# Back Propagation

Homework #3

32181928 박찬호

Dankook University

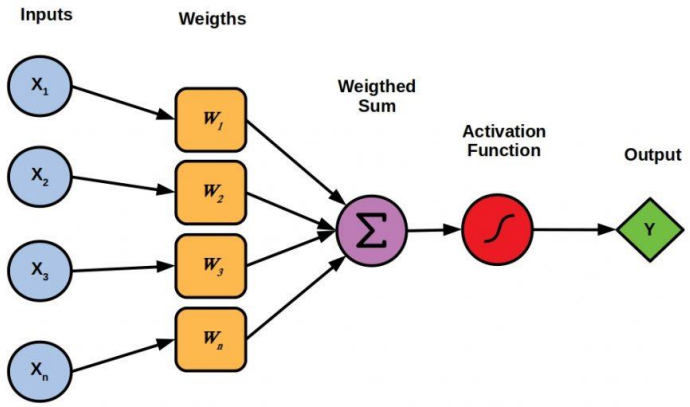
Mobile Systems Engineering

2022 Fall

서론

Perceptron and Transformation

인간의 뉴런을 모방한 Perceptron 은 여러 개의 input 에 weight 를 곱한 후 더하여, activation function 을 통과시켜 하나의 output 을 출력한다. 이는 아래의 그림처럼 표현할 수 있다.




https://velog.io/@tobe-honest/%EB%8B%A4%EC%B8%B5-%ED%8D%BC%EC%85%89%ED%8A%B8%EB%A1%A0Multi-layer-Perceptron-MLP

이러한 동작 방식을 선형대수학의 언어로 표현하면, 변수 $x_i$ 들이 모인 $\boldsymbol{x}$ 벡터와 가중치 $w_i$ 들이 모인 $\boldsymbol{w}$ 벡터의 선형 결합이라고 할 수 있다. 나아가 행렬 곱 역시 이러한 선형결합의 모음으로 이해할 수 있다. 이는 아래의 꼴과 같다.

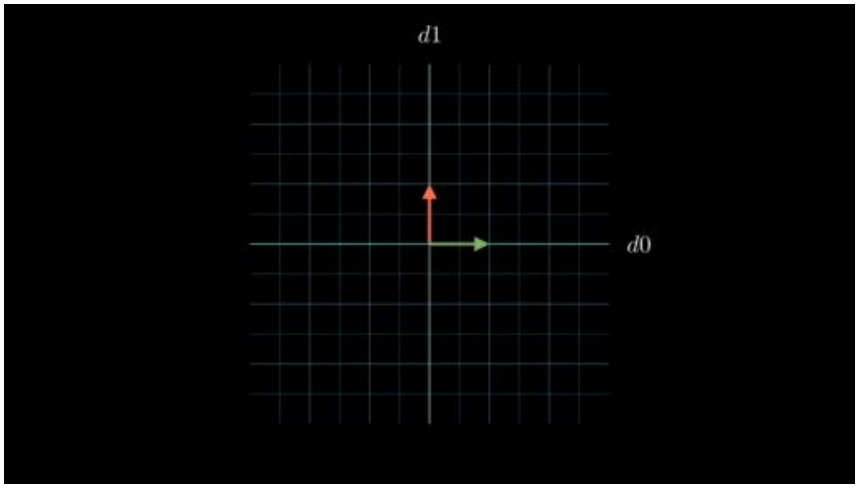
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \quad \therefore \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} ax+by \\ cx+dy \end{pmatrix}$$

여기서 유의할 것은 좌측과 우측의 식이 같다는 것이다. 좌측의 꼴은 선형 변환으로 볼 수 있고 우측의 꼴은 선형결합의 모음으로 볼 수 있다. 따라서 벡터 $\langle \begin{smallmatrix} x \\ y \end{smallmatrix} \rangle$와 $\langle \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \rangle$ 간의 행렬 곱을 통해 선형결합의 모음을 만드는 것은 벡터를 구성하는 기저 x, y 를 행렬 $\langle \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \rangle$ 에 대해 선형 변환을 하는 것으로 이해할 수 있다. 따라서 Multi-Layer Perceptron 에서의 선형 결합들은 x 에 대한 선형변환으로 볼 수 있다. 이는 아래의 수식처럼 이해할 수 있으며, 이러한 선형 변환에 의해 벡터 공간은 우측과 같이 변화한다. (수식과 변환 애니메이션은 다른 행렬에 대한 변환이다)

$$A = \begin{bmatrix} 2 & -3 \\ 1 & 1 \end{bmatrix}$$

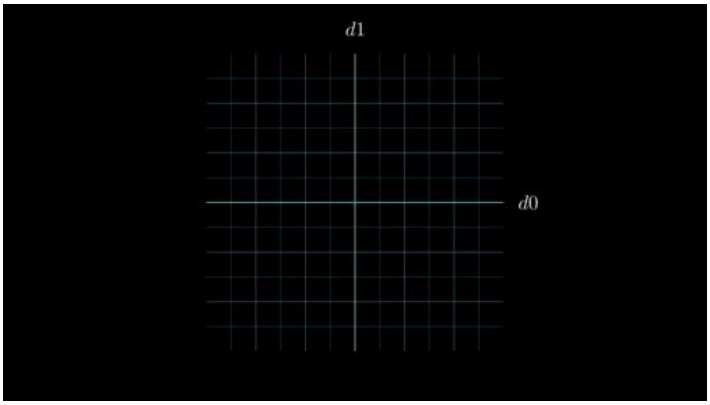
$$\vec{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

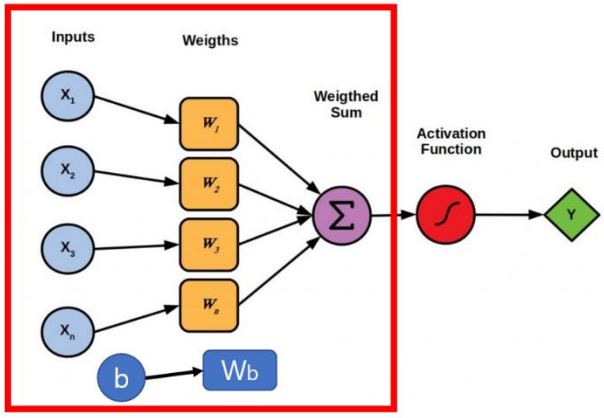
$$A\vec{x} = \begin{bmatrix} 2 & -3 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$



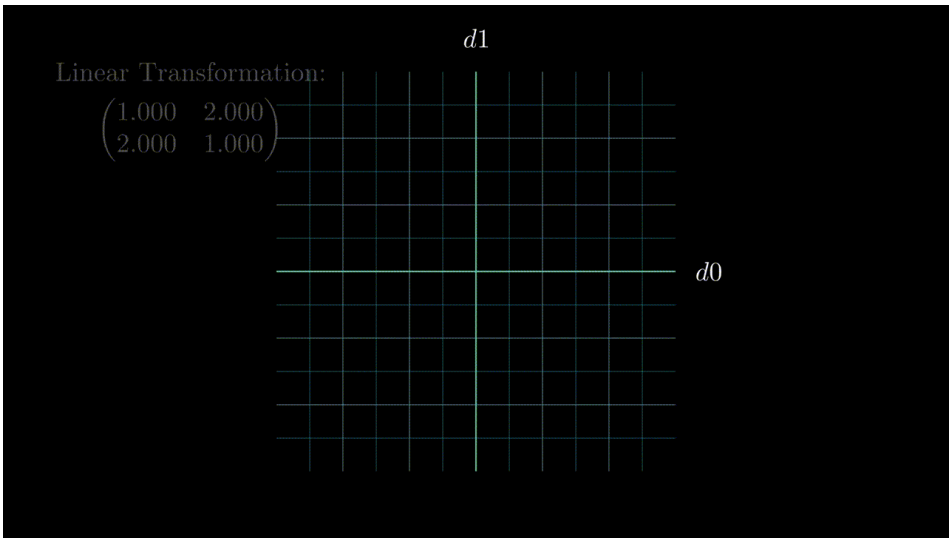

https://towardsdatascience.com/visualizing-the-mlp-a-composition-of-transformations-dec1c62d4eea

이때 선형 변환에 bias 값이 추가되면 이는 Affine 변환이라고 한다. 이 변환에 의한 공간의 변화는 좌측과 같고, 이때 perceptron 내에서 변환이 수행되는 영역은 우측 빨간 테두리 안의 영역이다.

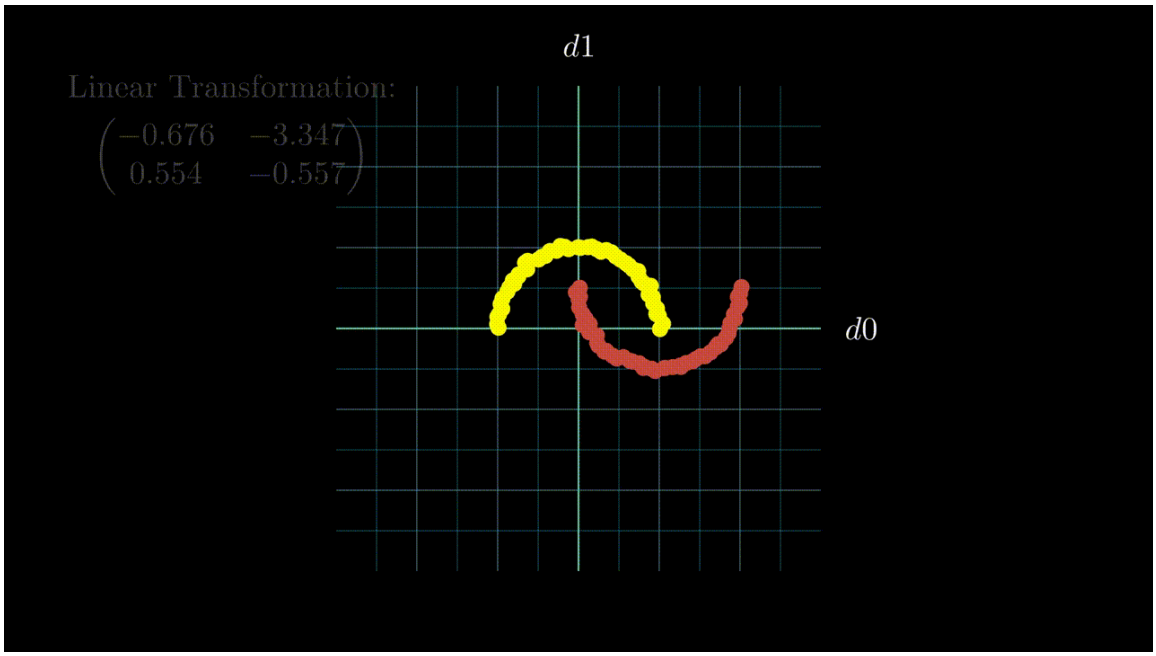



이렇게 선형변환을 수행한 후, 이를 non-linear activation function 을 통해 활성화하게 되면 공간은 다시 비선형변환을 거치게 된다. 이는 아래의 그림과 같다.



이러한 변환의 거듭된 적용은 공간을 일그러뜨려 어떤 복잡한 함수를 근사하는 효과를 가지게 된다. 이는 아래의 영상으로 확인할 수 있으며, 영상에서는 변환을 거듭하여 복잡한 데이터의 decision boundary 를 찾아낸다

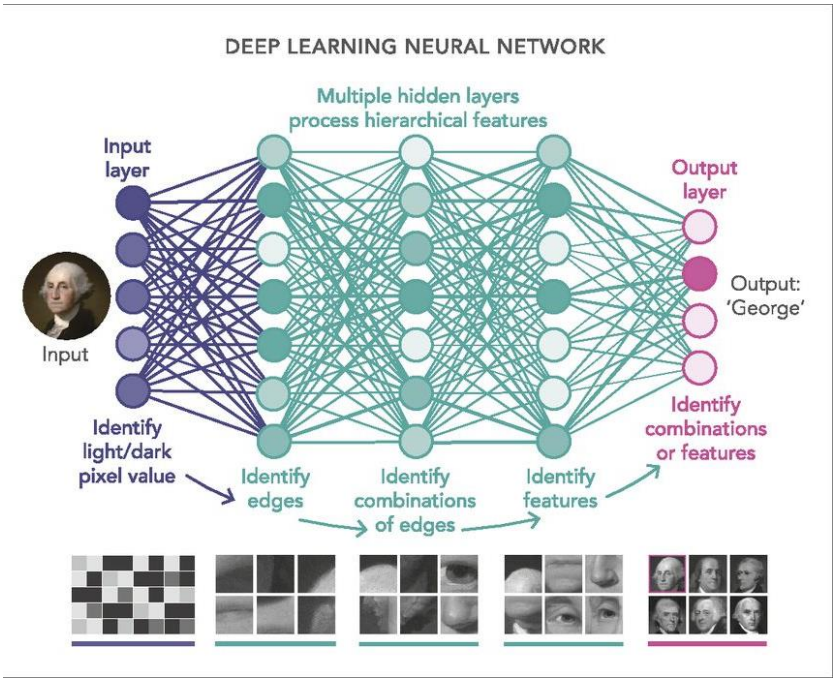



따라서 deep learning 에서의 은닉층은 이러한 Affine Transform + Non-Linear Transform 을 거듭해서 적용할 수 있게 만드는 역할을 수행하고 이러한 방법으로 우리는 복잡한 함수를 근사할 수 있다.

Back Propagation

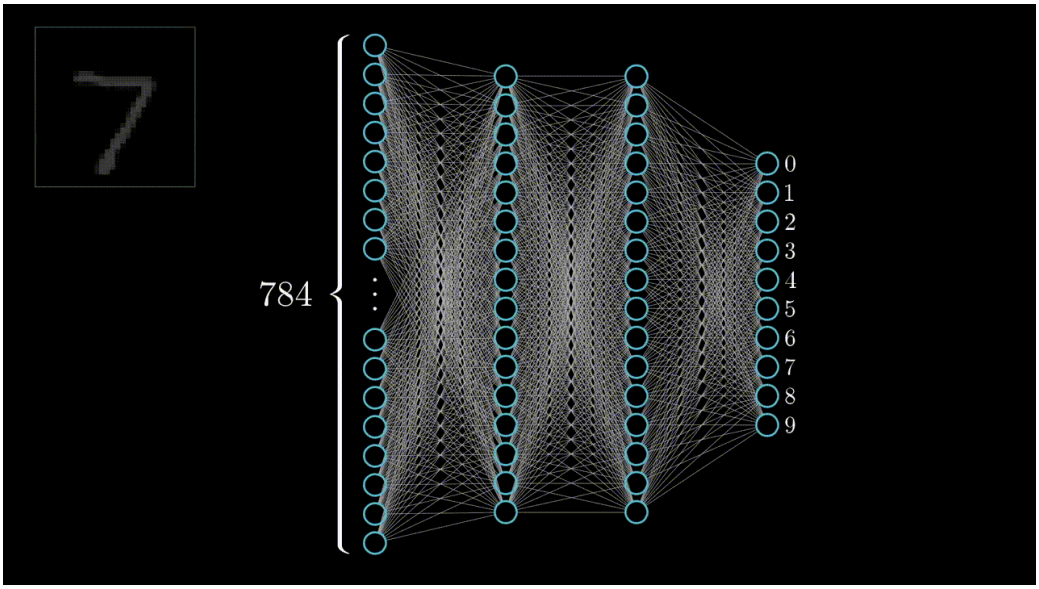
일반적으로 deep learning 은 artificial neural network 에서 hidden layer 가 2~3 개 이상일 때를 말한다. 즉, 은닉층의 깊이가 깊어 복잡한 함수를 잘 근사할 수 있는 것이다.

나아가 이러한 과정은 개별적인 feature 들로부터 추상적인 feature, 고차원적인 feature 를 찾아내는 representation learning 으로도 볼 수 있다. 각 layer 에서의 학습이 feature 의 hierarchy 를 만들어 내는 것이다.




https://www.pnas.org/doi/full/10.1073/pnas.1821594116

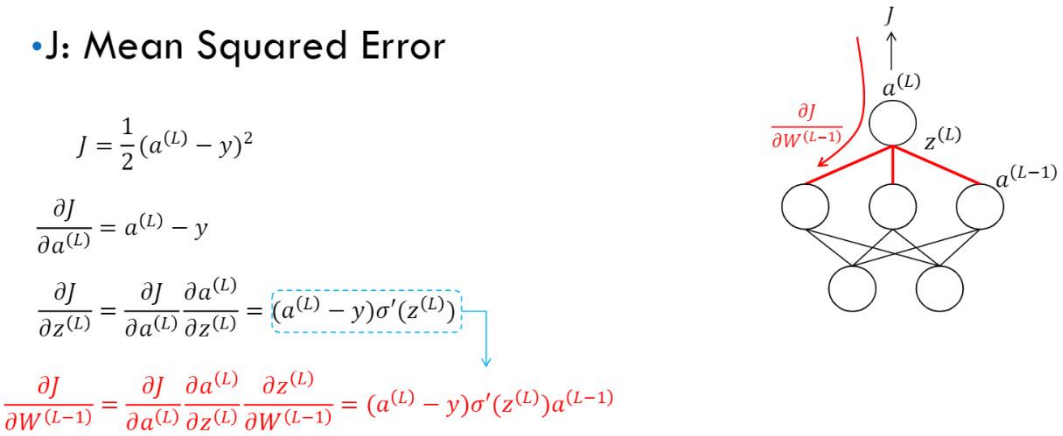
그런데 이는 아래와 같이 이미 적절한 weight 가 정해져 있고, 이를 Feed-Forward 시에 이용하여 activation 을 수행했을 때의 경우이다.




https://youtu.be/aircAruvnKk

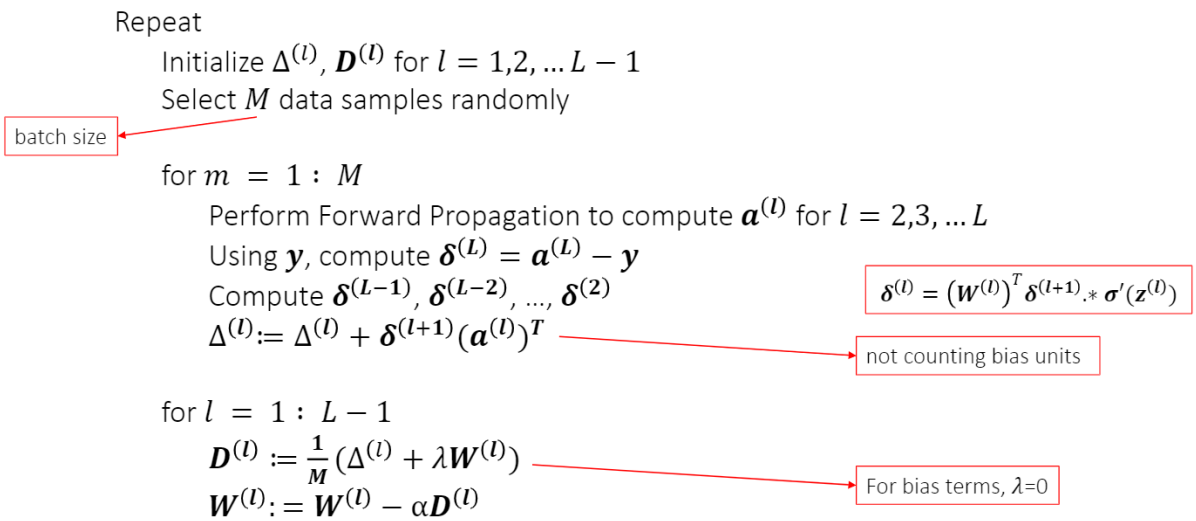
따라서 우리는 어떤 perceptron 과 연결이 얼만큼 활성화되는지를 찾아야 하고, 이는 training 과정을 통해 얻을 수 있다. Linear Regression 과 같은 일반적인 machine learning 방법에서 training 은 gradient descent 를 통해 수행된다. 그러나, Deep Neural Network 는 gradient 를 수행하기 위해서 각 weight 의 gradient 를 찾아야 하는데, 개별 weight 에 대해 gradient 를 찾고 gradient descent 를 수행하는 것은 상당한 cost 를 요구한다.

이때 등장하는 방법이 바로 Error Backpropagation 이다. Error 을 output layer 에서 찾은 후, 이를 전의 layer 로 거듭 propagation 을 수행하며 gradient 를 찾고 weight 를 업데이트 하는 방법이다. 이는 derivative (혹은 gradient)를 구할 때의 chain rule 에서 그 원리를 찾을 수 있다




출처 : 강의자료

이 과정을 layer 단위의 알고리즘으로 행렬을 사용하여 정리하면 아래와 같다.



본론

구현

본 프로그램에서도 이 알고리즘을 일부 차용한다.

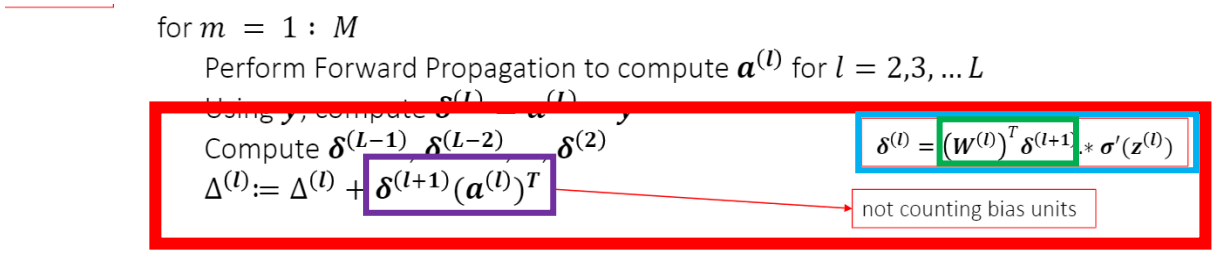
먼저 layer 단위에서 수행되는 backward 과정을 보면 아래와 같다.

**error = diff * activation 함수의 미분 값**

**dw = outer product(error, current_activation)**

**diff = dot product(weight.T, error)**

이는 알고리즘에서 빨간 부분에 해당하는 것이다.




먼저 diff 가 초기값 (이후 등장하겠지만 y_hat – y)이 준비된 이후,

**Error = diff * activation 의 미분값**은 하늘색 박스에 해당한다

**dw = outer product(error, current_activation)**은 보라색 박스에 해당한다. 이는 행렬 $u, v$의 outer product 가 $uv^T$와 같기 때문이다.

**diff = dot product(weight.T, error)** 은 초록색 박스에 해당한다.

따라서 이를 layer_back(diff) 라고 하면 전체 Network 에서 back propagation 은 아래와 같이 진행된다.

```
diff = y_hat - y

for layer in layer-reverse:

        diff = layer_back (diff)
```

그리고 이 과정에서 dw 를 저장하면, back propagation 을 마친 후 전체 layer 에 대해 weight update 를 수행한다.

제시한 알고리즘과 이 logic 이 차이가 있는 부분은, 알고리즘은 data 전체를 forwarding 한 후 back propagation 을 진행한다는 것인데 현재의 구현 방법에서 위의 방법을 사용할 경우 최종 layer 에서 구현의 미묘한 차이에 따라 overflow 나 invalid value 등의 문제가 발생하였다. 이러한 warning 는 현재의 주어진 데이터 외에 다른 데이터를 생성하며 테스트 해본 결과 데이터가 갖는 y 값의 범위나 x 값의 범위에 따라 발생하였으며, 이는 본 과제로 주어진 데이터에 대해 전혀 문제가 발생하지 않더라도 유효한 구현이 아니라고 판단했다. 따라서 모든 데이터를 하나씩 forwarding – back propagation – update weight 하는 과정으로 바꿨는데, 이 과정에서는 문제가 사라지고 학습도 정상적으로 이루어지는 것을 확인할 수 있었다.

따라서 이러한 back propagation, train 과정을 코드로 표현하면 아래와 같다

Layer wise

```
def backward(self, diff):
    error = diff * self.activation.grad()
    dw = np.outer(error, self.x)
    diff = np.dot(self.weight.T, error)
    return diff, dw
```

Network wise

```
def back_propagation(self, Y, Y_hat):
        diff = (Y_hat - Y) / Y_hat.shape[0]
        to_update = []
        for idx, layer in reversed(list(enumerate(self._model['layers']))):
            diff, dw = layer.backward(diff)
            to_update.append(dw)
        for dw, layer in zip(reversed(to_update), self._model['layers']):
            layer.weight -= self.alpha * dw


def fit(self, X, Y):
        for epoch in range(self.epochs):
            for x, y in zip(X, Y):
                y_hat = self.feed_forward(x)
                self.back_propagation(y, y_hat)
```

추가적으로 구현 상의 특장점을 나열하자면,

1. 객체지향적 설계에 충실하였다
2. Layer-wise, Network-wise method 를 명확히 구분하여 설계하였다.
3. Progress 를 단계별로 plotting 할 수 있다.
4. Network 의 구조를 description, graph 등으로 볼 수 있다.
5. 다양한 데이터와 activation function, 구조에 따라 수정이 가능하도록 구현하였다.

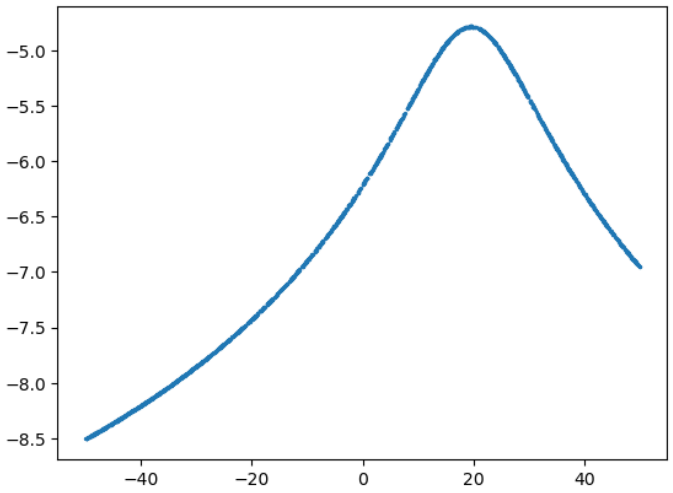
1 번에 대해 추가로 설명하자면, Layer 와 Sequential 클래스(network)를 생성하여 method 와 데이터를 관리하였고, 이를 통해 weight, bias, activation 등을 layer 에 종속시키고, 또한 alpha, layers, epochs, graph 등을 Sequential 에 종속시킬 수 있었다. 또한 forward, backward 를 각 단계별로 구현하였다. Activation function 들 역시 Function 클래스를 상속하여 동일한 method 로 activation 과 gradient 를 수행할 수 있으며, 특히 activation 시의 값을 저장하여 gradient 를 수행 시에 별도의 값을 input 으로 받지 않아도 되는 장점이 있다.

```
class Function:
    def __init__(self, name = ""):
        self.output = 0
        self.name = name

    def __call__(self, X):
        pass

    def grad(self):
        pass

    def string(self):
        pass
```

```
class Sigmoid(Function):
    def __init__(self):
        super().__init__("Sigmoid")

    def __call__(self, X):
        self.output = 1 / (1 +np.exp(-X))
        return self.output

    def grad(self):
        return self.output * (1 - self.output)

    def string(self):
        return self.name
```

결론

먼저 dataset 은 아래와 같이 구성되어 있다.




모델은 아래의 모델을 사용하였다

----------Sequential----------

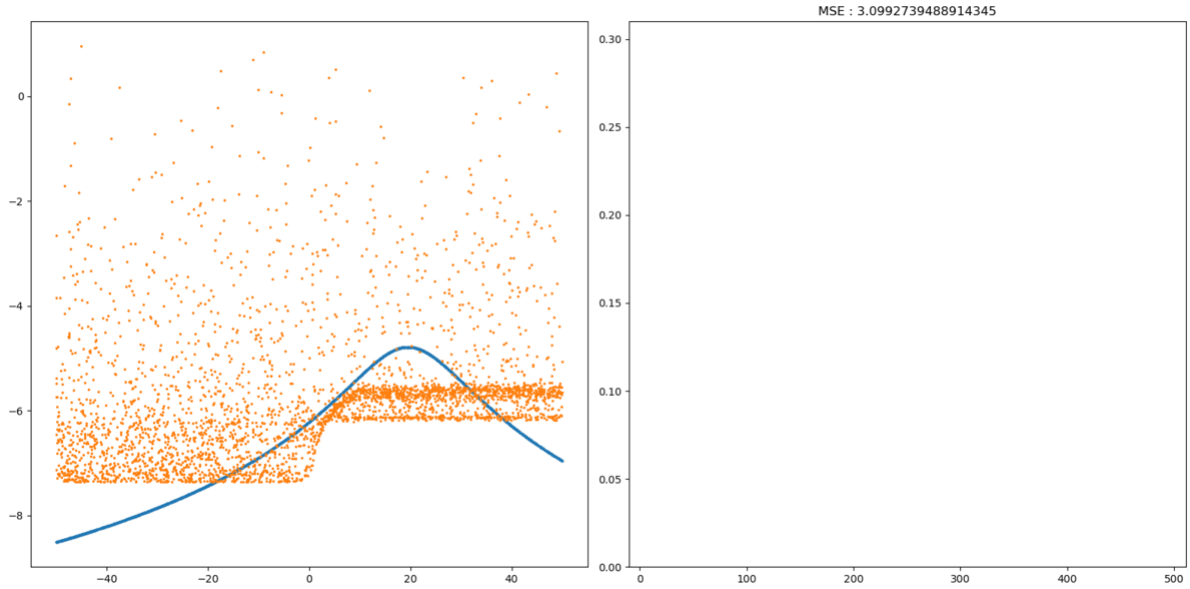
Alpha : 0.001, Epochs : 501

Cost Function : Mean Squared Error

| | | | | |
|---|---|---|---|---|
| Input size : | 1 | Output size : | 4 | Activation : Hyperbolic Tangent |
| Input size : | 4 | Output size : | 8 | Activation : Hyperbolic Tangent |
| Input size : | 8 | Output size : | 4 | Activation : Hyperbolic Tangent |
| Input size : | 4 | Output size : | 2 | Activation : Hyperbolic Tangent |
| Input size : | 2 | Output size : | 1 | Activation : Identity |

---

이를 train – test set 로 나누어 (test 30%) train 을 수행한 결과는 아래와 같다




학습 과정에서 cost 는 우측과 같이 감소하였고, 좌측처럼 데이터가 근사되는 것을 확인할 수 있다.

총 500 epoch 을 학습하였는데, 이때 더 학습을 진행할 경우 train set 에 대해서는 cost 가 감소하지만, test dataset 의 cost 는 큰 변화가 없었다.

dataset 의 cost 는 아래와 같다.

```
Train(epoch = 500)     | MSE : 0.004784159457067888
Test                   | MSE : 2.7667068630265046
```

이를 통해 도출된 최종 weight 및 bias 는 아래와 같다

Input – Hidden1

[{'w': array([[ 0.86559703], [ 0.16934807], [-0.04074102], [-0.04840727]]),

'b': array([[-0.62131118], [-1.69227828], [ 0.14486564], [ 0.2726653 ]])},

## Hidden1 – Hidden2

{'w': array([[-1.14317884, -0.19807719, -1.0956175 , -0.9464194 ],

[-0.42183356, -0.39634244, -0.28193942, -0.26519271],

[ 0.18835332, 0.4604627 , 0.196893 , 0.08042776],

[-0.04695649, -0.63528868, -0.65160954, -0.09517421],

[-0.57856558, -0.41202364, -1.35056763, -0.99919844],

[ 0.2338379 , 0.32957321, 0.12331997, -0.09561573],

[ 0.87113785, 0.49078162, 0.22489726, 0.62998866],

[ 0.31657414, 0.45646341, -0.17535194, -0.0993499 ]]),

'b': array([[-0.47160482], [ 0.65089026], [ 0.00868813], [-0.88830515], [ 1.4248608 ], [-0.65507567], [-0.99573567], [ 0.09300561]])}

## Hidden2 – Hidden3

{'w': array([[ 0.43042537, 0.24417309, 0.66767853, 0.21959702, 0.25334494, 0.79619809, 1.35585083, -0.44406356],

[-0.74933034, 0.25570265, 0.99173936, -1.55416215, 0.14050354, 0.31603618, 0.67858859, -0.08951873],

[ 1.19377676, 0.50130037, -0.02564011, -0.2950911 , -1.93611709, 0.18541462, -1.31558414, 1.05947651],

[-0.29165532, 0.04375395, 0.14600168, 0.44920982, 0.74565785, 0.32411473, -0.23576766, 0.65416953]]),

'b': array([[-0.37707676], [ 0.76933052], [-1.00380583], [-1.29499124]])},

## Hidden3 – Hidden4

{'w': array([[-0.00866982, -0.93338459, -1.68580254, 0.6389882 ],

[-1.99846778, 0.79053448, 0.1671767 , -0.09714783]]),

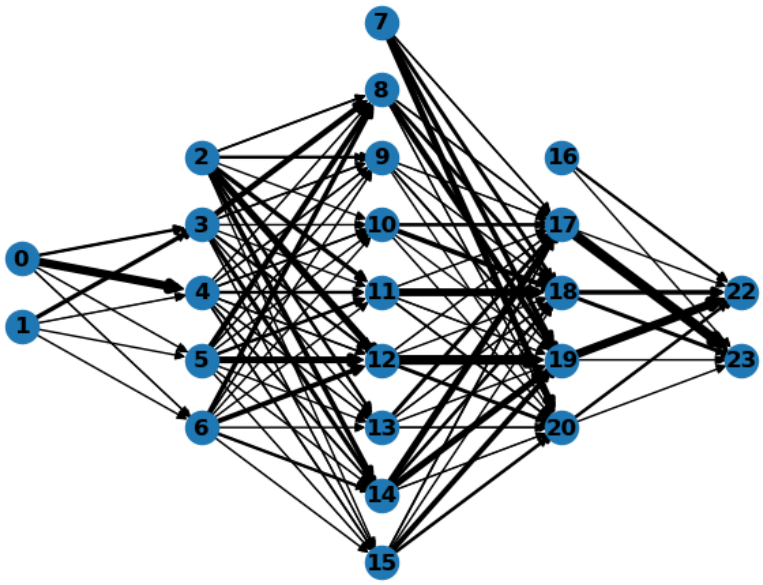
'b': array([[-0.57922942], [-0.00058854]])},

## Hidden4 – Output

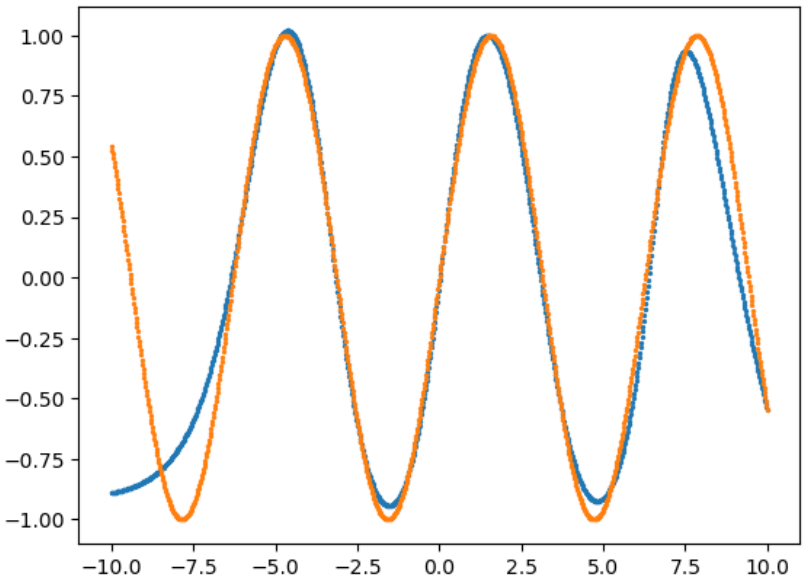
{'w': array([[ 5.35551942, -4.86250455]]),

'b': array([[-0.09226216]])}]

그러나 이는 시각적으로 확인하기 어렵기 때문에 이를 edge 의 width 로 사용하여 전체 network 를 graph 로 표현하였다. 0, 2, 7, 16 번 node 와 같이 각 layer 의 최상단에 있는 node 는 bias node 이다.




추가로 이 구현을 사용하여 sine 함수를 근사하였을 때에도 좋은 성능을 보임을 확인할 수 있었다.



Sine 함수 근사를 위하여 총 2000 epoch 동안 학습하였으며, 사용한 모델은 아래와 같다.

----------Sequential----------

Alpha : 0.001, Epochs : 2000

Cost Function : Mean Squared Error

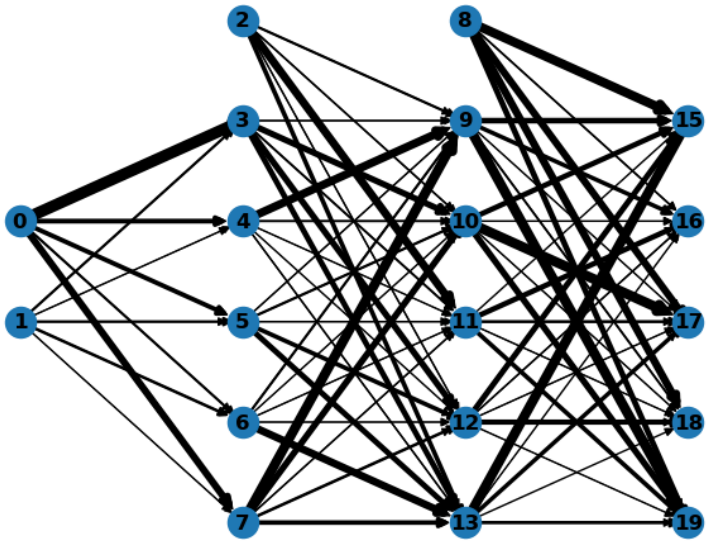
Input size : 1 Output size : 5 Activation : Hyperbolic Tangent

Input size : 5 Output size : 5 Activation : Hyperbolic Tangent

Input size : 5 Output size : 5 Activation : Hyperbolic Tangent

Input size : 5 Output size : 1 Activation : Identity

------------------------------



실행 방법

HW3.ipynb 를 jupyter notebook 이나 colab 에서 열고 하나씩 셀을 실행하면 됩니다. Numpy, matplotlib 이 설치되어 있어야합니다.

# 감사합니다