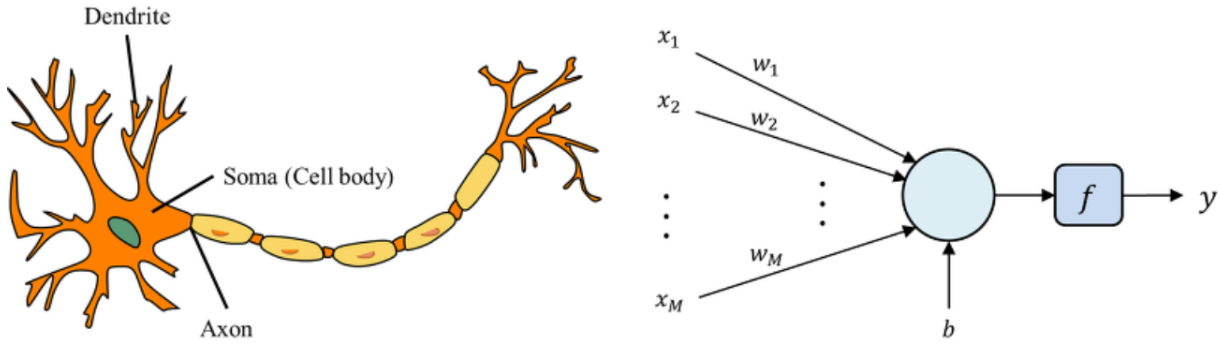
뉴럴 네트워크

> 뉴럴 네트워크의 목적은 사람이 직접 정했던 퍼셉트론의 파라미터를 알고리즘을 통해 결정하고자 하는 노력에서 출발한다.

뉴럴 네트워크. 한글로 인공 신경망은 두뇌의 신경세포인 뉴런이 서로 연결된 형태를 모방한 모델이다. 뉴럴 네트워크의 가장 작은 단위인 퍼셉트론이 뉴런과 유사하게 생겨 뉴럴 네트워크라고 부른다.(실제로 동작하는 방법도 유사하게 동작한다.)



[그림 1] 생물체의 neuron (좌)과 artificial neuron (우)

> 퍼셉트론 문서의 선숙지가 요구된다.

뉴럴 네트워크 알고리즘

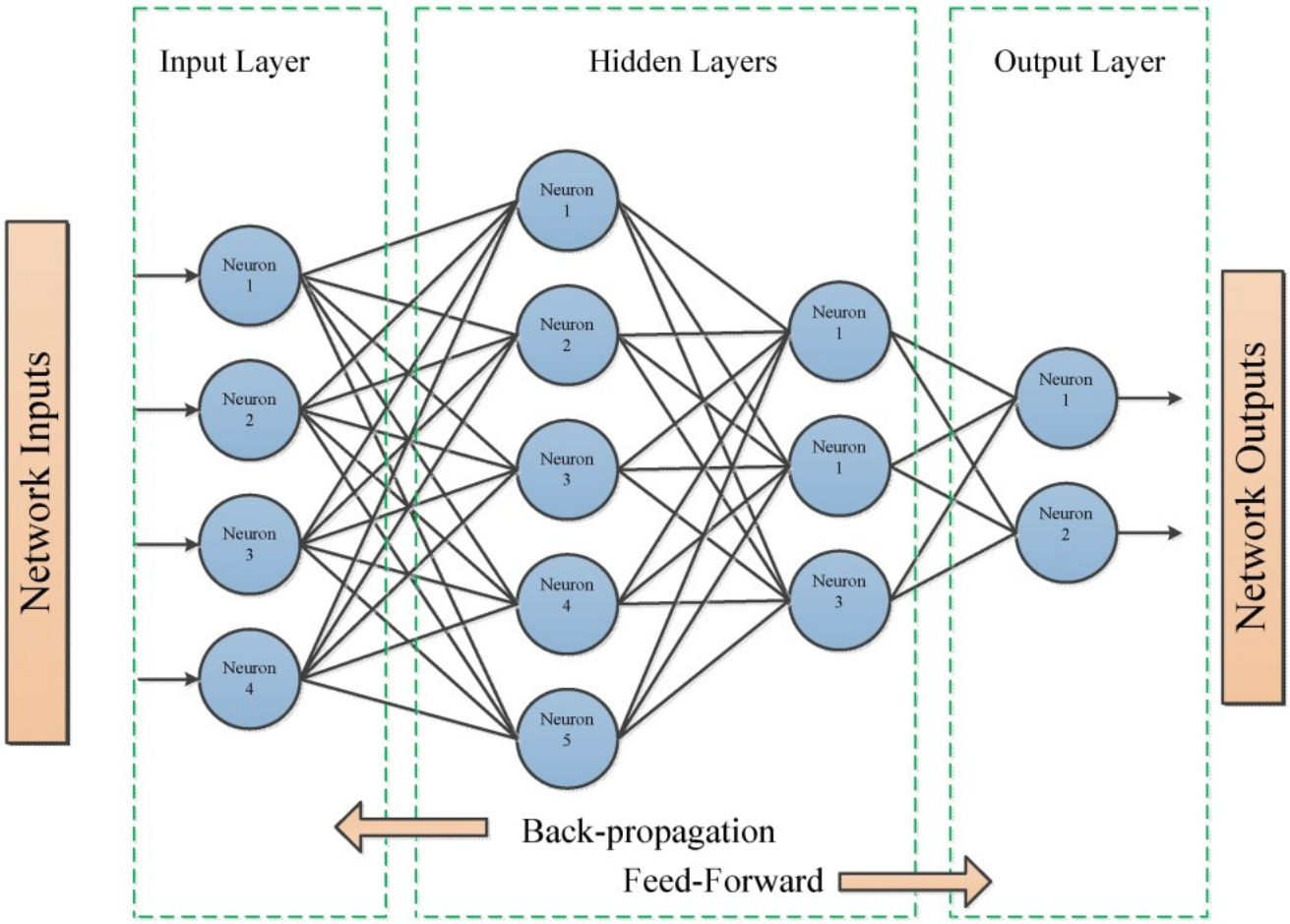
선형회귀모델, 로지스틱 회귀 모델, 등 여러 머신러닝 모델을 통해 데이터를 분석, 예측, 분류할 때 많은 경우에서 사람이 직접 모델을 설계하고 파라미터 사용 유무를 결정하였다. 이러한 일련의 과정들을 알고리즘을 통해 최적의 값을 선택하고자 하는 노력이 뉴럴 네트워크 알고리즘의 시작점이다.

뉴럴 네트워크 알고리즘은 아래 순서로 진행된다.

1. 각 층의 노드의 개수와 같은 하이퍼 파라미터 값을 결정한다.(이 부분은 아래에서 이야기하고 일단 각 층의 노드의 개수만 설정하자.)
2. 각 노드 간의 연결에 무작위 숫자의 가중치를 부여한다.(비교적 작은 숫자가 좋다).
3. 입력층의 노드값과 가중치값을 통해 출력층까지의 노드값을 계산한다.
4. 출력층에서 반환된 값과 실제 값과의 차이 즉, 오차를 계산한다.
5. 오차가 최소화 되도록 각 노드 간의 가중치를 조절한다.
6. 충분히 작은 오차가 나올 때까지 3,4,5번 과정을 반복한다.

- 1번 과정을 **'모델 구성'** 이라고 부른다. 순수히 사람이 직접 설계해야 하는 부분이기 때문이다.
- 3번 과정을 **'순전파 알고리즘(forward propagation)'** 이라고 부른다. 입력층부터 출력층까지 순서대로 계산하기 때문이다.
- 4번 과정에서 오차를 계산하는 함수를 **'비용 함수(cost function)'** 라고 부른다. 목적에 따라 다른 비용함수를 사용한다.
  - 이진 분류가 목적일 경우 비용 함수로 sigmoid 함수를 주로 사용한다.

    - 3개 이상의 클래스 분류가 목적일 경우 비용 함수로 softmax 함수를 주로 사용한다.
    - 예측이 목적일 경우 비용 함수로 MSE를 기본적으로 채택한다.(다른 것도 많다.)
- 5번 과정에서 오차가 최소화 되도록 가중치를 조절하는 것을 **'경사하강법(Gradient descent)'** 이라고 부른다.
    - 가장 먼저 고안된 경사하강법은 **역전파 알고리즘(Backpropagation)** 이다.
    - **아담(Adam)**: 아담은 모멘텀과 아다그라드를 결합한 방법으로, 경사 하강법에서 널리 사용되는 최적화 알고리즘이다. 아담은 학습률을 조정하면서도 모멘텀의 이점을 취하며, 그래디언트의 1차 및 2차 모멘트 추정을 사용하여 가중치 업데이트를 수행한다.



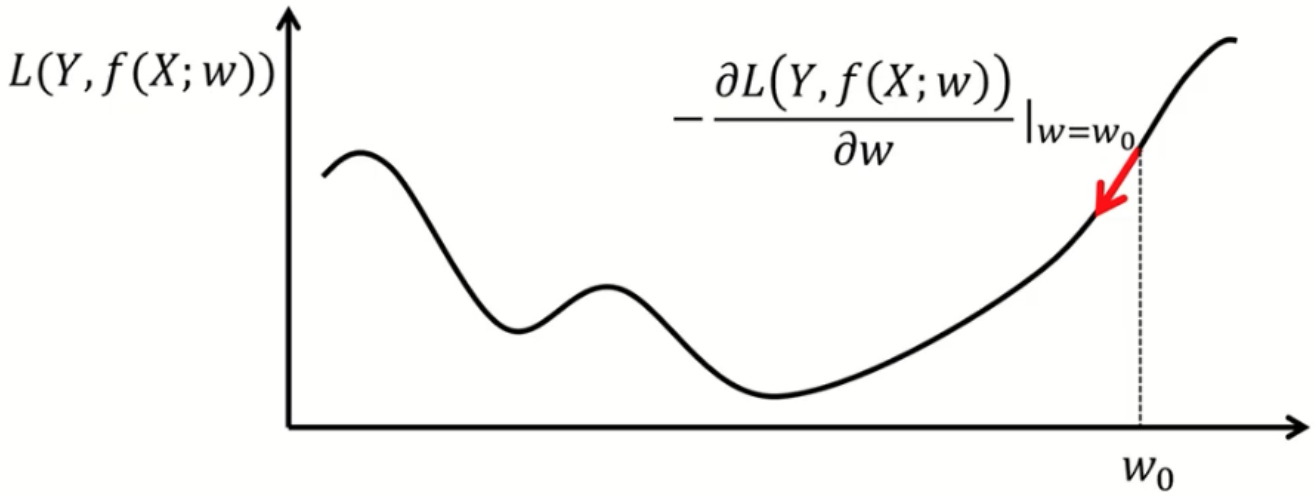
모델 구성은 코드 구현 단계에서 다루어 보고 순전파 알고리즘의 경우 단순히 계산하면 되기에 어렵지 않다.
4번 과정인 비용 함수의 경우 다른 문서에서 자세히 다루어 보고 본 문서에서는 익숙하게 아는 MSE를 비용 함수로 하여 연습해보자.

경사하강법

경사하강법이란 뉴럴 네트워크에서 가중치를 업데이트(수정)하는 알고리즘을 의미한다. 최초의 경사하강법으로 제안된 것은 역전파 알고리즘(Backpropagation)이며 현재 가장 많이 사용되는 알고리즘은 아담(Adam)이다.

본 장에서는 역전파 알고리즘(Backpropagation)을 통해 경사 하강법을 알아보자.

수치 예측이 목적일 경우 비용함수를 그래프로 표현하였을 때 아래와 같다고 가정하자.



$L()$는 비용함수이며 $f(X;w)$는 출력층의 output이다. 즉 아래 그래프는 x축은 가중치(w) 값이며 y축은 w값에 대한 비용함수값(오차)인 것이다.

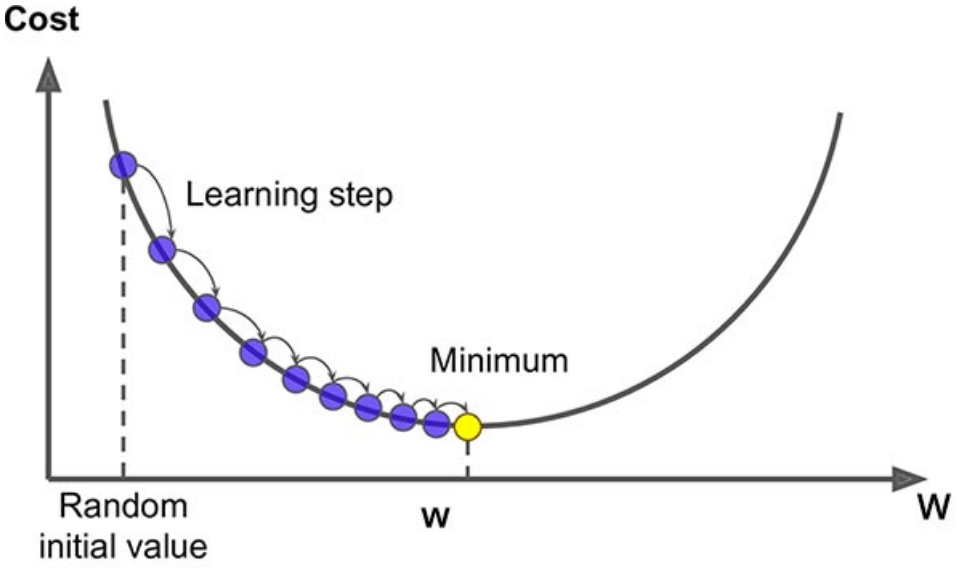
오차(비용함수값)이 최소가 되는 지점은 위 그래프에서 기울기가 0이 되는 지점일 것이다. 따라서 기울기가 0이 될 때까지 화살표 발향으로 w값을 이동시키면 오차가 최소화 되는 지점에 도달 할 수 있을 것으로 예상할 수 있다.

위 그래프의 기울기인 미분값은 $\frac{\partial L(Y, f(X;w))}{\partial w}$ 이니 한번에 이동할 w의 값은 $\alpha \times \frac{\partial L(Y, f(X;w))}{\partial w}$ 이다. α는 한번에 이동하는 크기로 learning rate 라고 부른다. learning rate 값이 작으면 한번에 움직이는 w값이 작음으로 조금씩 움직이며 최소가 되는 지점을 찾을 수 있으나 지정된 반복 횟수 안에 최소점을 찾지 못할 수도 있다.
반대로 learning rate 값이 너무 크면 한번에 움직이는 w값이 큼으로 최소가 되는 지점을 건너띄어 엉뚱한 지점을 찾을 수도 있다. 그렇기에 적절한 learning rate 값을 찾는 것이 중요하다.

learning rate를 고려하였을 때 w는 아래와 같이 식으로 작성될 수 있다.

$$w_{i+1} = w_i - \alpha \times \frac{\partial L(Y, f(X;w_i))}{\partial w}$$

최저점에 가까워 질수록 이동 횟수인 i는 증가한다. 그리고 최저점에 가까워 질수록 기울기 값이 감소하니 한번에 이동하는 w 값 또한 작아진다. 즉 경사하강법은 반복 할 수록 작은 폭으로 w값을 수정하여 아래 그림과 같이 최저점을 찾을 수 있다.



위와 같이 w 값을 변화하며 비용함수값(오차)이 최소가 되는 지점을 찾는 알고리즘을 경사하강법이라고 부르며 위와 같이 기울기와 learning rate를 이용해 w값을 이동시키는 방법을 역전파(backpropagation) 알고리즘이라고 부른다.

위 방법을 통해 w 값을 어떻게 수정해야 할지 알았다. 출력층 이전에 n개의 노드가 있었다면 각각의 출력층은 n개의 w를 가지게 된다. n개의 w들은 서로 다른 크기를 가지기에 출력층 노드에 서로 다른 영향을 준다. 그럼 n개의 w값들을 어떻게 조정해야 하는가?

의외로 답은 간단하다. 각각의 w값에 대해 편미분하여 서로 다른 업데이트 값을 적용하면 된다. i 번째 w값의 변화량을 식으로 작성하면 아래와 같다.

$$\triangle w_i = -\alpha \times \frac{\partial L(Y, f(X; w_i))}{\partial w_i}$$

엄밀히 말하면 출력층은 노드는 1개 이상 존재할 수 있기에 k번째 출력층의 j번째 w값의 변화량은 아래와 같다.

$$\triangle w_{ki} = -\alpha \times \frac{\partial L(Y, f(X; w_{ki}))}{\partial w_{ki}}$$

예를 들어 비용함수가 MSE이고 활성함수가 시그모이드 함수(로지스틱 함수)라고 가정했을 때 미분값을 아래와 같다. (계산 증명은 다루지 않는다).

> x는 입력값의 노드들, y는 실제값의 노드들, o는 출력값 즉 정답 노드들이며 h는 은닉층의 노드들을 의미한다. 예를 들어 h_j는 j번째 은닉층의 노드값인 것이다.

출력층과 은닉층 사이일 경우 : k 번째 출력 노드에 대해 i번째 은닉층에서 연결된 가중치(w) 변화량은 아래와 같다.

$$\triangle w_{ki} = -\alpha \times \frac{\partial L(Y, f(X; w_{ki}))}{\partial w_{ki}} = -\alpha(o_k - y_k)o_k(1 - o_k)h_i$$

은닉층과 은닉층 또는 입력층과 은닉층 사이일 경우 : j 번째 은닉(또는 입력) 노드에서 i 번째 은닉층에서 연경된 가중치(w) 변화량은 아래와 같다.

$$\triangle w_{ji} = -\alpha \times \frac{\partial L(Y, f(X; w_{ji}))}{\partial w_{ji}} = -\alpha x_i h_j(h_j - 1)\sum_{k=1}^{m} w_{kj}o_k(1 - o_k)(t_k - o_k)$$

다른 비용함수와 활성함수를 사용했다면 다른 식이 사용된다.

코드

위 내용을 numpy를 이용하여 하나하나 구현해보자.

1. 먼저 모델을 설계해야 한다. 필자는 입력층 1개, 은닉층 2개, 출력층 1개로 모델을 구성할 것이다. 노드의 개수는 순서대로 3개, 2개, 3개, 2개로 구성할 것이다. 은닉층의 노드 수에 맞게 각 노드를 연결하는 랜덤한 가중치를 임의로 배정한다.

```
import numpy as np
import random

# 0과 1사이에 num_count개의 랜덤한 실수 값을 가지는 list를 반환
random_num = lambda num_count: [random.random() for _ in range(num_count)]

# 입력값
```

```
X = np.array([0.5, 0.3, 0.2])

# 초기 가중치
# 입력층과 은닉층1을 연결하는 가중치(2 x 3)
W1 = np.array([random_num(3), random_num(3)])
# 은닉층1과 은닉층2를 연결하는 가중치(3 x 2)
W2 = np.array([random_num(2), random_num(2), random_num(2)])
# 은닉층2와 출력층을 연결하는 가중치(2 x 3)
W3 = np.array([random_num(3), random_num(3)])

# 실제값: 출력층이 맞추어야 하는 정답
Y = np.array([0.8, 0.6])
```

2. 노드들을 설정했다면 순전파 계산을 하자. 순전파 계산은 어렵지 않다. 가중치와 노드값의 선형 결합인 행렬 내적 연산을 한 다음 활성함수인 시그모이드 함수에 넣어 출력값(Z)를 반환하면 된다.

```
# 순전파 단계
# 은닉층1 노드값 계산
A1 = np.dot(W1, X)          # 선형 결합
Z1 = 1 / (1 + np.exp(-A1)) # 활성 함수

# 은닉층2 노드값 계산
A2 = np.dot(W2, Z1)
Z2 = 1 / (1 + np.exp(-A2))

# 출력층 노드값 계산
A3 = np.dot(W3, Z2)
Z3 = 1 / (1 + np.exp(-A3))
```

3. 출력층의 반환값(Z3)과 실제값(y)과의 오차를 계산한다. 비용함수(손실함수)는 MSE를 사용한다.

```
# 손실 계산
loss = np.sum((Y - Z3) ** 2)
print(f'[{i+1}] loss: {loss}')
```

4. 오차가 줄어들 수 있도록 각 노드로 들어오는 가중치값의 변화량(delta)를 계산한 뒤 가중치를 업데이트 한다.

```
# 역전파 단계
# 출력층 그래디언트 계산
delta3 = (Z3 - Y) * Z3 * (1 - Z3)

# 은닉층2 그래디언트 계산
delta2 = np.dot(W3.T, delta3) * Z2 * (1 - Z2)

# 은닉층1 그래디언트 계산
delta1 = np.dot(W2.T, delta2) * Z1 * (1 - Z1)


# 가중치 업데이트
learning_rate = 0.1
W3 -= learning_rate * np.outer(delta3, Z2)
```

```
W2 -= learning_rate * np.outer(delta2, Z1)
W1 -= learning_rate * np.outer(delta1, X)
```

5. 오차값이 충분히 줄어들 수 있도록 경사하강법은 1000번 반복할 것이다. 아래는 전체 코드이다. 1000번 반복한 뒤에 출력층의 값을 얼마나 실제값(Y)에 가까운지 확인해보자.

```
import numpy as np
import random

random_num = lambda num_count: [random.random() for _ in range(num_count)]

# 입력값
X = np.array([0.5, 0.3, 0.2])

# 초기 가중치
# W1 = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
# W2 = np.array([[0.2, 0.3], [0.4, 0.5], [0.6, 0.7]])
# W3 = np.array([[0.3, 0.4, 0.5], [0.6, 0.7, 0.8]])

W1 = np.array([random_num(3), random_num(3)])
W2 = np.array([random_num(2), random_num(2), random_num(2)])
W3 = np.array([random_num(3), random_num(3)])

# 실제값
Y = np.array([0.8, 0.6])

for i in range(1000):

    # 순전파 단계
    A1 = np.dot(W1, X)
    Z1 = 1 / (1 + np.exp(-A1))

    A2 = np.dot(W2, Z1)
    Z2 = 1 / (1 + np.exp(-A2))

    A3 = np.dot(W3, Z2)
    Z3 = 1 / (1 + np.exp(-A3))

        if(i == 0):
                print(Z3) # 초기값: [0.82129409 0.73044517]

    # 손실 계산
    loss = np.sum((Y - Z3) ** 2)
    # print(f'[{i+1}] loss: {loss}')

    # 역전파 단계
    # 출력층 그래디언트 계산
    delta3 = (Z3 - Y) * Z3 * (1 - Z3)

    # 은닉층2 그래디언트 계산
    delta2 = np.dot(W3.T, delta3) * Z2 * (1 - Z2)

    # 은닉층1 그래디언트 계산
    delta1 = np.dot(W2.T, delta2) * Z1 * (1 - Z1)
```

```
    # 가중치 업데이트
    learning_rate = 0.1
    W3 -= learning_rate * np.outer(delta3, Z2)
    W2 -= learning_rate * np.outer(delta2, Z1)
    W1 -= learning_rate * np.outer(delta1, X)

print(Z3) # 종료값: [0.80029981 0.60032069]
```

초기값에 비해 종료값은 상당히 오차가 줄어들었음을 확인 할 수 있다.