AI에 대한 수학적 이해

Jong Won

본 문서는 AI에 대한 수학적인 이해를 정리한다.

Contents

Abstract

Artificial Intelligence는 인공지능, 즉 컴퓨터로 인간이 지능을 사용하여 하는 일을 하도록 하는 것이 목적인 컴퓨터과학의 한 분야이다. 사람에게 질문을 하면 대답을 하고, 작업을 시키면 작업을 수행하는 것과 같이, 컴퓨터가 그런 활동을 모방하여 하도록 하는 것이다. 본 문서에서는 Neural Network의 기초적인 구조부터, 학습이 진행되는 방식과 수학적인 원리를 기술하고, 인공지능에 관련된 복잡한 구조와 수학적 이론들에 대해 다룬다.

1 Perceptron

퍼셉트론은 다수의 신호를 입력받아 하나의 신호를 출력하는 인공신경망을 구성하는 한 요소다.

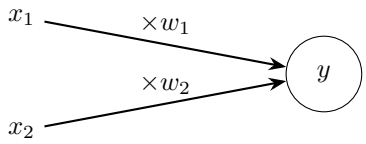

Figure 1: Perceptron

위 그림이 퍼셉트론 하나의 예시이다. 입력값 x_1, x_2을 그대로 더해서 출력값 y로 보내는 것은 큰 의미가 없다. 때문에 우린 가중치, weight를 곱하고, 편향 bias를 더하여 이를 출력 y로 보낼 것이다. 수식으로 나타내면 다음과 같다.

$$\begin{pmatrix} w_1 & w_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \end{pmatrix} = W \cdot X + B = y$$

퍼셉트론을 사용하여 컴퓨터가 x_1, x_2의 And, Or을 판별하는 상황을 생각해보자. x_1, x_2에 w_1, w_2를 곱하여 y로 보낸 다음, y가 특정 기준보다 크냐 작냐로 이를 판별할 수 있다.
편의를 위해 지금은 bias 행렬 B를 영행렬로 두겠다.

```python
def AndGate(x_1, x_2):
    weight_1 = 0.5
    weight_2 = 0.5
    criteria = 0.6
    if weight_1 * x_1 + weight_2 * x_2 >= criteria:
        return True
    else:
        return False
print(AndGate(True, False))
#False
print(AndGate(True, True))
#True
```

```python
def OrGate(x_1, x_2):
    weight_1 = 0.5
    weight_2 = 0.5
    criteria = 0.1
    if weight_1 * x_1 + weight_2 * x_2 >= criteria:
        return True
    else:
        return False
print(OrGate(False, False))
#False
print(OrGate(True, False))
#True
```

이와 같이 And나 Or 등의 연산은 입력값에 대해 어떤 가중치를 곱하고 더하는, 즉 선형 연산만으로 이 값들이 어떤 기준에 따라 참과 거짓이 구분되게 할 수 있다. 그런데 선형 연산으로 진리표를 만족시킬 수 없는 경우가 존재한다.

x_1	x_2	XOR
1	1	0
1	0	1
0	1	1
0	0	0

Table 1: XOR Gate Truth Table

어떤 선형변환과 상수 c가 존재하여

$$\begin{pmatrix} w_1 & w_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq c$$

를 위의 표와 같이 만족시킬 수 있는가? 이를 만족하려면 어떤 c와 $w_1, 2$가 존재하여 다음이 동시에 만족되어야 한다.

$$\begin{cases} w_1 + w_2 < c \\ w_1 \geq c \\ w_2 \geq c \\ 0 < c \end{cases}$$

자명하게 이는 불가능하다. bias가 추가되더라도 마찬가지이다. 이와 같이, 기존의 선형 연산만으론 XOR Gate와 같이 해결할 수 없는 상황이 존재한다. 선형연산은 몇번을 해도 여전히 선형이므로, 퍼셉트론을 늘린다고 해도 이 문제를 해결할 수는 없다. 때문에 우린 지금부터 비선형함수를 도입하여, 비선형 연산을 통해 우리가 원하는 값을 출력하는 비선형 함수를 만들 것이다.

신경망에 사용되는 비선형함수는 많은 종류가 있지만, 가장 많이 쓰이는 ReLU를 알아보자.

$$\mathcal{R}(x) = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

단순하게 0보다 크면 input을 그대로 내놓고, 아니면 0을 내놓는 함수이다. 이 단순한 비선형함수를 사용하여 우리는 신경망에 비선형성을 부여할 것이다. 이렇게 비선형성을 부여하기 위해 사용되는 함수들을 활성화함수라고 부르고, 이들은 퍼셉트론에서 출력되는 값을 입력으로 받아 퍼셉트론의 최종적인 출력값을 이 함수의 출력으로 내놓는다. 즉 활성화함수를 도입하여 기존의 퍼셉트론을 다음과 같이 바꿀 수 있다.

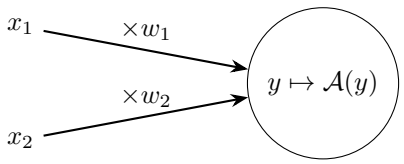

Figure 2: Perceptron with Activation function

수식적으론 다음과 같이 된다.

$$\mathcal{A}\left(\begin{pmatrix} w_1 & w_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \end{pmatrix}\right) = \mathcal{A}(W \cdot X + B) = \mathcal{A}(y)$$

이제 이 퍼셉트론을 사용해서 XOR Gate를 다시 구현해보자.

```python
def XorGate(x_1, x_2):
    weight_1 = 1.0
    weight_2 = -1.0
    weight_3 = -1.0
    weight_4 = 1.0
    weight_5 = 1.0
    weight_6 = 1.0
    y_1 = weight_1 * x_1 + weight_2 * x_2
    y_2 = weight_3 * x_1 + weight_4 * x_2
    y_3 = weight_5 * Relu(y_1) + weight_6 * Relu(y_2)
    if y_3 >= 0.5:
        print(True)
    else:
        print(False)

XorGate(1, 1)
# Fasle
XorGate(1, 0)
#True
XorGate(0, 1)
#True
XorGate(0, 0)
# Fasle
```

위 상황을 그림으로 표현하면 아래와 같다.

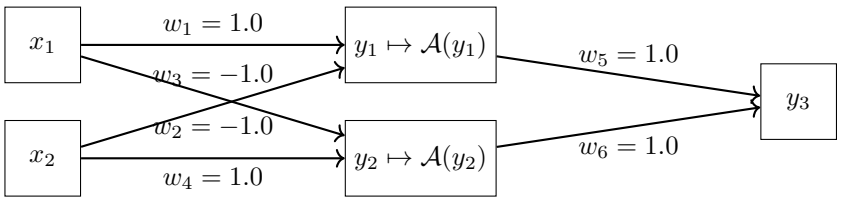

Figure 3: Perceptron network for XOR Gate.

이제 우린 아주 단순한 형태의 신경망을 만든 것이다. 이제 신경망의 일반화된 구조에 대해 구체적으로 알아본다.

2 Forward Propagation

위에서 퍼셉트론을 통해 단순한 신경망을 구성했다. 이제 이미지를 분류하는 신경망을 구성해보도록 하자. 8x8 크기의 흑백 이미지를 생각해보자.

이 이미지는 우리가 볼때는 이미지이지만, 컴퓨터가 받아들일때는 각 픽셀에 RGB값이 부여된 행렬이다.

$$\begin{pmatrix} 0 & 0 & 0 & 222 & 222 & 0 & 0 & 0 \\ 0 & 0 & 222 & 0 & 0 & 222 & 0 & 0 \\ 0 & 0 & 222 & 0 & 0 & 222 & 0 & 0 \\ 0 & 0 & 0 & 222 & 222 & 0 & 0 & 0 \\ 0 & 0 & 222 & 0 & 0 & 222 & 0 & 0 \\ 0 & 0 & 222 & 0 & 0 & 222 & 0 & 0 \\ 0 & 0 & 0 & 222 & 222 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

실제론 이렇게 되는 것이다. 이제 이 행렬을 신경망에 넣기 위해 평탄화flatten을 하자. (1행 2행 3행 ...) 꼴로 만든다고 하면 다음과 같이 된다. (어떻게 평탄화하든지 같은 규칙이면 상관없다.)

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 222 \\ 222 \\ \vdots \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

이런 64개의 정보를 입력받는 64개의 input node를 생각하자. (node는 perceptron과 같은 것으로 생각하면 된다.)

$$X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{61} \\ x_{62} \\ x_{63} \\ x_{64} \end{pmatrix}$$

이를 지금부터 0번째 layer로 생각하겠다.(왜 0번째로 부여하였는지는 이후 수식에서 나온다.)

이제 이런 64개의 노드를 1번째 layer들의 128개의 노드에 weight를 곱하고 bias를 더해 보내는 연산을 생각하자. 그림으로 보면 다음과 같다.

각 화살표들이 각각의 weight와 bias를 가져서 이를 곱하고 더하는 연산이다. 노드의 갯수가 l_{i-1} 개인 i-1번째 레이어의 j번째 노드를 그 다음 i번째 레이어의 k번째 노드로 보내는 weight를 $w^i_{k,j}$ 라 하면

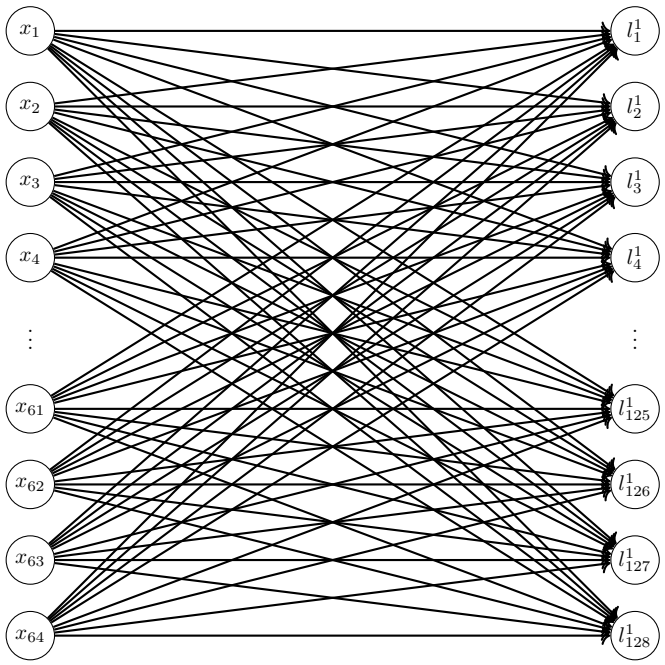

Figure 4: 64개의 입력 노드와 1번째 레이어의 128개의 노드

우린 이를 행렬 연산, 즉 선형변환으로 표현할 수 있고 이 행렬을 구체적으로 작성할 수 있다.

$$W_i = \left(w^i_{k,j}\right) = \begin{pmatrix} w^i_{1,1} & w^i_{1,2} & \cdots & w^i_{1,l_i-1} & w^i_{1,l_{i-1}} \\ w^i_{2,1} & w^i_{2,2} & \cdots & w^i_{2,l_i-1} & w^i_{2,l_{i-1}} \\ w^i_{3,1} & w^i_{3,2} & \cdots & w^i_{3,l_i-1} & w^i_{3,l_{i-1}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w^i_{l_i-2,1} & w^i_{l_i-2,2} & \cdots & w^i_{l_i-2,l_i-1} & w^i_{l_i-2,l_{i-1}} \\ w^i_{l_i-1,1} & w^i_{l_i-1,2} & \cdots & w^i_{l_i-1,l_i-1} & w^i_{l_i-1,l_{i-1}} \\ w^i_{l_i,1} & w^i_{l_i,2} & \cdots & w^i_{l_i,l_i-1} & w^i_{l_i,l_{i-1}} \end{pmatrix}$$

또, 노드의 갯수가 l_i 개인 i번째 레이어의 노드 각각에 bias를 더하는 행렬을

$$B_i = \begin{pmatrix} b^i_1 \\ b^i_2 \\ \vdots \\ b^i_{l_i-1} \\ b^i_{l_i} \end{pmatrix}$$

으로 정의하자.

위 상황에서 각 x_j 을 1번째 레이어로 보내는 weight 행렬은

$$W_1 = \begin{pmatrix} w^1_{1,1} & w^1_{1,2} & w^1_{1,3} & \cdots & w^1_{1,62} & w^1_{1,63} & w^1_{1,64} \\ w^1_{2,1} & w^1_{2,2} & w^1_{2,3} & \cdots & w^1_{2,62} & w^1_{2,63} & w^1_{2,64} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ w^1_{127,1} & w^1_{127,2} & w^1_{127,3} & \cdots & w^1_{127,62} & w^1_{127,63} & w^1_{127,64} \\ w^1_{128,1} & w^1_{128,2} & w^1_{128,3} & \cdots & w^1_{128,62} & w^1_{128,63} & w^1_{128,64} \end{pmatrix}$$

이제 1 번째 레이어는 다음과 같아진다.

$$W_1 \cdot X = \begin{pmatrix} w^1_{1,1} & w^1_{1,2} & w^1_{1,3} & \cdots & w^1_{1,62} & w^1_{1,63} & w^1_{1,64} \\ w^1_{2,1} & w^1_{2,2} & w^1_{2,3} & \cdots & w^1_{2,62} & w^1_{2,63} & w^1_{2,64} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ w^1_{127,1} & w^1_{127,2} & w^1_{127,3} & \cdots & w^1_{127,62} & w^1_{127,63} & w^1_{127,64} \\ w^1_{128,1} & w^1_{128,2} & w^1_{128,3} & \cdots & w^1_{128,62} & w^1_{128,63} & w^1_{128,64} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \cdots \\ x_{61} \\ x_{62} \\ x_{63} \\ x_{64} \end{pmatrix}$$

$$= \begin{pmatrix} w^1_{1,1} \cdot x_1 + w^1_{1,2} \cdot x_2 + \cdots + w^1_{1,63} \cdot x_{63} + w^1_{1,64} \cdot x_{64} \\ w^1_{2,1} \cdot x_1 + w^1_{2,2} \cdot x_2 + \cdots + w^1_{2,63} \cdot x_{63} + w^1_{2,64} \cdot x_{64} \\ w^1_{3,1} \cdot x_1 + w^1_{3,2} \cdot x_2 + \cdots + w^1_{3,63} \cdot x_{63} + w^1_{3,64} \cdot x_{64} \\ \vdots \\ w^1_{126,1} \cdot x_1 + w^1_{126,2} \cdot x_2 + \cdots + w^1_{126,63} \cdot x_{63} + w^1_{126,64} \cdot x_{64} \\ w^1_{127,1} \cdot x_1 + w^1_{127,2} \cdot x_2 + \cdots + w^1_{127,63} \cdot x_{63} + w^1_{127,64} \cdot x_{64} \\ w^1_{128,1} \cdot x_1 + w^1_{128,2} \cdot x_2 + \cdots + w^1_{128,63} \cdot x_{63} + w^1_{128,64} \cdot x_{64} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{64} w^1_{1,i} \cdot x_i \\ \sum_{i=1}^{64} w^1_{2,i} \cdot x_i \\ \sum_{i=1}^{64} w^1_{3,i} \cdot x_i \\ \vdots \\ \sum_{i=1}^{64} w^1_{126,i} \cdot x_i \\ \sum_{i=1}^{64} w^1_{127,i} \cdot x_i \\ \sum_{i=1}^{64} w^1_{128,i} \cdot x_i \end{pmatrix}$$

이제 여기에 bias 행렬 B_1을 더하면

$$W_1 \cdot X + B_1 = \begin{pmatrix} w^1_{1,1} & w^1_{1,2} & w^1_{1,3} & \dots & w^1_{1,62} & w^1_{1,63} & w^1_{1,64} \\ w^1_{2,1} & w^1_{2,2} & w^1_{2,3} & \dots & w^1_{2,62} & w^1_{2,63} & w^1_{2,64} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ w^1_{127,1} & w^1_{127,2} & w^1_{127,3} & \dots & w^1_{127,62} & w^1_{127,63} & w^1_{127,64} \\ w^1_{128,1} & w^1_{128,2} & w^1_{128,3} & \dots & w^1_{128,62} & w^1_{128,63} & w^1_{128,64} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \dots \\ x_{61} \\ x_{62} \\ x_{63} \\ x_{64} \end{pmatrix} + \begin{pmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \\ b^1_4 \\ \vdots \\ b^1_{125} \\ b^1_{126} \\ b^1_{127} \\ b^1_{128} \end{pmatrix}$$

$$= \begin{pmatrix} \sum_{i=1}^{64} w^1_{1,i} \cdot x_i + b^1_1 \\ \sum_{i=1}^{64} w^1_{2,i} \cdot x_i + b^1_2 \\ \sum_{i=1}^{64} w^1_{3,i} \cdot x_i + b^1_3 \\ \vdots \\ \sum_{i=1}^{64} w^1_{126,i} \cdot x_i + b^1_{126} \\ \sum_{i=1}^{64} w^1_{127,i} \cdot x_i + b^1_{127} \\ \sum_{i=1}^{64} w^1_{128,i} \cdot x_i + b^1_{128} \end{pmatrix} = L_1 \text{번째 레이어}$$

이제 마지막으로 활성화함수, ReLU를 사용하여 이 0번째 레이어에서 1번째 레이어로 보낸 뒤 최종적인 결과를 만들자.

Definition 1. *ReLU function* $\mathcal{R} : \mathbb{R} \to \mathbb{R}$

$$\mathcal{R}(x) = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

Definition 2. *Matrix ReLU :* $\mathcal{A} : \mathcal{M}_{n,m} \to \mathcal{M}_{n,m}$

$$\mathcal{A}(X) = \mathcal{A}\left(x_{i,j}\right) = \left(\mathcal{R}(x_{i,j})_{i,j}\right)$$

즉 elementwise하게 ReLU function을 적용한 것이다. 이제 최종적으로 우리가 원하는 1번째 레이

어의 결과는

$$\mathcal{A}(W_1 \cdot X + B_1) = \begin{pmatrix} \mathcal{R}(\sum_{i=1}^{64} w_{1,i}^1 \cdot x_i + b_1^1) \\ \mathcal{R}(\sum_{i=1}^{64} w_{2,i}^1 \cdot x_i + b_2^1) \\ \mathcal{R}(\sum_{i=1}^{64} w_{3,i}^1 \cdot x_i + b_3^1) \\ \vdots \\ \mathcal{R}(\sum_{i=1}^{64} w_{126,i}^1 \cdot x_i + b_{126}^1) \\ \mathcal{R}(\sum_{i=1}^{64} w_{127,i}^1 \cdot x_i + b_{127}^1) \\ \mathcal{R}(\sum_{i=1}^{64} w_{128,i}^1 \cdot x_i + b_{128}^1) \end{pmatrix}$$

이 된다. 이제 input layer X 로부터 n번째 layer L_n 를 다음과 같이 표현할 수 있다.

$$\begin{cases} L_0 = X \\ L_n = W_n \cdot \mathcal{A}(L_{n-1}) + B_n \quad (n \geq 1) \end{cases}$$

이렇게 input에 대해 weight를 곱하고 bias를 더해서 계속해서 다음 레이어를 구성할 수 있다. input layer, hidden layer 2개 층, output layer 총 4개 층으로 이뤄진 신경망을 구성해보자.

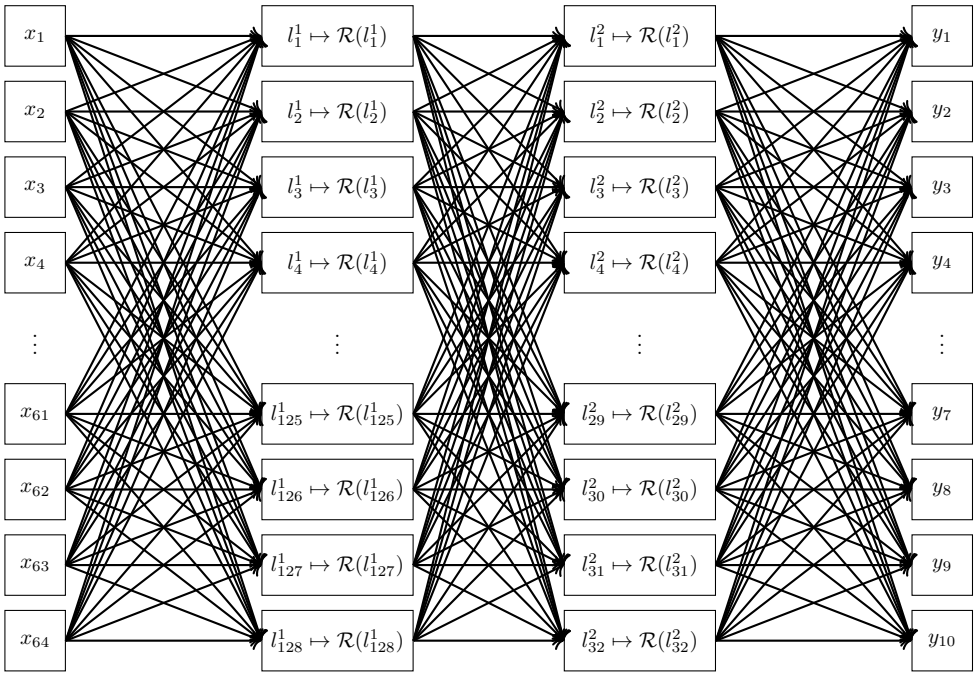

Figure 5: Neural network with 64 input nodes, 128 nodes in the first hidden layer, 32 nodes in the second hidden layer, and 10 output nodes.

위와 같이 64개의 scalar를 입력받아 10개의 scalar를 출력하는 신경망이 구성되었다. 마지막 레이어의 $y_1, y_2, \ldots, y_{10}$ 을 확률 분포로 바꾸는 함수를 거쳐 최종적으로 출력하면, 우린 어떤 이미지를 입력받아 그 이미지가 어떤 함수일지 내놓는 신경망을 구성한 것이다.

이렇게 input matrix에 대해 weight를 곱하고 bias를 더하고, 활성화함수를 통해 비선형성을 부여하는 과정을 반복하여 최종적인 결과값을 내놓는 과정을 순전파Forward Propagation이라 부른다. 위

의 수식적인 표현을 보면 알 수 있지만, 신경망은 결국 input을 다차원으로 받아서 ouput을 다차원으로 내놓는 함수이다.

이제 우리는 input에 대해 우린 여러개의 Layer를 겹쳐서 복잡한 신경망을 구성하고, 순전파를 통해 원하는 차원의 값을 내놓을 수 있다. 그런데 이렇게 구성된 신경망이 우리가 원하는 결과를 도출하냐는 다른 문제이다. 우리가 원하는 결과값을 내도록 하기 위해서는 weight matrix와 bias matrix를 우리가 원하는 행렬로써 찾아야 한다. 그 과정을 역전파Back Propagation이라 부르고, 이것이 우리가 학습이라 부르는 과정이다.

3 Back Propagation

역전파를 시작하기 전에, 경사하강법Gradient descent에 대해 알아보겠다.

Gradient descent

Gradient descent는 함수와 초깃값이 주어졌을 때, 가장 가까운 극값을 찾기 위해 함수의 gradient를 이용하는 방법이다. 아래와 같은 함수를 생각하자.

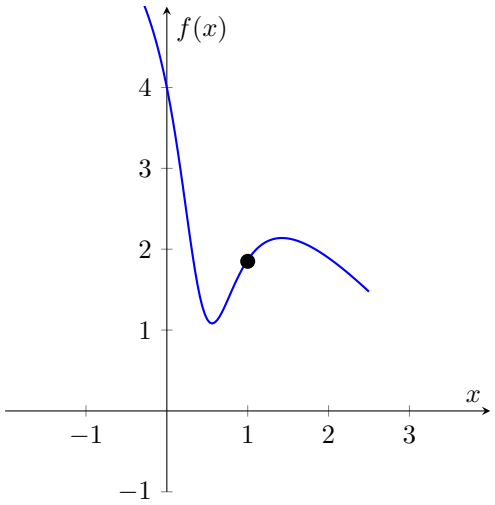

Figure 6: Unknown function

연속이고 매끄러운 적당한 함수 $f(x)$가 주어지고, 이 함수의 미분을 알고 있을 때 어떻게 임계점의 x좌표를 찾을 수 있을까? 당연히 미분계수가 0일때가 임계점이 되겠지만, 미분에 어떤 값을 넣어야 0이 되는지를 찾아야 하는 상황인 것이다. 모든 값을 다 넣어봐서 언제 0이 되는지 확인하면 좋겠지만, 이는 계산량이 몹시 많이 필요하다. 때문에 임계점만을 보다 적은 연산량으로 찾기 위해 Gradient descent라 불리는 수치적인 최적화 방법을 사용한다.

Gradient descent의 개념은 간단하다. 무작위로 혹은 어떤 기준에 의해 주어진 x 값에서, 함수의 gradient를 계산한 뒤 미리 정해놓은 step size h에 대해 gradient 벡터에 h를 곱한만큼 이동하는 과정을 반복하여 임계점의 x좌표에 다가가겠다는 것이다.

gradient라 해서 이해가 잘 가지 않을 수 있는데, 위와 같은 $\mathbb{R} \to \mathbb{R}$ 상황에서는 gradient가 곧 기울기다. 즉 위 그림에서 주어진 점에서의 기울기가 양수이므로, 그 기울기에 어떤 작은 양수값 h를 곱한 뒤 기존의 x 값에서 빼면 기울기 $\cdot h$ 만큼 음의 방향으로 x좌표값이 변하게 되는 것이므로, step size h를 적당한 크기로 설정했다면 이 과정을 반복함에 따라 임계점에 다가가게 될 것이다.

간단한 파이썬 예제로 임의의 일변수함수에 대해 임계점의 x좌표를 찾는 예제 코드를 살펴보자.

```python
def unknown_function(x):
    y = x ** 2 - 2 * x + 1
    return y
#have local minimum where x=1.

def derivative_function(x):
    y = 2 * x - 2
    return y

step_size = 0.001

def gradient_descent_step(function, derivative_function, x):
    next_x = x - derivative_function(x) * step_size
    return next_x

x = 10
for _ in range(1000):
    x = gradient_descent_step(unknown_function, derivative_function, x)
print(x)
# 2.215580702020154

x = 10
for _ in range(10000):
    x = gradient_descent_step(unknown_function, derivative_function, x)
print(x)
# 1.0000000181825743
```

위 코드에서 볼 수 있듯이, step을 많이 반복할 수록 실제 임계점 값에 가까워지고, 보다 더 정확한 값을 찾기 위해선 step size를 더 작게 조절함으로써 원하는 값에 더 근사할 수 있다. 물론 step size를 줄이면 줄인만큼 작은 거리를 이동하게 되므로 보다 많은 step을 반복해야 한다. 이런 문제를 해결하기 위해 Gradient descent를 변형한 여러 최적화 기법들이 사용되고 있지만, 이는 지금 하기엔 어려운 내용이므로 나중에 알아본다.

Gradient descent를 왜 신경망에서 사용해야 할까? 그걸 알아보기 위해 먼저 Loss function을 알아보자. 앞서 말한 것처럼, 신경망은 어떤 input값에 대해 output을 내놓는 함수다. 하지만 우린 input값에 대해 아무 의미없는 output을 내놓는걸 원하지 않는다. 어떤 input에 대해 특정한 output이 나오길 원한다. 예를 들면 신경망에 위와 같은 8 이미지를 넣었을 때 이 이미지가 8일 확률을 출력하기를 바라는 것이다. 처음에 무작위로 생성한 weight matrix와 bias matrix로 구성된 신경망이 input에 대해 우리가 원하는 output을 내놓지는 않는다. 때문에 우린 우리가 원하는 정답 label을 만들고, 그 label과 신경망의 output 사이의 차이, Loss를 계산한 뒤에 그 Loss를 최소화하도록 하기 위해 Gradient descent를 사용하게 되는 것이다. 이 계산에서 사용되는게 Loss function이다.

Derivative of Neural Network

앞에서 Gradient descent에 대해 알아보았다. 그런데 Gradient descent를 사용하기 위해선 현재 값에서의 함수의 미분을 알아야 한다. 그런데 위에서 정의한 신경망과 같은 함수에서 미분을 어떻게 구할 수 있을까?

지금부터 신경망의 미분을 구하기 위해, 구체적으로 정의하고 시작한다.

Definition 1. 노드의 갯수가 l_{i-1} 개인 *i-1*번째 레이어의 k번째 노드를 그 다음 i번째 레이어의 j번째 노드로 보내는 *weight matrix:*

$$W_i = \left(w^i_{j,k}\right) = \begin{pmatrix} w^i_{1,1} & w^i_{1,2} & \cdots & w^i_{1,l_{i-1}-1} & w^i_{1,l_{i-1}} \\ w^i_{2,1} & w^i_{2,2} & \cdots & w^i_{2,l_{i-1}-1} & w^i_{2,l_{i-1}} \\ w^i_{3,1} & w^i_{3,2} & \cdots & w^i_{3,l_{i-1}-1} & w^i_{3,l_{i-1}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w^i_{l_i-2,1} & w^i_{l_i-2,2} & \cdots & w^i_{l_i-2,l_{i-1}-1} & w^i_{l_i-2,l_{i-1}} \\ w^i_{l_i-1,1} & w^i_{l_i-1,2} & \cdots & w^i_{l_i-1,l_{i-1}-1} & w^i_{l_i-1,l_{i-1}} \\ w^i_{l_i,1} & w^i_{l_i,2} & \cdots & w^i_{l_i,l_{i-1}-1} & w^i_{l_i,l_{i-1}} \end{pmatrix} \in \mathcal{M}_{l_i,l_{i-1}}$$

Definition 2. 노드의 갯수가 l_i 개인 i번째 레이어의 노드에 *bias*를 더하는 *bias matrix:*

$$B_i = \begin{pmatrix} b^i_1 \\ b^i_2 \\ \vdots \\ b^i_{l_i-1} \\ b^i_{l_i} \end{pmatrix} \in \mathcal{M}_{l_i,1}$$

Definition 3. *Activation function*(*ReLU*) $\mathcal{R} : \mathbb{R} \to \mathbb{R}$

$$\mathcal{R}(x) = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases} = max(x, 0)$$

Definition 4. *Matrix ReLU :* $\mathcal{A} : \mathcal{M}_{n,m} \to \mathcal{M}_{n,m}$

$$\mathcal{A}(X) = \mathcal{A}\left(x_{i,j}\right) = \left(\mathcal{R}(x_{i,j})_{i,j}\right)$$

Definition 5. *Output function*(*Softmax*) $\mathcal{F}$: $\mathcal{M}_{n,1} \to \mathcal{M}_{n,1}$

$$\mathcal{F}(Y) = \begin{pmatrix} \exp(Y_{1,1}) \\ \exp(Y_{2,1}) \\ \vdots \\ \exp(Y_{n-1,1}) \\ \exp(Y_{n,1}) \end{pmatrix} \cdot \frac{1}{\sum_{i=1}^{n}(\exp Y_{i,1})}$$

Definition 6. *Loss function*(*MSE*) $\mathcal{L} : (\mathcal{M}_{n,1}, \mathcal{M}_{n,1}) \to \mathbb{R}$

$$\mathcal{L}(Y, T) = \frac{1}{n} \cdot \sum_{i=1}^{n}(Y_i - T_i)^2$$

Definition 7. *layer*가 $n+1$ 개인 신경망의 i 번째 *layer:*

$$\begin{cases} L_0 = X \in \mathcal{M}_{l_0,1} \\ L_1 = W_1 \cdot L_0 + B_1 \in \mathcal{M}_{l_1,1} \\ L_i = W_i \cdot \mathcal{A}(L_{i-1}) + B_i \in \mathcal{M}_{l_i,1} \quad (2 \leq i \leq n) \end{cases}$$

Definition 8. ***Feedforward Neural Network*** $\mathcal{N}$***:*** $\mathbb{F}^{l_0+\sum_{i=1}^{n} l_i \times l_{i-1}+\sum_{i=1}^{n} l_i \times 1} \to \mathbb{F}^{l_n}$

$$\mathcal{N}(X, W_1, \ldots, W_n, B_1, \ldots, B_n) = \mathcal{F}(L_n) = Y$$

이제 정의는 끝났다. Definition 8에서 정의한 것과 같이, 신경망은 input X, Weight matrix W, Bias matrix B, Activation function $\mathcal{A}$, Output function $\mathcal{F}$의 결합으로 이뤄진 다변수함수이다. 우린 input에 대해 우리가 원하는 output이 나오도록 Weight와 Bias를 조절해야 하고, 이를 하기 위해 Gradient descent를 사용하는 것이다. 이제 Gradient descent를 적용하기 위해, 구체적으로 우리가 어떤 함수에 대한 미분을 구해야하는지 알아보자.

우리는 input에 따른 output이 Loss function에 정답 label과 함께 들어갔을 때, 즉 output Y와 정답 T_X에 대해 $\mathcal{L}(Y, T_X)$의 값이 최소가 되게 하는 Weight와 Bias를 찾고 싶은 것이다. Loss function의 정의에 따라 이 함수값이 최소가 될 때가 output Y와 정답 T_X의 오차가 가장 적은 상태일 것이고, 즉 우리가 바라는 상황이 된다. (우리가 Gradient descent로 찾은 극솟값이 사실 최소가 아닐 수도 있다. Gradient descent는 주어진 초깃값에서 가까운 임계점을 찾는 것이지, domain 전체에서의 최소를 구하는게 아니다. Global minimum을 찾는 것은 많이 어려운 작업이다. 때문에 우린 Local minimum을 우선적으로 찾을 것이고, 신경망이 깊어질수록 Local minimum을 찾기만 해도 충분해진다. 이에 대한 이론적인 설명은 section 6 Artificial Intelligence Theory에서 다룬다.) 이제 Gradient descent를 적용하기 위해 각 weight와 bias에 대한 $\mathcal{L}(Y, T_X)$의 미분을 구해보자.

먼저 신경망을 수식적으로 바라보기 위해, layer가 3개인 신경망을 구성해보자.

$$
\begin{aligned}
&\mathcal{N}(X, W_1, W_2, B_1, B_2) \\
&= \mathcal{F}(W_2 \cdot \mathcal{A}(W_1 \cdot X + B_1) + B_2) = \mathcal{F}(W_2 \cdot \mathcal{A}(L_1) + B_2) = \mathcal{F}(L_2) \\
&= \mathcal{F}\left(\begin{pmatrix} w^2_{1,1} & \cdots & w^2_{1,l_1} \\ \vdots & \ddots & \vdots \\ w^2_{l_2,1} & \cdots & w^2_{l_2,l_1} \end{pmatrix} \cdot \mathcal{A}\left(\begin{pmatrix} w^1_{1,1} & \cdots & w^1_{1,l_0} \\ \vdots & \ddots & \vdots \\ w^1_{l_1,1} & \cdots & w^1_{l_1,l_0} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_{l_0} \end{pmatrix} + \begin{pmatrix} b^1_1 \\ \vdots \\ b^1_{l_1} \end{pmatrix}\right) + \begin{pmatrix} b^2_1 \\ \vdots \\ b^2_{l_2} \end{pmatrix}\right) \\
&= \mathcal{F}\left(\begin{pmatrix} \sum_{j=1}^{l_1} w^2_{1,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_1 \\ \sum_{j=1}^{l_1} w^2_{2,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_2 \\ \vdots \\ \sum_{j=1}^{l_1} w^2_{l_2,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_{l_2} \end{pmatrix}\right) \\
&= \begin{pmatrix} \dfrac{\exp(\sum_{j=1}^{l_1} w^2_{1,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_1)}{\sum_{k=1}^{l_2} \exp(\sum_{j=1}^{l_1} w^2_{k,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_1)} \\ \dfrac{\exp(\sum_{j=1}^{l_1} w^2_{2,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_2)}{\sum_{k=1}^{l_2} \exp(\sum_{j=1}^{l_1} w^2_{k,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_1)} \\ \vdots \\ \dfrac{\exp(\sum_{j=1}^{l_1} w^2_{l_2,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_{l_2})}{\sum_{k=1}^{l_2} \exp(\sum_{j=1}^{l_1} w^2_{k,j} \cdot \mathcal{R}(\sum_{i=1}^{l_0} w^1_{j,i} \cdot x_i + b^1_1) + b^2_1)} \end{pmatrix}
\end{aligned}
$$

이제 이 신경망을 Loss function(MSE)에 넣으면 최종적으로

$$\mathcal{L}(Y,T_X)=\mathcal{L}\left(\begin{pmatrix}\dfrac{\exp(\sum_{j=1}^{l_1}w^2_{1,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_1)}{\sum_{k=1}^{l_2}\exp(\sum_{j=1}^{l_1}w^2_{k,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_1)}\\ \dfrac{\exp(\sum_{j=1}^{l_1}w^2_{2,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_2)}{\sum_{k=1}^{l_2}\exp(\sum_{j=1}^{l_1}w^2_{k,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_1)}\\ \vdots\\ \dfrac{\exp(\sum_{j=1}^{l_1}w^2_{l_2,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_{l_2})}{\sum_{k=1}^{l_2}\exp(\sum_{j=1}^{l_1}w^2_{k,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_1)}\end{pmatrix},T_X\right)$$

$$=\frac{1}{l_2}\cdot\sum_{s=1}^{l_2}\left(\left(\frac{\exp(\sum_{j=1}^{l_1}w^2_{s,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_1)}{\sum_{k=1}^{l_2}\exp(\sum_{j=1}^{l_1}w^2_{k,j}\cdot\mathcal{R}(\sum_{i=1}^{l_0}w^1_{j,i}\cdot x_i+b^1_1)+b^2_1)}\right)-T_s\right)^2$$

이제 함수 $\mathcal{L}$은 명백하게 다변수에서 스칼라로 가는 함수가 되었고, $w^i_{k,j}$와 b^i_k에 영향받는 함수임을 확인할 수 있다. 하지만 위 수식으로부터 각 변수에 대한 미분을 구해내는건 몹시 까다로운 일이다. 위 식은 layer가 늘어남에 따라 훨씬 더 복잡해진다. 때문에 연쇄법칙을 사용한다. 지금부터 다변수에서의 연쇄법칙Chain rule을 유도한다. 다변수 미분에 대한 정의는 Appendix를 참고하라.

$\mathcal{L}(Y,T_X)=\mathcal{L}\circ\mathcal{F}(L_n)$ 이를 L_n으로 미분하면

$\dfrac{\partial\mathcal{L}}{\partial L_n}=\dfrac{\partial\mathcal{L}}{\partial\mathcal{F}}\cdot\dfrac{\partial\mathcal{F}}{\partial L_n}$ 이 된다.

왜냐하면, 아래 수식을 보라.

$$\frac{\partial \mathcal{L}}{\partial L_n} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial L_1^n} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial L_{l_n}^n} \end{pmatrix}$$

이때 연쇄법칙에 따라, 각각의 원소들은 다음과 같이 다시 쓰여진다.

$$= \begin{pmatrix} \sum_{i=1}^{l_n} \frac{\partial \mathcal{L}}{\partial \mathcal{F}_i} \cdot \frac{\partial \mathcal{F}_i}{\partial L_1^n} \\ \vdots \\ \sum_{i=1}^{l_n} \frac{\partial \mathcal{L}}{\partial \mathcal{F}_i} \cdot \frac{\partial \mathcal{F}_i}{\partial L_{l_n}^n} \end{pmatrix}$$

그리고 이는 사실 다음과 같고

$$= \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial \mathcal{F}_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \mathcal{F}_{l_n}} \end{pmatrix}^T \cdot \begin{pmatrix} \frac{\partial \mathcal{F}_1}{\partial L_1^n} & \cdots & \frac{\partial \mathcal{F}_1}{\partial L_{l_n}^n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{F}_{l_n}}{\partial L_1^n} & \cdots & \frac{\partial \mathcal{F}_{l_n}}{\partial L_{l_n}^n} \end{pmatrix}$$

결론적으로 이 수식들이 아래와 같아진다.

$$= \frac{\partial \mathcal{L}}{\partial \mathcal{F}} \cdot \frac{\partial \mathcal{F}}{\partial L_n}$$

이처럼 다변수에서도 우리가 알고 있는 연쇄법칙이 성립한다. 우리는 지금 마지막 layer에 대한 Loss function과 output fucntion의 합성함수의 편미분을 구한 것이다.
이제 이 편미분을 연쇄로 잇고 이어서 우리가 애초에 원하던 Weight에 대한 편미분을 구할 것이다.

이제 $(2 \leq i \leq n)$ 인 i 들에 대해 다음을 구하자.

$$\frac{\partial L_i}{\partial L_{i-1}} = \begin{pmatrix} \frac{\partial L_1^i}{\partial L_1^{i-1}} & \cdots & \frac{\partial L_1^i}{\partial L_{l_{i-1}}^{i-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L_{l_i}^i}{\partial L_1^{i-1}} & \cdots & \frac{\partial L_{l_i}^i}{\partial L_{l_{i-1}}^{i-1}} \end{pmatrix} \in \mathcal{M}_{l_i, l_{i-1}}$$

이때 아래를 상기하라.

$$L_i = W_i \cdot \mathcal{A}(L_{i-1}) + B_i \in \mathcal{M}_{l_i,1} \quad (n \geq i \geq 2)$$

여기서 위 layer를 아래와 같이 다시 쓸 수 있다.

$$L_i = \begin{pmatrix} L_1^i \\ L_2^i \\ \vdots \\ L_{l_i}^i \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{l_{i-1}} w_{1,j}^i \cdot \mathcal{R}(L_j^{i-1}) + B_1^i \\ \vdots \\ \sum_{j=1}^{l_{i-1}} w_{l_i,j}^i \cdot \mathcal{R}(L_j^{i-1}) + B_{l_i}^i \end{pmatrix}$$

이제 Jacobian[2] 의 정의에 따라 위 편미분을 직접 계산하면

$$\frac{\partial L_i}{\partial L_{i-1}} = \begin{pmatrix} w^i_{1,1} \cdot \frac{d\mathcal{R}}{dx}(L^{i-1}_1) & \dots & w^i_{1,l_{i-1}} \cdot \frac{d\mathcal{R}}{dx}(L^{i-1}_{l_{i-1}}) \\ \vdots & \ddots & \vdots \\ w^i_{l_i,1} \cdot \frac{d\mathcal{R}}{dx}(L^{i-1}_1) & \dots & w^i_{l_i,l_{i-1}} \cdot \frac{d\mathcal{R}}{dx}(L^{i-1}_{l_{i-1}}) \end{pmatrix} \in \mathcal{M}_{l_i,l_{i-1}}$$

그런데 사실 이는 아래와 같고

$$= \begin{pmatrix} w^i_{1,1} & \dots & w^i_{1,l_{i-1}} \\ \vdots & \ddots & \vdots \\ w^i_{l_i,1} & \dots & w^i_{l_i,l_{i-1}} \end{pmatrix} \cdot \begin{pmatrix} \frac{d\mathcal{R}}{dx}(L^{i-1}_1) & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \frac{d\mathcal{R}}{dx}(L^{i-1}_{l_{i-1}}) \end{pmatrix}$$

이는 결국 다음과 같아진다.

$$= \frac{\partial L_i}{\partial \mathcal{A}} \cdot \frac{\partial \mathcal{A}}{\partial L_{i-1}} = W_i \cdot diag(\frac{d\mathcal{R}}{dx}(L^{i-1}_1), \frac{d\mathcal{R}}{dx}(L^{i-1}_2), \dots, \frac{d\mathcal{R}}{dx}(L^{i-1}_{l_{i-1}}))$$

이제 우린 임의의 i번째 레이어 L_i 에 대해 $\mathcal{L}$의 변화율 $\frac{\partial \mathcal{L}}{\partial L_i}$ 을 다음과 같이 구할 수 있다.

$$\frac{\partial \mathcal{L}}{\partial L_i} = \frac{\partial \mathcal{L}}{\partial \mathcal{F}} \cdot \frac{\partial \mathcal{F}}{\partial L_n} \cdot \frac{\partial L_n}{\partial L_{n-1}} \cdots \frac{\partial L_{i+1}}{\partial L_i}$$

이제 마지막 단계다. $\frac{\partial L_i}{\partial W_i}$ 을 구하자.

Matrix에서의 Jacobian에 대한 아래 정의[3] 에 따라, 다음과 같이 쓰여진다.

$$\frac{\partial L_i}{\partial W_i} = \begin{pmatrix} \frac{\partial L_i}{\partial W^i_{1,}} \\ \vdots \\ \frac{\partial L_i}{\partial W^i_{l_i,}} \end{pmatrix}$$

$$\text{where } \frac{\partial L_i}{\partial W^i_{j,}} = \begin{pmatrix} 0 & 0 & \dots & 0 & 0 & \\ \vdots & \vdots & & \vdots & \vdots & \\ \mathcal{R}(L^{i-1}_1) & \mathcal{R}(L^{i-1}_2) & \dots & \mathcal{R}(L^{i-1}_{l_{i-2}}) & \mathcal{R}(L^{i-1}_{l_{i-1}}) & \leftarrow j\text{번째 row} \\ \vdots & \vdots & & \vdots & \vdots & \\ 0 & 0 & \dots & 0 & 0 & \end{pmatrix} \in \mathcal{M}_{l_i,l_{i-1}}$$

비로소 우리가 원하는걸 얻는다.

Proposition 1. i번째 *Weight*의 *Gradient:*

$$\frac{\partial \mathcal{L}}{\partial W_i} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_{1,1}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{1,l_{i-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{l_i,1}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{l_i,l_{i-1}}} \end{pmatrix} \in \mathcal{M}_{l_i,l_{i-1}}$$

$$\begin{aligned} &= \frac{\partial \mathcal{L}}{\partial \mathcal{F}} \cdot \frac{\partial \mathcal{F}}{\partial L_n} \cdot \frac{\partial L_n}{\partial L_{n-1}} \cdots \frac{\partial L_{i+1}}{\partial L_i} \cdot \frac{\partial L_i}{\partial W_i} \\ &= \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial \mathcal{F}_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \mathcal{F}_{l_n}} \end{pmatrix}^T \cdot \begin{pmatrix} \frac{\partial \mathcal{F}_1}{\partial L_1^n} & \cdots & \frac{\partial \mathcal{F}_1}{\partial L_{l_n}^n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{F}_{l_n}}{\partial L_1^n} & \cdots & \frac{\partial \mathcal{F}_{l_n}}{\partial L_{l_n}^n} \end{pmatrix} \\ &\cdot W_n \cdot \mathrm{diag}(\frac{d\mathcal{R}}{dx}(L_1^{n-1}), \frac{d\mathcal{R}}{dx}(L_2^{n-1}), \ldots, \frac{d\mathcal{R}}{dx}(L_{l_{n-1}}^{n-1})) \\ &\cdot W_{n-1} \cdot \mathrm{diag}(\frac{d\mathcal{R}}{dx}(L_1^{n-2}), \frac{d\mathcal{R}}{dx}(L_2^{n-2}), \ldots, \frac{d\mathcal{R}}{dx}(L_{l_{n-2}}^{n-2})) \\ &\qquad\qquad \vdots \\ &\cdot W_{i+1} \cdot \mathrm{diag}(\frac{d\mathcal{R}}{dx}(L_1^i), \frac{d\mathcal{R}}{dx}(L_2^i), \ldots, \frac{d\mathcal{R}}{dx}(L_{l_i}^i)) \\ &\cdot \begin{pmatrix} \frac{\partial L_i}{\partial W_{1,}^i} \\ \vdots \\ \frac{\partial L_i}{\partial W_{l_i,}^i} \end{pmatrix} \end{aligned}$$

이제 우린 Weight의 미분을 구체적으로 알게 되었다. 복잡한 함수이지만, 결국 우리가 그 미분을 아는 비선형함수와 선형변환의 결합이기 때문에 미적분학의 기본적인 법칙인 연쇄법칙과, 기초적인 선형대수만으로 신경망의 미분을 알아낸 것이다. 마찬가지의 방법으로 Bias의 미분도 구할 수 있다.

Proposition 2. i번째 *Bias*의 *Gradient:*

$$\frac{\partial \mathcal{L}}{\partial B_i} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_{1,1}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{1,l_{i-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{l_i,1}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{l_i,l_{i-1}}} \end{pmatrix} \in \mathcal{M}_{l_i,l_{i-1}}$$

$$\begin{aligned} &= \frac{\partial \mathcal{L}}{\partial \mathcal{F}} \cdot \frac{\partial \mathcal{F}}{\partial L_n} \cdot \frac{\partial L_n}{\partial L_{n-1}} \cdots \frac{\partial L_{i+1}}{\partial L_i} \cdot \frac{\partial L_i}{\partial B_i} \\ &= \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial \mathcal{F}_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \mathcal{F}_{l_n}} \end{pmatrix}^T \cdot \begin{pmatrix} \frac{\partial \mathcal{F}_1}{\partial L_1^n} & \cdots & \frac{\partial \mathcal{F}_1}{\partial L_{l_n}^n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{F}_{l_n}}{\partial L_1^n} & \cdots & \frac{\partial \mathcal{F}_{l_n}}{\partial L_{l_n}^n} \end{pmatrix} \\ &\cdot W_n \cdot \mathrm{diag}(\frac{d\mathcal{R}}{dx}(L_1^{n-1}), \frac{d\mathcal{R}}{dx}(L_2^{n-1}), \ldots, \frac{d\mathcal{R}}{dx}(L_{l_{n-1}}^{n-1})) \\ &\cdot W_{n-1} \cdot \mathrm{diag}(\frac{d\mathcal{R}}{dx}(L_1^{n-2}), \frac{d\mathcal{R}}{dx}(L_2^{n-2}), \ldots, \frac{d\mathcal{R}}{dx}(L_{l_{n-2}}^{n-2})) \\ &\qquad\qquad \vdots \\ &\cdot W_{i+1} \cdot \mathrm{diag}(\frac{d\mathcal{R}}{dx}(L_1^i), \frac{d\mathcal{R}}{dx}(L_2^i), \ldots, \frac{d\mathcal{R}}{dx}(L_{l_i}^i)) \cdot I_{l_i} \end{aligned}$$

마지막에 항등행렬이 붙는건 $\frac{\partial L_i}{\partial B_i}$ 이 결국 항등행렬이 되기 때문이다(실제로 그런지는 직접 계산해 보라).

이제 우린 우리가 원하는 Weight, Bias의 미분을 얻었고, 역전파를 할 수 있다. 다음을 반복하면 된다.

$$W_i^+ = W_i - \frac{\partial \mathcal{L}}{\partial W_i} \cdot \mathrm{step\_size}$$
$$B_i^+ = B_i - \frac{\partial \mathcal{L}}{\partial B_i} \cdot \mathrm{step\_size}$$

각 i들에 대해 위 과정을 반복하면, Gradient descent를 통해 우리가 원하는 Loss값의 극솟값을 찾게 될 것이고, 이것이 신경망의 output과 우리가 원하는 정답 target의 차이가 극소가 될 때이다. 이제 코드를 구현하여 이미지를 넣고 그 이미지가 어떤 이미지인지 추정하는 확률 분포를 학습시켜 보자.

4 Construct Code

필요한 함수들을 아래와 같이 정의하고 사용한다.

```python
def ReLU(x):
    return np.maximum(x, 0)

def ReLU_Derivative(x):
    return np.where(x > 0, 1, 0)

def softmax(L):
    exp_L = np.exp(L - np.max(L))
    softmax_values = exp_L / np.sum(exp_L)
    return softmax_values

def softmax_derivative(L):
    s = softmax(L).reshape(-1, 1)
    return np.diagflat(s) - s @ s.T

def MSE(Y, T):
    return np.mean((Y - T) ** 2)

def mse_derivative(Y, T):
    n = len(Y)
    return (2 / n) * (Y - T)

def nthnrowmatrix(L_i, L_i_minus_1):
    l_i = len(L_i)
    l_i_minus_1 = len(L_i_minus_1)
    jacobian_matrix = np.zeros((l_i, l_i, l_i_minus_1))

    for j in range(l_i):
        jacobian_matrix[j, j, :] = ReLU(L_i_minus_1).flatten()
    return jacobian_matrix
```

```python
import numpy as np

class Neural_Net:
    def __init__(self, layer_size):
        self.layer_number = len(layer_size)
        self.W = ['Weight']
        for i in range(self.layer_number - 1):
            self.W.append(0.01 * np.random.randn(layer_size[i + 1], layer_size[
                i]))
        self.B = ['Bias']
        for i in range(self.layer_number - 1):
            self.B.append(0.01 * np.random.randn(layer_size[i + 1], 1))

        self.h = 0.001  # step size

    def forward_propagation(self, input):
        self.L = ['Unassigned'] * self.layer_number
        X = np.array(input).reshape(len(input), 1)
        self.L[0] = X
        self.L[1] = self.W[1] @ self.L[0] + self.B[1]
        for i in range(2, self.layer_number):
            self.L[i] = self.W[i] @ ReLU(self.L[i - 1]) + self.B[i]
        return softmax(self.L[-1])

    def back_propagation(self, input, target):
        N = self.layer_number - 1
        Y = self.forward_propagation(input)
        T = np.array(target).reshape(-1, 1)

        self.dL = ['Unassigned'] * (N + 1)
        self.dW = ['D_Weight'] * (N + 1)
        self.dB = ['D_Bias'] * (N + 1)

        self.dL[N] = mse_derivative(Y, T).T @ softmax_derivative(self.L[N])
        for i in range(N - 1, 0, -1):
            self.dL[i] = self.dL[i + 1] @ self.W[i + 1] @ np.diagflat(
                ReLU_Derivative(self.L[i]))

        for i in range(1, N + 1):
            self.dW[i] = (self.dL[i] @ nthnrowmatrix(self.L[i], self.L[i - 1]))
                .reshape(self.W[i].shape)
            self.dB[i] = self.dL[i]

        for i in range(1, N + 1):
            self.W[i] -= self.h * self.dW[i]
            self.B[i] -= self.h * self.dB[i].T
        Loss = MSE(Y, T)
        return Loss
```

Appendix

Definition 9. *Gradient of* $f : \mathbb{R}^n \to \mathbb{R} :=$

$$\nabla f = \frac{\partial f}{\partial X} = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{pmatrix} \in \mathcal{M}_{1,n}$$

Definition 10. *Jacobian of* $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m :=$

$$\frac{\partial \boldsymbol{f}}{\partial X} = \begin{pmatrix} \nabla f_1 \\ \nabla f_2 \\ \vdots \\ \nabla f_m \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \in \mathcal{M}_{m,n}$$

Definition 11. *Jacobian of* $\boldsymbol{F} : \mathcal{M}_{n,m} \to \mathcal{M}_{k,1} :=$

$$\frac{\partial \boldsymbol{F}}{\partial X} = \begin{pmatrix} \frac{\partial F}{\partial X_{1,}} \\ \vdots \\ \frac{\partial F}{\partial X_{k,}} \end{pmatrix}$$

여기서 $X_{i,}$는 X의 i번째 행을 의미한다.
즉, 각각의 $\frac{\partial F}{\partial X_{i,}}$들은 *Definition* 10에 따른 *Jacobian*을 의미한다.

Definition 12. *Hadamard product:*
For same dimension matrix A *and* B,
Hadamard product $A \odot B$ *given by*

$$(A \odot B)_{i,j} = (A)_{i,j} \cdot (B)_{i,j}$$

Proposition 3. *Chain Rule:*
For function $f(x_1(t_1, \ldots, t_m), x_2(t_1, \ldots, t_m), \ldots, x_n(t_1, \ldots, t_m))$,

$$\frac{\partial f}{\partial t_i} = \sum_{j=1}^{n} \frac{\partial f}{\partial x_j} \cdot \frac{\partial x_j}{\partial t_i}$$

Lemma 1. *For linear transformation* $Y = W \cdot X$,

$$\frac{\partial Y}{\partial X} = W$$

Lemma 2. *For Softmax function* $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^n$,

$$\begin{aligned} \frac{\partial \mathcal{F}}{\partial X} &= \begin{pmatrix} \frac{\partial \mathcal{F}_1}{\partial x_1} & \cdots & \frac{\partial \mathcal{F}_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{F}_n}{\partial x_1} & \cdots & \frac{\partial \mathcal{F}_n}{\partial x_n} \end{pmatrix} \\ &= \begin{pmatrix} \mathcal{F}_1 & 0 & \ldots & 0 \\ 0 & \mathcal{F}_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \mathcal{F}_n \end{pmatrix} - \begin{pmatrix} \mathcal{F}_1 \cdot \mathcal{F}_1 & \ldots & \mathcal{F}_1 \cdot \mathcal{F}_n \\ \mathcal{F}_2 \cdot \mathcal{F}_1 & \ldots & \mathcal{F}_2 \cdot \mathcal{F}_n \\ \vdots & \ddots & \vdots \\ \mathcal{F}_n \cdot \mathcal{F}_1 & \ldots & \mathcal{F}_n \cdot \mathcal{F}_n \end{pmatrix} \end{aligned}$$