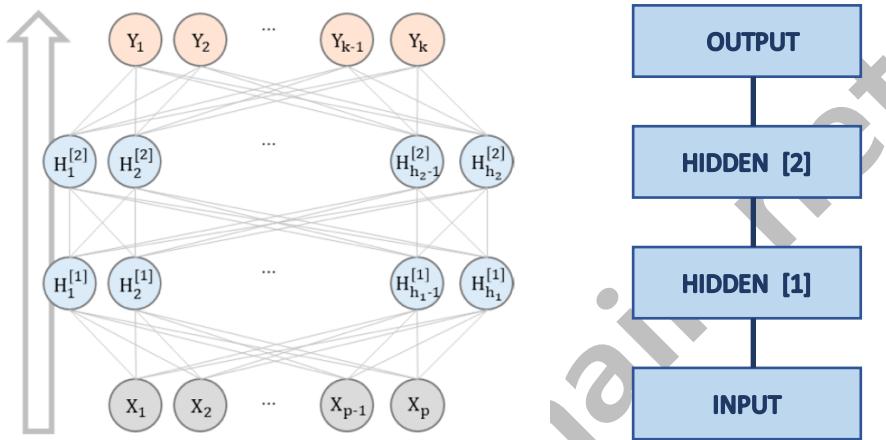
그렇다면, 하나의 층안에서 일어나는 계산과정을 살펴보자. 사전에 학습된 모형이 없는 경우 파라미터(가중치와 편향)의 초기값을 직접 모델에 입력해야 한다. loss를 줄이기 위해 초기값을 어떻게 설정할 것인지에 대해서 앞에서 다루었다. 파라미터의 초기값이 주어지면 각 훈련 샘플을 네트워크에 입력(input layer)하고 연속되는 각 층마다 출력을 계산(hidden layer)한다. 이때, 각 층에서는 $output = f(W \cdot input + b)$ 의 연산을 한다. 전체 알고리즘을 따라 하나의 입력 데이터(하나의 관측치)에 대한 하나의 최종 출력 값(output layer)을 계산할 수 있다. 또한 계산된 출력값을 이용해 실제값과의 오차 및 loss를 계산한다.

순전파와 역전파는 가중치(W)를 계산하기 위한 최적화 단계에서 그래디언트를 어떻게 계산할 것인지에 대한 개념이다.

a) 순전파(forward propagation)

순전파(forward propagation)은 네트워크의 입력층부터 출력층까지 순서대로 변수들을 계산하고 저장하는 것을 의미한다.

입력층과 2개의 은닉층, 출력층으로 이루어진 분류모형을 예시로 보자. MLP에서 각 층과 층사이의 노드는 완전 연결되어 있다.

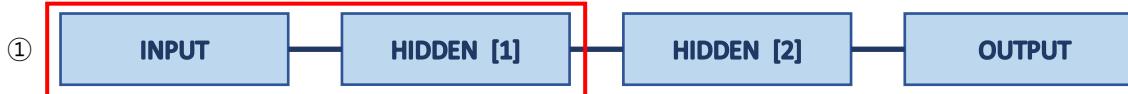


m	배치 사이즈* (한번의 연산에 사용되는 입력 데이터의 수)
p	입력층의 노드 개수 (=입력 특징의 수)
h_1	은닉층 1의 노드 개수
h_2	은닉층 2의 노드 개수
k	출력층의 노드 개수(=출력 클래스의 수)

NOTE)

학습 한번에 모든 데이터를 사용할 수도 있지만 한번에 일부 데이터(m 개)만을 사용해 적합하는 미니배치 방법론을 주로 많이 사용하므로 이 방법론으로 연산 과정을 설명하였다. 배치 사이즈에 대한 자세한 설명은 뒤에서 다시 다루기로 한다.

각 층의 계산 과정을 행렬로 표현하면 다음과 같다.



$$X_{batch} \cdot W^{[1]} + b^{[1]} = z^{[1]}, \quad a^{[1]} = activation(z^{[1]})$$

$$\begin{bmatrix} X_{1,1} & \cdots & X_{1,p} \\ \vdots & \ddots & \vdots \\ X_{m,1} & \cdots & X_{m,p} \end{bmatrix} \begin{bmatrix} W_{1,1} & \cdots & W_{1,h_1} \\ \vdots & \ddots & \vdots \\ W_{p,1} & \cdots & W_{p,h_1} \end{bmatrix}^{[1]} + \begin{bmatrix} b_1 \\ \vdots \\ b_{h_1} \end{bmatrix}^{[1]} = \begin{bmatrix} z_{1,1} & \cdots & z_{1,h_1} \\ \vdots & \ddots & \vdots \\ z_{m,1} & \cdots & z_{m,h_1} \end{bmatrix}^{[1]} \xrightarrow{f} \begin{bmatrix} a_{1,1} & \cdots & a_{1,h_1} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,h_1} \end{bmatrix}^{[1]}$$

$(m, p) \times (p, h_1)$

$(1, h_1)$

(m, h_1)



$$a^{[1]} \cdot W^{[2]} + b^{[2]} = z^{[2]}, \quad a^{[2]} = activation(z^{[2]})$$

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,h_1} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,h_1} \end{bmatrix}^{[1]} \begin{bmatrix} W_{1,1} & \cdots & W_{1,h_2} \\ \vdots & \ddots & \vdots \\ W_{h_1,1} & \cdots & W_{h_1,h_2} \end{bmatrix}^{[2]} + \begin{bmatrix} b_1 \\ \vdots \\ b_{h_2} \end{bmatrix}^{[2]} = \begin{bmatrix} z_{1,1} & \cdots & z_{1,h_2} \\ \vdots & \ddots & \vdots \\ z_{m,1} & \cdots & z_{m,h_2} \end{bmatrix}^{[2]} \xrightarrow{f} \begin{bmatrix} a_{1,1} & \cdots & a_{1,h_2} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,h_2} \end{bmatrix}^{[2]}$$

$(m, h_1) \times (h_1, h_2)$

$(1, h_2)$

(m, h_2)



$$a^{[2]} \cdot W^{[3]} + b^{[3]} = z^{[3]}, \quad \hat{p} = softmax(z^{[3]}) = \frac{\exp(z^{[3]})}{\sum \exp(z^{[3]})}$$

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,h_2} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,h_2} \end{bmatrix}^{[2]} \begin{bmatrix} W_{1,1} & \cdots & W_{1,k} \\ \vdots & \ddots & \vdots \\ W_{h_2,1} & \cdots & W_{h_2,k} \end{bmatrix}^{[3]} + \begin{bmatrix} b_1 \\ \vdots \\ b_{h_2} \end{bmatrix}^{[3]} = \begin{bmatrix} z_{1,1} & \cdots & z_{1,k} \\ \vdots & \ddots & \vdots \\ z_{m,1} & \cdots & z_{m,k} \end{bmatrix}^{[3]}$$

$(m, h_2) \times (h_2, k)$

$(1, k)$

$$\xrightarrow{softmax} \begin{bmatrix} \frac{\exp(z_{1,1})}{\sum_i \exp(z_{1,i})} & \cdots & \frac{\exp(z_{1,k})}{\sum_i \exp(z_{1,i})} \\ \vdots & \ddots & \vdots \\ \frac{\exp(z_{m,1})}{\sum_i \exp(z_{m,i})} & \cdots & \frac{\exp(z_{m,k})}{\sum_i \exp(z_{m,i})} \end{bmatrix}$$

④ LOSS

입력한 관측치 하나당 정의되는 LOSS를 계산하고 미니배치 안에서 m개 관측치의 평균 LOSS(E)를 구한다.

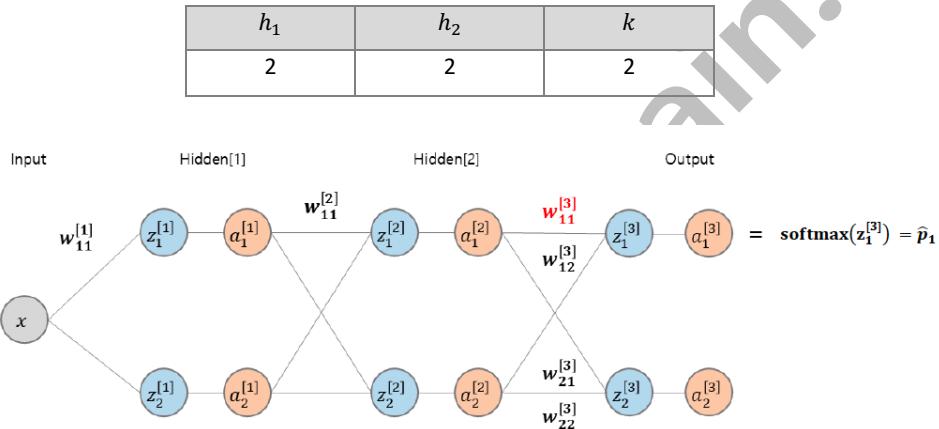
$$\text{관측치 } i\text{의 LOSS}_i = -\sum_k I(y_i \in C_k) \cdot \log \hat{p}_i, \quad E = \frac{1}{m} \sum_{i=1}^m \text{LOSS}_i$$

b) 역전파(Back Propagation)

심층신경망에는 층과 층 사이에 매우 많은 가중치 파라미터가 존재하고, 학습 한번 당 각 층의 모든 가중치를 경사 하강법을 이용해 업데이트해야 한다. 이 연산을 빠르고 효율적으로 하는 방법으로 역전파(Back Propagation) 알고리즘을 사용한다.

역전파는 입력층의 입력 데이터에 대해 전방향 연산(Forward Propagation)을 통해 계산된 오차를 네트워크에 존재하는 각각의 노드(퍼셉트론)에 역으로 전파하는 과정이라고 할 수 있다. 마지막 은닉층에서 각 뉴런들의 가중치에 대해 loss의 편미분을 다시 계산하고, 다시 이전 은닉층의 뉴런의 가중치에 대한 해당 그래디언트의 편미분을 계산하는 과정을 반복하여 입력 층에 도달할 때까지 역으로 계산한다. 이 역방향 과정은 오차 그래디언트를 후방으로 전파한다고 하여 '역전파(Back Propagation)'라고 한다.

$k - 1$ 번째 층(입력층)의 i 번째 노드와 k 번째 층(출력층)의 j 번째 노드를 연결하는 가중치 w 를 $w_{ij}^{[k]}$ 라고 한다. 순전파와 동일하게 입력층과 출력층 사이에 2개의 은닉층이 있는 신경망을 예시로 사용하였다.



순전파 과정에서 입력 x 에 대해 마지막 층과 층 사이에서는 다음의 연산이 일어난다.

$$\begin{aligned} z_1^{[3]} &= a_1^{[2]} w_{11}^{[3]} + a_2^{[2]} w_{21}^{[3]}, & z_2^{[3]} &= a_1^{[2]} w_{12}^{[3]} + a_2^{[2]} w_{22}^{[3]} \\ a_1^{[3]} &= f(z_1^{[3]}), & a_2^{[3]} &= f(z_2^{[3]}) \end{aligned}$$

마지막 활성화를 거쳐 계산된 출력, 즉 예측값과 실제값을 사용해 LOSS(E)를 구하고 나면 역전파 알고리즘을 시작한다. 전체 가중치의 업데이트를 위해 먼저 출력층과 마지막 은닉층으로 출발한다. $w_{11}^{[3]}$ 을 업데이트하기 위한 $\frac{\partial E}{\partial w_{11}^{[3]}}$ 의 계산은 Chain Rule에 의해 다음과 같이 계산할 수 있다.

$$\begin{aligned} \frac{\partial E}{\partial w_{11}^{[3]}} &= \frac{\partial E}{\partial a_1^{[3]}} \cdot \frac{\partial a_1^{[3]}}{\partial z_1^{[3]}} \cdot \frac{\partial z_1^{[3]}}{\partial w_{11}^{[3]}} \\ \frac{\partial E}{\partial a_1^{[3]}} &= \frac{\partial}{\partial a_1^{[3]}} E(a_1^{[3]}, y_1), \\ \frac{\partial a_1^{[3]}}{\partial z_1^{[3]}} &= f(z_1^{[3]}) \cdot f'(z_1^{[3]}) = a_1^{[3]} f'(f^{-1}(a_1^{[3]})) \\ \frac{\partial z_1^{[3]}}{\partial w_{11}^{[3]}} &= a_1^{[3]} \end{aligned}$$

이 과정에서 E 의 가중치에 대한 미분 값이 결국 역전파의 출발 노드의 활성함수 값과 도착 노드의 활성함수 값과 실제 출력 값으로만 표현 가능하다는 것을 알 수 있다. 계산된 그래디언트 $\frac{\partial E}{\partial a_1^{[3]}} \cdot \frac{\partial a_1^{[3]}}{\partial z_1^{[3]}}$ 을 $\delta_1^{[3]}$ 라고 정의하고 이것을 전 단계로 전파하여 마지막 은닉층으로부터의 가중치 $w_{11}^{[3]}$ 와 $w_{21}^{[3]}$ 를 업데이트 한다.

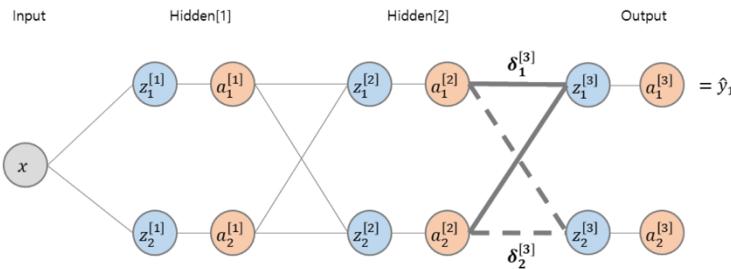
$$w_{11}^{[3]} \leftarrow w_{11}^{[3]} - \eta \cdot \delta_1^{[3]} \cdot a_1^{[2]}$$

$$w \leftarrow w_{21}^{[3]} - \eta \cdot \delta_1^{[3]} \cdot a_1^{[2]}$$

마찬가지로, $\delta_2^{[3]} = \frac{\partial E}{\partial a_2^{[3]}} \cdot \frac{\partial a_2^{[3]}}{\partial z_2^{[3]}}$ 을 이용해 $w_{12}^{[3]}$ 와 $w_{22}^{[3]}$ 를 업데이트 한다. 이와 같이 역전파를 통해 전달되는 값이 $\delta_1^{[3]}$ 과 $\delta_2^{[3]}$ 뿐이더라도 해당 층에서 관련된 가중치를 모두 업데이트 할 수 있다.

$$w_{12}^{[3]} \leftarrow w_{12}^{[3]} - \eta \cdot \delta_2^{[3]} \cdot a_2^{[2]}$$

$$w_{22}^{[3]} \leftarrow w_{22}^{[3]} - \eta \cdot \delta_2^{[3]} \cdot a_2^{[2]}$$



다음으로 이전 단계의 가중치 $w_{11}^{[2]}$ 를 업데이트하기 위해서는 $\frac{\partial E}{\partial w_{11}^{[2]}}$ 을 계산해야 한다.

$$\frac{\partial E}{\partial w_{11}^{[2]}} = \frac{\partial E}{\partial a_1^{[2]}} \cdot \frac{\partial a_1^{[2]}}{\partial z_1^{[2]}} \cdot \frac{\partial z_1^{[2]}}{\partial w_{11}^{[2]}}$$

여기서 첫번째 편미분인 $\frac{\partial E}{\partial a_1^{[2]}}$ 은 다음과 같이 분할 가능하고 이것은 $a_1^{[2]}$ 가 전체 E 계산에 영향을 미치는 모든 경로의 합이라고 할 수 있다.

$$\frac{\partial E}{\partial a_1^{[2]}} = \frac{\partial E_1}{\partial a_1^{[2]}} + \frac{\partial E_2}{\partial a_1^{[2]}} = \frac{\partial E_1}{\partial z_1^{[3]}} \cdot \frac{\partial z_1^{[3]}}{\partial a_1^{[2]}} + \frac{\partial E_2}{\partial z_2^{[3]}} \cdot \frac{\partial z_2^{[3]}}{\partial a_1^{[2]}} = \delta_1^{[3]} w_{11}^{[3]} + \delta_2^{[3]} w_{12}^{[3]}$$

두 번째, 세 번째 편미분의 계산 역시 이전 과정의 동일한 과정을 거쳐 $w_{11}^{[2]}$ 의 업데이트는 다음과 같다.

$$w_{11}^{[2]} \leftarrow w_{11}^{[2]} - \delta_1^{[2]} a_1^{[1]}$$

이렇게 역으로 오차의 그래디언트를 연달아 전파하여 입력층까지 연결되는 모든 가중치들을 업데이트 할 수 있다.

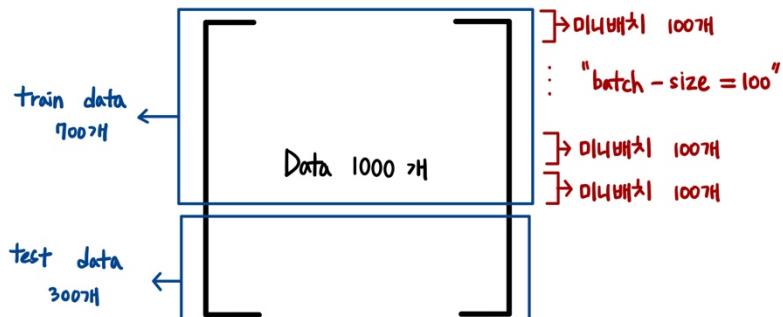
7) ITERATION

네트워크의 모든 가중치가 업데이트되고 나면 다시 처음의 단계로 돌아가 다음으로 입력되는 미니배치의 샘플에 대한 출력 값 및 오차를 정방향으로 계산하고, 그래디언트 계산 및 역전파를 거쳐 가중치를 다시 업데이트 한다. 즉 학습 과정은 LOSS가 충분히 작아질 때 까지, 또는 정해진 반복 수에 도달할 때까지 정방향과 역방향 연산을 번갈아 진행하며 파라미터 값을 조정하는 과정이라고 할 수 있다.

마지막으로 딥러닝 모델을 완성하기 위해 두가지 개념이 남았다. 한번에 몇개의 샘플을 학습할 것인지 결정하는 '배치 사이즈(batch size)'와 네트워크의 각 반복을 의미하는 '에포크(epoch)'이다. 학습률, 배치 사이즈, 에포크는 모델을 훈련하기 위해서 정의해야 될 값으로 이를 하이퍼 파라미터(hyper parameter)라고 한다.

a) 배치 사이즈(batch size)

전체 훈련 데이터 셋을 여러 작은 그룹으로 나누었을 때 batch size는 하나의 소그룹에 해당되는 데이터 수를 의미한다. 전체 훈련 셋을 미니 배치로 분할하는 이유는 훈련 데이터를 통째로 네트워크에 넣는 경우 학습 시간이 오래 걸리기 때문이다.

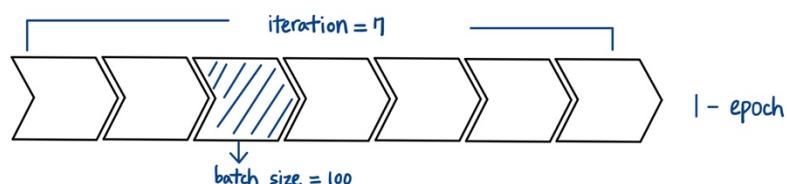


b) 에포크(epoch)

에포크는 전체 훈련 셋이 신경망을 통과한 횟수를 의미한다. 예를 들어, 1-epoch는 전체 훈련 셋이 하나의 신경망에 적용되어 순전파와 역전파를 통해 신경망을 한번 통과했다는 것을 의미한다.

c) iteration

iteration은 1-epoch를 마치는데 필요한 미니배치 갯수를 의미한다. 다시 말해, 1-epoch를 마치는데 필요한 파라미터 업데이트 횟수이기도 하다. 각 미니 배치마다 파라미터 업데이트가 한번씩 진행되므로 iteration은 파라미터 업데이트 횟수이자 미니배치 갯수이다.



이 반복적인 연산을 이용한 학습 과정은 다수의 완전 연결 층으로만 이루어진 MLP 모형뿐만 아니라 합성곱 계층과 풀링 계층을 포함하는 CNN의 구조에서도 수학적으로 동일하게 이루어진다.

8) 그레디언트 폭주/소실 문제

심층신경망을 훈련하는 과정에서 다음과 같은 문제를 마주할 수 있다.

a) 그레디언트 소실과 폭주(Vanishing and Exploding Gradient)

역전파 알고리즘은 출력층에서 입력층으로 오차 그레디언트를 전파하면서 진행된다. 알고리즘이 신경망의 모든 파라미터에 대한 오차 함수의 그레디언트를 계산하면서 경사 하강법 단계에서 이 그레디언트를 사용하여 각 파라미터를 수정한다.

그런데 알고리즘이 하위층으로 진행될수록 그레디언트가 점점 작아지는 경우가 많다. 이때, 경사 하강법이 하위층의 연결 가중치를 변경되지 않은 채로 둔다면 훈련이 좋은 솔루션으로 수렴되지 않는다. 이 문제를 그레디언트 소실(Vanishing Gradient)이라고 한다. 어떤 경우엔 반대 현상이 일어나는데, 그레디언트가 점점 커져서 여러 층이 비정상적으로 큰 가중치로 갱신되면 알고리즘은 발산한다. 이 문제를 그레디언트 폭주(Exploding Gradient)라고 하며 RNN에서 주로 일어난다. 일반적으로 불안정한 그레디언트는 심층신경망의 훈련을 어렵게 한다. 층마다 학습속도가 달라질 수 있기 때문이다.

이런 좋지 않은 학습 패턴이 꽤 오랫동안 경험적으로 관측되어왔다. 2000년대 초까지 심층신경망이 거의 방치되었던 이유 중 하나이다. 심층 신경망을 훈련할 때 그레디언트를 불안정하게 만드는 원인이 무엇인지 명확하게 밝혀지지 않았지만 2010년 Xavier Glorot과 Yoshua Bengio가 발표한 논문으로 이에 대한 이해가 진전되었다. 저자들은 몇가지 의심되는 원인을 발견했는데, 그중에는 많이 사용되는 로지스틱 시그모이드 활성화 함수와 그 당시 가장 인기 있던 가중치 초기화 방법(즉, 평균이 0이고 표준편차가 1인 정규분포)의 조합이었다. 이 활성화 함수와 초기화 방식을 사용했을 때 각 층에서 출력의 분산이 입력의 분산보다 더 크다는 것을 밝혀냈다. 신경망의 위쪽으로 갈수록 층을 지날때마다 분산이 계속 커져 가장 높은 층에서는 활성화 함수가 0이나 1로 수렴한다. 이는 로지스틱 함수의 평균이 0이 아니고 0.5라는 사실 때문에 더 나빠진 것이다.

9) 대표적인 deep learning model

심층 신경망(DNN)의 기본 단위는 퍼셉트론으로 복수의 퍼셉트론을 어떻게 연결하느냐에 따라 새로운 구조를 형성할 수 있다. 데이터의 속성에 따라 효과적으로 적용할 수 있으며 가장 많이 쓰이는 3가지 심층 신경망에 대해 소개한다.

- 완전 연결 신경망(fully-connected neural network)
- 합성곱 신경망(CNN, Convolutional Neural Network)
- 순환 신경망(RNN, Recurrent Neural Network)

10) Feature scaling

Feature Scaling 은 Feature 들의 크기와 범위를 정규화 시키는 것을 말한다. 아래 데이터 프레임에서 두 개의 변수인 height 와 weight 는 각각 다른 단위를 가지고 있다. 딥러닝 모델은 feature 의 단위를 고려하지 않고 숫자의 크기를 feature 의 중요도로 받아들이기 때문에 model 을 fitting 하기 이전에 feature scaling 과정이 필요하다.

a) Min-Max Normalization

Min-Max Normalization 은 최솟값이 0, 최대값이 1 이 되도록 scaling 한다.

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

```
import pandas as pd
from sklearn import preprocessing

height_weight_people = {'height' : [6.1, 5.9, 5.2],
                       'weight' : [140, 175, 115]}

df = pd.DataFrame(height_weight_people, index = ['A', 'B', 'C'])
df
```

	height	weight
A	6.1	140
B	5.9	175
C	5.2	115

```
# Min-Max Scaling
minmax_scaler = preprocessing.MinMaxScaler()
minmax_scaler.fit(df)
print(minmax_scaler.transform(df))

[[1.          0.41666667]
 [0.77777778 1.          ]
 [0.          0.          ]]
```

b) Standardization

Standardization 은 정규화 과정과 동일하다.

$$x' = \frac{x - \bar{x}}{\sigma}$$

```
standard_scaler = preprocessing.StandardScaler()
standard_scaler.fit(df)
print(standard_scaler.transform(df))

[[ 0.95025527 -0.13545709]
 [ 0.43193421  1.28684238]
 [-1.38218948 -1.15138528]]
```

Standardization 은 데이터가 Gaussian distribution 을 따르는 경우 이용하지만 반드시 이 가정을 지켜야 할 필요는 없다. 또한, Min-Max normalization 과는 다르게 standardization 은 bounding range 가 존재하지 않는다. 가장 좋은 방법은 raw data, Min-Max Normalization data, standardization data 를 각각 학습시킨 후 가장 좋은 performance 를 보이는 방법을 선택하는 것이다.

jmkim21@ewhain.net

3. regression and classification

1) Deep learning model

딥러닝 모델을 만들기 위한 대표적인 라이브러리는 tensorflow, pythoch 등 여러 라이브러리들이 존재한다. 이 중 keras를 이용하여 model을 만들어 보도록 한다.

우선 라이브러리를 불러들인다.

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

Sequential 모델은 층을 선형으로 연결하여 구성한다. 딥러닝 모델을 만들기 위한 첫단주이다.

```
from tensorflow import keras
model = keras.Sequential()
```

다음으로 add() 메서드를 이용하여 층을 추가할 수 있다.

```
model.add(Dense(32, activation = 'relu', input_dim=784))
model.add(Dense(32, activation = 'relu'))
```

모델을 학습시키기 이전에, compile 메소드를 통해서 학습 방식에 대한 환경설정이 필요하다. compile은 다음 세 개의 인자를 입력으로 받는다.

- **optimizer**: rmsprop나 adagrad와 같은 기존의 optimizer에 대한 문자열 식별자 또는 optimize 클래스의 인스턴스를 사용할 수 있다.
- **loss function** : 모델이 최적화에 사용되는 목적 함수이다. 회귀분석으로 경우 mse, 분류 문제의 경우 categorical_crossentropy로 설정한다.
- **metric** : 분류 문제에 대해서는 metrics=['accuracy']로 설정한다.

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')
```

2) Regression

심층신경망은 회귀분석 작업에 사용할 수 있다. 값 하나를 예측하는데(예를 들어 여러 특성을 이용하여 주택 가격을 예측할 때) 출력 뉴런이 하나만 필요하다. 이 뉴런의 출력이 예측된 값인데, 다변량 회귀의 경우(동시에 여러 값을 예측하는 경우) 출력 차원마다 출력 뉴런이 하나씩 필요하다. 예를 들어 이미지에서 물체의 중심 위치를 파악하려면 2D 좌표를 예측해야 하므로 두개의 출력값이 필요하다. 따라서 출력 뉴런 두 개가 필요하다.

일반적으로 회귀분석에 이용할 MLP를 만들 때 출력 뉴런에 활성화 함수를 사용하지 않고 어떤 범위의 값도 출력되도록 한다. 하지만 출력이 항상 양수여야 하는 경우 출력에 ReLU 활성화 함수를 사용하면 된다. 또한 softplus 활성화 함수를 사용할 수 있는데, 이 함수는 ReLU의 변종으로 다음과 같다.

$$\text{softplus}(z) = \log(1 + \exp(z))$$

이 함수는 z 가 음수일 때 0에 가까워지고 z 가 양수일수록 1에 가까워진다.

마지막으로 어떤 범위 안의 값을 예측해야 하는 경우 로지스틱 함수나 하이퍼볼릭 탄젠트 함수를 사용하고 레이블의 스케일을 적절한 범위로 조정할 수 있다. 로지스틱 함수는 0에서 1 사이를 출력하고 하이퍼 볼릭 탄젠트 함수는 -1에서 1 사이를 출력한다.

훈련에 사용하는 손실 함수는 일반적으로 mse를 사용한다. 하지만 훈련 데이터에 이상치가 많다면 대신 mae를 사용할 수 있다. 또는 이 둘을 조합한 Huber도 있다.

Regression MLP의 일반적인 hyper-parameter	
입력 뉴런 수	특성마다 하나
은닉층 수	문제에 따라 다름. 일반적으로 1에서 5 사이
은닉층의 뉴런 수	문제에 따라 다름. 일반적으로 10에서 100 사이
출력 뉴런 수	예측 차원마다 하나
은닉층의 활성화 함수	ReLU
출력층의 활성화 함수	일반적으로 설정하지 않으나 identity
손실함수	MSE(이상치가 많다면 MAE/Huber)

① Regression Example : Data in machine learning

Chapter 2의 multiple linear regression에서 다루었던 당뇨병 데이터로 딥러닝 모델을 정의 후 모델의 성능을 비교해보도록 한다. 딥러닝 모델을 정의할 때 모델의 하이퍼 파라미터는 모델의 성능에 영향을 준다. 따라서, 좋은 성능의 모델을 찾기 위해선 은닉층의 개수, 층마다 있는 뉴런의 개수, 각 층에서 사용하는 활성화 함수, 가중치 초기화, 에포크, 배치 사이즈 등 다양한 하이퍼 파라미터에 변화를 주며 최적의 조합을 찾아내야 한다. 이 예제에서는 5가지 하이퍼 파라미터의 조합을 시도하여 최적의 성능을 보이는 모델을 찾을 것이다.

a) import the dataset

```
# import package
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Library for regression
from sklearn import datasets

# import dataset
mydata = datasets.load_diabetes()

# feature columns
df = pd.DataFrame(mydata['data'], columns = mydata['feature_names'])

# target column 추가
df['diabetes_score'] = mydata['target']

df.head()
```

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	diabetes_score
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646	151.0
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204	75.0
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930	141.0
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362	206.0
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641	135.0

df.describe().T

	count	mean	std	min	25%	50%	75%	max
age	442.0	-3.634285e-16	0.047619	-0.107226	-0.037299	0.005383	0.038076	0.110727
sex	442.0	1.308343e-16	0.047619	-0.044642	-0.044642	-0.044642	0.050680	0.050680
bmi	442.0	-8.045349e-16	0.047619	-0.090275	-0.034229	-0.007284	0.031248	0.170555
bp	442.0	1.281655e-16	0.047619	-0.112400	-0.036656	-0.005671	0.035644	0.132044
s1	442.0	-8.835316e-17	0.047619	-0.126781	-0.034248	-0.004321	0.028358	0.153914
s2	442.0	1.327024e-16	0.047619	-0.115613	-0.030358	-0.003819	0.029844	0.198788
s3	442.0	-4.574646e-16	0.047619	-0.102307	-0.035117	-0.006584	0.029312	0.181179
s4	442.0	3.777301e-16	0.047619	-0.076395	-0.039493	-0.002592	0.034309	0.185234
s5	442.0	-3.830854e-16	0.047619	-0.126007	-0.033249	-0.001948	0.032433	0.133599
s6	442.0	-3.412882e-16	0.047619	-0.137767	-0.033179	-0.001078	0.027917	0.135612
diabetes_score	442.0	1.521335e+02	77.093005	25.000000	87.000000	140.500000	211.500000	346.000000

데이터의 개수는 442개이며, feature column 10개와 target column 1개로 구성된 데이터이다. 또한, 당뇨병 데이터는 feature scaling이 완료된 상태이며, 모든 변수가 numeric이므로 원-핫 인코딩이나 라벨인코딩 과정이 필요하지 않다.

b) data preprocessing

데이터를 훈련용 데이터와 테스트 데이터로 분할한다.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:, :-1], df.iloc[:, -1],
                                                    test_size = 0.2,
                                                    shuffle = True, random_state = 22)
```

모델을 만들고 학습하기에 앞서 모델안의 여러 하이퍼파라미터들을 바꿔가며 각 모델마다의 성능을 비교해보도록 한다. DNN 모델의 하이퍼파라미터는 다음과 같다.

- Dense layer 안의 배치사이즈와 활성화함수, 파라미터 초기화
- hidden layer 의 개수
- BN(BatchNoramlization layer) 추가
- learning rate(0.01, 0.05, 0.001)
- drop out layer 추가

Model 1

c) building model

```

import tensorflow as tf

model = tf.keras.Sequential([
    # input Layer
    tf.keras.layers.Dense(64, activation = "relu", input_dim = (len(df.columns)-1)),

    # hidden Layer
    tf.keras.layers.Dense(32, activation = "relu"),
    tf.keras.layers.Dense(16, activation = "relu"),

    # output Layer
    tf.keras.layers.Dense(1, activation = "linear")
])

# model compile
model.compile(optimizer = "adam", loss = "mse", metrics = [ 'mse'])

# model summary
model.summary()

Model: "sequential"

Layer (type)          Output Shape         Param #
===== 
dense (Dense)         (None, 64)           704
dense_1 (Dense)       (None, 32)            2080
dense_2 (Dense)       (None, 16)            528
dense_3 (Dense)       (None, 1)             17
=====
Total params: 3,329
Trainable params: 3,329
Non-trainable params: 0

```

모델의 hidden layer은 2개이며, 각 층의 활성화 함수는 relu이다. optimizer는 adam으로 설정하였으며, 회귀 문제이므로 loss function은 mse로 설정한다.

d) training model

모델에 대한 훈련은 1000번의 에포크(epoch)로 지정하며, 배치사이즈를 32로 지정해 미니배치로 학습 시킨다.

```

history = model.fit(X_train, y_train, batch_size = 32,
                     epochs = 100, validation_split = 0.2, verbose = 1)

```

학습의 과정을 시각화시키면 각 에포크마다 loss와 mse를 관찰할 수 있다.

```

# Loss function
def plot_loss_curve(total_epoch = 10, start = 1):

    # package for visualization
    import matplotlib.pyplot as plt
    import seaborn as sns
    sns.set_theme(color_codes = True)

    plt.figure
    plt.plot(figsize = (15,5))
    plt.plot(range(start, total_epoch +1), history.history['loss'][start-1:total_epoch],
             label = "Train")
    plt.plot(range(start, total_epoch +1), history.history['val_loss'][start-1:total_epoch],
             label = "Validation")
    plt.xlabel("Epochs")
    plt.ylabel("loss")
    plt.legend()
    plt.show()

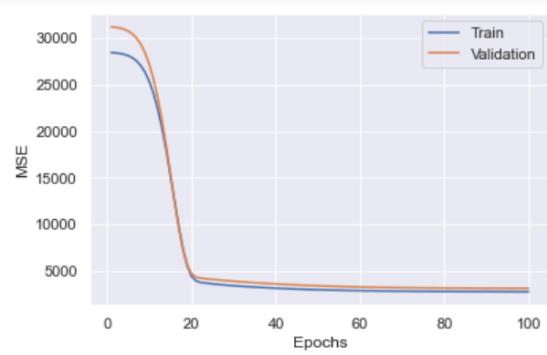
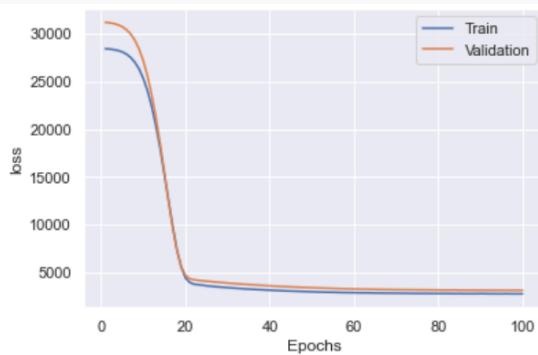
# mse function
def plot_mse_curve(total_epoch = 10, start = 1):

    # package for visualization
    import matplotlib.pyplot as plt
    import seaborn as sns
    sns.set_theme(color_codes = True)

    plt.figure
    plt.plot(figsize = (15,5))
    plt.plot(range(start, total_epoch +1), history.history['mean_squared_error'][start-1:total_epoch],
             label = "Train")
    plt.plot(range(start, total_epoch +1), history.history['val_mean_squared_error'][start-1:total_epoch],
             label = "Validation")
    plt.xlabel("Epochs")
    plt.ylabel("MSE")
    plt.legend()
    plt.show()

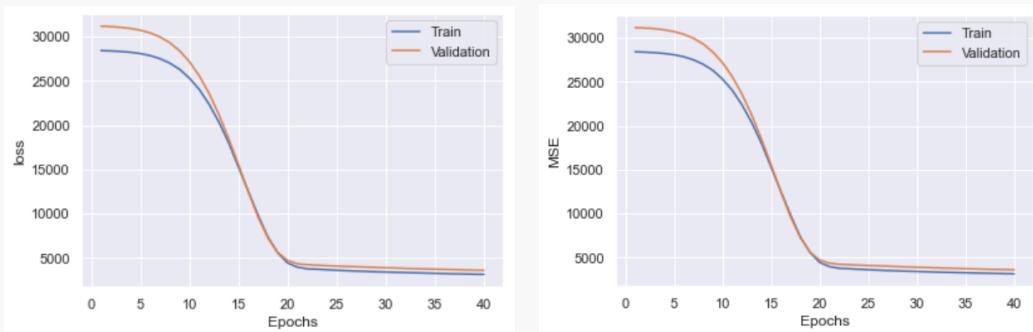
plot_loss_curve(total_epoch=100, start = 1)
plot_mse_curve(total_epoch=100, start = 1)

```



시작 지점을 자세하게 보도록 한다.

```
plot_loss_curve(total_epoch=40, start = 1)
plot_mse_curve(total_epoch=40, start = 1)
```



epoch가 30번이 지난후부터는 loss와 mse의 두드러진 감소가 나타나지 않고 있다.

e) Test model

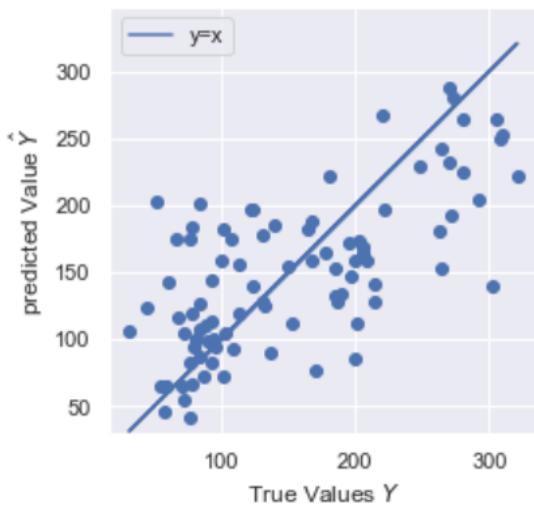
```
res = model.evaluate(X_test, y_test, verbose = 1)
print(f'테스트 세트의 MSE : {round(res[1], 4)}')
```

테스트 세트의 MSE : 3283.789306640625

모델의 성능을 판단하기 위해 test data를 이용하여 predict값을 구하고, regression 문제이므로 plot과 R^2 를 이용하여 판단한다.

```
# plot
import matplotlib.pyplot as plt

plt.scatter(y_test, model.predict(X_test))
plt.xlabel("True Values $Y$")
plt.ylabel("predicted Value $\hat{Y}$")
plt.axis("equal")
plt.axis("square")
plt.plot(y_test, y_test, label = "y=x")
plt.legend()
plt.show()
```



```
# R_square
from sklearn.metrics import r2_score
score = r2_score(y_test, model.predict(X_test))
score

0.4643671202819214
```

chapter 2의 머신러닝 모델의 R^2 는 0.518로 딥러닝 모델의 예측력이 더 낫다.

Model 2

c) building model

```
import tensorflow as tf

model = tf.keras.Sequential([
    # input layer
    tf.keras.layers.Dense(256, activation = "relu", input_dim = (len(df.columns)-1)),

    # hidden layer
    tf.keras.layers.Dense(128, activation = "relu"),
    tf.keras.layers.Dense(64, activation = "relu"),
    tf.keras.layers.Dense(32, activation = "relu"),
    tf.keras.layers.Dense(16, activation = "relu"),

    # output layer
    tf.keras.layers.Dense(1, activation = "linear")
])

# model compile
model.compile(optimizer = "adam", loss = "mse", metrics = [ 'mse'])

# model summary
model.summary()

Model: "sequential_1"

Layer (type)          Output Shape         Param #
=====
dense_4 (Dense)      (None, 256)          2816
dense_5 (Dense)      (None, 128)          32896
dense_6 (Dense)      (None, 64)           8256
dense_7 (Dense)      (None, 32)           2080
dense_8 (Dense)      (None, 16)           528
dense_9 (Dense)      (None, 1)            17
=====
Total params: 46,593
Trainable params: 46,593
Non-trainable params: 0
```

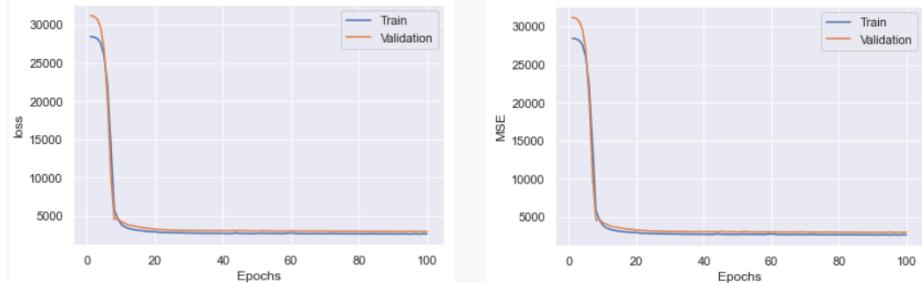
모델의 hidden layer 은 4 개이며, 각 층의 활성화 함수는 relu 이다. optimizer 는 adam 으로 설정하였으며, 회귀 문제이므로 loss function 은 mse 로 설정한다.

d) training model

모델에 대한 훈련은 100 번의 에포크(epoch)로 지정하며, 배치사이즈를 32 로 지정해 미니 배치 방법으로 학습한다.

```
history = model.fit(X_train, y_train, batch_size = 32,
                     epochs = 100, validation_split = 0.2, verbose = 0)

plot_loss_curve(total_epoch=100, start = 1)
plot_mse_curve(total_epoch=100, start = 1)
```



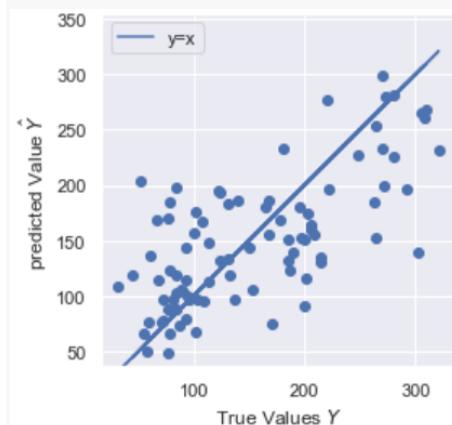
e) Test model

```
res = model.evaluate(X_test, y_test, verbose = 1)
print(f'테스트 세트의 MSE : {round(res[1], 4)}')
```

테스트 세트의 MSE : 3163.21728515625

```
# plot
import matplotlib.pyplot as plt

plt.scatter(y_test, model.predict(X_test))
plt.xlabel("True Values $Y$")
plt.ylabel("predicted Value $\hat{Y}$")
plt.axis("equal")
plt.axis("square")
plt.plot(y_test, y_test, label = "y=x")
plt.legend()
plt.show()
```



```
# R_square
from sklearn.metrics import r2_score
score = r2_score(y_test, model.predict(X_test))
score

0.48403412189639805
```

R^2 값이 hidden layer 를 추가한 후 높아졌다. Model 1에 비해 Model 2의 성능이 좋아졌다고 할 수 있다.

Model 3

c) building model

```
import tensorflow as tf

model = tf.keras.Sequential([
    # input layer
    tf.keras.layers.Dense(256, activation = "relu", input_dim = (len(df.columns)-1)),
    tf.keras.layers.BatchNormalization(),

    # hidden layer
    tf.keras.layers.Dense(128, activation = "relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(64, activation = "relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(32, activation = "relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(16, activation = "relu"),
    tf.keras.layers.BatchNormalization(),

    # output layer
    tf.keras.layers.Dense(1, activation = "linear")
])

# model compile
model.compile(optimizer = "adam", loss = "mse", metrics = [ 'mse' ])

# model summary
model.summary()
```

```
Model: "sequential_2"
-----
```

Layer (type)	Output Shape	Param #
dense_10 (Dense)	(None, 256)	2816
batch_normalization (BatchN ormalization)	(None, 256)	1024
dense_11 (Dense)	(None, 128)	32896
batch_normalization_1 (BatchNormalization)	(None, 128)	512
dense_12 (Dense)	(None, 64)	8256
batch_normalization_2 (BatchNormalization)	(None, 64)	256
dense_13 (Dense)	(None, 32)	2080
batch_normalization_3 (BatchNormalization)	(None, 32)	128
dense_14 (Dense)	(None, 16)	528
batch_normalization_4 (BatchNormalization)	(None, 16)	64
dense_15 (Dense)	(None, 1)	17

```
=====
Total params: 48,577
Trainable params: 47,585
Non-trainable params: 992
```

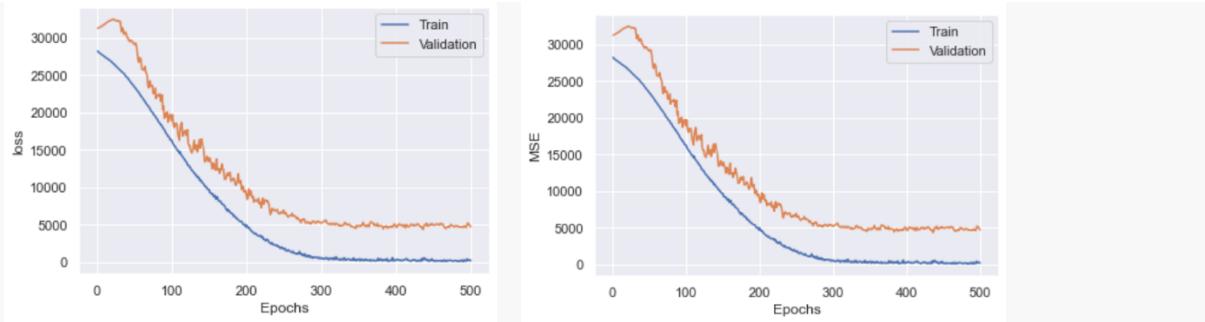
모델의 hidden layer은 4개이며, 각 층의 활성화 함수는 relu이다. optimizer는 adam으로 설정하였으며, 회귀 문제이므로 loss function은 mse로 설정한다. 또한, hidden layer 사이에 Batch-Normalization layer를 추가하여 층을 쌓는다.

d) training model

BN-layer 추가 후 loss와 mse의 감소 속도가 느려졌기 때문에 epoch를 증가시킬 필요가 있다.

```
history = model.fit(X_train, y_train, batch_size = 32,
                     epochs = 500, validation_split = 0.2, verbose = 0)

plot_loss_curve(total_epoch=500, start = 1)
plot_mse_curve(total_epoch=500, start = 1)
```



BN-layer 를 추가한 후, Train 과 validation 의 loss 와 mse 의 변동량의 폭이 커졌다.

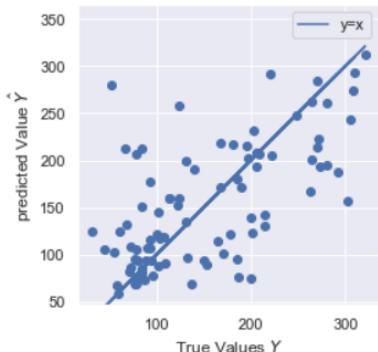
e) Test model

```
res = model.evaluate(X_test, y_test, verbose = 1)
print(f'테스트 세트의 MSE : {round(res[1], 4)}')
```

테스트 세트의 MSE : 3770.05908203125

```
# plot
import matplotlib.pyplot as plt

plt.scatter(y_test, model.predict(X_test))
plt.xlabel("True Values $Y$")
plt.ylabel("predicted Value $\hat{Y}$")
plt.axis("equal")
plt.axis("square")
plt.plot(y_test, y_test, label = "y=x")
plt.legend()
plt.show()
```



```
# R_square
from sklearn.metrics import r2_score
score = r2_score(y_test, model.predict(X_test))
score
```

0.38504956948269753

BN-layer 를 추가한 후 R^2 값이 감소하였다.

Model 4

c) building model

```

import tensorflow as tf

model = tf.keras.Sequential([
    # input Layer
    tf.keras.layers.Dense(256, activation = "relu", input_dim = (len(df.columns)-1),
                          kernel_initializer = "he_normal"),

    # hidden layer
    tf.keras.layers.Dense(128, activation = "relu"),
    tf.keras.layers.Dense(64, activation = "relu"),
    tf.keras.layers.Dense(32, activation = "relu"),
    tf.keras.layers.Dense(16, activation = "relu"),

    # output layer
    tf.keras.layers.Dense(1, activation = "linear")
])

# model compile
model.compile(optimizer = "adam", loss = "mse", metrics = ['mse'])

# model summary
model.summary()

Model: "sequential_3"

-----  

Layer (type)          Output Shape         Param #  

=====
dense_16 (Dense)     (None, 256)          2816  

dense_17 (Dense)     (None, 128)          32896  

dense_18 (Dense)     (None, 64)           8256  

dense_19 (Dense)     (None, 32)            2080  

dense_20 (Dense)     (None, 16)            528  

dense_21 (Dense)     (None, 1)             17  

=====  

Total params: 46,593  

Trainable params: 46,593  

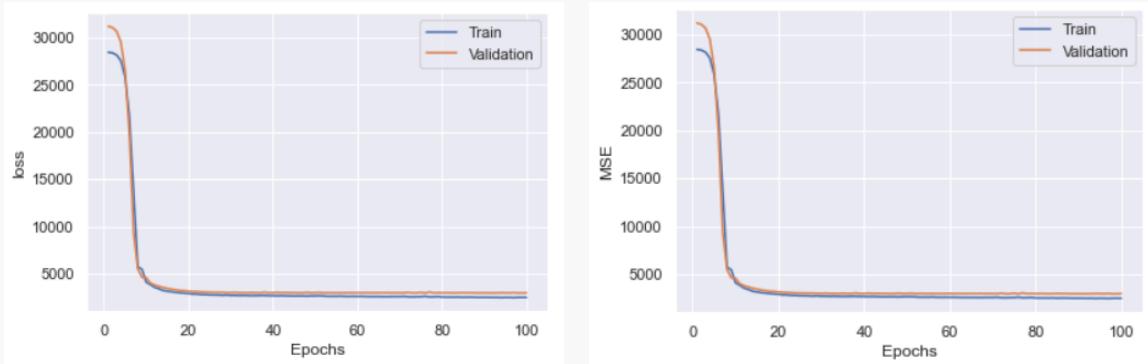
Non-trainable params: 0
-----
```

성능이 가장 좋았던 Model 2 의 input layer에 kernel_initializer = "he_normal" 옵션을 추가한다. 이 옵션은 파라미터 초기화 방법이다.

d) training model

```
history = model.fit(X_train, y_train, batch_size = 32, epochs = 100, validation_split = 0.2,
verbose = 0)
```

```
plot_loss_curve(total_epoch=100, start = 1)
plot_mse_curve(total_epoch=100, start = 1)
```



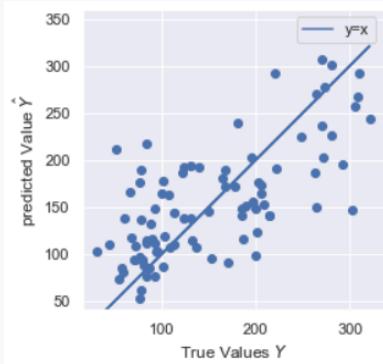
e) test model

```
res = model.evaluate(X_test, y_test, verbose = 1)
print(f'테스트 세트의 MSE : {round(res[1], 4)}')
```

테스트 세트의 MSE : 3187.583251953125

```
# plot
import matplotlib.pyplot as plt

plt.scatter(y_test, model.predict(X_test))
plt.xlabel("True Values $Y$")
plt.ylabel("predicted Value $\hat{Y}$")
plt.axis("equal")
plt.axis("square")
plt.plot(y_test, y_test, label = "y=x")
plt.legend()
plt.show()
```



```
# R_square
from sklearn.metrics import r2_score
score = r2_score(y_test, model.predict(X_test))
score
```

0.48005973062976426

파라미터 초기화 방법을 이용한 후 R^2 값에 큰 변화가 없다.

Model 5

c) building model

```

import tensorflow as tf

model = tf.keras.Sequential([
    # input layer
    tf.keras.layers.Dense(256, activation = "relu", input_dim = (len(df.columns)-1)),
    tf.keras.layers.Dropout(0.2),

    # hidden layer
    tf.keras.layers.Dense(128, activation = "relu"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(64, activation = "relu"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(32, activation = "relu"),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(16, activation = "relu"),
    tf.keras.layers.Dropout(0.2),

    # output Layer
    tf.keras.layers.Dense(1, activation = "linear")
])

# model compile
model.compile(optimizer = "adam", loss = "mse", metrics = ['mse'])

# model summary
model.summary()

Model: "sequential_4"

```

Layer (type)	Output Shape	Param #
<hr/>		
dense_22 (Dense)	(None, 256)	2816
dropout (Dropout)	(None, 256)	0
dense_23 (Dense)	(None, 128)	32896
dropout_1 (Dropout)	(None, 128)	0
dense_24 (Dense)	(None, 64)	8256
dropout_2 (Dropout)	(None, 64)	0
dense_25 (Dense)	(None, 32)	2080
dropout_3 (Dropout)	(None, 32)	0
dense_26 (Dense)	(None, 16)	528
dropout_4 (Dropout)	(None, 16)	0
dense_27 (Dense)	(None, 1)	17

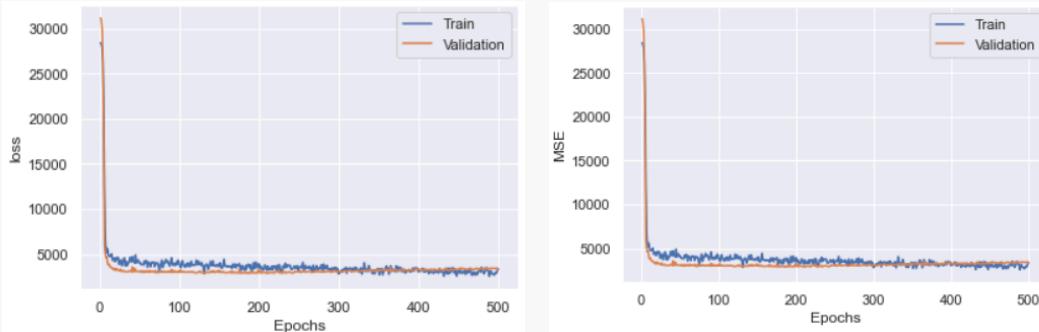
```
=====
Total params: 46,593
Trainable params: 46,593
Non-trainable params: 0
```

hidden layer 사이에 drop out layer 를 추가한다.

d) training model

```
history = model.fit(X_train, y_train, batch_size = 32, epochs = 500, validation_split = 0.2,
                     verbose = 0)
```

```
plot_loss_curve(total_epoch=500, start = 1)
plot_mse_curve(total_epoch=500, start = 1)
```



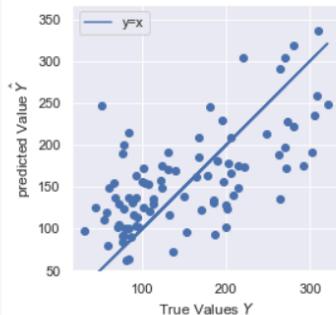
e) test model

```
res = model.evaluate(X_test, y_test, verbose = 1)
print(f'테스트 세트의 MSE : {round(res[1], 4)}')
```

테스트 세트의 MSE : 3649.260498046875

```
# plot
import matplotlib.pyplot as plt

plt.scatter(y_test, model.predict(X_test))
plt.xlabel("True Values $Y$")
plt.ylabel("predicted Value $\hat{Y}$")
plt.axis("equal")
plt.axis("square")
plt.plot(y_test, y_test, label = "y=x")
plt.legend()
plt.show()
```



```
# R_square
from sklearn.metrics import r2_score
score = r2_score(y_test, model.predict(X_test))
score

0.4047535516366586
```

drop out layer 를 추가한 후 모델의 성능이 낮아졌다.

cf. Table of result

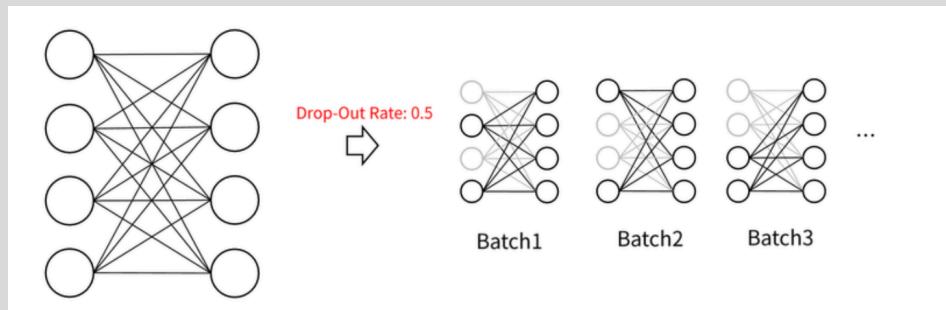
Parameter	Number of hidden layer	BN	initialization	R^2
Model 1	2	X	X	0.4584
Model 2	4	X	X	0.4789
Model 3	4	O	X	0.3685
Model 4	4	X	O	0.4787
Model 5	4	X	X	0.3898

Model 2의 성능이 가장 좋다.

cf. Dropout

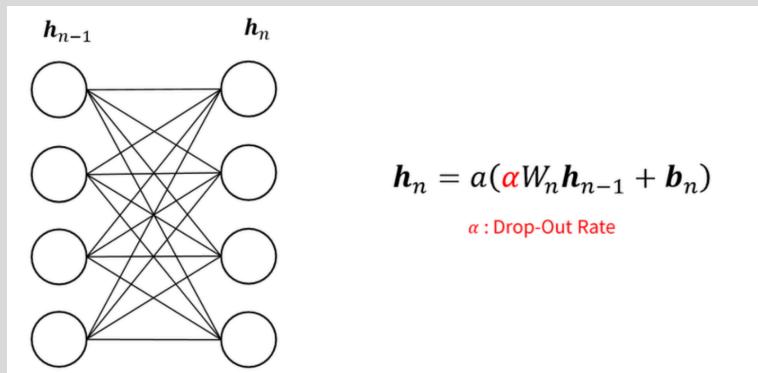
Dropout은 MLP에서 가장 인기있는 규제 기법 중 하나이다. 이 방법은 2012년에 Geoffrey E. Hinton이 제안했고, Nitish Srivastava의 2014년 논문에서 잘 작된다고 입증되었다. 최고 성능을 내는 신경망에서도 dropout을 적용함으로써 정확도를 1~2% 높였다. 이 수치가 크게 느껴지지 않을 수 있지만 모델의 정확도가 95%일 때 2% 상승하는 것은 오차율이 거의 40% 정도 줄어드는 것을 의미한다.

이 알고리즘은 간단하다. 매 훈련 스텝에서 각 뉴런(입력 뉴런은 포함하지만 출력 뉴런은 포함하지 않는다)은 임시적으로 dropout될 확률 p 를 가진다. 즉, 이번 훈련 스텝에는 완전히 무시되지만 다음 스텝에는 활성화될 수 있다. 하이퍼파라미터 p 를 dropout rate라고 하며 일반적으로 0.1 ~ 0.5 사이의 값을 지정한다. RNN에서는 0.2~0.3에 가깝고 CNN에서는 0.4 ~ 0.5에 가깝다.



dropout은 일부 feature가 overfitting되는 문제를 막는 Ensemble 효과를 주기도 한다. 각 훈련 스텝에서 고유한 네트워크가 생성된다고 생각했을 때, 각각의 뉴런이 사용되거나 혹은 사용되지 않을 수 있는 2가지는 경우의 수가 존재하므로 2^N 개의 네트워크가 가능하다.(N 은 dropout이 가능한 뉴런의 수) 이는 아주 큰 값이기 때문에 같은 네트워크가 두 번 선택될 가능성은 거의 없다. 10000번의 훈련 스텝을 진행하면 10000개의 다른 신경망을 훈련하게 된다. 이 신경망은 대부분의 가중치를 공유하고 있기 때문에 아주 독립적이지는 않지만, 그럼에도 모두 다르다. 결과적으로 만들어진 신경망은 이 모든 신경망을 평균한 양상으로 볼 수 있는 것이다.

마지막으로 결과를 출력할 때는 dropout rate를 적용한다.



② Regression Example : Boston-Housing(built-in dataset)

회귀분석에 사용될 데이터셋은 Boston Housing 1970 데이터의 일부 변수를 추출한 데이터이며 Sklearn에 내장되어 있다. 미국 매사추세츠주 92개 도시(TOWN)에 대한 506개의 데이터로 주택 가격 및 기타 지역 특성이 포함되어 있다.

feature	
1. CRIM	지역 범죄율(per capita crime)
2. ZN	25000 제곱 피트 이상의 주택지 비율
3. INDUS	상업적 비즈니스에 활용되지 않는 농지 면적
4. CHAS	Charles 강과 접하고 있는지 여부 (dummy variable, 1: 강의 경계에 위치한 경우, 0: 강의 경계에 위치하지 않은 경우)
5. NOX	일산화질소 농도
6. RM	주택당 평균 방 갯수
7. AGE	1940년 이전에 건설된 주택의 비율
8. DIS	5개 보스턴 고용 센터와의 거리에 따른 가중치 거리
9. RAD	Radial 고속도로(방사형 고속도로)와의 접근성 지수
10. TAX	10000달러당 재산세율
11. PTRATIO	학생-교사의 비율
12. B	흑인 지수 $1000(B_K - 0.63)^2$, B_K 는 흑인의 비율
13. LSTAT	빈곤층 비율

target	
14. MEDV	1978년 보스턴 주택 가격으로 타운의 주택 가격 중앙값(단위 1000달러)

a) import the dataset

```
# import package
import random

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers import Dense

# 랜덤 시드 고정
SEED = 22
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

```
# import dataset
from sklearn import datasets
boston_housing = datasets.load_boston()
```

b) data preprocessing

```
df = pd.DataFrame(boston_housing.data, columns = boston_housing.feature_names)
df['MEDV'] = boston_housing.target
df.head(3)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   CRIM        506 non-null    float64
 1   ZN          506 non-null    float64
 2   INDUS       506 non-null    float64
 3   CHAS         506 non-null    float64
 4   NOX          506 non-null    float64
 5   RM           506 non-null    float64
 6   AGE          506 non-null    float64
 7   DIS           506 non-null    float64
 8   RAD           506 non-null    float64
 9   TAX           506 non-null    float64
 10  PTRATIO      506 non-null    float64
 11  B             506 non-null    float64
 12  LSTAT        506 non-null    float64
 13  MEDV         506 non-null    float64
dtypes: float64(14)
memory usage: 55.5 KB
```

```
df['CHAS'].value_counts()

0.0    471
1.0     35
Name: CHAS, dtype: int64
```

boston data에서 CHAS는 범주형 변수이므로 원-핫 인코딩이 필요하다.

```
# OneHotEncoder
df = pd.get_dummies(data = df, columns = ['CHAS'], prefix = ['CHAS'])
df.head(3)
```

	CRIM	ZN	INDUS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV	CHAS_0.0	CHAS_1.0
0	0.00632	18.0	2.31	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0	1	0
1	0.02731	0.0	7.07	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6	1	0
2	0.02729	0.0	7.07	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7	1	0

모델에 학습을 시키기 위해 데이터를 훈련용 데이터와 테스트 데이터로 분할한다.

```
# df 의 column 순서 변경
df = df[['CRIM', 'ZN', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS',
          'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'CHAS_0.0', 'CHAS_1.0', 'MEDV']]
```

```
X = df.iloc[:, :-1]
Y = df.iloc[:, -1]
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
                                                    shuffle=True, random_state = SEED)
```

```
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(404, 14) (404,)
(102, 14) (102,)
```

```
# feature scaling
from sklearn import preprocessing
```

```
# normalization
minmax_scaler = preprocessing.MinMaxScaler()
norm_fit = minmax_scaler.fit(X_train)
X_train_norm = norm_fit.transform(X_train)
X_test_norm = norm_fit.transform(X_test)
```

```
# standardization
standard_scaler = preprocessing.StandardScaler()
stan_fit = standard_scaler.fit(X_train)
X_train_stan = stan_fit.transform(X_train)
X_test_stan = stan_fit.transform(X_test)
```

test data를 정규화하는 경우 train data의 통계량을 이용한다.

c) building model

회귀분석에서 은닉층의 활성화 함수는 일반적으로 ReLU로 정의한다.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

```
def build_model(num_input=1):
    model = Sequential()

    # hidden layer
    model.add(Dense(128, activation='relu', input_dim=num_input))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(16, activation='relu'))

    # output layer
    model.add(Dense(1, activation='linear'))
```

```
# model compile
```

```

    model.compile(optimizer='adam', loss='mse', metrics=['mse'])

    return model

model = build_model(num_input=14) # feature 의 개수 = 14

```

summary 메서드를 사용해 모델에 대한 간단한 정보를 출력할 수 있다.

```

model.summary()

Model: "sequential_5"
=====
Layer (type)                 Output Shape              Param #
=====
dense_28 (Dense)            (None, 128)             1920
dense_29 (Dense)            (None, 64)              8256
dense_30 (Dense)            (None, 32)              2080
dense_31 (Dense)            (None, 16)              528
dense_32 (Dense)            (None, 1)               17
=====
Total params: 12,801
Trainable params: 12,801
Non-trainable params: 0

```

d) training model & test model

d-1) Min-max normalization

모델의 훈련은 200번의 에포크(epoch)로 지정하며, 배치사이즈를 32로 지정해 미니배치로 학습을 시킨다.

```

EPOCHS = 200

history = model.fit(X_train_norm, y_train, batch_size = 32,
                     epochs=EPOCHS, validation_split = 0.2, verbose=2)

```

history에 저장된 통계치를 사용해 모델의 훈련과정을 데이터 프레임으로 시각화 할 수 있다.

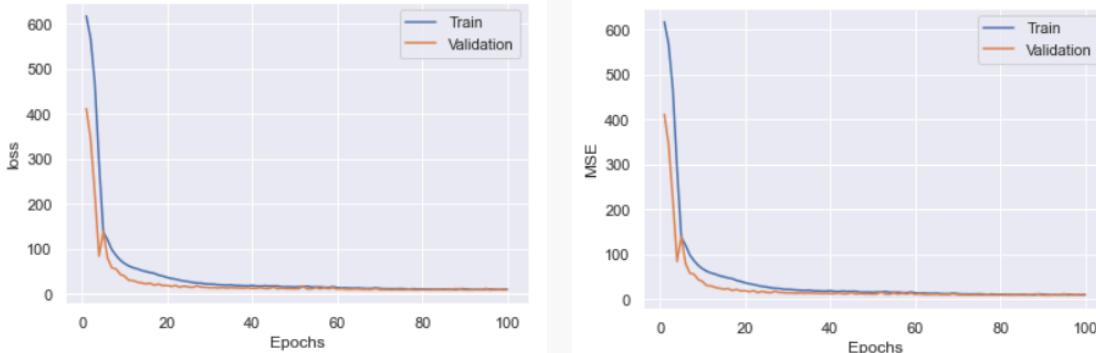
```

hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()

      loss      mse   val_loss   val_mse  epoch
195  7.136828  7.136828  10.594507  10.594507    195
196  6.722535  6.722535  10.506862  10.506862    196
197  7.602661  7.602661  10.856474  10.856474    197
198  7.364977  7.364977  10.514135  10.514135    198
199  6.581415  6.581415  9.396604  9.396604    199

```

```
plot_loss_curve(total_epoch=100, start=1)
plot_mse_curve(total_epoch=100, start=1)
```



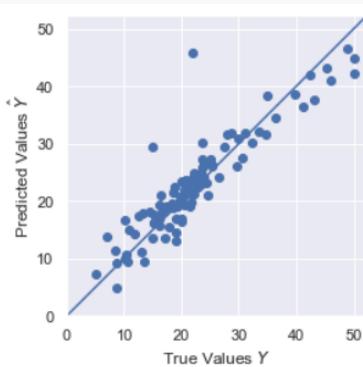
```
res = model.evaluate(X_test_norm, y_test, verbose=2) # Loss = res[0], mse = res[1]
print(f"테스트 세트의 MSE: {round(res[1],3)*1000}$$")
```

테스트 세트의 MSE: 17572.00050354004\$

모델의 성능을 확인하기 위해 test data를 사용한 후, plot과 R^2 를 이용하여 판단한다.

```
test_predictions = model.predict(X_test_norm).flatten()

plt.scatter(y_test, test_predictions)
plt.xlabel('True Values $Y$ ')
plt.ylabel('Predicted Values $\hat{Y}$ ')
plt.axis('equal')
plt.axis('square')
plt.xlim([0,plt.xlim()[1]])
plt.ylim([0,plt.ylim()[1]])
_ = plt.plot([-100, 100], [-100, 100])
```



예측값과 실제값이 거의 선형관계를 이루고 있다. 다음으로 R^2 를 이용하여 y 의 변동량을 얼마큼 설명하고 있는지 확인한다.

```
from sklearn.metrics import r2_score
r2_y_predict = r2_score(y_test, test_predictions)
r2_y_predict
```

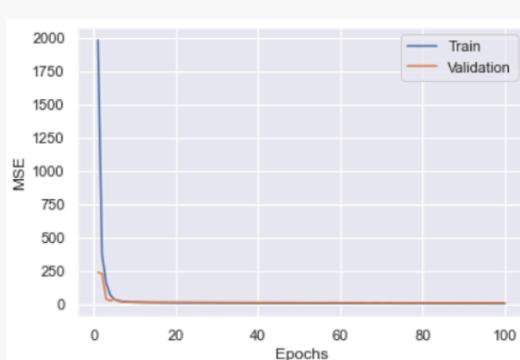
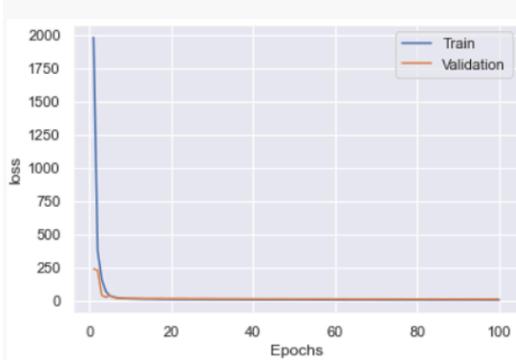
0.8018204183769247

d-2) Standardization

```
EPOCHS = 200

history = model.fit(X_train_stan, y_train, batch_size = 32,
                     epochs=EPOCHS, validation_split = 0.2, verbose=0)

plot_loss_curve(total_epoch=100, start=1)
plot_mse_curve(total_epoch=100, start=1)
```



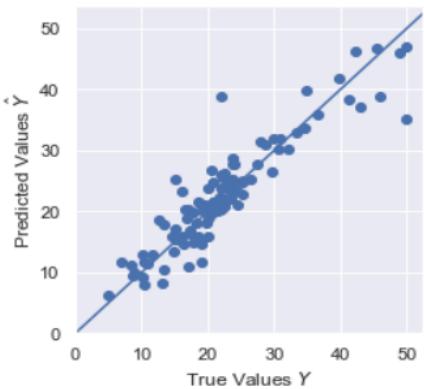
```
res = model.evaluate(X_test_stan, y_test, verbose=2) # Loss = res[0], mse = res[1]
print(f"테스트 세트의 MSE: {round(res[1],3)*1000}$$")
```

테스트 세트의 MSE: 15385.99967956543\$

모델의 성능을 확인하기 위해 test data를 사용한 후, plot과 R^2 를 이용하여 판단한다.

```
test_predictions = model.predict(X_test_stan).flatten()

plt.scatter(y_test, test_predictions)
plt.xlabel('True Values $Y$ ')
plt.ylabel('Predicted Values $\hat{Y}$ ')
plt.axis('equal')
plt.axis('square')
plt.xlim([0,plt.xlim()[1]])
plt.ylim([0,plt.ylim()[1]])
_ = plt.plot([-100, 100], [-100, 100])
```



```
from sklearn.metrics import r2_score
r2_y_predict = r2_score(y_test, test_predictions)
r2_y_predict
```

0.8264778561768521

③ Regression Example : MPG

Auto MPG data set에는 1970년대 후반부터 1980년대 초반의 자동차에 대한 정보(feature)와 연비(target)이 들어 있다. 총 398개의 데이터가 들어있으며 feature의 개수는 7개이다.

target	
1. mpg	Continuous
feature	
2. cylinder	Multi-valued discrete
3. displacement	Continuous
4. horsepower	Continuous
5. weight	Continuous
6. acceleration	Continuous
7. model year	Multi-valued discrete
8. origin	Multi-valued discrete

NOTE)

데이터 주소 : <http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data>

a) import dataset

```
# import package
import pathlib
import random

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
sns.set_theme(color_codes = True)

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers import Dense

# 랜덤 시드 고정
SEED = 22
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

Auto MPG 데이터 셋은 UCI에서 다운로드 할 수 있다.

```
# keras.utils.get_file("파일명", "인터넷 상 파일경로")는 컴퓨터에 "파일명"에 해당하는 파일의 경로를
불러들인다.
dataset_path = keras.utils.get_file("auto-mpg.data", "http://archive.ics.uci.edu/ml/machine-
learning-databases/auto-mpg/auto-mpg.data")
dataset_path
```

b) data preprocessing

pandas를 사용하여 데이터를 데이터 프레임 형태로 읽는다.

```
# 변수명
column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
                 'Acceleration', 'Model Year', 'Origin']

# 데이터 로드
mydata = pd.read_csv(dataset_path, names=column_names, na_values = "?", comment='\t',
                     sep=" ", skipinitialspace=True)

mydata.head()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
0	18.0	8	307.0	130.0	3504.0	12.0	70	1
1	15.0	8	350.0	165.0	3693.0	11.5	70	1
2	18.0	8	318.0	150.0	3436.0	11.0	70	1
3	16.0	8	304.0	150.0	3433.0	12.0	70	1
4	17.0	8	302.0	140.0	3449.0	10.5	70	1

데이터셋에 누락된 값이 없는지 확인한다.

```
mydata.info()

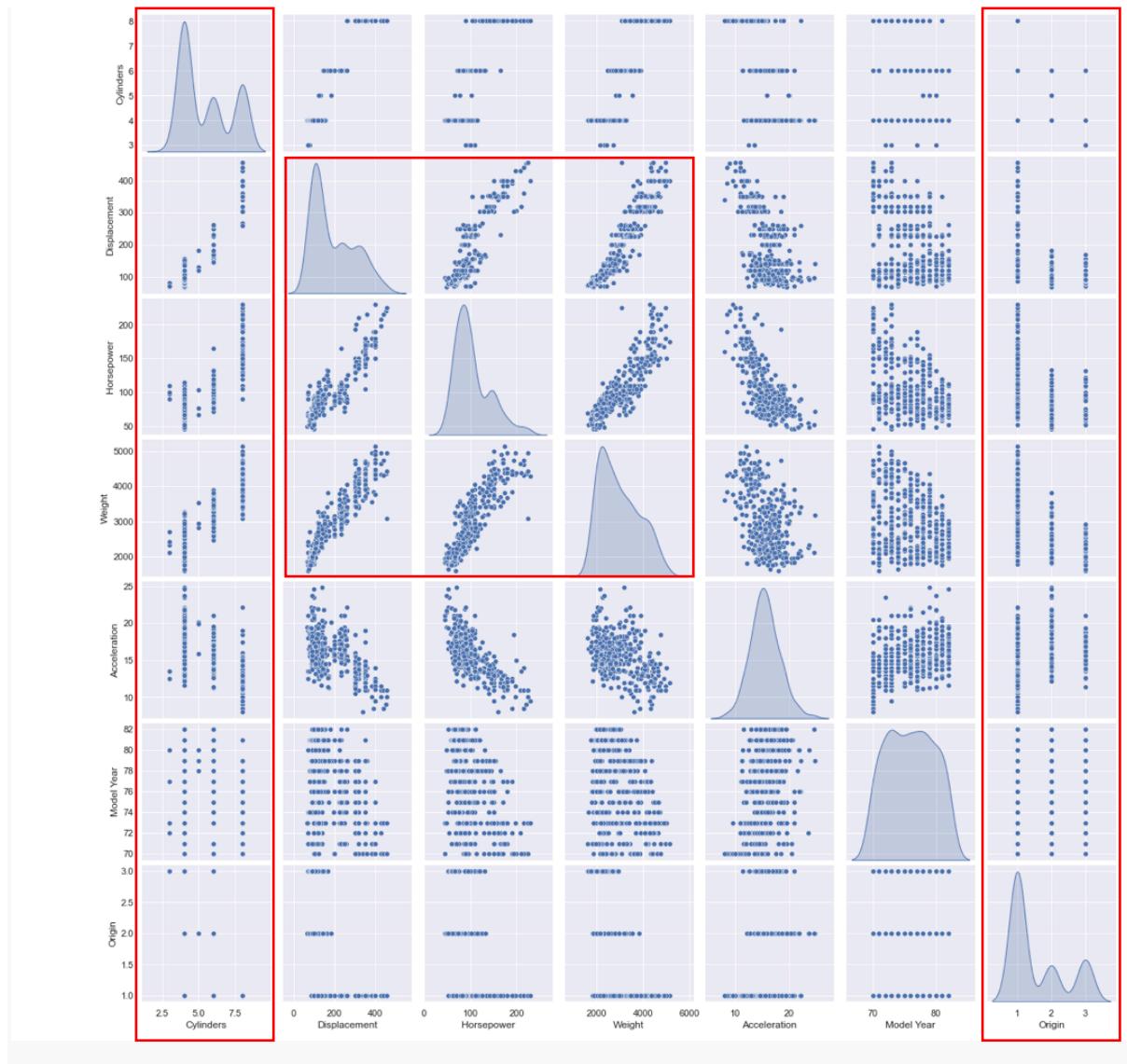
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
--- 
 0   MPG          398 non-null    float64
 1   Cylinders    398 non-null    int64  
 2   Displacement 398 non-null    float64
 3   Horsepower   392 non-null    float64
 4   Weight        398 non-null    float64
 5   Acceleration 398 non-null    float64
 6   Model Year   398 non-null    int64  
 7   Origin        398 non-null    int64  
dtypes: float64(5), int64(3)
memory usage: 25.0 KB
```

누락된 행을 삭제한다.

```
mydata = mydata.dropna()
```

모델 학습으로 들어가기 이전에 데이터에 대해 간단히 살펴보기 위해 산점도와 특성별 통계량을 구해본다.

```
# pair plot
sns.pairplot(mydata[['Cylinders', 'Displacement', 'Horsepower', 'Weight',
                      'Acceleration', 'Model Year', 'Origin']], diag_kind="kde")
plt.show()
```



Cylinders와 Origin은 범주형 변수이므로 원-핫 인코딩이 필요하다. 또한, Weight과 Horsepower, Displacement 사이에 선형성이 관찰된다.

```
# OneHotencoder
mydata = pd.get_dummies(data = mydata, columns = ['Cylinders', 'Origin'], prefix = ['Cylinders', 'Origin'])
```

데이터를 훈련 데이터와 테스트 데이터로 분리한다.

```
X = mydata.iloc[:, 1:]
Y = mydata.iloc[:, 0]

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
                                                    shuffle=True, random_state = SEED)

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)

(313, 13) (313,)
(79, 13) (79,)
```

다음으로 feature들을 scaling한다. feature의 스케일과 범위가 다르기 때문에 feature scaling 과정이 권장된다. feature를 정규화하지 않고 모델이 수렴할 수 있지만, 훈련시키기 어렵고 입력 단위에 의존적인 모델이 만들어지기 때문이다. 이때, 훈련된 모델에 테스트 데이터를 입력하기 전에 테스트 데이터 또한 동일하게 정규화를 시킨 뒤 입력해야하며 test data의 scaling에 사용되는 통계량은 train data의 통계량이어야 한다.

```
# feature scaling
from sklearn import preprocessing

# normalization
minmax_scaler = preprocessing.MinMaxScaler()
norm_fit = minmax_scaler.fit(X_train)
X_train_norm = norm_fit.transform(X_train)
X_test_norm = norm_fit.transform(X_test)

# standardization
standard_scaler = preprocessing.StandardScaler()
stan_fit = standard_scaler.fit(X_train)
X_train_stan = stan_fit.transform(X_train)
X_test_stan = stan_fit.transform(X_test)
```

c) building model

두 개의 완전 연결 은닉층으로 Sequential 모델을 만든다. 회귀분석 모델이므로 출력층은 numeric값을 반환한다. 또한 모델을 반복적으로 사용하는 경우를 위해 모델을 함수로 정의한다. 학습에 사용되는 데이터의 개수가 적기 때문에 층을 깊게 만들지 않고 두개만 만들도록 한다.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = tf.keras.Sequential([
    # hidden layer
    tf.keras.layers.Dense(64, activation = 'relu', input_shape = (13,)),
    tf.keras.layers.Dense(64, activation = 'relu'),
    # output layer
    tf.keras.layers.Dense(1, activation = 'linear')
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(0.001), loss = 'mse',
              metrics = ['mae', 'mse'])
```

summary 메서드를 사용해 모델에 대한 간단한 정보를 출력할 수 있다.

```

model.summary()

Model: "sequential_6"
=====
Layer (type)          Output Shape         Param #
=====
dense_33 (Dense)     (None, 64)           896
dense_34 (Dense)     (None, 64)           4160
dense_35 (Dense)     (None, 1)            65
=====
Total params: 5,121
Trainable params: 5,121
Non-trainable params: 0
=====
```

d) training model & test model

d-1) Min-max normalization

모델을 훈련은 1000번의 에포크(epoch)로 지정한다. 훈련 정확도와 검증 정확도는 history 객체에 저장된다.

```

history = model.fit(X_train_norm, y_train, epochs = 1000, validation_split = 0.2, verbose = 0)

import matplotlib.pyplot as plt

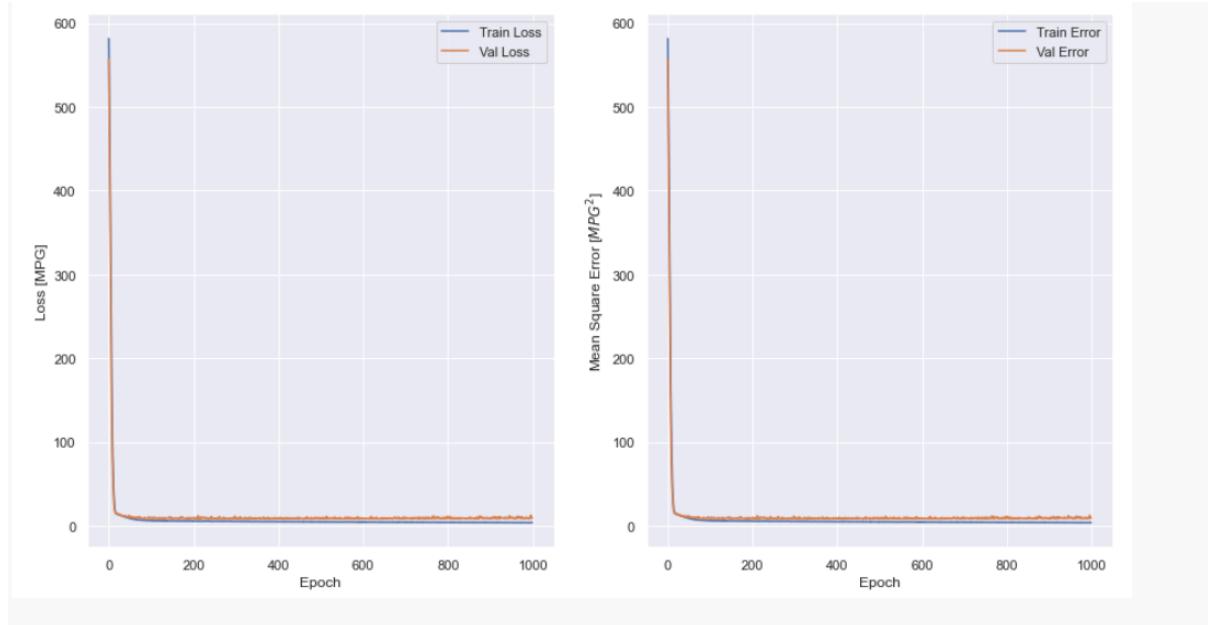
def plot_history(history):
    hist = pd.DataFrame(history.history)
    hist['epoch'] = history.epoch

    plt.figure(figsize=(15,8))

    plt.subplot(1,2,1)
    plt.xlabel('Epoch')
    plt.ylabel('Loss [MPG]')
    plt.plot(hist['epoch'], hist['loss'], label='Train Loss')
    plt.plot(hist['epoch'], hist['val_loss'], label = 'Val Loss')
    plt.legend()

    plt.subplot(1,2,2)
    plt.xlabel('Epoch')
    plt.ylabel('Mean Square Error [$MPG^2$]')
    plt.plot(hist['epoch'], hist['mean_squared_error'], label='Train Error')
    plt.plot(hist['epoch'], hist['val_mean_squared_error'], label = 'Val Error')
    plt.legend()
    plt.show()

plot_history(history)
```

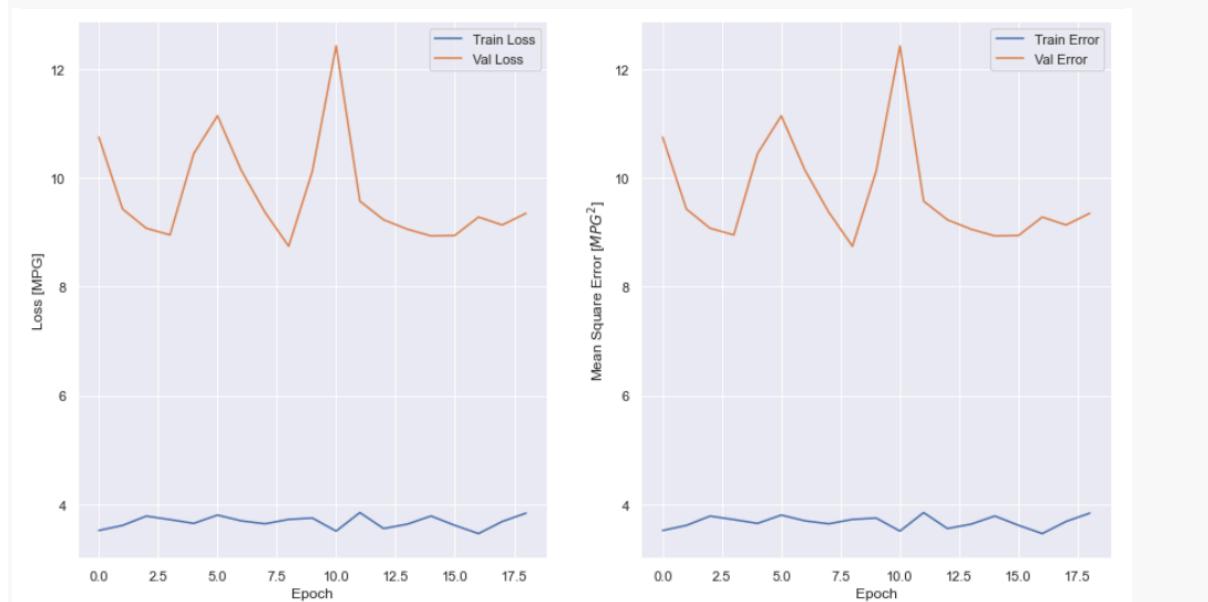


그래프에서 epoch가 수백번 진행한 후에는 모델의 성능이 거의 향상되지 않는 것을 볼 수 있다. `model.fit` 메서드를 수정하여 검증 점수가 향상되지 않는 경우 자동으로 훈련을 멈추도록 할 수 있다. 에포크마다 훈련 상태를 점검하기 위해 **EarlyStopping** 콜백(callback)을 사용한다. 이 옵션은 지정된 에포크 횟수동안 성능 향상이 없으면 자동으로 훈련이 멈춘다.

```
# patience 매개변수는 성능 향상을 체크할 에포크 횟수
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)

history = model.fit(X_train_norm, y_train, epochs=1000,
                     validation_split = 0.2, verbose=0, callbacks=[early_stop])

plot_history(history)
```



epoch가 18번 정도 진행된 후 학습은 중단하였다. 모델의 성능을 확인하기 위해 테스트 세트를 사용한다.

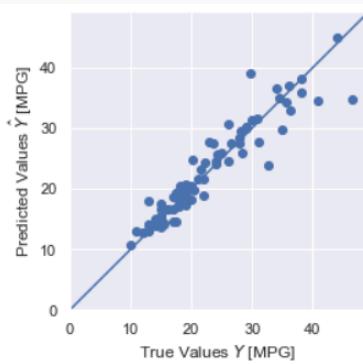
```
loss, mae, mse = model.evaluate(X_train_norm, y_train, verbose=2)
print(f"테스트 세트의 MSE: {round(mse,3)} MPG")
```

테스트 세트의 MSE: 4.5970001220703125 MPG

마지막으로 테스트 세트에 있는 샘플을 사용하여 MPG 값을 예측한다.

```
test_predictions = model.predict(X_test_norm).flatten()

plt.scatter(y_test, test_predictions)
plt.xlabel('True Values $Y$ [MPG]')
plt.ylabel('Predicted Values $\hat{Y}$ [MPG]')
plt.axis('equal')
plt.axis('square')
plt.xlim([0,plt.xlim()[1]])
plt.ylim([0,plt.ylim()[1]])
_ = plt.plot([-100, 100], [-100, 100])
```



```
from sklearn.metrics import r2_score
r2_y_predict = r2_score(y_test, test_predictions)
r2_y_predict
```

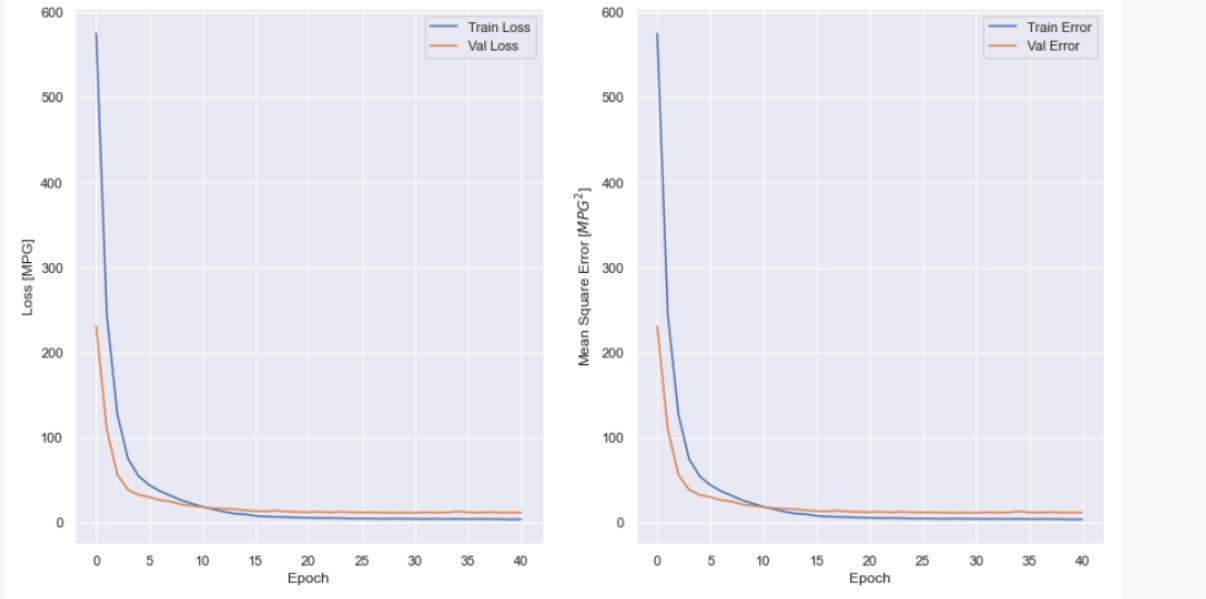
0.8863605478622524

d-2) standardization

```
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)

history = model.fit(X_train_stan, y_train, epochs=1000,
                     validation_split = 0.2, verbose=0, callbacks=[early_stop])

plot_history(history)
```

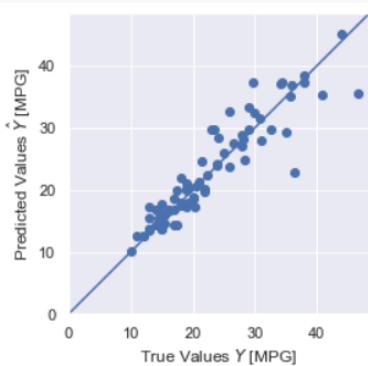


```
loss, mae, mse = model.evaluate(X_train_stan, y_train, verbose=2)
print(f"테스트 세트의 MSE: {round(mse,3)} MPG")
```

테스트 세트의 MSE: 5.189000129699707 MPG

```
test_predictions = model.predict(X_test_stan).flatten()

plt.scatter(y_test, test_predictions)
plt.xlabel('True Values $Y$ [MPG]')
plt.ylabel('Predicted Values $\hat{Y}$ [MPG]')
plt.axis('equal')
plt.axis('square')
plt.xlim([0,plt.xlim()[1]])
plt.ylim([0,plt.ylim()[1]])
_ = plt.plot([-100, 100], [-100, 100])
```



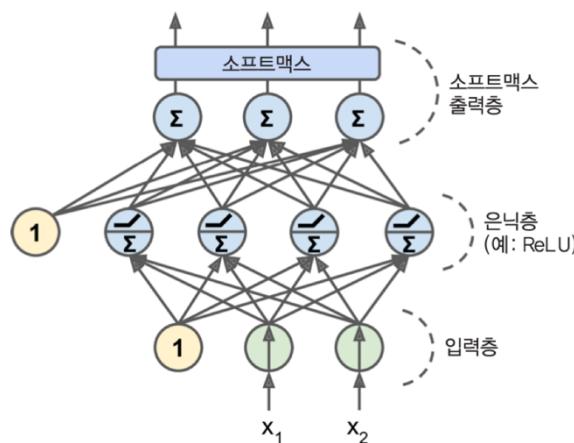
```
from sklearn.metrics import r2_score
r2_y_predict = r2_score(y_test, test_predictions)
r2_y_predict
```

0.7638210367313423

3) Classification

MLP는 분류 작업에서도 사용가능하다. 이진 분류 문제의 경우 로지스틱 활성화 함수를 가진 하나의 출력 뉴런만 필요하다. 출력은 0과 1 사이의 실수이다. 이를 양성 클래스(positive class)에 대한 예측 확률로 해석할 수도 있다. 음성 클래스(negative class)에 대한 예측 확률은 1에서 양성 클래스의 예측 확률을 뺀 값이다.

MLP를 이용하여 다중 분류 작업을 하는 경우도 생각해보자. 각 샘플이 3개 이상의 클래스 중 한 클래스에만 속할 수 있다면 (예를 들어 숫자 이미지 분류에서 클래스 0에서 9까지) 클래스마다 하나의 출력 뉴런이 필요하다. 출력층에는 소프트맥스 활성화 함수를 사용해야한다. 소프트맥스 함수는 모든 예측 확률을 0과 1 사이로 만들어 더했을 때 1이 되도록 만든다.(클래스가 서로 배타적이여야 함)



확률 분포를 예측해야 하므로 손실 함수에는 일반적으로 크로스 엔트로피(cross-entropy)를 손실함수로 선택하는 것이 좋다.

Hyperparameter	이진 분류	다중 분류
입력층과 은닉층	회귀와 동일	회귀와 동일
출력 뉴런 수	1개	클래스마다 1개
출력층의 활성화 함수	logistic	Softmax
손실함수	Cross-entropy	Cross-entropy

학습한 모델의 성능을 판단하기 위해 분류 문제에서는 항상 confusion matrix를 이용한다. 이때, binary class의 경우 confusion matrix가 다음과 같은 형태로 나온다.

		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

① Classification Example : Wine(Binary data)

target	
1. type	Red wine = 0, white wine = 1
feature	
2. fixed acidity	Continuous
3. volatile acidity	Continuous
4. citric acid	Continuous
5. residual sugar	Continuous
6. chlorides	Continuous
7. free sulfur dioxide	Continuous
8. total sulfur dioxide	Continuous
9. density	Continuous
10. pH	Continuous
11. sulphates	Continuous
12. alcohol	Continuous
13. quality	Categorical
14. type	Continuous

주어진 데이터의 feature는 와인에 대한 정보이며, target은 레드와인과 화이트와인로 분류되어 있다.

a) import dataset

```
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
import random
sns.set_theme(color_codes = True)

SEED = 22
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

UCI에서 제공하는 데이터를 이용한다. 데이터는 pandas를 이용하여 data frame 형태로 불러온다.

```
red = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-
quality/winequality-red.csv', sep=';')
white = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-
quality/winequality-white.csv', sep=';')
```

NOTE) 데이터 주소 :

<http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv>

<http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv>

b) data preprocessing

데이터는 red, white 두개를 불러오게 되는데 모델에 넣기 위해선 이 두개의 데이터를 하나의 데이터로 병합해야한다.

```
# 새로운 변수로 'type'을 설정 후 red -> 0, white -> 1로 할당
red['type'] = 0
white['type'] = 1

# 데이터 병합
wine = pd.concat([red, white])
wine_copied = wine.copy()
wine.head(3)

fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  free sulfur dioxide  total sulfur dioxide  density  pH  sulphates  alcohol  quality  type
0            7.4            0.70       0.00          1.9      0.076           11.0             34.0    0.9978   3.51      0.56     9.4      5      0
1            7.8            0.88       0.00          2.6      0.098           25.0             67.0    0.9968   3.20      0.68     9.8      5      0
2            7.8            0.76       0.04          2.3      0.092           15.0             54.0    0.9970   3.26      0.65     9.8      5      0

print(wine.shape)

(6497, 13)
```

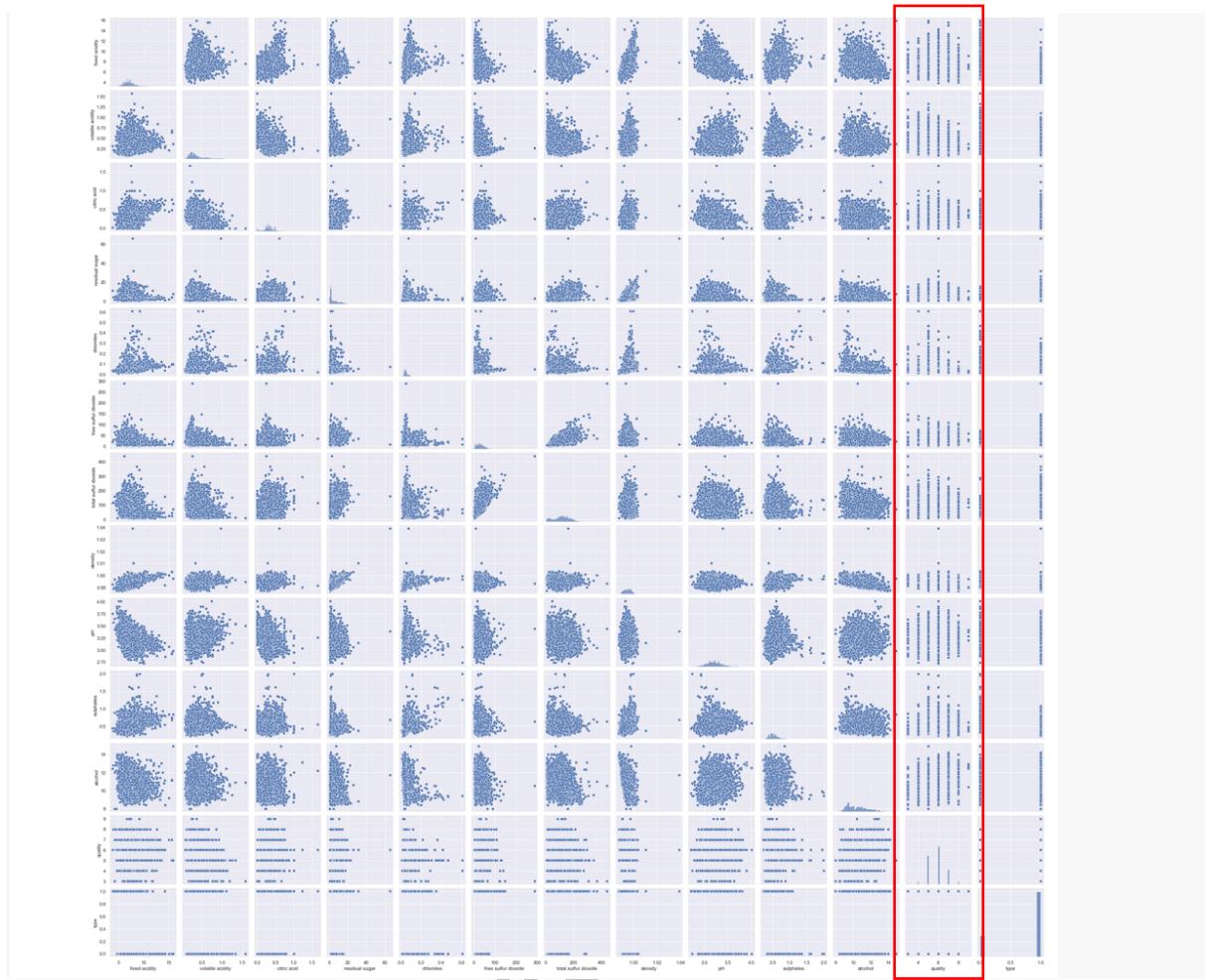
데이터의 개수는 6497이며 feature의 개수는 12개이다. 다음으로 데이터에 결측치가 없는지 확인한다.

```
wine.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 6497 entries, 0 to 4897
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   fixed acidity      6497 non-null   float64
 1   volatile acidity   6497 non-null   float64
 2   citric acid        6497 non-null   float64
 3   residual sugar     6497 non-null   float64
 4   chlorides          6497 non-null   float64
 5   free sulfur dioxide 6497 non-null   float64
 6   total sulfur dioxide 6497 non-null   float64
 7   density            6497 non-null   float64
 8   pH                 6497 non-null   float64
 9   sulphates          6497 non-null   float64
 10  alcohol            6497 non-null   float64
 11  quality            6497 non-null   int64  
 12  type               6497 non-null   int64  
dtypes: float64(11), int64(2)
memory usage: 710.6 KB
```

결측치는 존재하지 않는다.

```
sns.pairplot(wine)
plt.show()
```



pair plot에서 quality는 범주형 변수임을 알 수 있다. 따라서, 원-핫 인코딩 과정이 필요하다.

```
# quality => 3, 4, 5, 6, 7, 8, 9
wine['quality'].value_counts()

6    2836
5    2138
7    1079
4     216
8     193
3      30
9       5
Name: quality, dtype: int64

# OneHotEncoder
wine = pd.get_dummies(data = wine, columns = ['quality', 'type'], prefix = ['quality', 'type'])
```

다음으로 히스토그램을 이용하여 레드 와인(type=0)과 화이트 와인(type=1)의 개수를 확인한다.

```
plt.hist(wine_copied['type'])
plt.xticks([0, 1])
plt.show()
```



```
wine_copied['type'].value_counts()

1    4898
0    1599
Name: type, dtype: int64
```

red와 white의 비율이 1:3 정도로 화이트 와인의 개수가 많다. 학습을 위해 훈련용 데이터와 테스트 데이터로 분리한다.

```
# train/test split
X = wine.iloc[:, :-2]
y = wine.iloc[:, -2:]

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    shuffle = True, random_state = SEED)
```

다음으로 실수형 feature에 대하여 scaling을 한다. normalization과 standardization을 모두 적용시킨 후 각각의 데이터로 훈련시킨 모델들의 성능을 비교해보도록 한다.

```
# feature scaling
from sklearn import preprocessing

# normalization
minmax_scaler = preprocessing.MinMaxScaler()
norm_fit = minmax_scaler.fit(X_train)
X_train_norm = norm_fit.transform(X_train)
X_test_norm = norm_fit.transform(X_test)

# standardization
standard_scaler = preprocessing.StandardScaler()
stan_fit = standard_scaler.fit(X_train)
X_train_stan = stan_fit.transform(X_train)
X_test_stan = stan_fit.transform(X_test)
```

NOTE)

Min-Max Normalization(최소-최대 정규화)는 모델에 투입될 모든 데이터 중에서 가장 작은 값을 0, 가장 큰 값으로 1로 설정 후 나머지 값들은 비율을 맞춰서 0과 1사이 값으로 스케일링 해주는 것이다.

c) building model

분류 모델은 마지막 계층의 활성화 함수로 소프트맥스(softmax) 함수를 사용한다. (이런 분류 문제에서

logistic과 softmax는 동일하다) 소프트맥스 함수의 출력값의 합은 1.0이기 때문에 분류(확률 계산)에 있어서 유리하기 때문이다.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = tf.keras.Sequential([
    # hidden layer
    tf.keras.layers.Dense(units=48, activation='relu', input_shape=(18,)), # input_shape = number of feature
    tf.keras.layers.Dense(units=24, activation='relu'),
    tf.keras.layers.Dense(units=12, activation='relu'),

    # output layer
    tf.keras.layers.Dense(units=2, activation='softmax')
])

# model compile
# 분류 문제의 경우 손실함수는 categorical_crossentropy
# 분류 문제는 정확도로 평가하기 때문에 metrics=['accuracy']를 반드시 설정
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.05),
              loss='categorical_crossentropy', metrics=['accuracy'])

model.summary()

Model: "sequential_7"

Layer (type)          Output Shape         Param #
=====
dense_36 (Dense)      (None, 48)           912
dense_37 (Dense)      (None, 24)            1176
dense_38 (Dense)      (None, 12)             300
dense_39 (Dense)      (None, 2)              26
=====
Total params: 2,414
Trainable params: 2,414
Non-trainable params: 0
```

d) training model & Test model

훈련 데이터 중 25%를 검증 데이터로 분리 후 학습을 진행한다.

d-1) MinMax normalization data

```
# wine_norm
history = model.fit(X_train_norm, y_train, epochs=25, batch_size=32, validation_split=0.25,
verbose = 0)
```

```

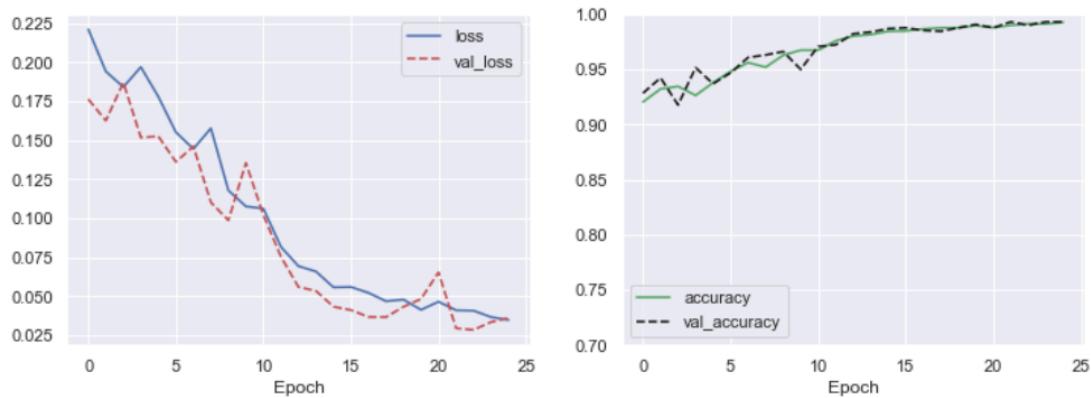
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['acc'], 'g-', label='accuracy')
plt.plot(history.history['val_acc'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()

```



```

loss, acc = model.evaluate(X_test_norm, y_test)
acc

0.9969231

```

정확도가 99%로 높은 정답률을 보여주고 있다. 다음으로 confusion matrix를 이용해 오분류된 개수를 확인한다.

```

# wine = 1, red = 0 / type_1 column 주출
y_test_c = y_test.iloc[:, 1]

# 0.9 이상인 확률은 1, 아닐 경우 0
predictions = model.predict(X_test_norm)
y_pred = (predictions > 0.9)

# confusion matrix
from sklearn import metrics
matrix = metrics.confusion_matrix(y_pred.argmax(axis = 1), y_test_c)
matrix

array([[313, 15],
       [1, 971]])

```

```
import pandas as pd
table = pd.DataFrame(matrix, columns = ["white", "red"], index = ["white", "red"])
table
```

	white	red
white	313	15
red	1	971

총 16개의 데이터를 잘못 분류 하였다.

d-2) standardized data

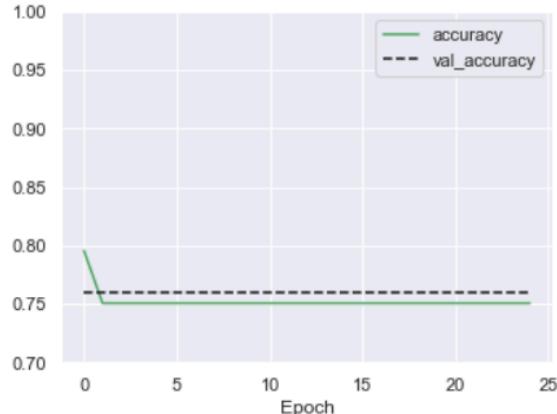
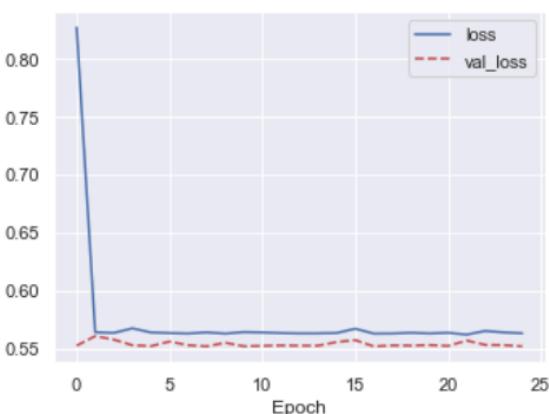
```
# wine_stan
history = model.fit(X_train_stan, y_train, epochs=25, batch_size=32, validation_split=0.25,
verbose = 1)
```

```
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['acc'], 'g-', label='accuracy')
plt.plot(history.history['val_acc'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.ylim(0.7, 1)
plt.legend()

plt.show()
```



```
loss, acc = model.evaluate(X_test_stan, y_test)
acc

0.9808
```

test data에서 accuracy가 99%인 normalization data의 training model performance가 가장 좋다. confusion matrix를 이용해 오분류된 개수를 확인한다.

```
# wine = 1, red = 0 の type_1 column 추출
y_test_c = y_test.iloc[:, 1]

# 0.9 이상인 확률은 1, 아닐 경우 0
predictions = model.predict(X_test_stan)
y_pred = (predictions > 0.9)

# confusion matrix
from sklearn import metrics
matrix = metrics.confusion_matrix(y_pred.argmax(axis = 1), y_test_c)
matrix

array([[300, 11],
       [ 14, 975]])

import pandas as pd
table = pd.DataFrame(matrix, columns = ["white", "red"], index = ["white", "red"])
table
```

	white	red
white	300	11
red	14	975

총 25개의 데이터를 잘못 분류하였다.

② Classification Example : Wine(Categorical data)

위에서 이용한 데이터를 동일하게 사용하되 feature 중 하나였던 quality로 target을 변경한다.

a) import dataset

```
import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
import random
sns.set_theme(color_codes = True)

SEED = 22
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

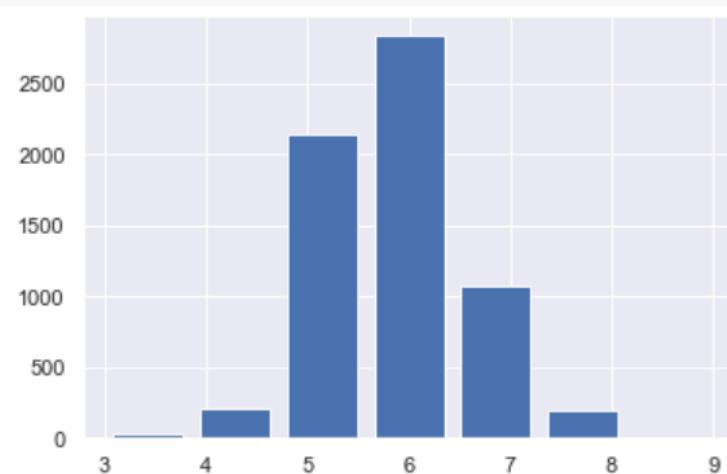
red = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-
quality/winequality-red.csv', sep=';')
white = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-
quality/winequality-white.csv', sep=';')
```

b) data preprocessing

```
# 와인 데이터셋 병합
wine = pd.concat([red, white])
wine.head(3)
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5

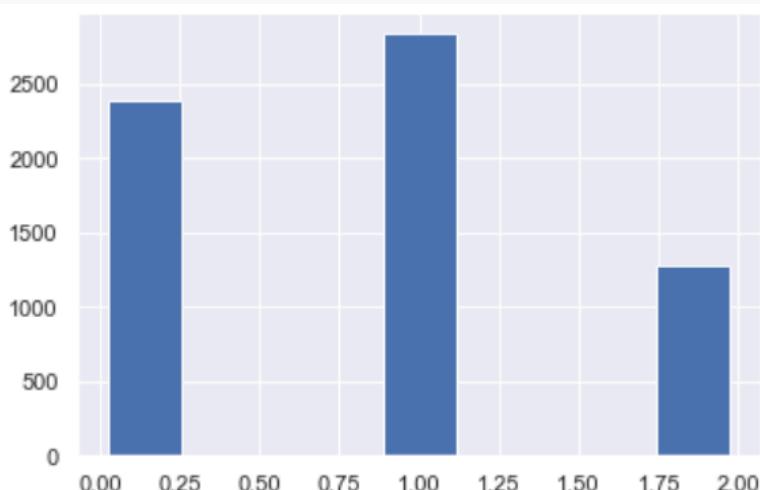
```
# 품질 히스토그램
plt.hist(wine['quality'], bins=7, rwidth=0.8)
plt.show()
```



와인의 품질이 3~9까지 7개의 등급으로 나누어져있다. 7개의 분류마다 샘플의 수가 일정하지 않아 분류가 어려워 보이므로 세 가지 범주로 나누어 재분류한다.

```
# 품질을 3 개의 범주(좋음, 보통, 나쁨)로 재분류
wine.loc[wine['quality'] <= 5, 'new_quality'] = 0
wine.loc[wine['quality'] == 6, 'new_quality'] = 1
wine.loc[wine['quality'] >= 7, 'new_quality'] = 2

# histogram
plt.hist(wine['new_quality'], bins=7, rwidth=0.8)
plt.show()
```



기존의 quality column을 제거한다.

```
wine = wine.drop("quality", axis = 1)
```

target인 new_quality를 원-핫 인코딩 한 후, train/test로 데이터를 분리한다.

```
# OneHotEncoder
wine = pd.get_dummies(data = wine, columns = ['new_quality'], prefix = ['new_quality'])

# train/test split
X = wine.iloc[:, :-3]
y = wine.iloc[:, -3:]

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    shuffle = True, random_state = SEED)
```

다음으로 모델에 학습시키기 이전에 feature scaling을 한다.

```
# feature scaling
from sklearn import preprocessing

# normalization
minmax_scaler = preprocessing.MinMaxScaler()
norm_fit = minmax_scaler.fit(X_train)
X_train_norm = norm_fit.transform(X_train)
```

```
X_test_norm = norm_fit.transform(X_test)

# standardization
standard_scaler = preprocessing.StandardScaler()
stan_fit = standard_scaler.fit(X_train)
X_train_stan = stan_fit.transform(X_train)
X_test_stan = stan_fit.transform(X_test)
```

NOTE)

train data를 기준으로 train/test data 모두를 scaling해야 하므로 train/test data split 후에 feature scaling을 해야한다.

c) building model

와인의 품질을 예측할 다항 분류 모델을 생성한다.

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=48, activation='relu', input_shape=(11,)),
    tf.keras.layers.Dense(units=24, activation='relu'),
    tf.keras.layers.Dense(units=12, activation='relu'),
    tf.keras.layers.Dense(units=3, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy', metrics=['accuracy'])

model.summary()

Model: "sequential_8"

-----  

Layer (type)           Output Shape        Param #  

-----  

dense_40 (Dense)      (None, 48)          576  

dense_41 (Dense)      (None, 24)          1176  

dense_42 (Dense)      (None, 12)          300  

dense_43 (Dense)      (None, 3)           39  

-----  

Total params: 2,091  

Trainable params: 2,091  

Non-trainable params: 0
```

d) training model & Test model

d-1) normalized data

```
history = model.fit(X_train_norm, y_train, epochs=100, batch_size=1, validation_split=0.25,
verbose = 0)
```

```
# 다항 분류 모델 학습 결과 시각화
plt.figure(figsize=(12, 4))
```

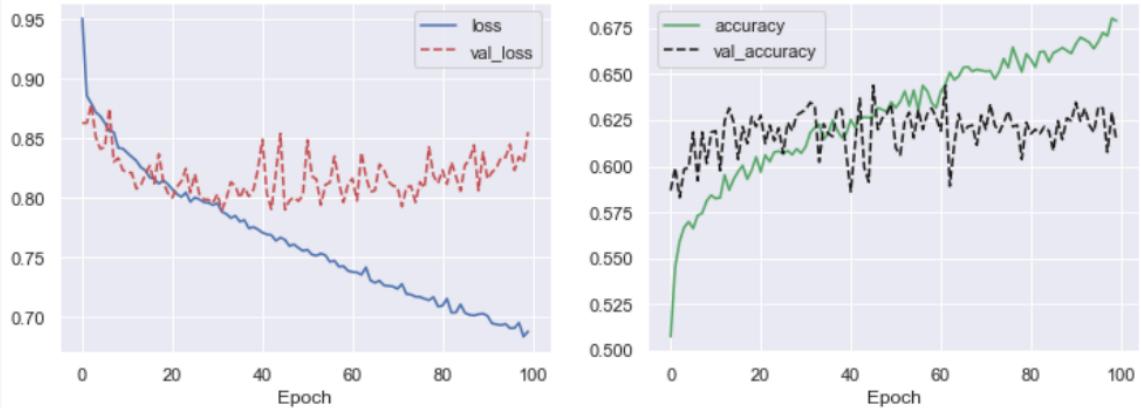
```

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['acc'], 'g-', label='accuracy')
plt.plot(history.history['val_acc'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.legend()

plt.show()

```



```

# 다형 분류 모델 평가
model.evaluate(X_test_norm, y_test)

[0.8667048811912537, 0.6176922917366028]

```

정확도가 61%로 raw data를 이용하여 학습했을 때 보다 모델의 성능이 향상되었다.

```

predictions = model.predict(X_test_norm)

y_pred = []

for i in range(0, len(y_test)):
    if np.argmax(predictions[i]) == 0:
        y_pred.append(0)
    elif np.argmax(predictions[i]) == 1:
        y_pred.append(1)
    elif np.argmax(predictions[i]) == 2:
        y_pred.append(2)

y_test_c = []

for i in range(0, len(y_test)):
    if np.argmax(y_test.iloc[i, :]) == 0:
        y_test_c.append(0)
    elif np.argmax(y_test.iloc[i, :]) == 1:
        y_test_c.append(1)
    elif np.argmax(y_test.iloc[i, :]) == 2:
        y_test_c.append(2)

```

```
# confusion matrix
from sklearn import metrics
matrix = metrics.confusion_matrix(y_pred, y_test_c)
matrix

array([[356, 172, 14],
       [114, 372, 163],
       [ 0, 34, 75]])

label = ["new_quality_1", "new_quality_2", "new_quality_3"]

import pandas as pd
table = pd.DataFrame(matrix, columns = label, index = label)
table
```

	new_quality_1	new_quality_2	new_quality_3
new_quality_1	356	172	14
new_quality_2	114	372	163
new_quality_3	0	34	75

d-2) standardized data

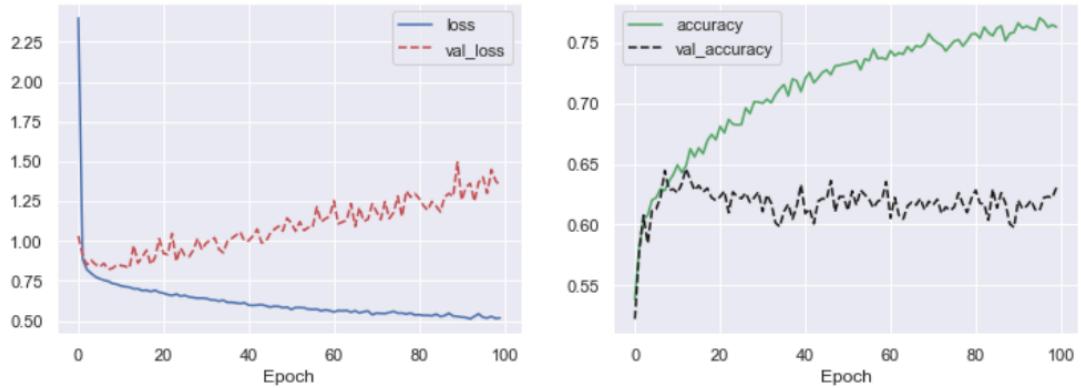
```
history = model.fit(X_train_stan, y_train, epochs=100, batch_size=1, validation_split=0.25,
verbose = 0)

# 단행 분류 모델 학습 결과 시각화
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['acc'], 'g-', label='accuracy')
plt.plot(history.history['val_acc'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.legend()

plt.show()
```



```
# 향후 분류 모델 평가
model.evaluate(X_test_stan, y_test)

[1.414917230606079, 0.6215384602546692]
```

마지막으로 Confusion matrix를 구한다.

```
predictions = model.predict(X_test_stan)

y_pred = []

for i in range(0, len(y_test)):
    if np.argmax(predictions[i]) == 0:
        y_pred.append(0)
    elif np.argmax(predictions[i]) == 1:
        y_pred.append(1)
    elif np.argmax(predictions[i]) == 2:
        y_pred.append(2)

y_test_c = []

for i in range(0, len(y_test)):
    if np.argmax(y_test.iloc[i, :]) == 0:
        y_test_c.append(0)
    elif np.argmax(y_test.iloc[i, :]) == 1:
        y_test_c.append(1)
    elif np.argmax(y_test.iloc[i, :]) == 2:
        y_test_c.append(2)

# confusion matrix
from sklearn import metrics
matrix = metrics.confusion_matrix(y_pred, y_test_c)
matrix

array([[341, 164, 21],
       [112, 327, 91],
       [17, 87, 140]])

label = ["new_quality_1", "new_quality_2", "new_quality_3"]

import pandas as pd
table = pd.DataFrame(matrix, columns = label, index = label)
table
```

	new_quality_1	new_quality_2	new_quality_3
new_quality_1	341	164	21
new_quality_2	112	327	91
new_quality_3	17	87	140

jmkim21@ewhain.net



③ Classification Example : Dry beans dataset(UCI machine learning repository)

Dry beans dataset은 13611개의 데이터로 16개의 feature와 1개의 target으로 구성되었다. target은 범주형 변수로 Dermason, Sira, Seker, Horoz, Cali, Barbunya, Bombay 7의 클래스로 분류된다. 이 데이터의 분석의 목적은 7개의 변수를 분류하는 것이다.

target	
1. Class	(Dermason, Sira, Seker, Horoz, Cali, Barbunya, Bombay)
feature	
2. Area	The area of a bean zone and the number of pixels within its boundaries
3. Perimeter	Bean circumference is defined as the length of its border.
4. Major axis length(L)	The distance between the ends of the longest line that can be drawn from a bean
5. Minor axis length(l)	The longest line that can be drawn from the bean while standing perpendicular to the main axis
6. Aspect ratio	Defines the relationship between L and l
7. Eccentricity	Eccentricity of the ellipse having the same moments as the region
8. Convex area	Number of pixels in the smalles convex polygon that can contain the area of a bean seed.
9. Equivalent diameter	The diameter of a circle having the same area as a bean seed area
10. Extent	The ratio of the pixels in the bounding box to the bean area
11. Solidity	Also known as convexity. The ration of the pixles in the convex shell to those found in beans.
12. Roundness	Calculated with the following formula $(4\pi A/P^2)$
13. Compactness	Measures the roundness of an object
14. ShapeFactor1	
15. ShapeFactor2	
16. ShapeFactor3	
17. ShapeFactor4	

a) import datasets

```
# import package
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf

sns.set_theme(color_codes = True)

# import dataset
mydata = pd.read_excel("/Users/jiminpopo/jupyter/LAB/DryBeanDataset/Dry_Bean_Dataset.xlsx")
```

엑셀파일 이므로 `read_excel()`을 이용하여 데이터를 불러온다.

```
mydata.head()
```

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRatio	Eccentricity	ConvexArea	EquivDiameter	Extent	Solidity	roundness	C
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	28715	190.141097	0.763923	0.988856	0.958027	
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	29172	191.272750	0.783968	0.984986	0.887034	
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	29690	193.410904	0.778113	0.989559	0.947849	
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	30724	195.467062	0.782681	0.976696	0.903936	
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	30417	195.896503	0.773098	0.990893	0.984877	

```
mydata.shape
```

```
(13611, 17)
```

데이터의 개수는 13611개이며 feature의 개수는 16개이다.

```
# feature
mydata.describe().T
```

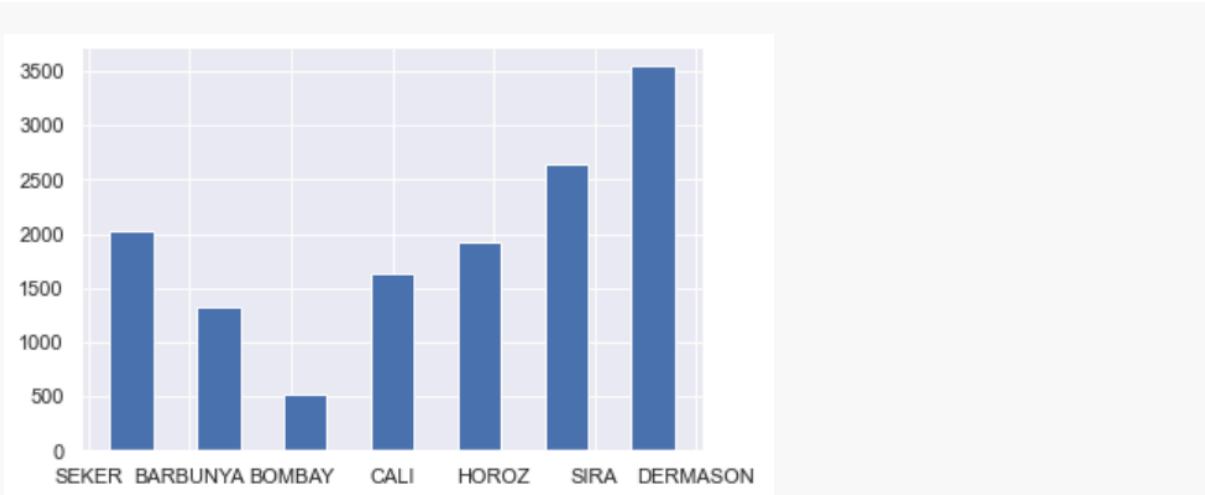
	count	mean	std	min	25%	50%	75%	max
Area	13611.0	53048.284549	29324.095717	20420.000000	36328.000000	44652.000000	61332.000000	254616.000000
Perimeter	13611.0	855.283459	214.289696	524.736000	703.523500	794.941000	977.213000	1985.370000
MajorAxisLength	13611.0	320.141867	85.694186	183.601165	253.303633	296.883367	376.495012	738.860153
MinorAxisLength	13611.0	202.270714	44.970091	122.512653	175.848170	192.431733	217.031741	460.198497
AspectRatio	13611.0	1.583242	0.246678	1.024868	1.432307	1.551124	1.707109	2.430306
Eccentricity	13611.0	0.750895	0.092002	0.218951	0.715928	0.764441	0.810466	0.911423
ConvexArea	13611.0	53768.200206	29774.915817	20684.000000	36714.500000	45178.000000	62294.000000	263261.000000
EquivDiameter	13611.0	253.064220	59.177120	161.243764	215.068003	238.438026	279.446467	569.374358
Extent	13611.0	0.749733	0.049086	0.555315	0.718634	0.759859	0.786851	0.866195
Solidity	13611.0	0.987143	0.004660	0.919246	0.985670	0.988283	0.990013	0.994677
roundness	13611.0	0.873282	0.059520	0.489618	0.832096	0.883157	0.916869	0.990685
Compactness	13611.0	0.799864	0.061713	0.640577	0.762469	0.801277	0.834270	0.987303
ShapeFactor1	13611.0	0.006564	0.001128	0.002778	0.005900	0.006645	0.007271	0.010451
ShapeFactor2	13611.0	0.001716	0.000596	0.000564	0.001154	0.001694	0.002170	0.003665
ShapeFactor3	13611.0	0.643590	0.098996	0.410339	0.581359	0.642044	0.696006	0.974767
ShapeFactor4	13611.0	0.995063	0.004366	0.947687	0.993703	0.996386	0.997883	0.999733

각 데이터들의 범위 차이가 크기 때문에 feature scaling이 필요해 보인다.

```
# target
mydata.Class.value_counts()

DERMASON      3546
SIRA          2636
SEKER          2027
HOROZ          1928
CALI           1630
BARBUNYA      1322
BOMBAY         522
Name: Class, dtype: int64
```

```
# histogram
plt.hist(mydata['Class'], bins = 7, rwidth = 0.5)
plt.show()
```



dermason 종이 가장 많으며 bombay 종이 가장 적다.

b) data preprocessing

target column에 대하여 라벨인코딩 후 원-핫 인코딩을 한다.

```
# target Label encoding
from sklearn.preprocessing import LabelEncoder
items = mydata['Class']
le = LabelEncoder()

mydata['Class'] = le.fit_transform(items)

# Label encoding // 해당되는 문자열 출력
print(le.inverse_transform([0,1,2,3,4,5,6]))

['BARBUNYA' 'BOMBAY' 'CALI' 'DERMASON' 'HOROZ' 'SEKER' 'SIRA']

# One-Hot Encoder
mydata = pd.get_dummies(data = mydata, columns = ['Class'], prefix = ['Class'])
```

feature scaling을 하기 이전에 train/test data로 split 한다.

```
# train/test split
X = mydata.iloc[:, :-7]
Y = mydata.iloc[:, -7:]

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.2,
                                                    shuffle = True, random_state = 22)
```

```
# feature scaling
from sklearn import preprocessing

# normalization
minmax_scaler = preprocessing.MinMaxScaler()
norm_fit = minmax_scaler.fit(X_train)
X_train_norm = norm_fit.transform(X_train)
X_test_norm = norm_fit.transform(X_test)
```

```
# standardization
standard_scaler = preprocessing.StandardScaler()
stan_fit = standard_scaler.fit(X_train)
X_train_stan = stan_fit.transform(X_train)
X_test_stan = stan_fit.transform(X_test)
```

c) building model

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units = 512, activation = 'relu', input_shape = (16,)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units = 256, activation = 'relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units = 128, activation = 'relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units = 64, activation = 'relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units = 32, activation = 'relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units = 7, activation = 'softmax'),
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-4),
              loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

```

model.summary()

Model: "sequential_1"
-----  

Layer (type)          Output Shape       Param #
-----  

dense_4 (Dense)      (None, 512)        8704  

dropout (Dropout)    (None, 512)        0  

dense_5 (Dense)      (None, 256)        131328  

dropout_1 (Dropout)  (None, 256)        0  

dense_6 (Dense)      (None, 128)        32896  

dropout_2 (Dropout)  (None, 128)        0  

dense_7 (Dense)      (None, 64)         8256  

dropout_3 (Dropout)  (None, 64)         0  

dense_8 (Dense)      (None, 32)         2080  

dropout_4 (Dropout)  (None, 32)         0  

dense_9 (Dense)      (None, 7)          231  

-----  

Total params: 183,495
Trainable params: 183,495
Non-trainable params: 0
-----
```

hidden layer는 4개이며, hidden layer 사이에 dropout layer를 추가한다. 활성화 함수는 Relu를 이용한다. 분류 문제이므로 출력층의 활성화 함수는 softmax를 이용한다. 하이퍼파라미터에서 optimizer는 RMSprop를 사용하며, 학습률은 0.004로 설정한다. 분류 문제이므로 손실함수는 categorical cross-entropy를 이용하며, metrics = accuracy로 지정한다.

d) training model & test model

d-1) Min-Max normalization

```

early_stop = tf.keras.callbacks.EarlyStopping(patience = 6, restore_best_weights = True,
                                              monitor = 'val_loss')
history = model.fit(X_train_norm, y_train, epochs = 100, batch_size = 32, validation_split = 0.2,
                     verbose = 0)

plt.figure(figsize=(17, 4))

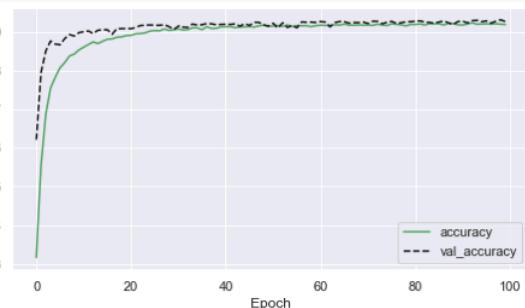
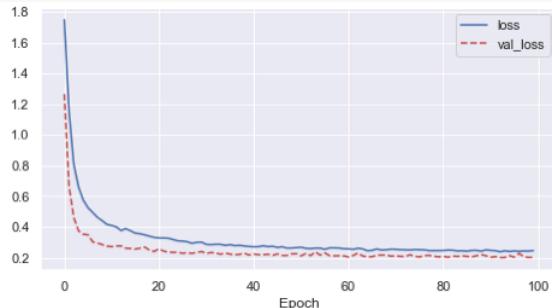
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()
```

```

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.legend()

plt.show()

```



```

model.evaluate(X_test_norm, y_test)

[0.2133258730173111, 0.9298567771911621]

```

accuracy가 92%로 매우 높다. 다음으로 confusion matrix를 구한다.

```

predictions = model.predict(X_test_norm)

y_pred = []

for i in range(0, len(y_test)):
    k = np.argmax(predictions[i])
    y_pred.append(k)

y_test_c = []

for i in range(0, len(y_test)):
    k = np.argmax(y_test.iloc[i,:])
    y_test_c.append(k)

from sklearn import metrics
matrix = metrics.confusion_matrix(y_pred, y_test_c)
matrix

```

array([[239, 0, 17, 0, 1, 5, 2],
[0, 106, 0, 0, 0, 0, 0],
[14, 0, 315, 0, 4, 0, 0],
[0, 0, 0, 639, 7, 7, 54],
[1, 0, 5, 1, 379, 0, 3],
[2, 0, 1, 14, 0, 407, 7],
[1, 0, 1, 28, 7, 9, 447]])

```

# Label encoding 이전의 target column 의 문자열 출력
label = le.inverse_transform([0,1,2,3,4,5,6])

import pandas as pd
table = pd.DataFrame(matrix, columns = label, index = label)
table

```

	BARBUNYA	BOMBAY	CALI	DERMASON	HOROZ	SEKER	SIRA
BARBUNYA	239	0	17	0	1	5	2
BOMBAY	0	106	0	0	0	0	0
CALI	14	0	315	0	4	0	0
DERMASON	0	0	0	639	7	7	54
HOROZ	1	0	5	1	379	0	3
SEKER	2	0	1	14	0	407	7
SIRA	1	0	1	28	7	9	447

NOTE)

np.argmax는 R에서 which.max와 동일하다.

dermason 종을 sira 종으로 잘못 분류하는 경우가 많다.

d-2) standardization

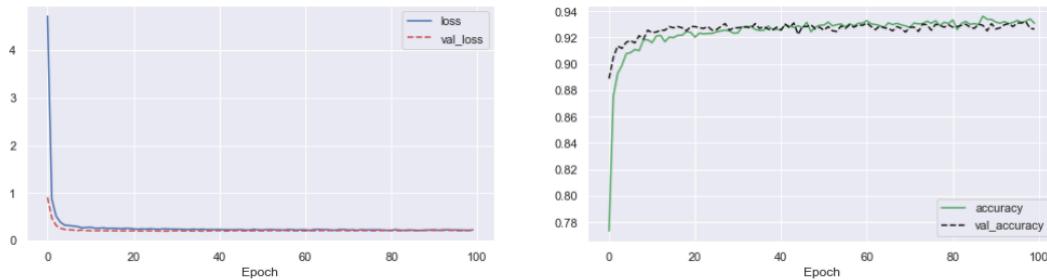
```
early_stop = tf.keras.callbacks.EarlyStopping(patience = 6, restore_best_weights = True,
                                              monitor = 'val_loss')
history = model.fit(X_train_stan, y_train, epochs = 100, batch_size = 32, validation_split = 0.2,
                     verbose = 0)
```

```
plt.figure(figsize=(17, 4))
```

```
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], 'b-', label='loss')
plt.plot(history.history['val_loss'], 'r--', label='val_loss')
plt.xlabel('Epoch')
plt.legend()
```

```
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], 'g-', label='accuracy')
plt.plot(history.history['val_accuracy'], 'k--', label='val_accuracy')
plt.xlabel('Epoch')
plt.legend()
```

```
plt.show()
```



```
model.evaluate(X_test_stan, y_test)
```

```
[0.2183748483657837, 0.9349981546401978]
```

accuracy가 93%로 매우 높다. 다음으로 confusion matrix를 구한다.

```

predictions = model.predict(X_test_stan)

y_pred = []

for i in range(0, len(y_test)):
    k = np.argmax(predictions[i])
    y_pred.append(k)

y_test_c = []

for i in range(0, len(y_test)):
    k = np.argmax(y_test.iloc[i,:])
    y_test_c.append(k)

from sklearn import metrics
matrix = metrics.confusion_matrix(y_pred, y_test_c)
matrix

```

array([[234, 0, 7, 0, 1, 2, 1],
 [0, 106, 1, 0, 0, 0, 0],
 [18, 0, 320, 0, 2, 0, 0],
 [0, 0, 636, 6, 6, 43],
 [1, 0, 8, 1, 381, 0, 4],
 [2, 0, 13, 0, 411, 7],
 [2, 0, 32, 8, 9, 458]])

```

label = le.inverse_transform([0,1,2,3,4,5,6])

import pandas as pd
table = pd.DataFrame(matrix, columns = label, index = label)
table

```

	BARBUNYA	BOMBAY	CALI	DERMASON	HOROZ	SEKER	SIRA
BARBUNYA	234	0	7	0	1	2	1
BOMBAY	0	106	1	0	0	0	0
CALI	18	0	320	0	2	0	0
DERMASON	0	0	0	636	6	6	43
HOROZ	1	0	8	1	381	0	4
SEKER	2	0	1	13	0	411	7
SIRA	2	0	2	32	8	9	458

Min-Max Normalization의 결과와 마찬가지로 dermason 종을 sira 종으로 잘못 분류하는 경우가 가장 많다.