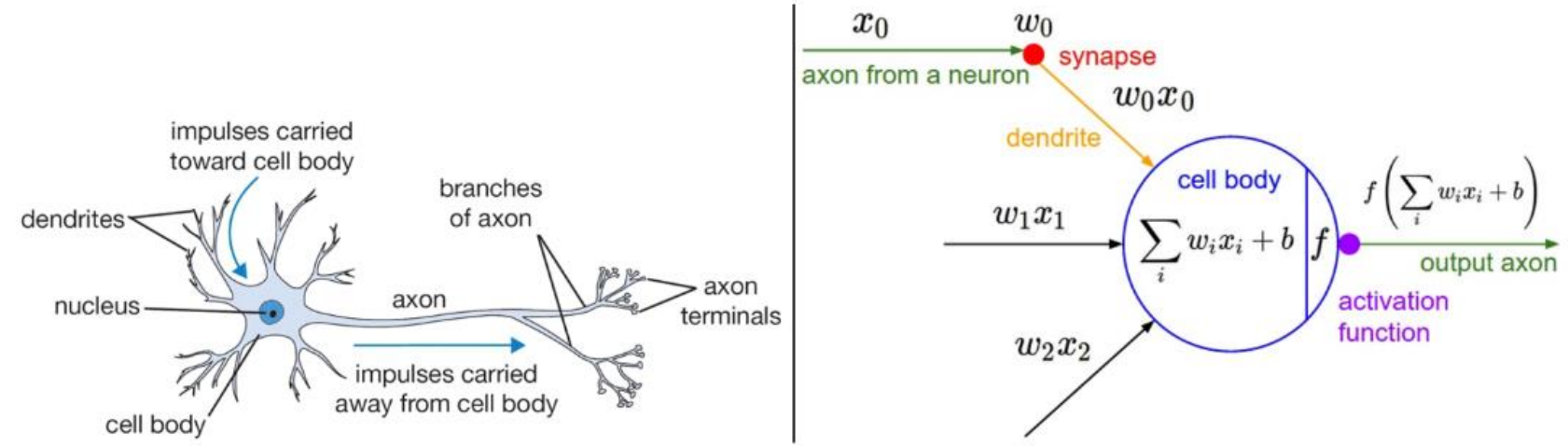
# 스마트 에너지를 위한

## Deep Learning 기초

2024년 1월 22일~1월 23일

# 신경망 개요




A cartoon drawing of a biological neuron (left) and its mathematical model (right).

(출처: http://cs231n.github.io/neural-networks-1/)

- axon (축삭돌기) : 뉴런에서 뻗어나와 다른 뉴런의 수상돌기와 연결
- dendrite (수상돌기) : 다른 뉴런의 축삭 돌기와 연결, 몸체에 나뭇가지 형태로 붙어 있다
- synapse (시냅스) : 축삭돌기와 수상돌기가 연결된 지점, 여기서 한 뉴런이 다른 뉴런으로 신호가 전달

# 신경망 개요

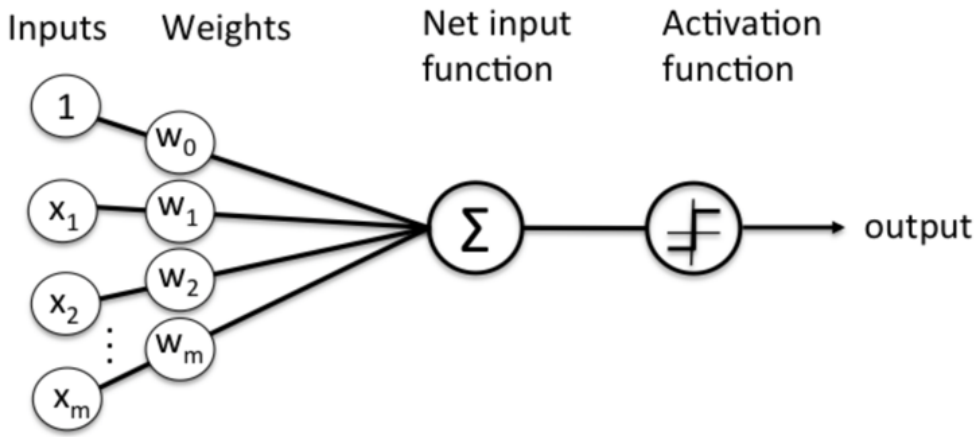
- 하나의 뉴런은 여러 다른 뉴런의 축삭돌기와 연결
- 연결된 시냅스의 강도가 연결된 뉴런들의 영향력이 결정
- 이러한 영향력의 합이 어떤 값을 초과하면 신호가 발생하여 축삭돌기를 통해서 다른 뉴런에게 신호가 전달

  - x0, x1, x2 : 입력되는 뉴런의 축삭돌기로부터 전달되는 신호의 양
  - w0, w1, w2 : 시냅스의 강도, 즉 입력되는 뉴런의 영향력
  - w0*x0* + *w1*x1 + w2*x2 : 입력되는 신호의 양과 해당 신호의 시냅스 강도가 곱해진 값의 합계
  - f : 최종 합계가 다른 뉴런에게 전달되는 신호의 양을 결정짓는 규칙, 이를 **활성화 함수**라고 부름

# 퍼셉트론 (Perceptron)

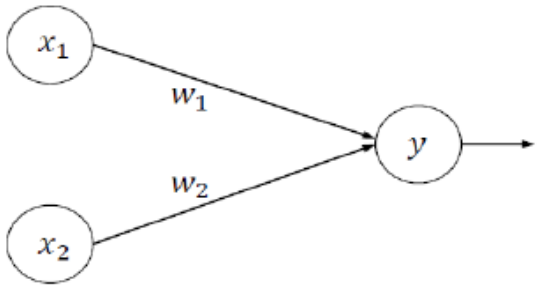
- 인공신경망의 개념
  - 1943년에 최초로 제안
  - 신경망은 인공신경망(Artificial Neural Net, ANN)을 줄여서 부름

- 퍼셉트론(Perceptron)
  - 신경망의 최초 모델



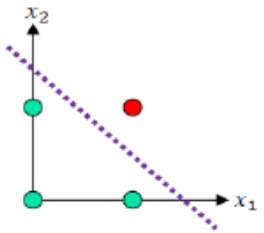
# 퍼셉트론 (Perceptron)

- 퍼셉트론은 실제로 간단한 XOR 기능도 구현하지 못하는 것으로 판명
  - 실효성에 의문
- 후에 XOR 과 같은 연산은 여러 층으로 구성된 다층 퍼셉트론 (MLP)으로 해결될 수 있다는 것이 증명



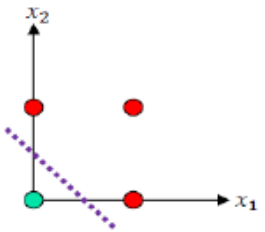

$$y = \begin{cases} 0 & (w_1 \cdot x_1 + w_2 \cdot x_2 \le \theta) \\ 1 & (w_1 \cdot x_1 + w_2 \cdot x_2 > \theta) \end{cases}$$

AND



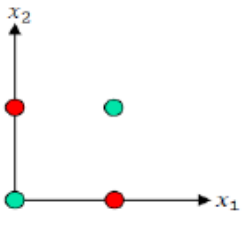

$w_1 = 1.0$
$w_2 = 1.0$
$\theta = 1.5$

OR




$w_1 = 1.0$
$w_2 = 1.0$
$\theta = 0.5$

XOR



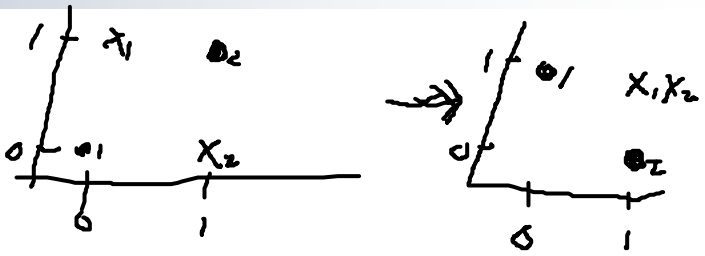

$w_1 = ?$
$w_2 = ?$
$\theta = ?$

# 네트워크 (network)

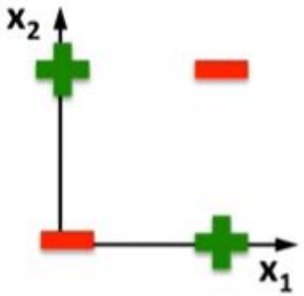
- 신경망 모델은 보통 여러개의 계층으로 구성되며 이를 “네트워크”라 함

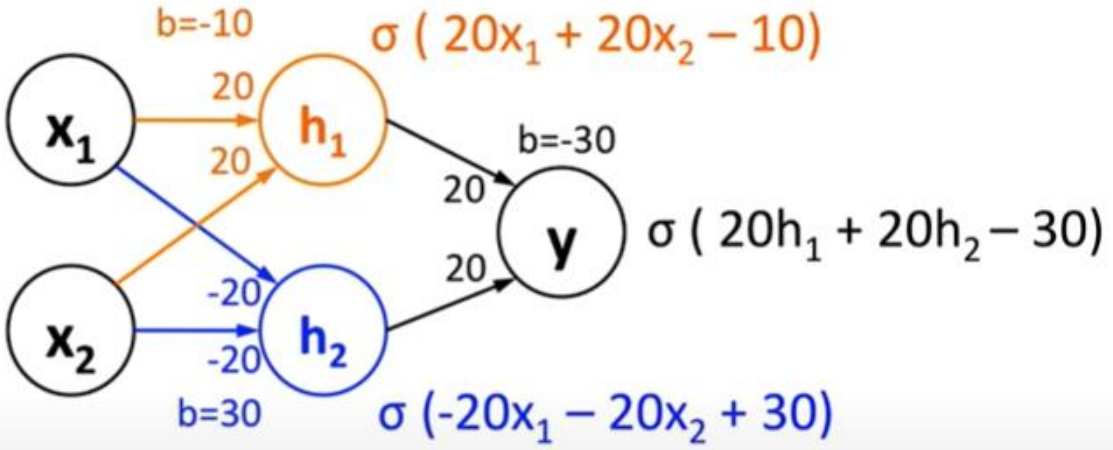
# 퍼셉트론 (Perceptron)

- Solving XOR Problem using MLP



Linear classifiers
cannot solve this



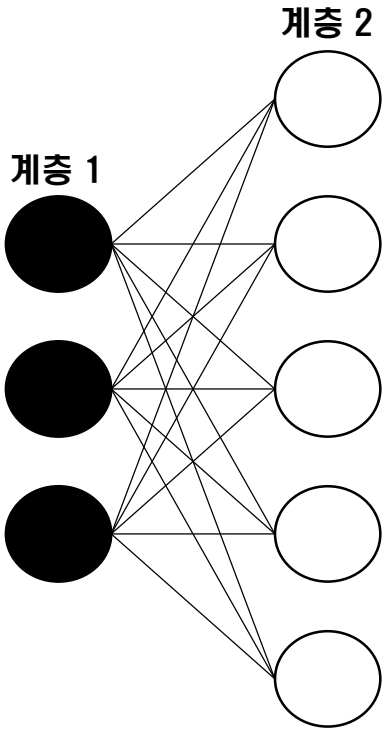




| | | |
|---|---|---|
| σ(20*0 + 20*0 – 10) ≈ 0 | σ (-20*0 – 20*0 + 30) ≈ 1 | σ (20*0 + 20*1 – 30) ≈ 0 |
| σ(20*1 + 20*1 – 10) ≈ 1 | σ (-20*1 – 20*1 + 30) ≈ 0 | σ (20*1 + 20*0 – 30) ≈ 0 |
| σ(20*0 + 20*1 – 10) ≈ 1 | σ (-20*0 – 20*1 + 30) ≈ 1 | σ (20*1 + 20*1 – 30) ≈ 1 |
| σ(20*1 + 20*0 – 10) ≈ 1 | σ (-20*1 – 20*0 + 30) ≈ 1 | σ (20*1 + 20*1 – 30) ≈ 1 |

(Ref: https://www.youtube.com/watch?v=kNPGXgzxoHw)

# 전 결합망 (Fully Connected Network, FCN)

- 가장 기본적인 신경망 구조
- 각 계층의 **모든 출력**이 다음 단계의 **모든 입력으로 연결**되는 선이 존재하고 여기에 가중치가 곱해지는 구조
  - 가중치를 곱하는 것 외에 스칼라 값인 편이(bias)가 더해진다.




$$Y = \Sigma(W_{ij} \times X_{ij} + b_i)$$

# 신경망과 딥러닝

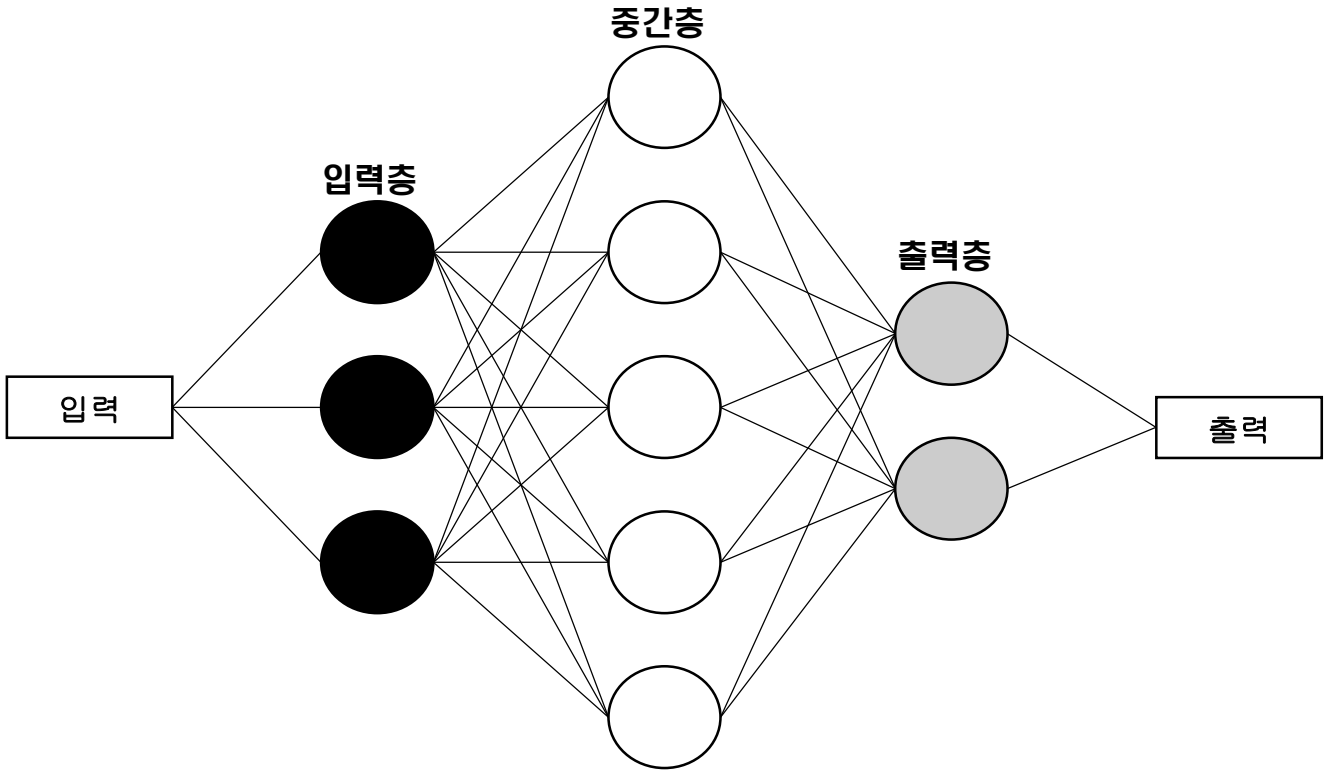
- 신경망 모델
  - 뇌를 구성하는 신경 뉴런(neuron)의 동작을 모방
  - 기본적으로는 입력 신호 벡터에 어떤 가중치를 곱하고 그 결과를 더하거나 비선형 처리를 하여 유용한 정보를 추출하는 구조
  - 이러한 작업을 계층(layer) 단위로 여러 번 수행

- 딥(deep) 러닝 모델
  - 계층이 2개 이상인 신경망 모델

- 성능
  - 이제 거의 사람처럼 듣고, 읽고, 보고 쓰는 능력이 발전
  - 2012년 ILSVRC(ImageNet Scale Visual Recognition Challenge)에서 기존의 알고리즘보다 월등히 우수한 성능을 보이면서 (AlexNet) 딥러닝이 머신러닝 모델로 널리 채택되기 시작

# 신경망 구성 – 기본 구조

$$Y = WX + b$$



# 텐서 (tensor)

- 신경망에서는 **계층별 입출력 정보**를 벡터 보다 **텐서**라고 부름

- 텐서는 개념적으로는 벡터와 같은 의미이나 신경망에서는 다차원 벡터를 통칭하여 텐서라고 하고, "벡터"라고 하면 아래와 같이 1차원이면서 항목이 **여러 개**인 신호를 지칭한다.

  - X = [-0.82, 0.94, -1.15, 0.25]

- 텐서에서는 차원 (dimension) 이라는 표현대신 **랭크**(rank)를 사용하는데, 벡터는 랭크가 1인 텐서이다.

# 텐서 (tensor) – 예

- rank = 2
  - 행(row) : 입력 신호의 sample들
  - 열(column) : 각 sample의 특성(feature) 값

```
X = [[-0.82, -0.25, -1.16, -0.64, 1.13, -0.68],
     [-0.18, -0.42, 1.34, -0.21, -0.57, 1.23],
     [ 2.32, 1.22, -0.43, 0.2 , -0.29, -0.88],
     [-2.02, -0.44, 0.2 , 0.5 , 0.01, -0.88]]
```

- 만일 입력 신호 X가 200 장의 컬러 이미지이고, 각 이미지가 (244, 244) 크기의 픽셀로 구성되며, 각 픽셀이 3원색으로 구성되어 있다면 X의 모양은 다음과 같으며 랭크는 4가 된다.

  - X.shape = (200, 244, 244, 3)

# 활성화 함수 (Activation Function)

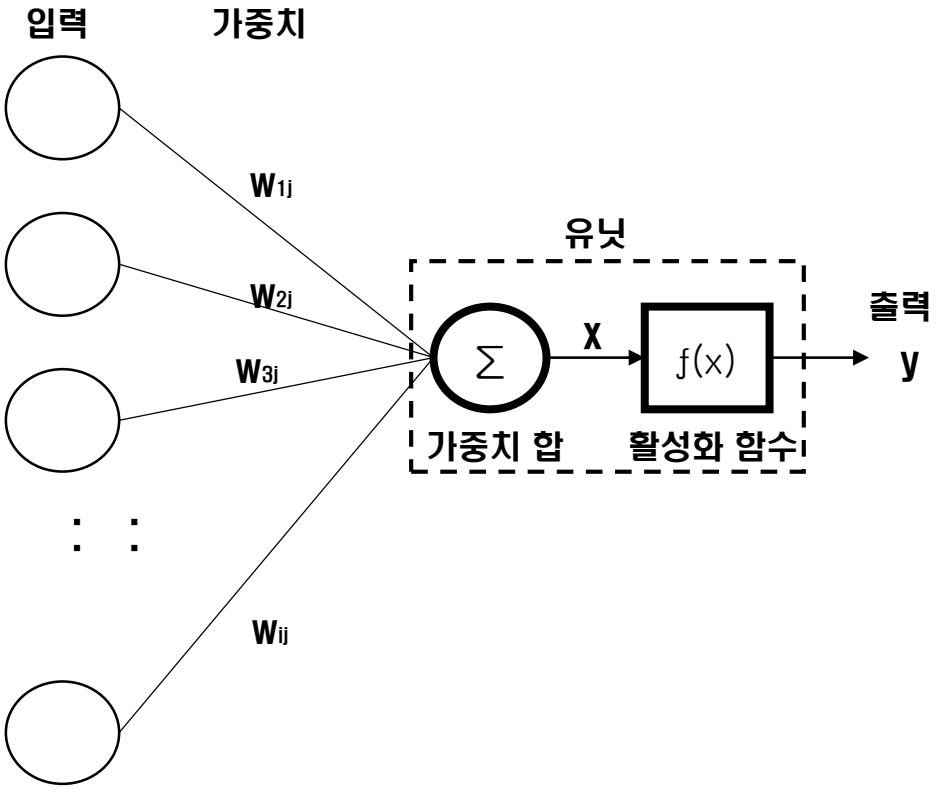
- 신경망은 (선형회귀와 달리) 한 계층의 신호를 다음 계층으로 그대로 전달하지 않고 비선형적인 활성화 함수를 거친 후에 전달

- 활성화 함수 사용 이유
  - 생물학적인 신경망을 모방
  - 약한 신호는 전달하지 않고, 어느 이상의 신호도 전달하지 않는 “S”자 형 곡선과 같이 “비선형적”인 반응을 한다고 생각

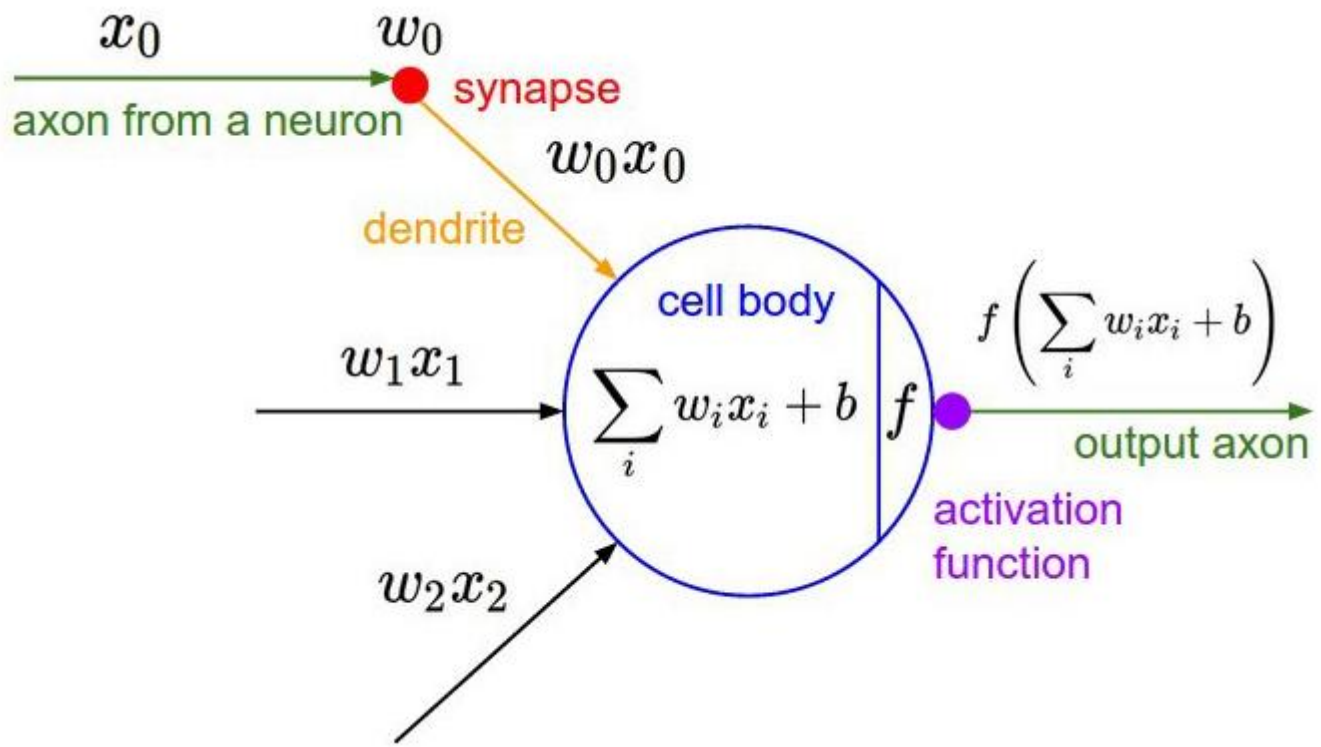
- 실제로 비선형의 활성화 함수를 도입한 신경망이 잘 동작

# 활성화 함수 (Activation Function)

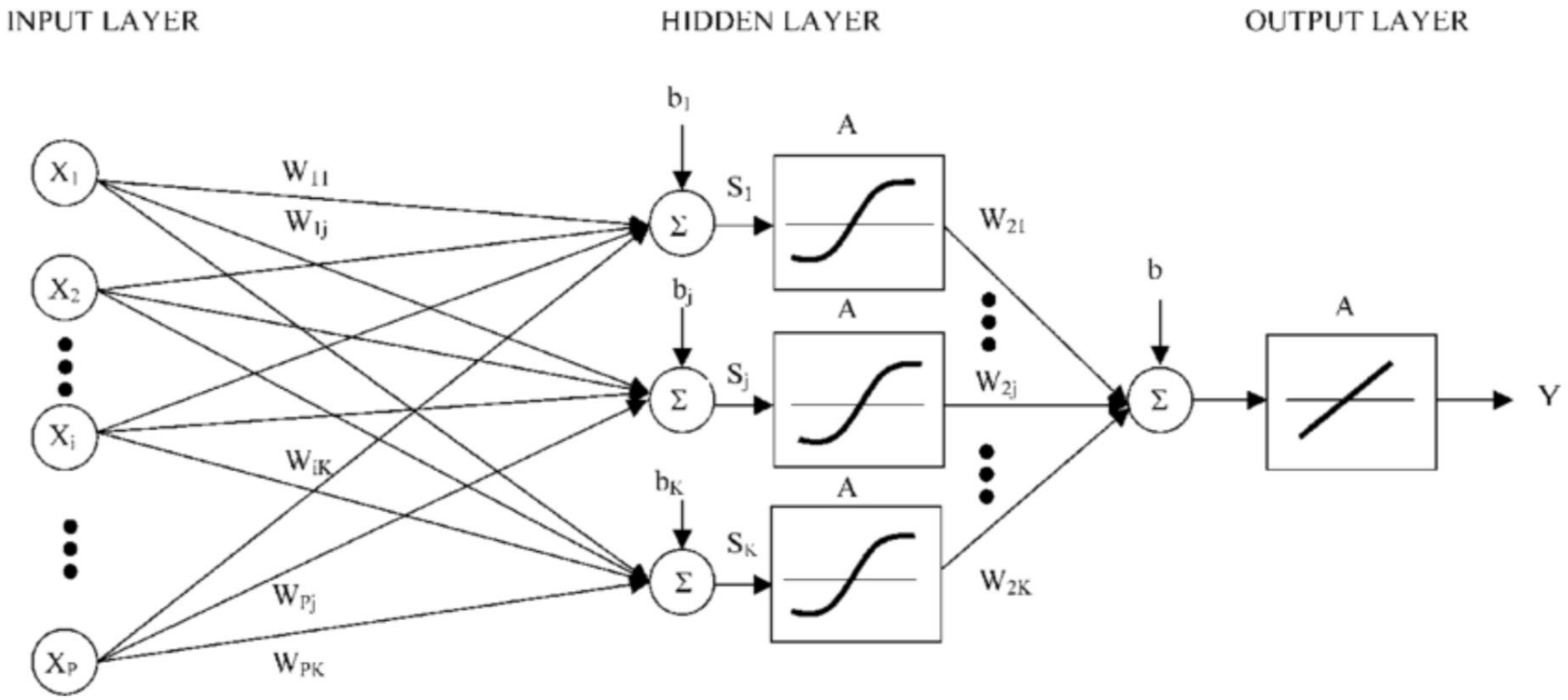
- 다음 계층으로 신호를 전달할 때, 어떤 범위의 신호를 "활성화(activate)" 하여 전달할지를 정하는 함수



# 활성화 함수 (Activation Function)



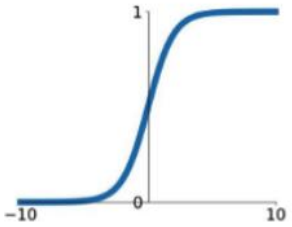
# 활성화 함수 (Activation Function)



# 활성화 함수 종류

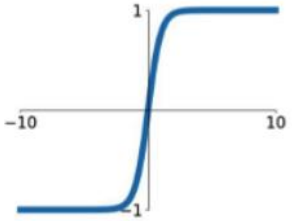
**Sigmoid**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



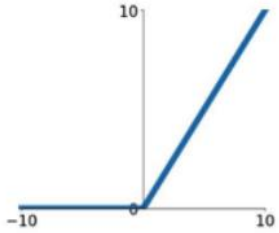

**tanh**

$$\tanh(x)$$



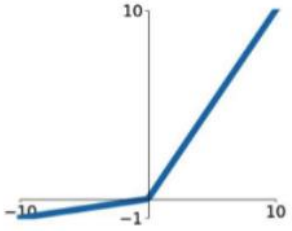

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

$$\max(0.1x, x)$$

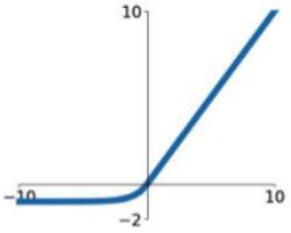



**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

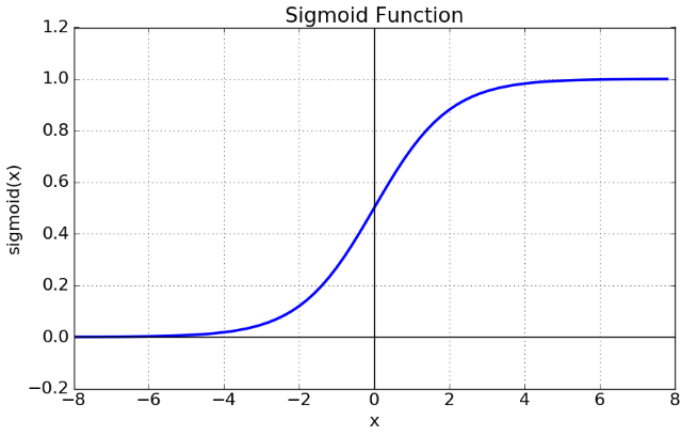
**ELU**

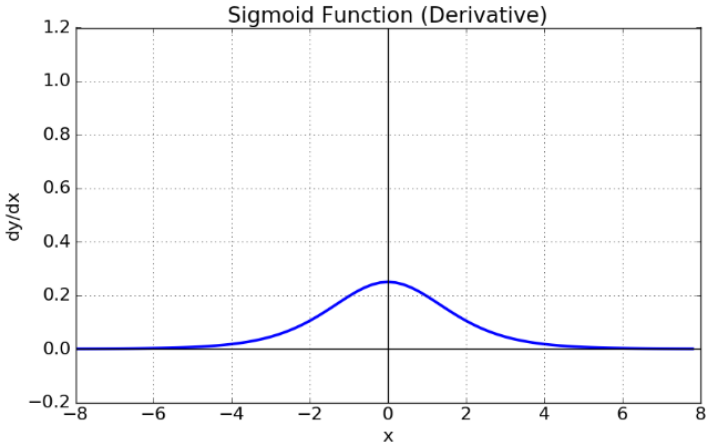
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# 활성화 함수 종류

- 시그모이드(Sigmoid) 함수
  - 입력과 출력의 관계가 S 커브를 따른다고 가정한 함수, 생물학적으로 입출력 관계를 설명할 때 많이 사용되는 모델
  - 출력을 0~1 사이로 제한
  - 단점 : 입력값이 일정 범위를 벗어나서 너무 크거나 작으면 모두 1이나 0으로 일정하게 평탄해져서 큰 신호는 무시된다는 문제가 있다. 또한 이 범위에서는 기울기가 모두 0이므로 입력의 변화에 반응하지 못하는 한계



# 활성화 함수 종류

- Hyperbolic tangent (tanh) 함수
  - 시그모이드와 유사한 모양
  - 출력의 범위가 다르다 : -1 ~ 1 사이
  - tanh함수 역시 크거나 작은 값이 무시되는 단점이 존재

- ReLU 함수
  - 입력이 양수이면 신호를 그대로(선형적으로) 전달
  - 입력이 음수이면 출력을 '0'으로 제한
  - 크기가 **큰 신호**를 출력으로 전달하는 구조를 가지며, 시그모이드를 사용할 때 발생하는 Vanishing Gradient 문제를 해결
  - 현재 많은 신경망에서 ReLU를 채택

# 활성화 함수 선택

- **활성화 함수의 선택**
  - 신경망의 각 계층마다 다르게 택할 수 있다

- **활성화 함수를 사용하지 않는다는 것**
  - 신호를 선형적으로 그대로 출력으로 전달
  - 이는 신경망을 **회귀분석**에 사용할 때 필요

- **분류 문제나 어떤 사건의 발생 확률을 구할 때**
  - 즉, 0~1 범위의 확률에 해당하는 값을 예측할 때에는 출력단에서 시그모이드 함수를 주로 사용

- **다중 분류를 사용할 때**
  - 소프트맥스(softmax)를 사용

# 다층 퍼셉트론 (MLP)

# 다층 퍼셉트론 (MLP, Multi Layer Perceptron)

- 전결합망과 활성화 함수를 합한 기능이 하나의 계층(layer)이며 이러한 계층을 여러 개 쌓은 구조

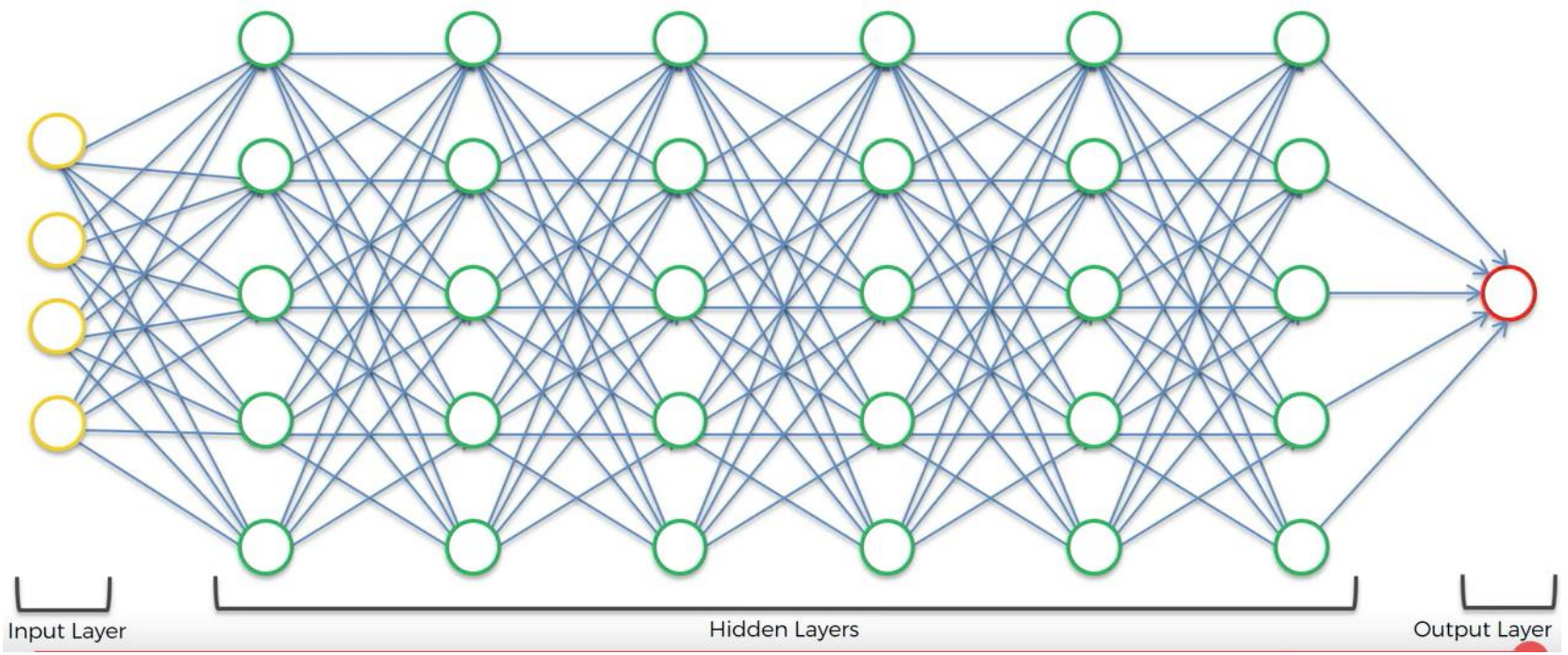
- 은닉(hidden) 계층
  - 입력과 출력 계층을 제외한 중간 계층

- 초기 신경망은 MLP로 구성했는데 필기체 숫자 인식 등에서 상용화되기도 함

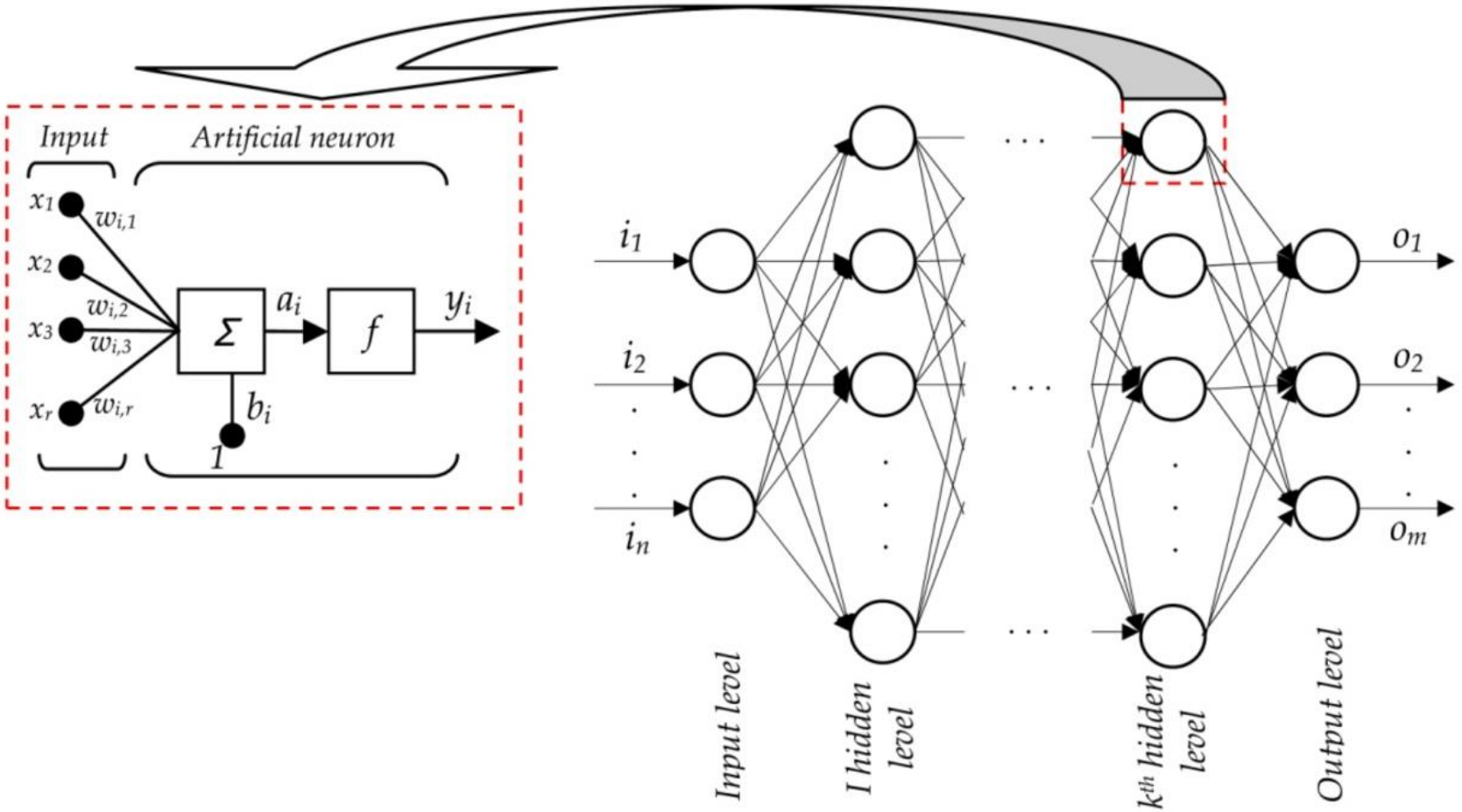
- 지금은 MLP를 개선한 CNN, RNN 등이 신경망으로 사용되며 성능도 급격히 향상됨

# 다층 퍼셉트론 (MLP)

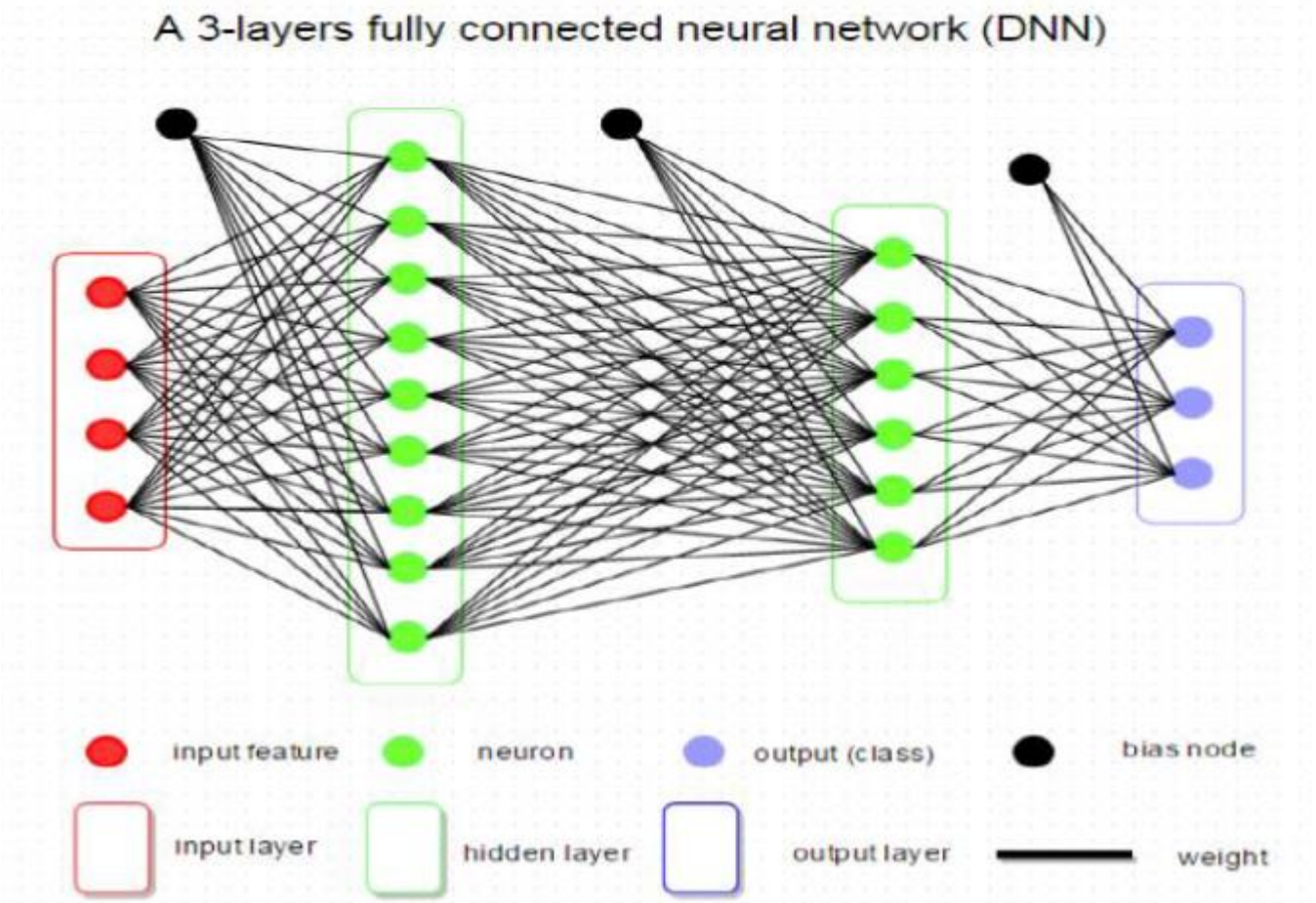
- 은닉(hidden) 계층
  - 입력과 출력 계층을 제외한 중간 계층



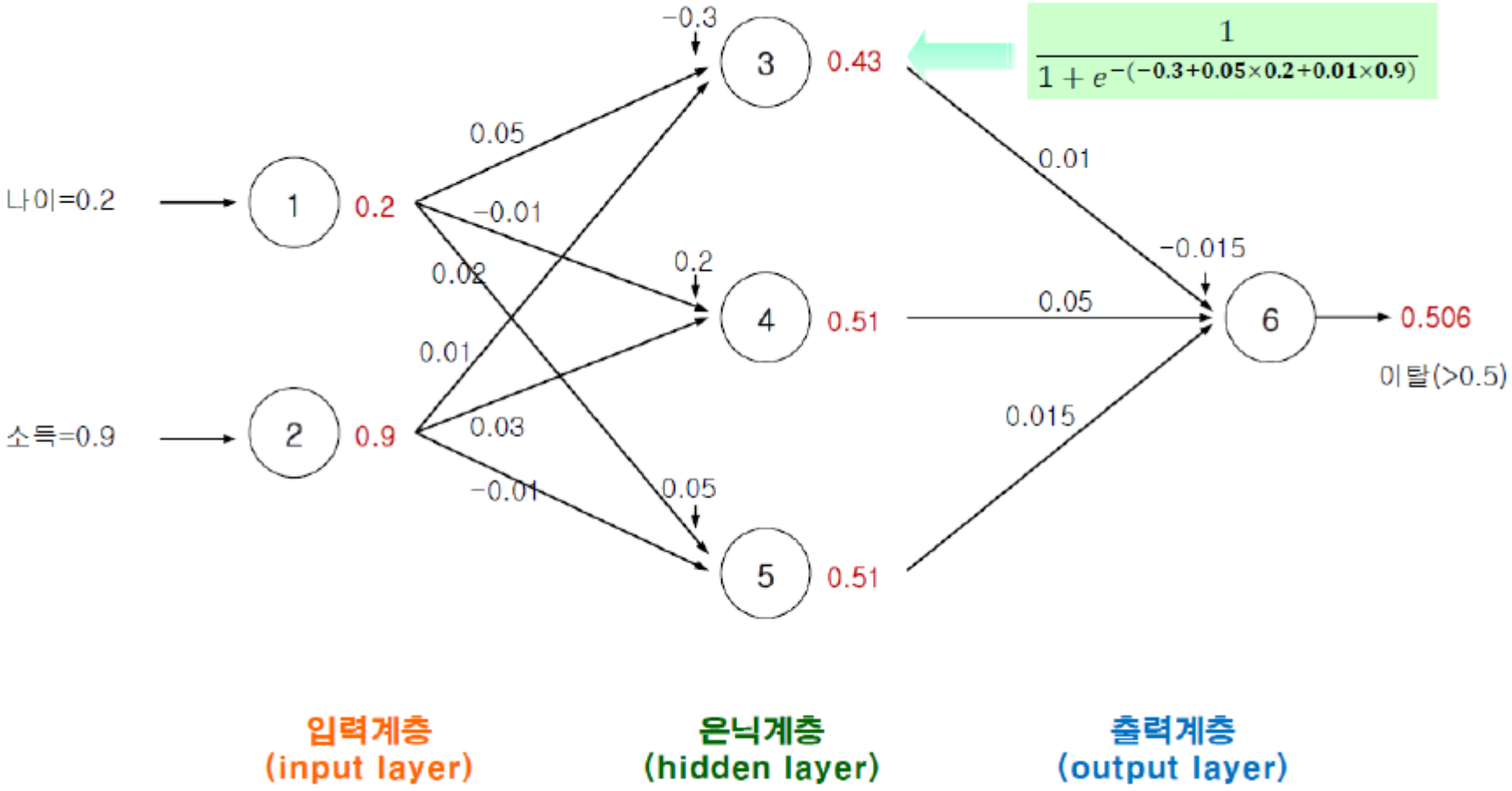
# 다층 퍼셉트론 (MLP)


Input
Artificial neuron
$x_1$
$x_2$
$x_3$
$x_r$
$w_{i,1}$
$w_{i,2}$
$w_{i,3}$
$w_{i,r}$
$\Sigma$
$a_i$
$f$
$y_i$
$b_i$
1
$i_1$
$i_2$
$i_n$
$o_1$
$o_2$
$o_m$
Input level
I hidden level
$k^{th}$ hidden level
Output level

# 다층 퍼셉트론 (MLP)

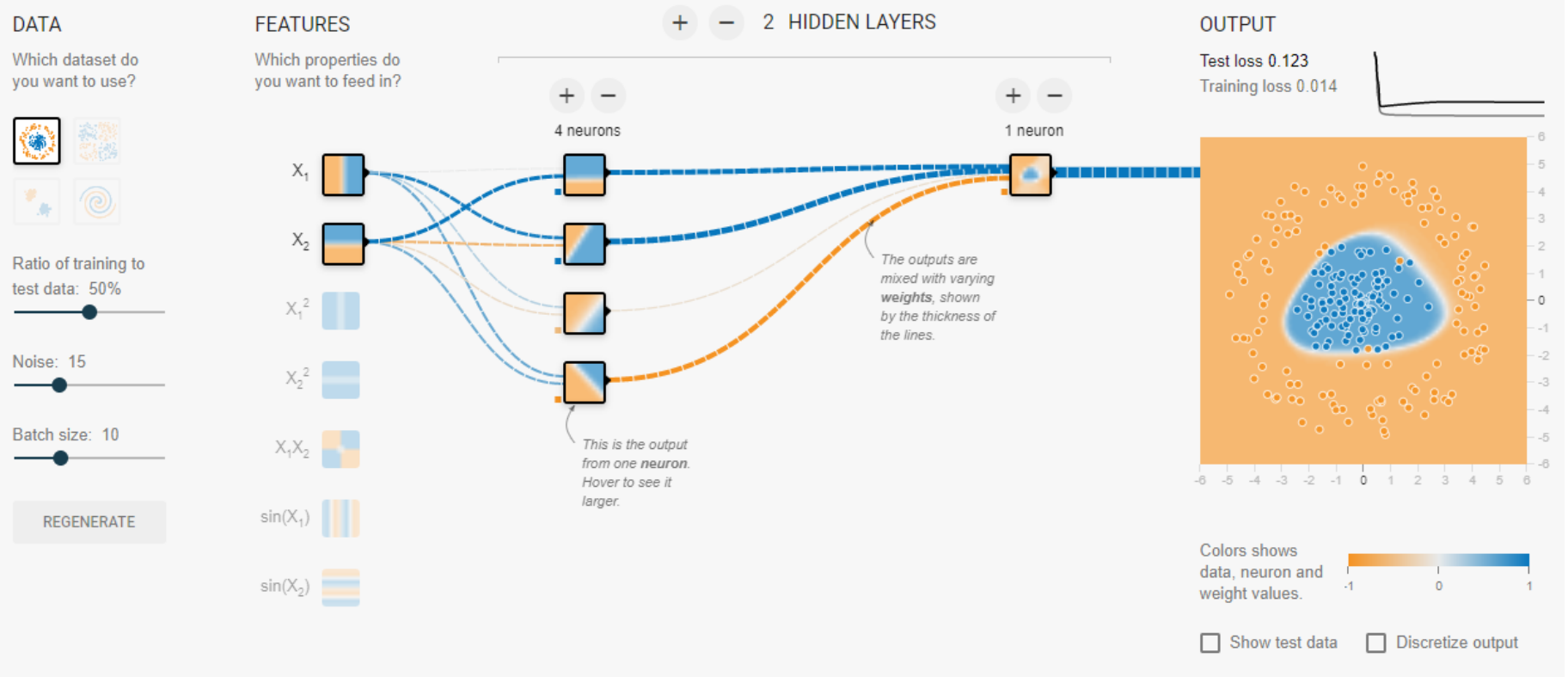

A 3-layers fully connected neural network (DNN)
input feature
neuron
output (class)
bias node
input layer
hidden layer
output layer
weight

# 다층 퍼셉트론 (MLP)


나이=0.2
소득=0.9
1
2
0.2
0.9
3
4
5
0.43
0.51
0.51
6
0.506
이탈(>0.5)
−0.3
0.2
0.05
−0.01
0.02
0.01
0.03
−0.01
0.05
0.01
−0.015
0.05
0.015
$\frac{1}{1+e^{-(-0.3+0.05\times0.2+0.01\times0.9)}}$
입력계층
(input layer)
은닉계층
(hidden layer)
출력계층
(output layer)

# 신경망 동작

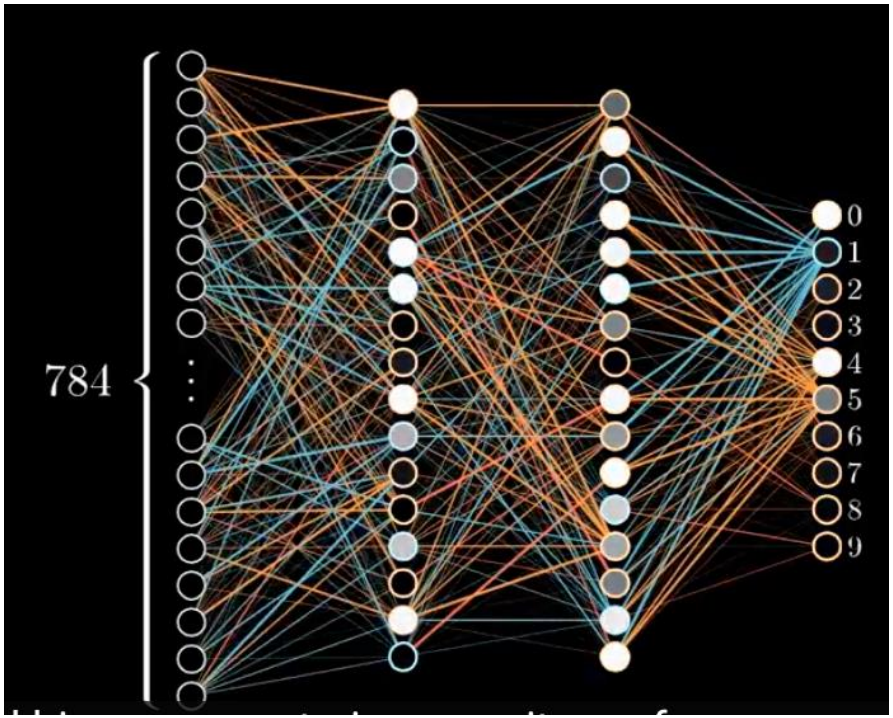
- 신경망 (Playground): https://playground.tensorflow.org/




- 입력 데이터의 분포가 복잡해질수록
  - 더 깊은 신경망 구조와 다수의 신경 유닛이 필요

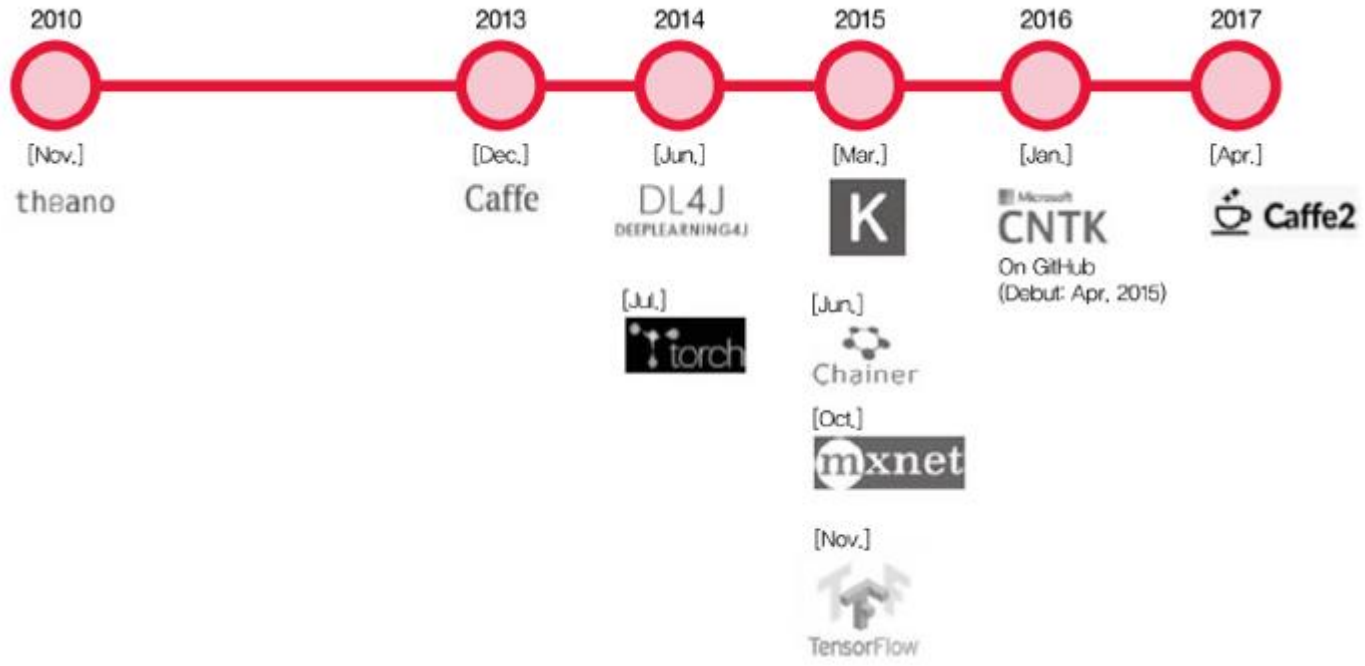
# 신경망 동작 상세 설명

- 동영상 (21:00)



# 텐서플로우

# 딥러닝 소프트웨어


2010
2013
2014
2015
2016
2017
[Nov.]
theano
[Dec.]
Caffe
[Jun.]
DL4J
DEEPLEARNING4J
[Jul.]
torch
[Mar.]
K
[Jun.]
Chainer
[Oct.]
mxnet
[Nov.]
TensorFlow
[Jan.]
CNTK
On GitHub
(Debut: Apr, 2015)
[Apr.]
Caffe2

# 텐서플로우 (Tensorflow)

- 데이터 처리 및 머신러닝에 특화된 소프트웨어로, 신경망 구축을 지원하는 라이브러리
  - 2015년 구글에서 공개

- Python의 머신러닝 라이브러리인 sklearn와 호환되는 tf.learn 라이브러리를 제공

- 시각화 도구로 텐서보드(TensorBoard)를 제공

- 텐서플로우의 가장 큰 특징
  - 다수의 GPU 칩을 병렬로 사용하거나 여러 대의 컴퓨터에서 분산 실행하여 대용량의 신경망 알고리즘을 쉽게 구축할 수 있다

텐서플로우 초보자를 위한 가이드
- https://www.tensorflow.org/tutorials/

# 계산 그래프 (Computational Graph)

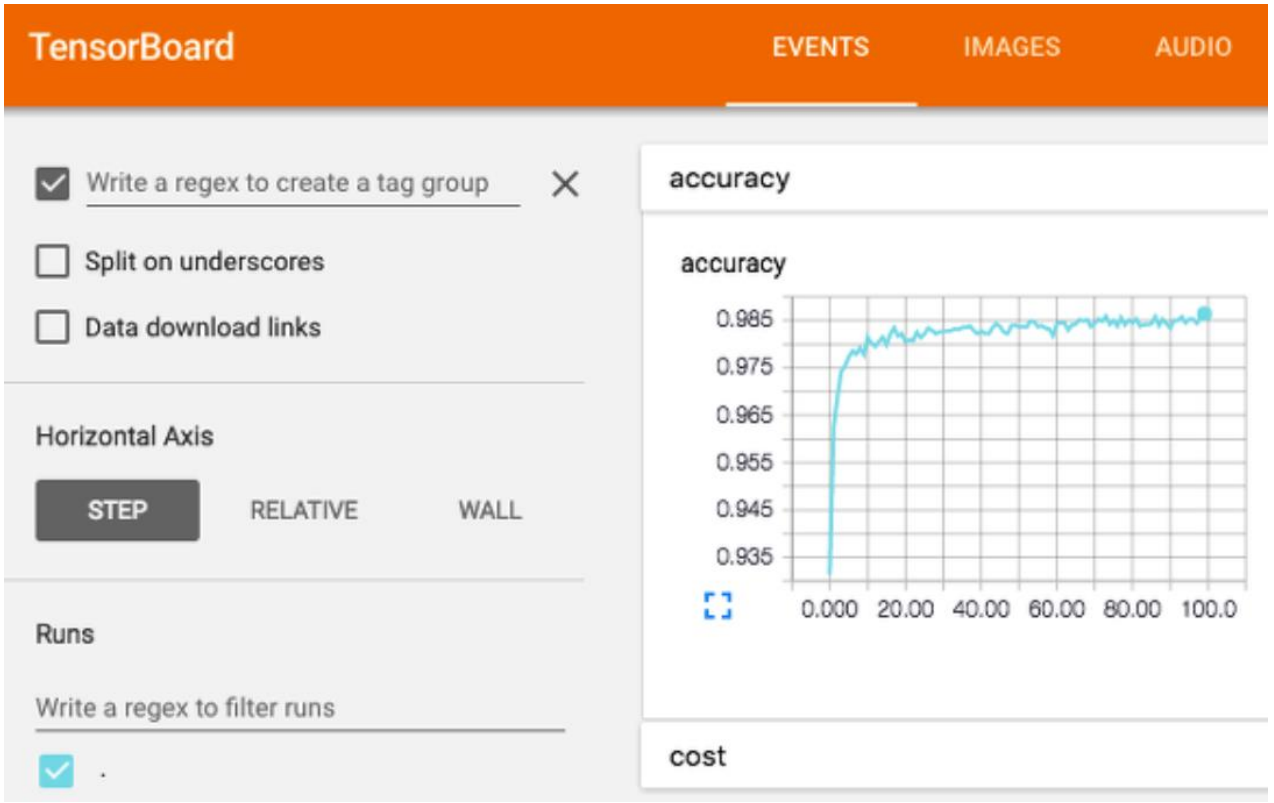
- 텐서플로우의 동작을 그래프로 나타낸 것으로 노드들을 링크로 연결한 구조를 갖는다.
  - 노드 : 어떤 작업이 실행
  - 링크를 따라서 텐서 (즉 데이터)가 전송
  - 노드에는 상수를 담을 수 있다

```
node1 = tf.constant(3.0, tf.float32)
node2 = tf.constant(4.0)                    # also tf.float32 implicitly
print(node1, node2)
```

# 텐서 보드 (tensorBoard)

- 계산 그래프의 내용을 그래픽하게 보여주는 기능을 제공

Tensorboard tutorials



# 텐서플로우 모듈, 클래스, 함수

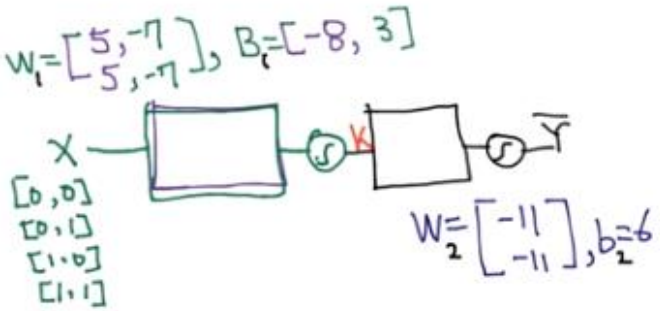
텐서플로우 모듈, 클래스, 함수

텐서플로우 텐서 연산

텐서플로우 2.0 변환

# Tensorflow 표현 (2-stage network)

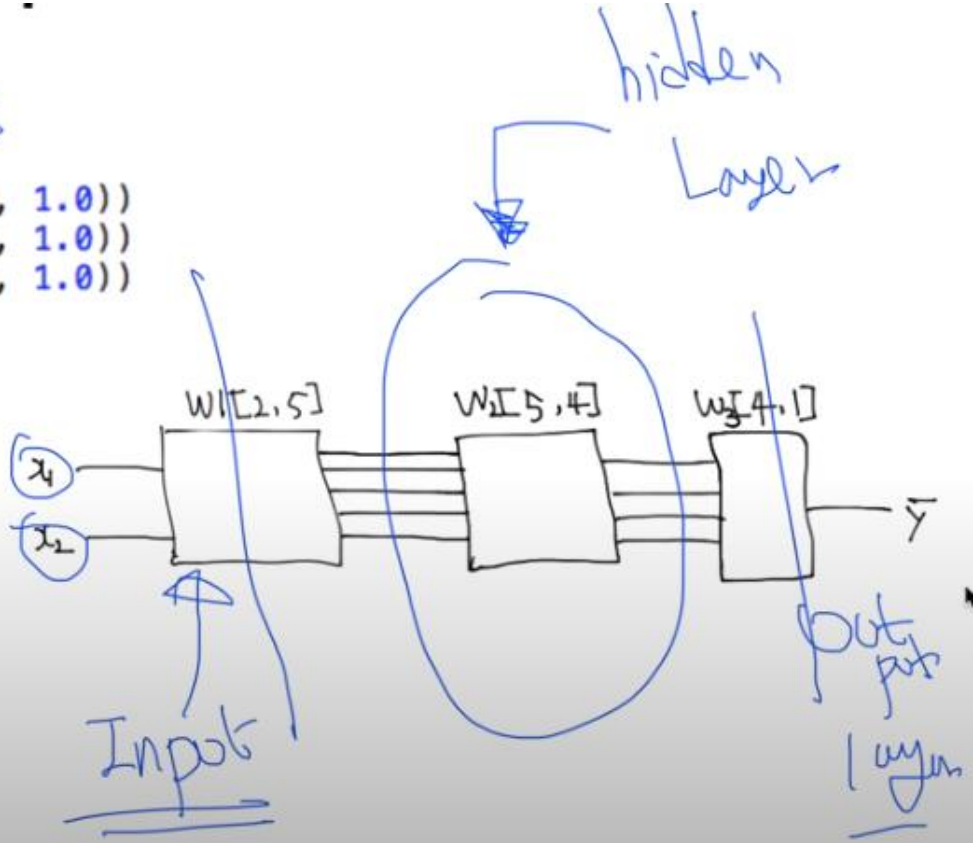
## NN for XOR




```
W1 = tf.Variable(tf.random_uniform([2, 2], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([2, 1], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([2]), name="Bias1")
b2 = tf.Variable(tf.zeros([1]), name="Bias2")

# Our hypothesis
L2 = tf.sigmoid(tf.matmul(X, W1) + b1)
hypothesis = tf.sigmoid(tf.matmul(L2, W2) + b2)
```

# Tensorflow 표현 (3-stage network)

```
W1 = tf.Variable(tf.random_uniform([2, 5], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([5, 4], -1.0, 1.0))
W3 = tf.Variable(tf.random_uniform([4, 1], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([5]), name="Bias1")
b2 = tf.Variable(tf.zeros([4]), name="Bias2")
b3 = tf.Variable(tf.zeros([1]), name="Bias2")

# Our hypothesis
L2 = tf.sigmoid(tf.matmul(X, W1) + b1)
L3 = tf.sigmoid(tf.matmul(L2, W2) + b2)
hypothesis = tf.sigmoid(tf.matmul(L3, W3) + b3)
```



# 고급 API (Keras)

# 고급 API의 필요성

- 심층 신경망(Deep neural networks)이 유행이기는 하나 텐서플로우 등 신경망의 주요 프레임워크들이 원시적인 코딩이 가능하고 세세하게 기능을 구현할 수 있는 라이브러리이지만 사용의 복잡성으로 인해 초보자가 사용법이 다소 복잡

- 초보자도 쉽게 신경망을 구현할 수 있는 고급 API 형태의 라이브러리의 제공이 필요함

https://keras.io/applications/

# TensorFlow APIs

**TensorFlow** provides multiple APIs (Application Programming Interfaces).

These can be classified into 2 major categories:

1. Low level API:
    - complete programming control
    - recommended for machine learning researchers
    - provides fine levels of control over the models
    - **TensorFlow Core** is the low level API of TensorFlow.
2. High level API:
    - built on top of **TensorFlow Core**
    - easier to learn and use than **TensorFlow Core**
    - make repetitive tasks easier and more consistent between different users
    - **tf.contrib.learn** is an example of a high level API.

# 텐서플로우 Core API : 저 수준 API

- https://www.tensorflow.org/guide/eager
- 텐서플로우 계산 그래프
- 텐서
- 연산
- 세션
- 텐서플로우를 처음 접하는 사람에게는 이해하기가 어려운 부분이 있다
- 사용할 경우 얻는 이점도 있으므로(대부분 디버깅 관련) 고수준과 저수준 텐서플로우 API를 필요에 따라 섞어 사용하면 된다.

# 텐서플로우 Core API : 저 수준 API

  - 신경층(neural layer)
  - 비용 함수(cost function)
  - 옵티마이저(optimizer)
  - 초기화 방식(initialization scheme)
  - 활성화 함수(activation function)
  - 정규화 방식(regularization scheme)

- 모두 독립적인 모듈이며 결합을 통해 새로운 모델을 만들 수 있다.
- 새로운 모듈을 새 클래스와 함수로 간단히 추가할 수 있다.
- 모델은 별도의 모델 구성 파일이 아닌 파이썬 코드로 정의된다.

# 텐서플로우 사용 딥러닝 모델 개발 단계

1. 데이터 준비 : 수집 및 전처리
2. 모델 구성
   - 입력층, 은닉층, 출력층 등 계층 구성 포함
3. 손실 함수 정의
4. 최적화 알고리즘 선택
5. 모델 훈련
6. 모델 평가 및 예측
7. 모델 저장 및 재사용
   - 모델 구조 & 가중치 저장 및 로드

# 케라스 (Keras)

- 딥러닝 모델을 python으로 쉽게 구축해주는 패키지 (고급 API)
  - Python으로 작성
- 다양한 Backend 신경망 엔진 지원
  - Tensorflow, Theano, CNTK(Microsoft), MXNet, Plai의 등과 같은 플랫폼으로 기반으로 동작
- 고수준 Keras API
  - tf.keras
  - https://www.tensorflow.org/guide/keras
- 케라스 설치 (colab – 미리 설치되어 있음)
  - conda, pip 명령으로 설치
  - Pip3 install keras

https://keras.io/applications/

# 케라스 (Keras)

- 텐서 곱(tensor products), 합성곱(convolutions)과 같은 저수준 작업을 자체적으로 수행하지 않고 백엔드에 의존
- 여러 백엔드 엔진을 지원하지만 주 백엔드이자 기본 백엔드는 텐서플로우
- 케라스의 가장 큰 지지 기업도 구글

# 케라스 (Keras) – model

- 핵심 데이터 구조
- 두 가지 주 모델 유형
  - Sequential Model: linear stack of layers
  - Functional API: way to define complex models, such as multi-output models, directed acyclic models, or shared layers

# 케라스 (Keras) – Sequential model

- 케라스 시퀀셜(Sequential) 모델
  - 계층의 선형적인 스택
    - 입력과 출력이 각각 하나
  - 계층은 아주 단순하게 기술이 가능 : *model.add()* 사용
  - 각 계층 정의에 한 줄의 코드가 필요
  - 컴파일(학습 프로세스 정의)에 한 줄의 코드가 필요
  - 피팅(학습), 평가(손실 및 메트릭 계산), 학습된 모델에서의 예측 출력에서 각각 한 줄의 코드를 사용

```
1 model = Sequential([
2             Dense(32, input_shape=(784,)),
3             Activation('relu'),
4             Dense(10),
5             Activation('softmax'),
6 ])
```

```
10 model = Sequential()
11 model.add(Dense(32, input_dim=784))
12 model.add(Activation('relu'))
```

# 케라스 (Keras) – Sequential model

- Specifying Input shape
  - *input_shape* (or *input_dim* and *input_length*)
  - *batch_size*
- Compilation
  - Optimizer
  - Loss function
  - List of metrics
- Training
  - Train on Numpy arrays of features and label

```
model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

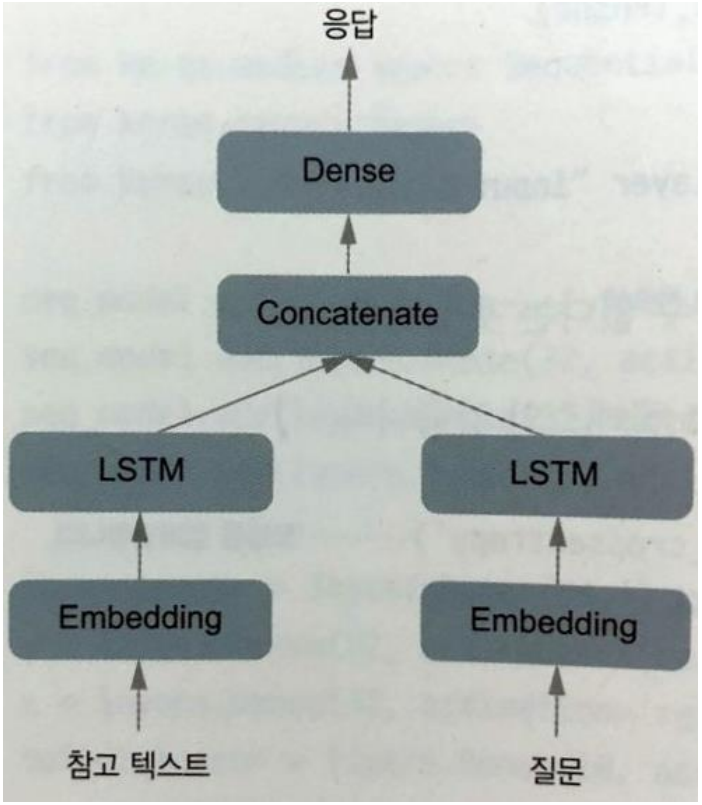
# 케라스 (Keras) – 함수 API model

- 케라스 함수 API
  - 케라스 시퀀셜 모델은 간소하지만 모델 토폴로지는 제한적
  - 다중 입력/다중 출력 모델
  - 방향성 비순환 그래프(DAG)
  - 공유된 계층이 있는 모델과 같은 복잡한 모델을 만드는 데 유용
  - 시퀀셜 모델과 같은 계층을 사용하지만 조합 측면에서 더 높은 유연성 제공
  - 먼저 계층을 정의한 다음 모델을 생성하고 컴파일하고 피팅(학습)한다.
  - 평가와 예측은 기본적으로 시퀀셜 모델과 동일

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)

# 입력과 출력 텐서를 지정하여 Model 클래스의 객체를 만듭니다.
model = Model(input_tensor, output_tensor)
```

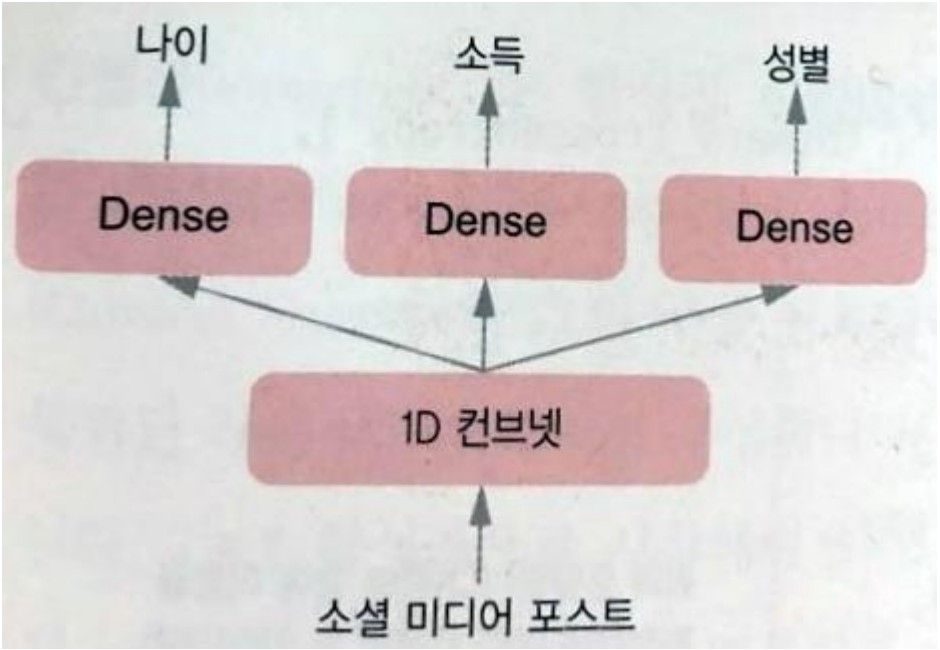
# 케라스 (Keras) – 함수 API model

- 다중 입력 모델
  - ➢ 질문-응답(Question-Answering) 모델



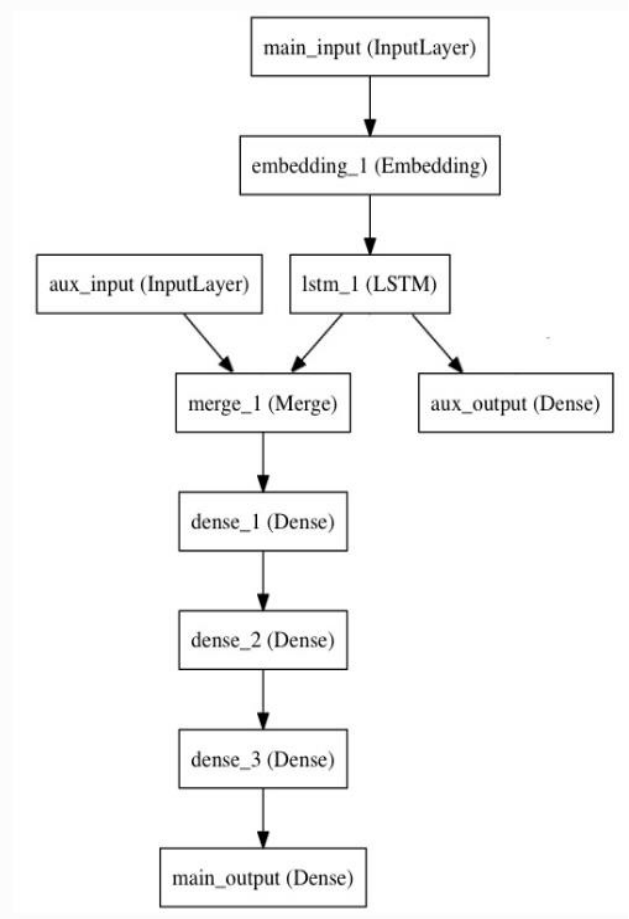
# 케라스 (Keras) – 함수 API model

- 다중 출력 모델
  - ➢ 익명 사용자의 포스트로부터 그 사람의 나이, 성별, 소득 수준 예측



# 케라스 (Keras) – 함수 API model

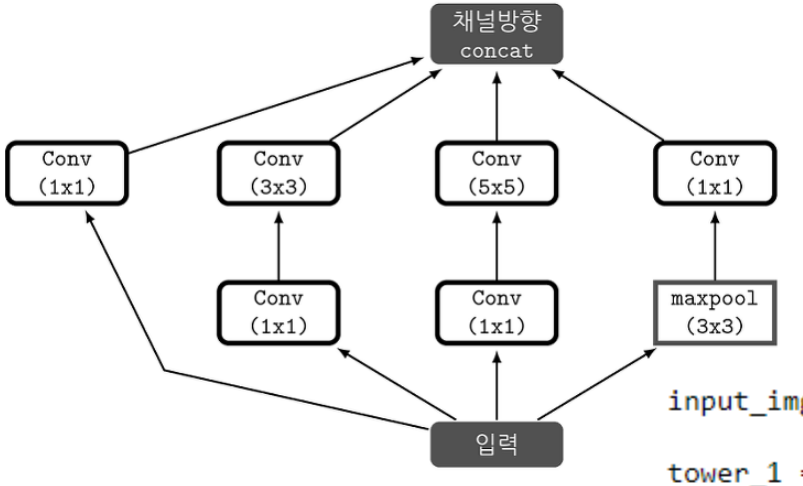
- multi-input multi-output models example (다중 입력  다중 출력 모델)




```
# Headline input: meant to receive sequences of 100 integers, between 1 and 10000
# Note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# This embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# A LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)

auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)

auxiliary_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])

# We stack a deep densely-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# And finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')(x)

model = Model(inputs=[main_input, auxiliary_input], outputs=[main_output, auxiliary_output])
```

# 케라스 (Keras) – 함수 API model

- 내부 토폴로지가 복잡한 모델
  - ➢ Inception 모듈 (GoogleNet)




```
input_img = Input(shape=(256, 256, 3))

tower_1 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_1 = Conv2D(64, (3, 3), padding='same', activation='relu')(tower_1)

tower_2 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_2 = Conv2D(64, (5, 5), padding='same', activation='relu')(tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), padding='same')(input_img)
tower_3 = Conv2D(64, (1, 1), padding='same', activation='relu')(tower_3)

output = keras.layers.concatenate([tower_1, tower_2, tower_3], axis=1)
```

# GPU (Graphics Processing Unit)

- 3D 그래픽 연산 전용의 processor
  - 엔비디아(NVIDIA)사 처음 개발, 가장 널리 사용
- 신경망 모델을 학습하려면 GPU를 사용해야 속도가 빠르다
  - driver인 CUDA와 cuDNN(cuda Deep Neural Network)을 설치
  - 텐서플로우도 gpu 버전(Tensorflow-gpu) 을 설치
- GPU를 동시에 사용 – 케라스 명령
  - Data parallelism: replicating the target model once on each device, and each device processes a different fraction of the input data
    - ✓ Refer to keras.utils.multi_gpu_model
  - Device parallelism: running different parts of a same model on different devices
- Colab 사용하면 GPU를 무료로 사용