LAB03 기본 신경망 구현

학습 내용

- 용어 이해하기
- 활성화 함수 종류 알아보기
- 입력 데이터, 출력 데이터를 이용하여 간단한 기본 신경망 모델을 만들어본다.

용어 이해하기

- 인공신경망(Artificial neural network)의 개념은 뇌를 구성하는 신경세포, 즉 뉴런(Neuron)의 동작원리에 기초한다.
- 입력(X)에 가중치(W)를 곱하고, 편향(b)를 더한 후, (X * W + b)
- 활성화함수(Sigmoid, ReLU, tanh 기타)를 거쳐 y를 만든다.

y = sigmoid(X x W + b)

y : 출력
Sigmoid : 활성화 함수
X : 입력
W : 가중치
b : 편향

학습이란 최적의 y의 값을 찾기 위한 W, b의 값을 찾아내는 것을 말한다.

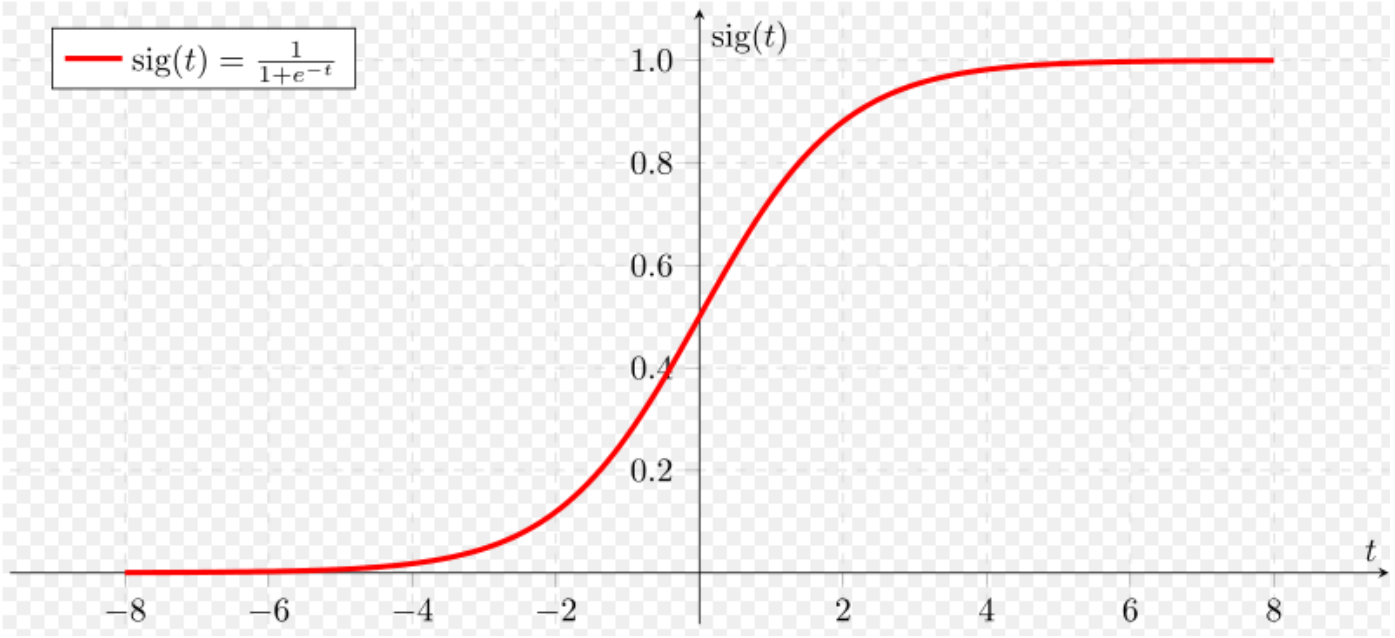
3-1 활성화 함수(activation function)

활성화 함수(activation function)는 인공 신경망을 통과해 값이 최종적으로 어떤 값으로 만들지를 결정한다.

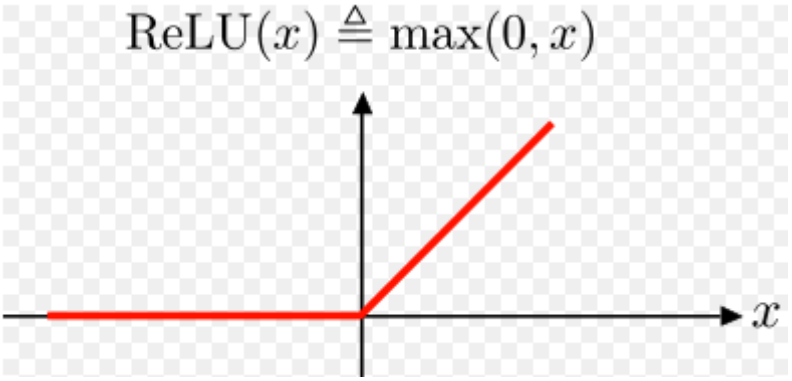
대표적인 종류 : Sigmoid(시그모이드), ReLu(렐루), tanh(쌍곡탄젠트)

- 인공뉴런은 가중치와 활성화 함수의 연결로 이루어진 간단한 구조이다.
- 간단한 개념의 인공 뉴런을 충분히 많이 연결된 것으로 인간이 인지하기 어려운 복잡한 패턴까지도 스스로 학습한다.
- 최근의 활성화 함수는 ReLU를 많이 사용함.

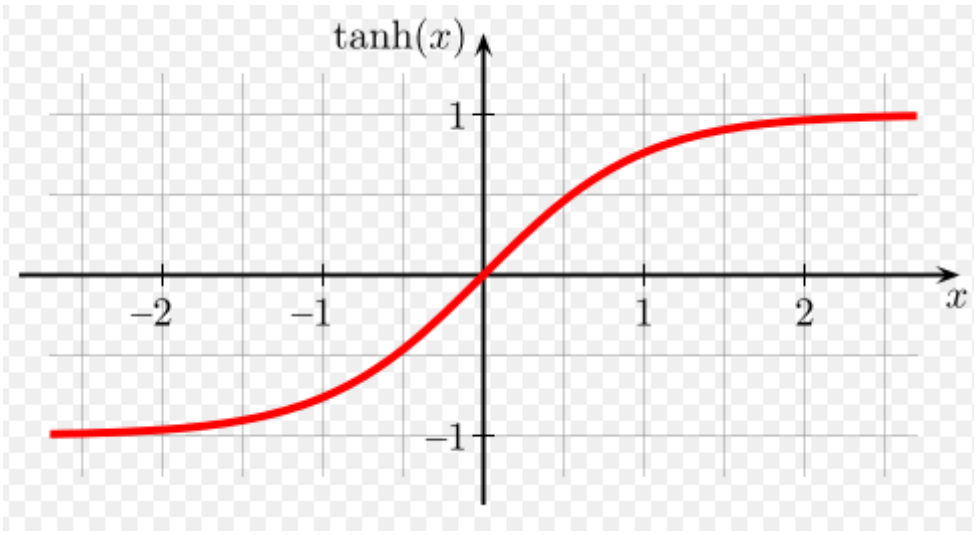
Sigmoid



Relu



Tanh



3-2 신경망 알고리즘 역사

초기 신경망의 한계

- 수천~수만개의 W와 b의 값을 일일이 변경시켜가며 계산하는데 오랜 시간이 걸림.
- 신경망의 층이 깊어질 수록 시도해봐야 하는 경우가 수가 많아, 유의미한 신경망을 만들기가 거의 불가능하다고 여겨짐.

신경망의 주목

- 제안 : 제프리 힌트(Geoffrey Hinton) 교수가 **제한된 볼트만 머신(Restricted Boltzmann Machine, RBM)**
- RBM를 통해 심층 신경망을 효율적으로 학습이 가능해짐을 증명함.
- 계속된 발전 : 드롭아웃 기법, ReLu 활성화 함수, GPU 발전, 역전파(backpropagation)

역전파

- 결과값의 활용한 오차를 앞쪽으로 전파하면서 가중치를 갱신.

3-3 간단한 분류 모델 구현하기

- 딥러닝에서 가장 폭넓게 활용되는 분야는 패턴 인식을 통한 영상처리 분야이다.
- 패턴을 파악해 여러종류로 구분하는 작업을 분류(classification)이라 한다.

In [20]:

```
import tensorflow as tf
import numpy as np
```

(1) 데이터 정의

- IRIS 꽃의 종류 setosa, vesicolor, virginica
- setosa를 우린 원핫을 이용하여 [1, 0, 0]
- vesicolor를 우린 원핫(0,1로 표현)하여 [0, 1, 0]
- virginica를 우린 원핫(0,1로 표현)하여 [0, 0, 1]로 표현

그리고 우린 x의 2가지 값(X1, X2)를 정하고, 두개의 값에 따라 setosa, vesicolor, virginica를 답을 매칭시킨다.

```
[0,0] -> [1,0,0]  # setosa
[1,0] -> [0 1 0]  # vesicolor
[1,1] -> [0 0 1]  # virginica
[0,1] -> [0 0 1]  # virginica
```

In [21]:

```
# x_data를 이용하여 우리는 꽃의 종류를 예측한다.
# 6개의 데이터
x_data = np.array( [[0, 0], [1, 0], [1, 1], [0, 0], [0, 0], [0, 1]])

y_data = np.array([
    [1, 0, 0],  # setosa
    [0, 1, 0],  # vesicolor
    [0, 0, 1],  # virginica
    [1, 0, 0],  # setosa
    [1, 0, 0],  # setosa
    [0, 0, 1]   # virginica
])
```

In [22]:

```
print(x_data.shape)
print(y_data.shape)
```

```
(6, 2)
(6, 3)
```

(2) 신경망 모델 구성

- 플레이스 홀더 구성(입력(X), 출력(Y)
- 가중치(W) : 특징수(2개) X 레이블수(3개) 로 설정
- 편향(B) 설정

In [23]:

```
X = tf.placeholder(tf.float32)   #  X에 들어갈 값(공간)
Y = tf.placeholder(tf.float32)   #  Y에 들어갈 값(공간)

# 신경망은 2차원으로 [입력층(특성), 출력층(레이블)] -> [2, 3] 으로 정합니다.
# 임의의 값을 지정한다.
# tf.random_uniform([shape], 시작범위, 끝범위)
W = tf.Variable(tf.random_uniform([2,3], -1., 1.))

# 편향을 각각 각 레이어의 아웃풋 갯수로 설정합니다.
# 편향은 아웃풋의 갯수, 즉 최종 결과값의 분류 갯수인 3으로 설정합니다.
b = tf.Variable(tf.zeros([3]))
```

- (가) 가중치를 곱하고 편향을 더한다.
- (나) (가)의 결과를 활성화 함수를 적용한다.(ReLu)
- (다) 신경망을 통해 나온 출력값을 softmax 함수를 이용하여 사용하기 쉽게 다듬어준다.

In [24]:

```
# (가) X(입력) * W(가중치) + b(편향)
L = tf.add(tf.matmul(X,W), b)

# (나) 활성화 함수 적용
# 가중치와 편향을 이용해 계산한 결과 값에
# 텐서플로우에서 기본적으로 제공하는 활성화 함수인 ReLU 함수를 적용합니다.
L = tf.nn.relu(L)

# (다) softmax 함수 적용
# 마지막으로 softmax 함수를 이용하여 출력값을 사용하기 쉽게 만듭니다
# softmax 함수는 출력층의 결과값을 전체합이 1인 확률로 만들어주는 함수입니다.
# 예) [8.04, 2.76, -6.52] -> [0.53 0.24 0.23]
model = tf.nn.softmax(L)
```

(3) 손실함수(loss) 작성

- 교차 엔트로피(Cross-Entropy) : 대부분의 모델에서 사용하는 Loss 함수
- reduce_XXX : 텐서의 차원을 줄여준다.
- reduce_mean, reduce_sum, reduce_min... -> 차원을 없앤다.

In [25]:

```
# 신경망을 최적화하기 위한 비용(cost) 함수를 작성.
# 각 개별 결과에 대한 합을 구한 뒤 평균을 내는 방식을 사용
# 전체 합이 아닌, 개별 결과를 구한 뒤 평균을 내는 방식을 사용하기 위해
# axis 옵션을 사용합니다. (0 : 열의 합, 1 : 행의 합)
# axis 옵션이 없으면 -1.09 처럼 총합인 스칼라값으로 출력됩니다.
#        Y         model         Y * tf.log(model)   reduce_sum(axis=1)
# 예) [[1 0 0]  [[0.1 0.7 0.2]  -> [[-1.0  0    0]  -> [-1.0, -0.09]
#     [0 1 0]]  [0.2 0.8 0.0]]     [ 0   -0.09 0]]
#  마지막 값 ([-1.0, -0.09])의 평균을 내면 이 값이 교차 엔트로피(-0.545)가 된다.
# 즉, 이것은 예측값과 실제값 사이의 확률 분포의 차이를 비용으로 계산한 것이며,
# 이것을 Cross-Entropy 라고 합니다.
cost = tf.reduce_mean(-tf.reduce_sum(Y*tf.log(model), axis=1))
cost
```

Out[25]:

```
<tf.Tensor 'Mean_7:0' shape=() dtype=float32>
```

(4) 최적화 알고리즘

- 기본적인 경사하강법을 이용한 최적화
- cost가 최소가 되도록 최적화 한다.

In [26]:

```
# cost = tf.reduce_mean(-tf.reduce_sum(Y*tf.log(model), axis=1))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
train_op = optimizer.minimize(cost)
print(train_op)
```

```
name: "GradientDescent_2"
op: "NoOp"
input: "^GradientDescent_2/update_Variable_2/ApplyGradientDescent"
input: "^GradientDescent_2/update_Variable_3/ApplyGradientDescent"
```

In [27]:

```
print(x_data.shape, y_data.shape)
```

```
(6, 2) (6, 3)
```

(5) 그래프 실행

- 세션 초기화
- 학습 : x_data -> 특징1, 특징2(0,1), y_data : 꽃의 종류
- 학습단위10번마다 cost 출력

In [28]:

```
# 텐서플로 세션 초기화
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

# 레이블 데이터를 이용하여 학습을 진행
for step in range(100):
  sess.run(train_op,feed_dict={X:x_data, Y:y_data})

  # 학습도중 10번씩 손실값을 출력
  if (step+1)%10 == 0:
    print(step+1, sess.run(cost, feed_dict={X:x_data, Y:y_data}))
```

```
10 1.1135373
20 1.1025534
30 1.0920216
40 1.081915
50 1.0722083
60 1.0629126
70 1.0539868
80 1.045455
90 1.037209
100 1.0293268
```

(6) 예측값과 실제값의 비교

- tf.argmax(model, 1) 예측값 중의 가장 높은 값을 갖는 위치 출력
- tf.argmax(Y, 1) 실제값 중의 가장 높은 값을 갖는 위치 출력
- tf.equal() 의 결과값 True, False
- tf.cast() 값을 실수값의 형태로 변경

In [29]:

```
# tf.argmax: 예측값과 실제값의 행렬에서 tf.argmax를 이용해
# 가장 큰 값의 위치(인덱스)을 가져옵니다.
# 예) [[0   1   0]  [1    0    0]] -> [1 0]   # 두번째(1), 첫번째(0)
#    [[0.2 0.7 0.1] [0.9  0.1  0.]] -> [1 0]  # 두번째(1), 첫번째(0)

prediction = tf.argmax(model, 1)
target = tf.argmax(Y, 1)

print('예측값:', sess.run(prediction, feed_dict={X: x_data}))
print('실제값:', sess.run(target, feed_dict={Y: y_data}))

is_correct = tf.equal(prediction, target)
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))

print('is_correct :', sess.run(is_correct, feed_dict={X: x_data,Y: y_data}))
print('정확도: %.2f' % sess.run(accuracy * 100, feed_dict={X: x_data, Y: y_data}))
```

```
예측값: [2 0 2 2 2 2]
실제값: [0 1 2 0 0 2]
is_correct : [False False  True False False  True]
정확도: 33.33
```