# 딥러닝의 이해

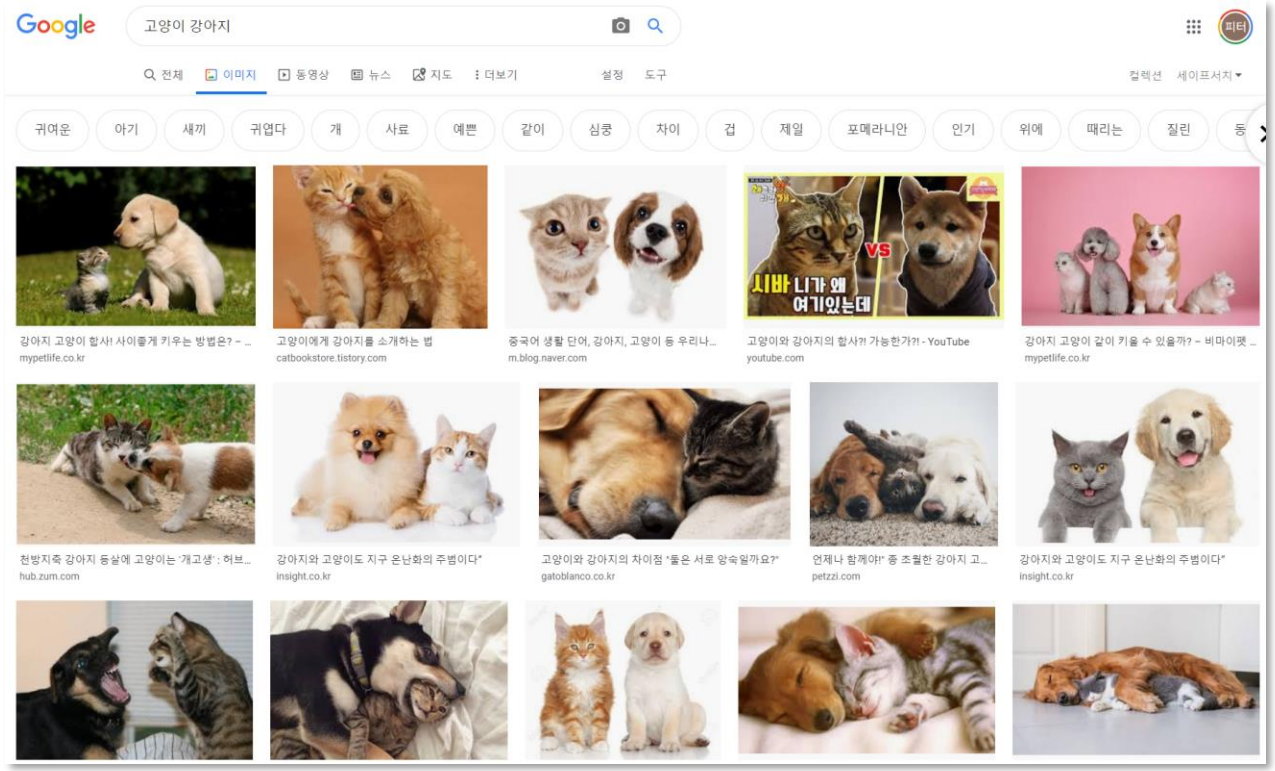
## 딥러닝vs머신러닝

# 개와 고양이 사진 분류하기

딥러닝은 머신러닝과 같이 "모델을 학습하고 데이터를 분류할 수 있다"
그럼 차이점은 무엇일까?

***수많은 개와 고양이 이미지들을 분류하고 싶다면,***

머신러닝vs딥러닝 둘 중 무엇을 써야 할까?



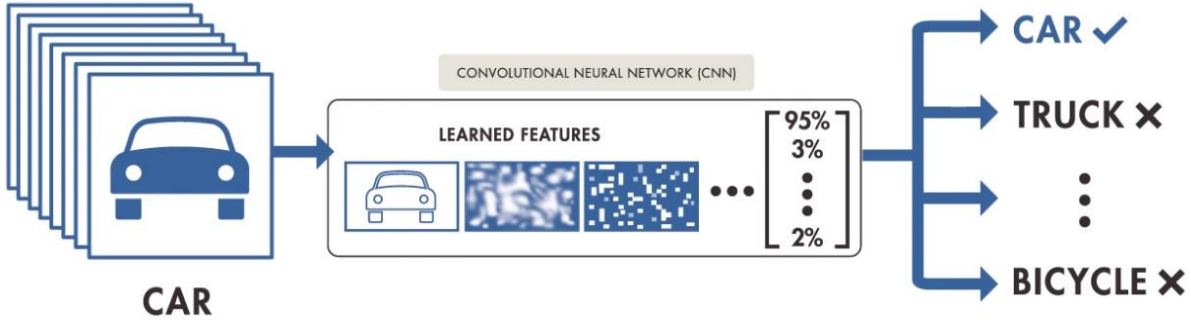
# 머신러닝 vs 딥러닝

머신러닝


1. 우선 이미지에서 관련 특징들을 추출한다.
2. 물체의 특징을 예측하는 모델을 만든다.

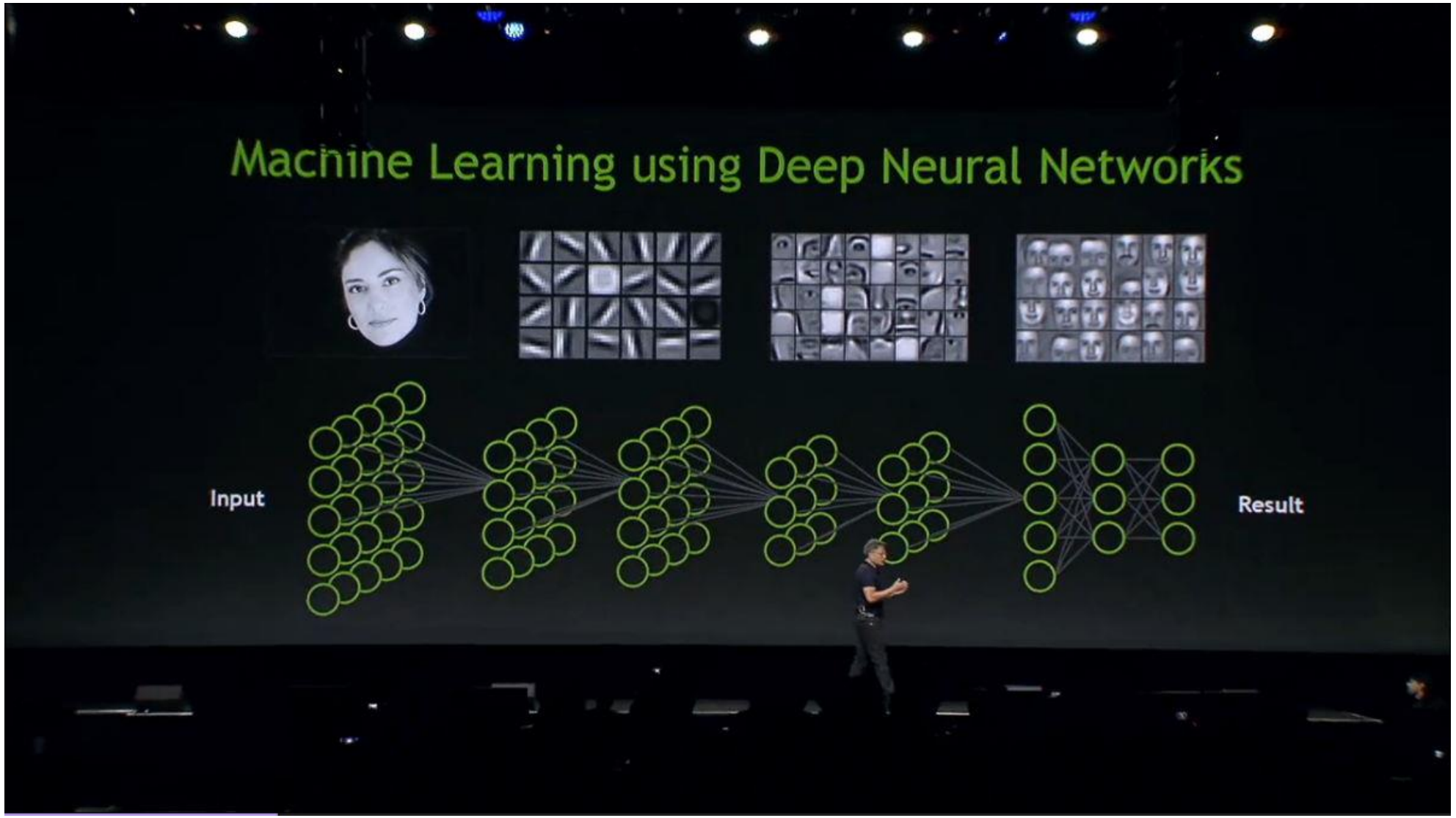
✓ 반면에 딥러닝은 수동으로 이미지에서 특징 추출을 할 필요가 없다.

딥러닝




1. 이미지 자체를 딥러닝 알고리즘에 제공함으로써 예측이 가능하다.
   즉, 특징 추출이 필요 없다.

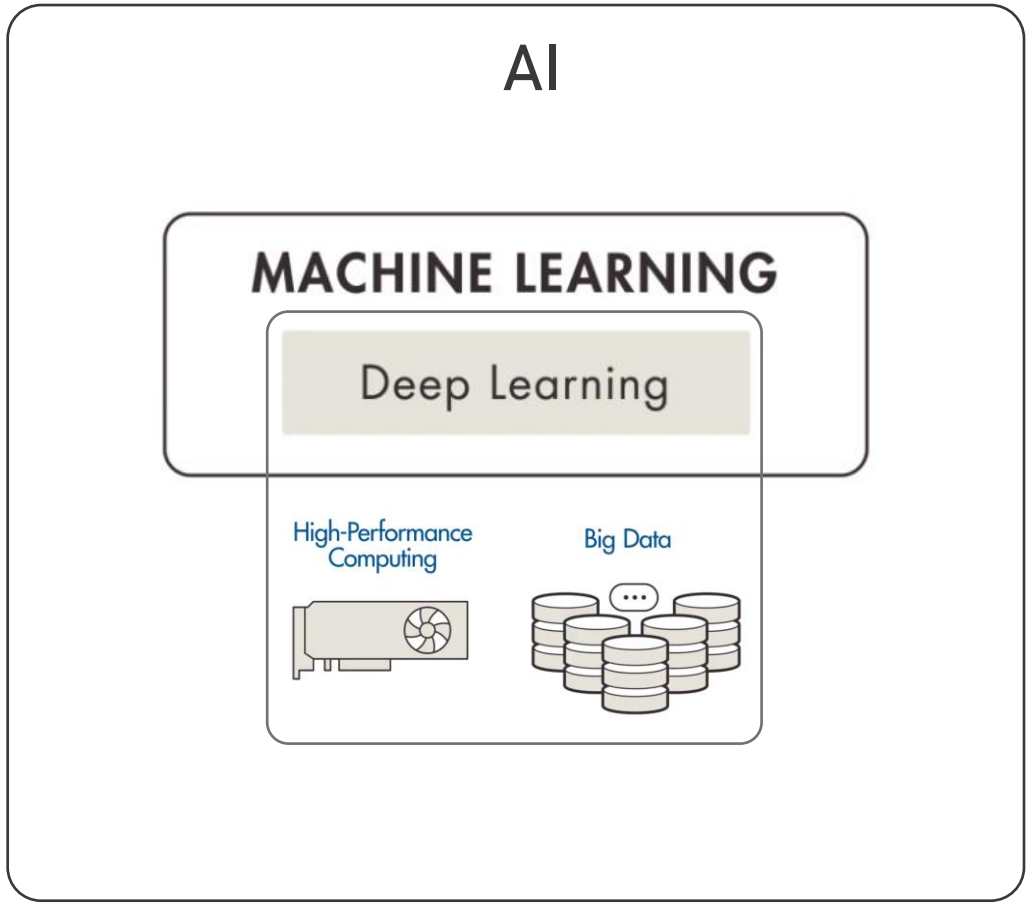
# 딥러닝은 스스로 피쳐를 통계적으로 뽑아낸다.




딥러닝이 피쳐들을 계층적으로 추출하는 모습

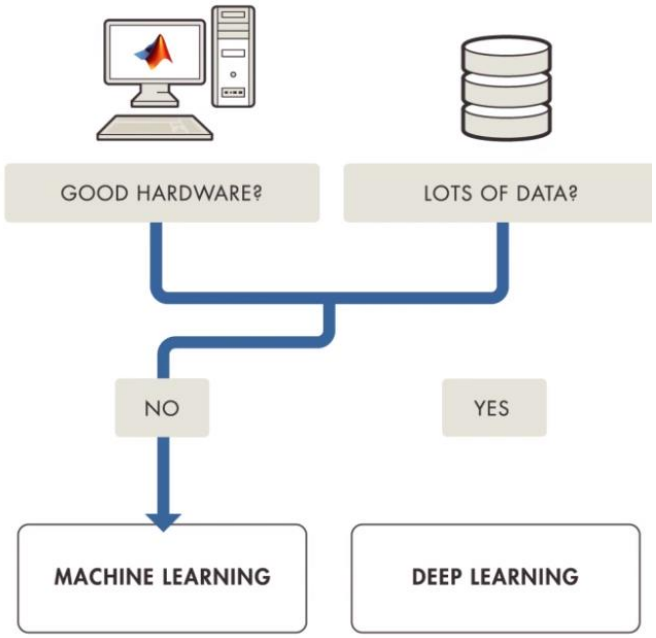
# 딥러닝이란

딥러닝은 머신러닝의 한 부분집합이다.
이미지를 입력 그대로 직접 처리할 수 있으며, 더 복잡할 수 있다.



# 딥러닝 사용이 가능한 경우

1. 데이터가 많을 때
2. 컴퓨팅 파워가 좋을 때(특히 GPU-그래픽카드 좋은 것, 램 용량 큰 것 등등)




GPU Nvidia RTX2070 8G (61만)
RTX2080Ti 11G (151만)
GTX1060 6G (35만)

*성능 : 2080Ti 〉 2070 〉 1080Ti 〉 1070Ti*
*가성비: 2070 〉 1060 〉 1070Ti 〉 1080Ti*

- 딥러닝은 수많은 뉴런들이 동시에 연산을 한다.
그래서 병렬 처리가 중요!

- GPU(Graphics processing unit) : 그래픽 처리 장치
원래 모니터에서 일어나는 수많은 모니터 픽셀 연산을 병렬 처리하는데 사용하는 장치였는데

최근 들어
✓비트코인 채굴,
✓딥러닝
에 사용하고 있다.

# 컴퓨터 사양이 별로 여도 딥러닝이 가능은 하다.

물론, 노트북으로도 딥러닝이 가능하다.
EX) 구글 COLAB에 접속해서 딥러닝을 돌릴 수 있다.

즉, COLAB은 구글에서 컴퓨터를 빌리는 것

단점: 그런데 답답하다.
- 쥬피터 노트북에 익숙한 사용자
- 인터넷 환경에 따라 버벅거림
- 클라우드는 얼마 이상의 데이터를 사용하면 돈이 든다.

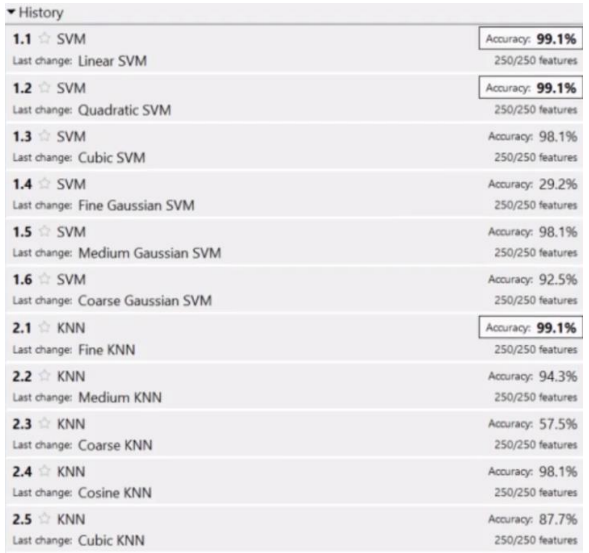

일단, 딥러닝을 COLAB으로 어느정도 돌려보다가 본격적으로 딥러닝 하려면 GPU 컴퓨터를 사는 게 낫다.
클라우드 비용 몇십만원 나올 거면 GPU를 사고 만다.

Google Cloud

## 머신러닝과 딥러닝은 각각 특징이 명확하다.

만약 조건(컴퓨팅 파워, 데이터 양)이 충분하다면 머신러닝보다 딥러닝의 정확도가 더 높은 경향이 있다.
어느 것을 사용할 지는 해결하고자 하는 과제와 그 데이터의 성질에 달려 있다.

| | Machine Learning | Deep Learning |
|---|---|---|
| Training dataset | Small | Large |
| Choose your own features | Yes | No |
| # of classifiers available | Many | Few |
| Training time | Short | Long |



# 딥러닝 기초

## 딥러닝의 발전 과정

# History of Deep learning

1943 **Neural Net** – 인공지능의 시작 (Warren Sturgis McCulloch)

1958 **Perceptron** (Frank Rosenblatt)

1969~1979 **XOR 문제** – 인공신경망의 첫번째 위기 (Marvin Minsky)

1980 **다층 퍼셉트론** (쿠니히코 후쿠시마)

1986 **역전파** (Geoffrey Everest Hinton)

1989 **CNN** (Yann Lecun)

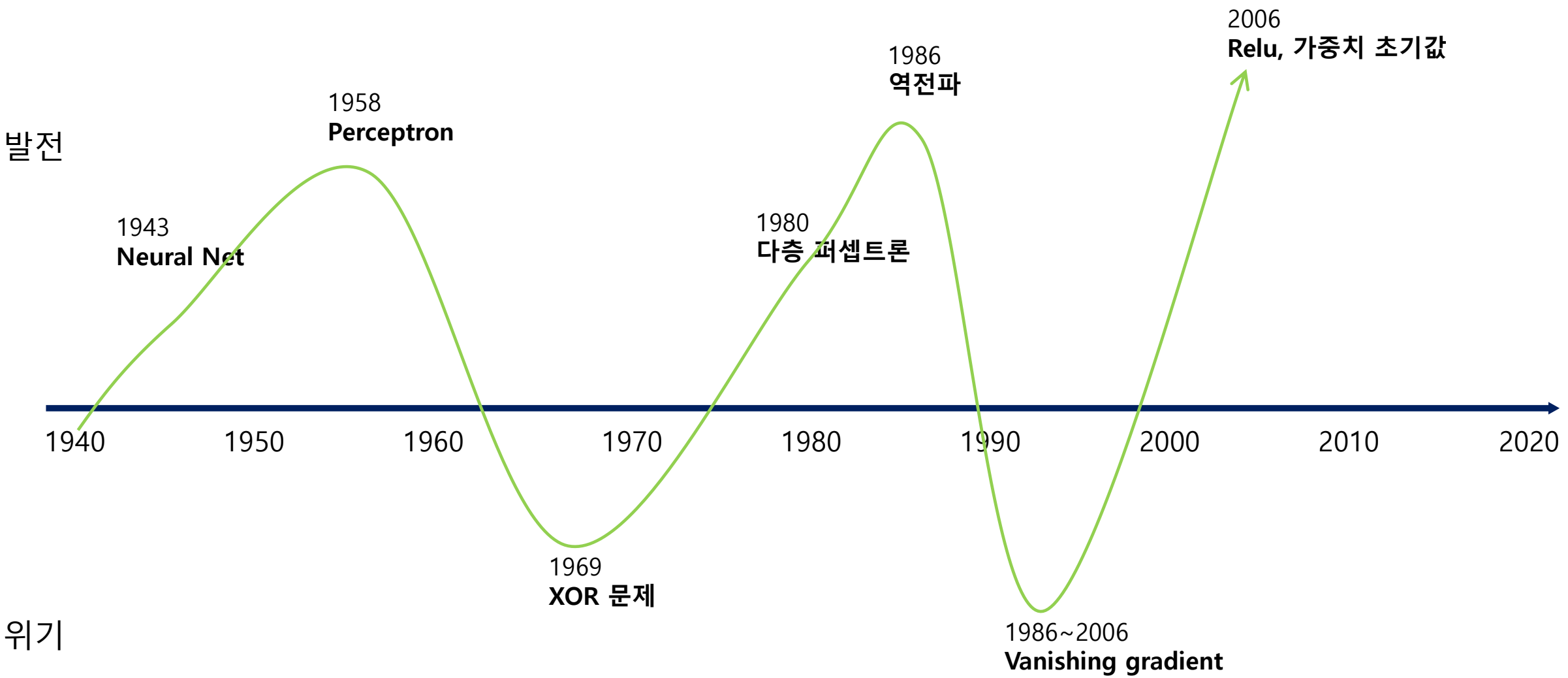
1990~2005 **Vanishing gradient 문제** – 인공신경망의 두번째 위기

2006 **Relu, 가중치 초기값** (Geoffrey Everest Hinton)

2011 **Drop out** (Geoffrey Everest Hinton)

2012 **이미지넷 대회 breakthrough** (Geoffrey Everest Hinton)

# 딥러닝의 발전과 위기


발전
위기
1943
Neural Net
1958
Perceptron
1969
XOR 문제
1980
다층 퍼셉트론
1986
역전파
1986~2006
Vanishing gradient
2006
Relu, 가중치 초기값
1940
1950
1960
1970
1980
1990
2000
2010
2020

# History of Artificial Intelligence

Alan Turing

"Artificial Intelligence"
By John McCarthy

Eliza,
The first Chabot
By MIT, 1965

Deep Blue,
Chess Playing
Computer
By IBM, 1997

Self-driving
cars
By Google, 2009

DeepMind
AlphaGo
By Google, 2016

1950 | 1960 | 1970 | 1980 | 1990 | 2000 | 2010

Symbolic AI,
Explicit rules

Expert systems

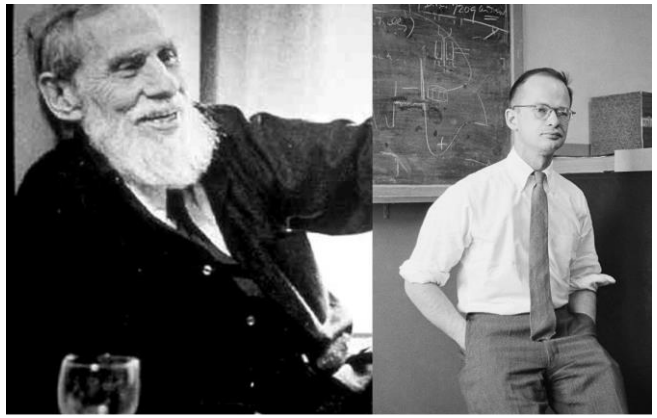
**The first AI Winter**
**1974 ~ 1980**

**The Second AI Winter**
**1987 ~ 1993**

XOR Problem,
No one on earth had found a viable way to train MLP(Multilayer perceptions)
By Marvin Minskey, 1969

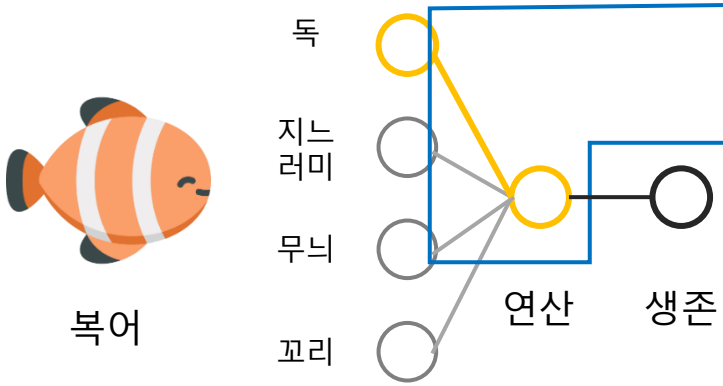
# 1943 Neural Net – 인공지능의 시작

1943년 뉴런 인공신경망 모델을 처음 생각해낸 사람은 신경생리학자 워렌 맥컬록과 수학자 윌터 피츠



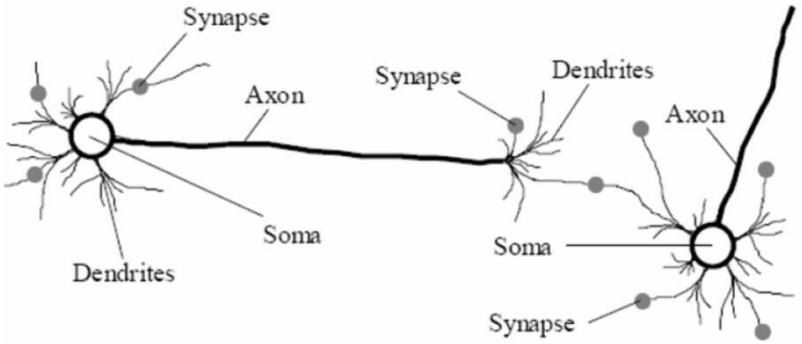
워렌 맥컬록(좌), 윌터 피츠(우)

논문 "신경 활동에 내재한 개념들의 논리적 계산"(1943년)

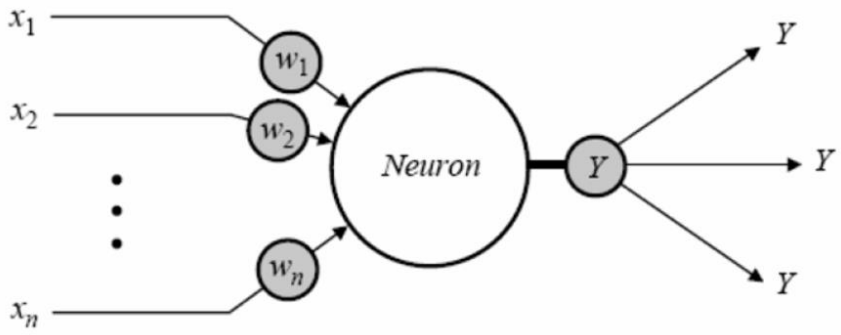



**인공신경망(Artificial Neural Network)**
**(줄여서 Neural Net)**

아직까진
딥러닝




신경세포(Neuron)의 구조

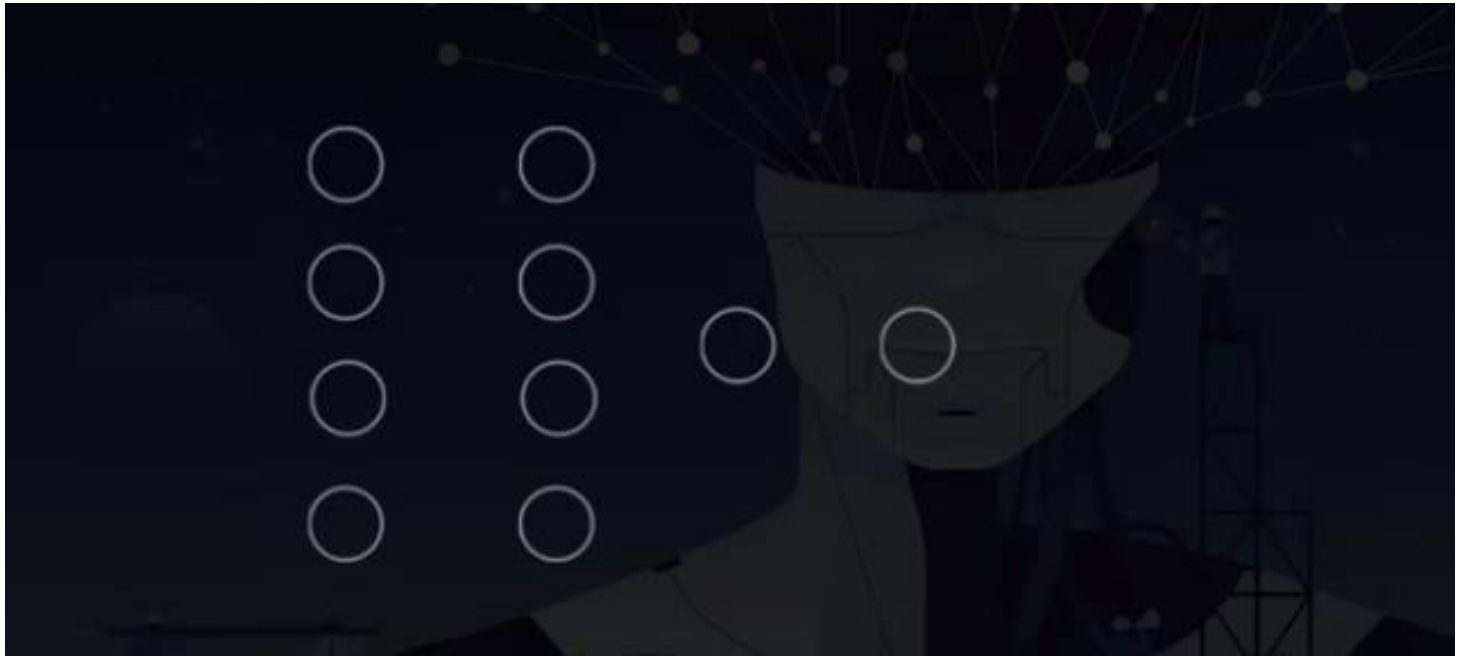



$$X = \sum_{i=1}^{n} x_i w_i \qquad Y = \begin{cases} +1, & \text{if } X \geq \theta \\ -1, & \text{if } X < \theta \end{cases}$$

신호 전달의 원리

# 1943 Neural Net – 인공지능의 시작

Neural Net의 학습 과정

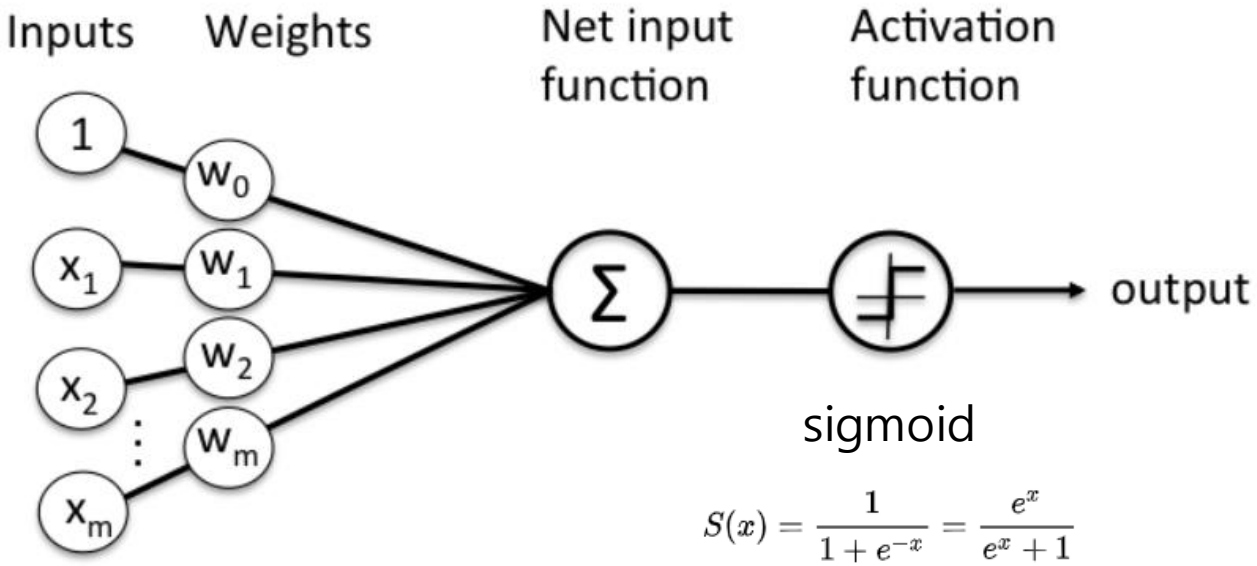


# 1958 Perceptron

Perceptron : Neural Net을 응용해서 실제 문제를 해결

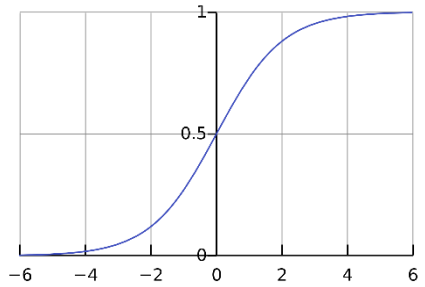
Frank Rosenblatt

논문 "The perceptron: A probabilistic model for information storage and organization in the brain."




sigmoid

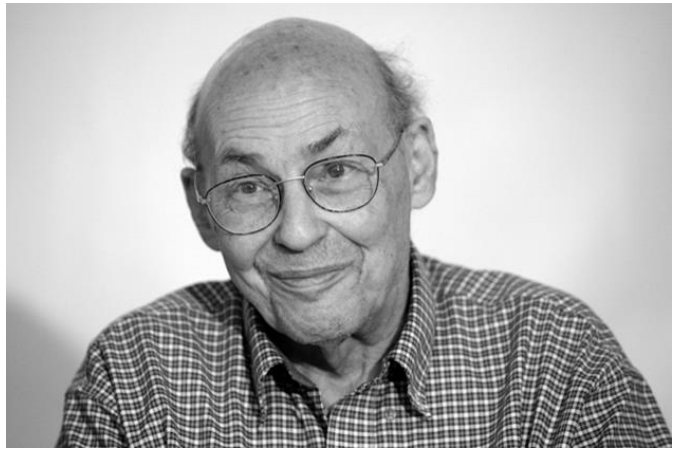
$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$




Perceptron
Linear Classification
Feed forward neural network
Input, weight
Activation fuction(sigmoid)

-> AI 투자 붐이 일어남

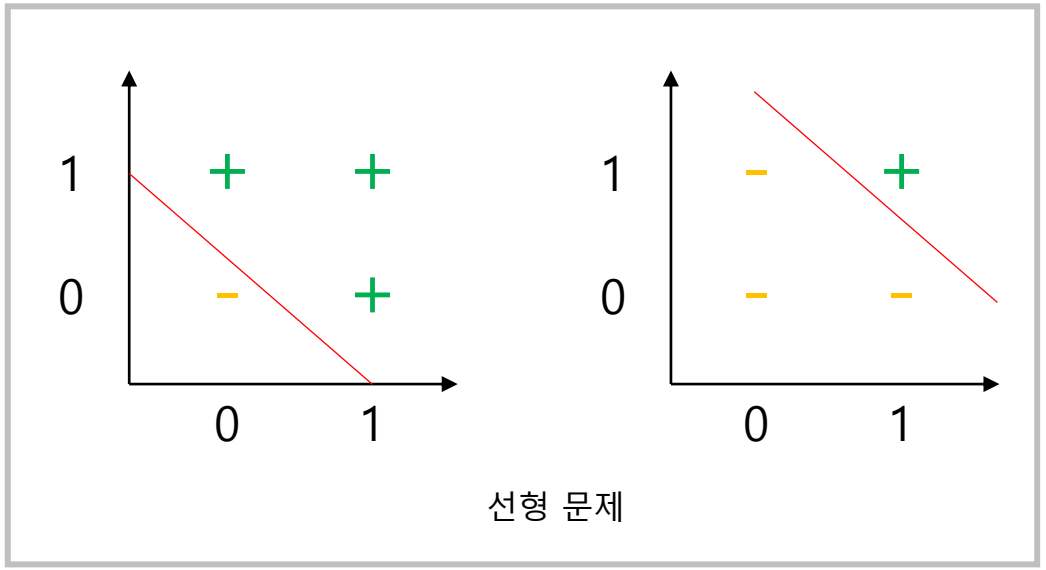
# 1969 XOR 문제 – 인공신경망의 첫번째 위기



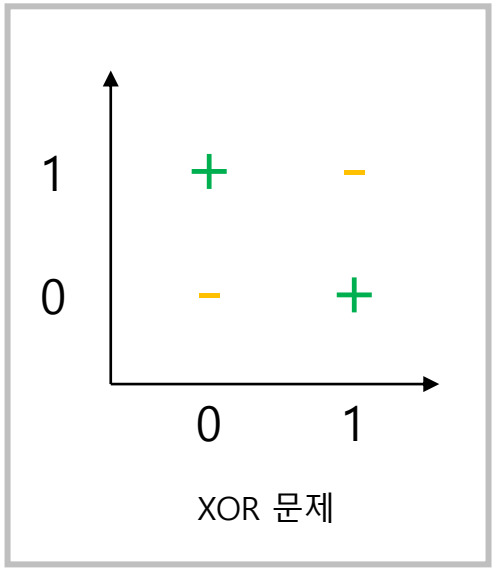
Marvin Minsky

책 "Perceptrons: an introduction to computational geometry"

"Perceptrons은 단순한 선형 분류기이다.
비선형 문제를 해결할 수 없다.
예를 들어 XOR 문제가 바로 그것이다."




퍼셉트론이
해결 가능

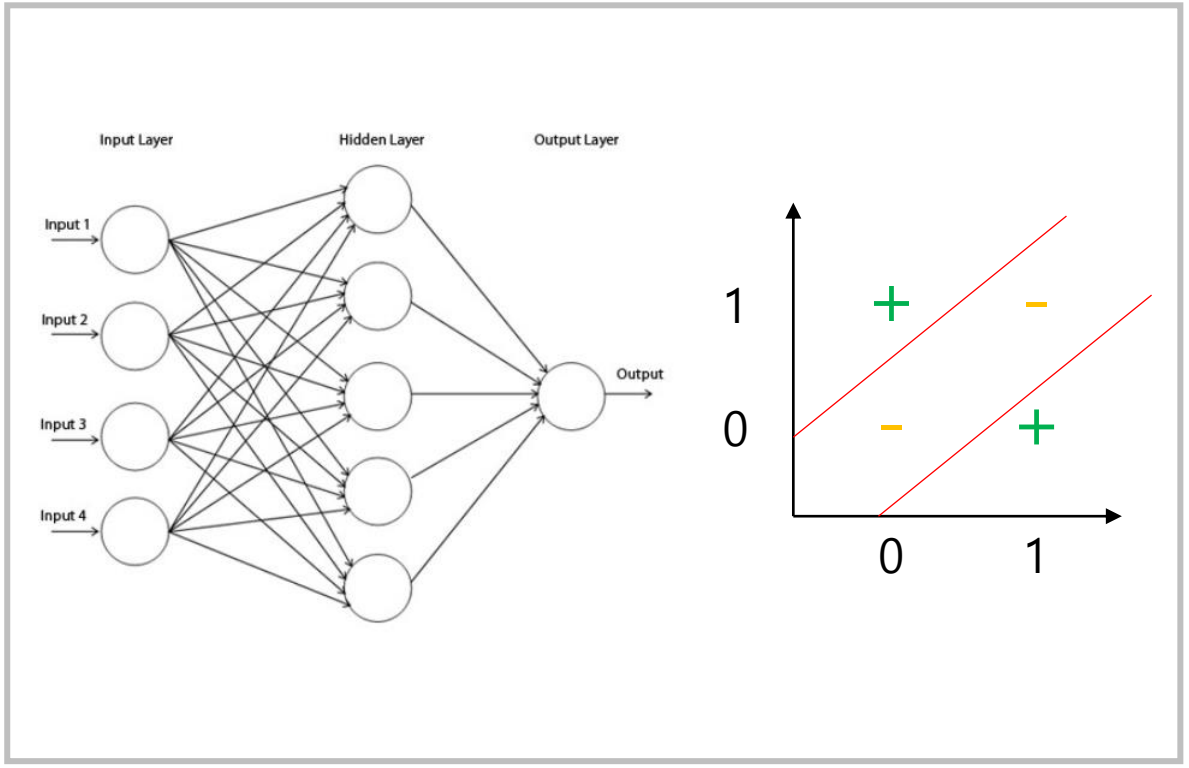



**퍼셉트론이
해결 불가능**

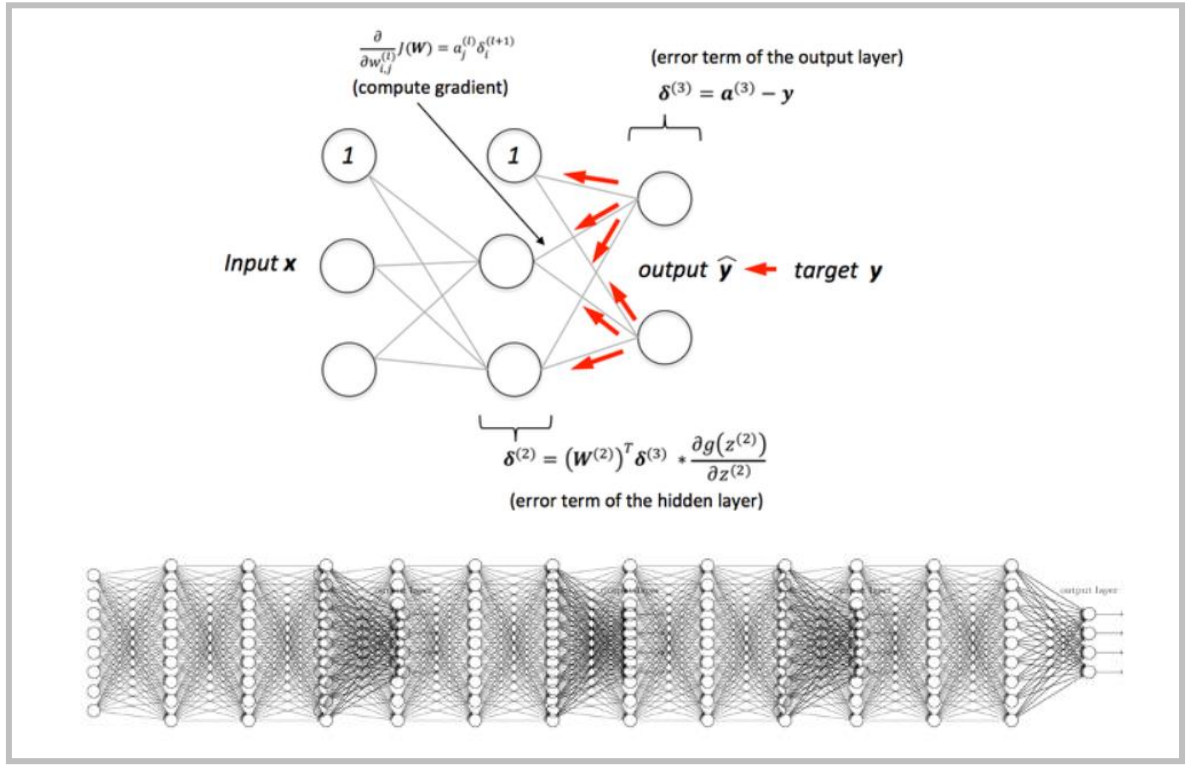
독
지느러미
무늬
꼬리
?

# 1980 다층 퍼셉트론(쿠니히코 후쿠시마), 1986 역전파(제프리 힌튼)

다층 퍼셉트론(Multi-layer Perceptron, MLP), 역전파(backpropagation)를 이용해서 XOR 문제 해결




다층 퍼셉트론(좌)과
MLP를 이용한 XOR(비선형) 문제 해결(우)



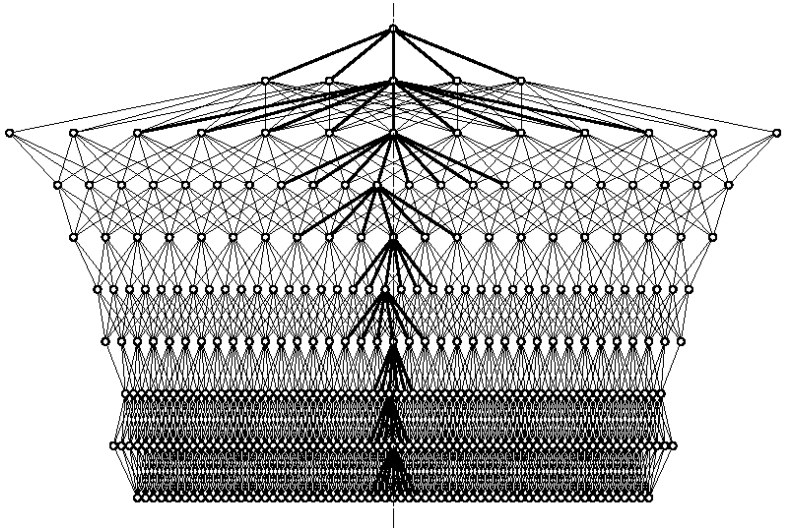

역전파법

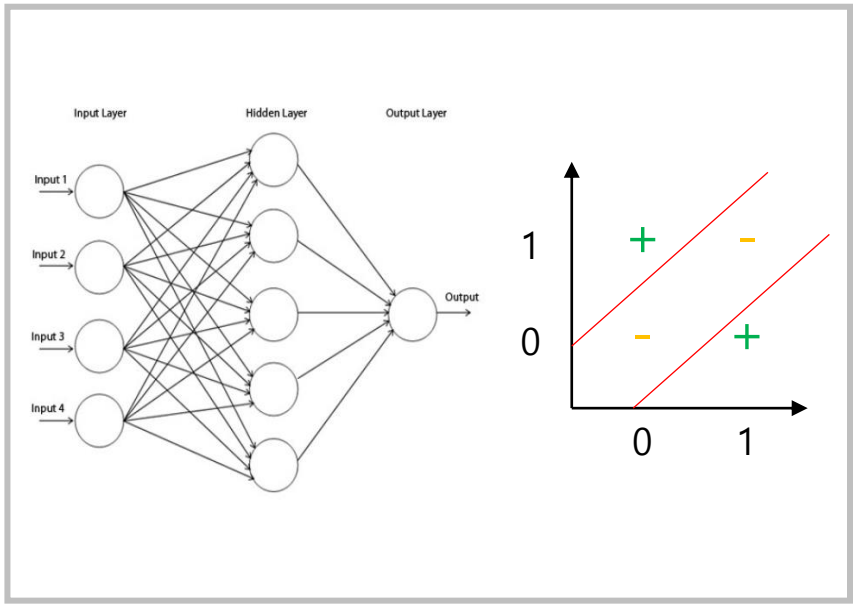
여러 층(다층 퍼셉트론)을 거치면서 가중치(weight)를 계산하고,
거꾸로 돌아가면서 에러율을 조정하는 역전파법을 통해 XOR 문제는 해결된다.

# 1980 다층 퍼셉트론(쿠니히코 후쿠시마)

네오코그니트론(Neoognitron) : 고양이의 시각 피질에서 아이디어를 얻은 합성곱 필터를 접목하여 CNN을 구현하였다.
네오코그니트론의 구조는 인공 지능의 깊은 신경망, 특히 시각적 패턴 인식을 사용하는 네트워크의 핵심이다.

쿠니히코 후쿠시마



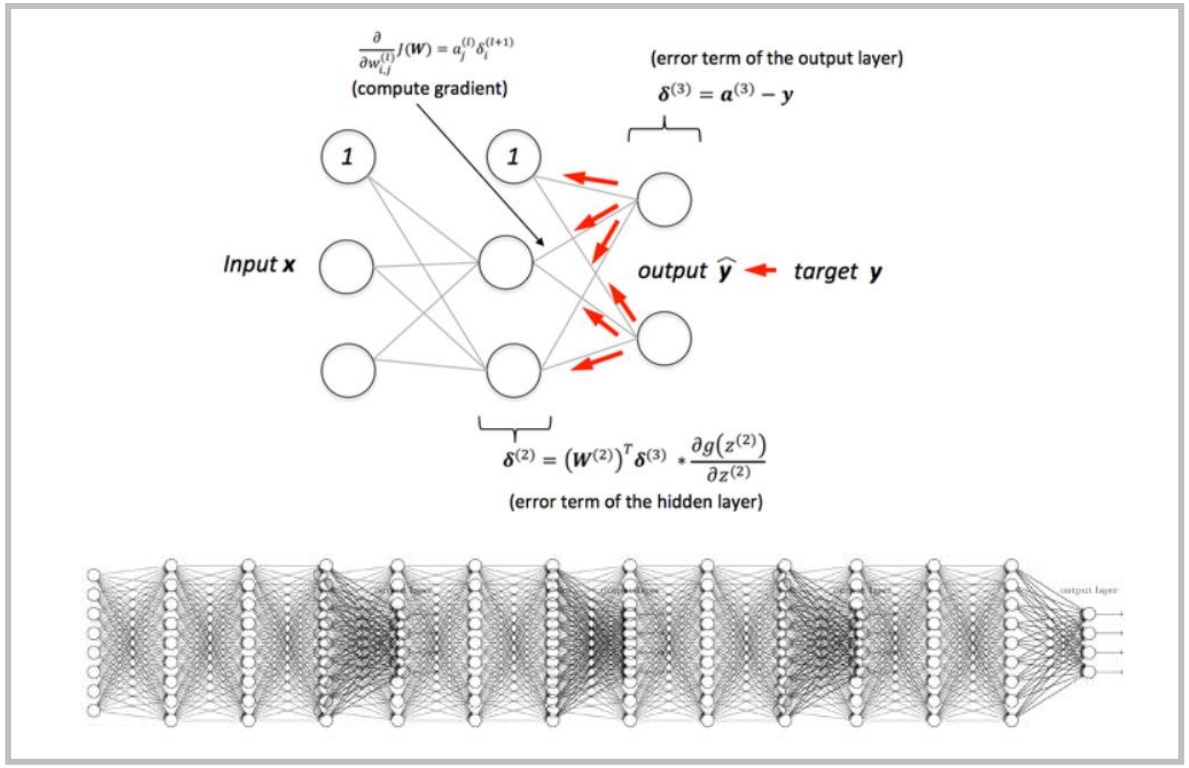



다층 퍼셉트론(좌)
MLP를 이용한 XOR(비선형) 문제 해결(우)

# 1986 역전파(제프리 힌튼)

역전파(backpropagation)를 이용해서 XOR 문제 해결

Geoffrey Everest Hinton




역전파법

여러 층(다층 퍼셉트론)을 거치면서 가중치(weight)를 계산하고,
거꾸로 돌아가면서 에러율을 조정하는 역전파법을 통해 XOR 문제는 해결된다.

# 1989 CNN(Convolution Neural Net)의 초기 모델 -> **딥러닝(Deep learning)의 시작**

MNIST 손글씨 분류 성공. 복잡한 이미지 인식은 아직 힘든 상황

Yann Lecun
제프리 힌튼의 제자

LeNet1 시연 영상

얀 르쿤은 1989년 AT&T 벨 연구소에 있을 때 CNN(Convolutional Neural Network)을 처음 발표했습니다. 이 연구는 쿠니히코 후쿠시마의 시각 피질 모델링에 영감을 얻어 단순/복잡 세포 위계를 지도 학습(supervised training)과 역전파(backpropagation)에 적용함으로써 가능했던 결과였습니다. 해당 연구는 320개의 mouser-written 숫자들을 사용했습니다. MNIST 데이터는 이때 만들어졌습니다.

1998년 LeNet5는 은행에서 수표 검사를 하는데 사용 되었습니다.

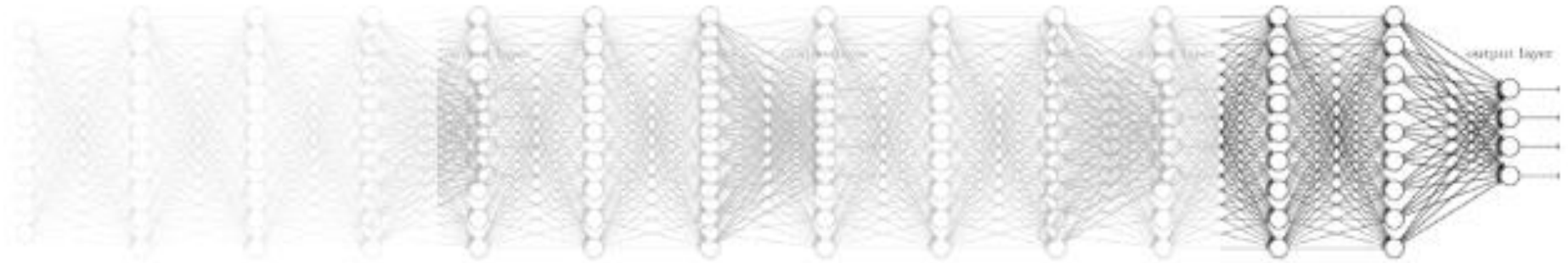
하지만 당시에는 간단한 이미지만 인식이 가능했습니다. 보다 큰 사이즈의 복잡한 사진을 처리하려면 모델이 더 커져야 합니다. 그렇게 하면 깊은 층으로 학습이 안되는 그래디언트 소실과, 훈련 데이터에만 딱 맞는 과대적합이 발생합니다. 이런 문제들을 해결하지 못했기 때문에 신경망에 대한 관심이 다시 사라지게 됩니다.

논문 “Handwritten digit recognition with a back-propagation network”

성과는 있었지만 아직 한계점들이 존재
vanishing gradient, overfitting

# 1990~2005 Vanishing gradient 문제 – 인공신경망의 두번째 위기

Activation function으로 sigmoid를 사용하는데 이 값은 0과 1사이 값만 전달하므로,
Hidden layer를 여러 층 거치면서 Vanishing gradient 된다(값이 현저히 줄어든다)
한 층당 1/10로 줄어든다면, 세 층을 거치면 1/1000로 줄어든다.

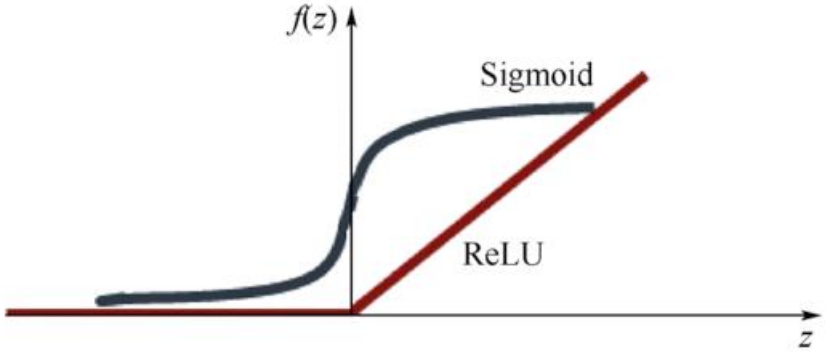


Sigmoid말고 좋은 activation function이 없을까

# 2006 1. ReLU

Activation function으로 Relu(Rectified Linear Unit)를 사용해보니,
0보다 작은 값은 0으로, 0보다 큰 값은 그 값 그대로 출력해준다. -> Vanishing gradient 문제 해결!

Geoffrey Everest Hinton




**Sigmoid(0~1)**

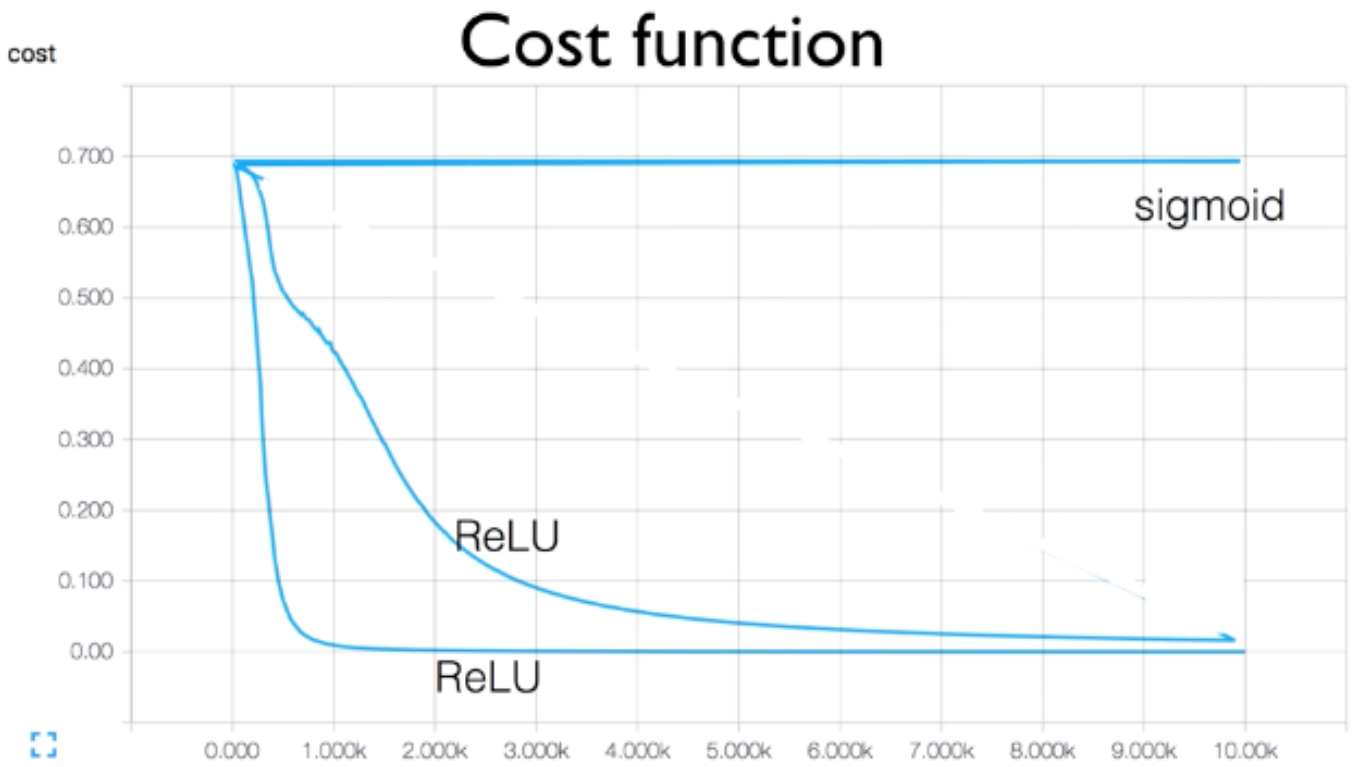
$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

**Relu(0 or X)**

**= max(0, x)**

# 2006 2. 가중치 초기값 세팅

가중치(weight) 초기값 세팅을 어떻게 하느냐에 따라 반복을 적게 해도 좋은 결과가 나온다.



# 2011 Drop out – 과적합 방지(일반화 능력 향상)

훈련 시 랜덤하게 신경망의 연결을 끊어버리기. 그럼 특정 노드에 의존하지 않아 일반화 능력이 향상됨.
모의고사 때 일부러 어려운 문제를 풀어서 실제 수능 볼 때 문제해결력 향상!

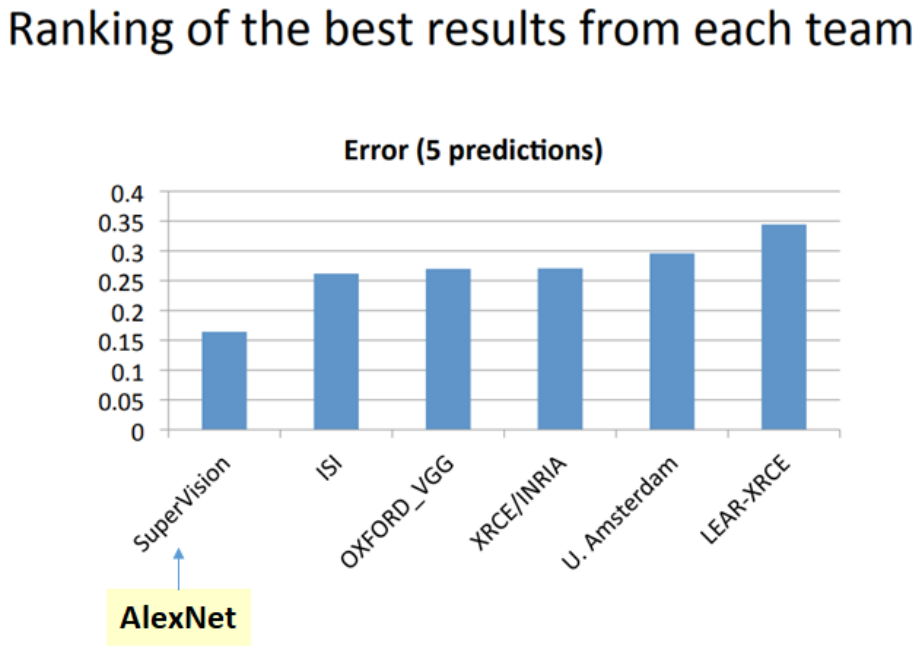
**드롭아웃 발견 스토리**

2004
- 레드포드 닐이 해준 얘기: 인간의 뇌의 용량이 이렇게 큰 이유는 어쩌면 뇌 안에 여러 모델이 있고 그 모델을 합치는(ensemble) 것 때문일지도 모른다는 것이었죠.
- 얼마 지나지 않아서 은행을 갈 일이 있었습니다. 그런데 은행을 갈때마다 창구 직원이 매번 바뀌더라구요. 같은 논리로 계속 다른 뉴런의 부분집합을 제거하면 뉴런들의 음모 - 즉 과적합(overfitting)을 막을 수 있지 않을까 하는 생각을 했어요.
- 당시엔 잘 돌아가지가 않아서 l2-reguralisation이 더 나은것으로 결론을 내리고 잊고 있었습니다.

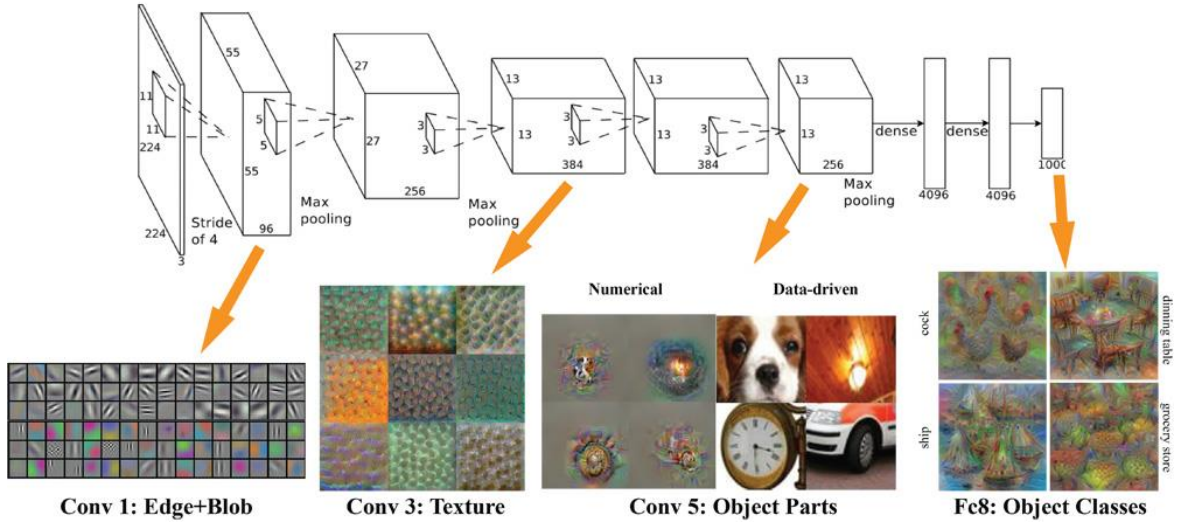
2011
- 2011년에 크리스토스 파파디미트리우가 토론토에서 강의하는걸 들었습니다.
  생물의 2세 생산이 (두 유전자를 임의로 합치는 과정에서) co-adaptation을 막는 의미를 갖는다는 내용이 있었습니다.
- 대학원생들과 함께 좀 더 열심히 구현을 해봤고 결과적으로 이 이론이 잘 작동한다는 것을 밝혀냈습니다.

# 2012 ILSVRC(ImageNet Large Scale Visual Recognition Competition) – 이미지넷 대회 우승

힌튼 연구팀은 신경망 방식의 Alexnet(얀 르쿤 CNN의 업그레이드 버전)로 기존보다 10% 성능 향상시키고 대회 우승(다른 팀들은 대부분 SVM 같은 전통적인 머신러닝 기법 사용)

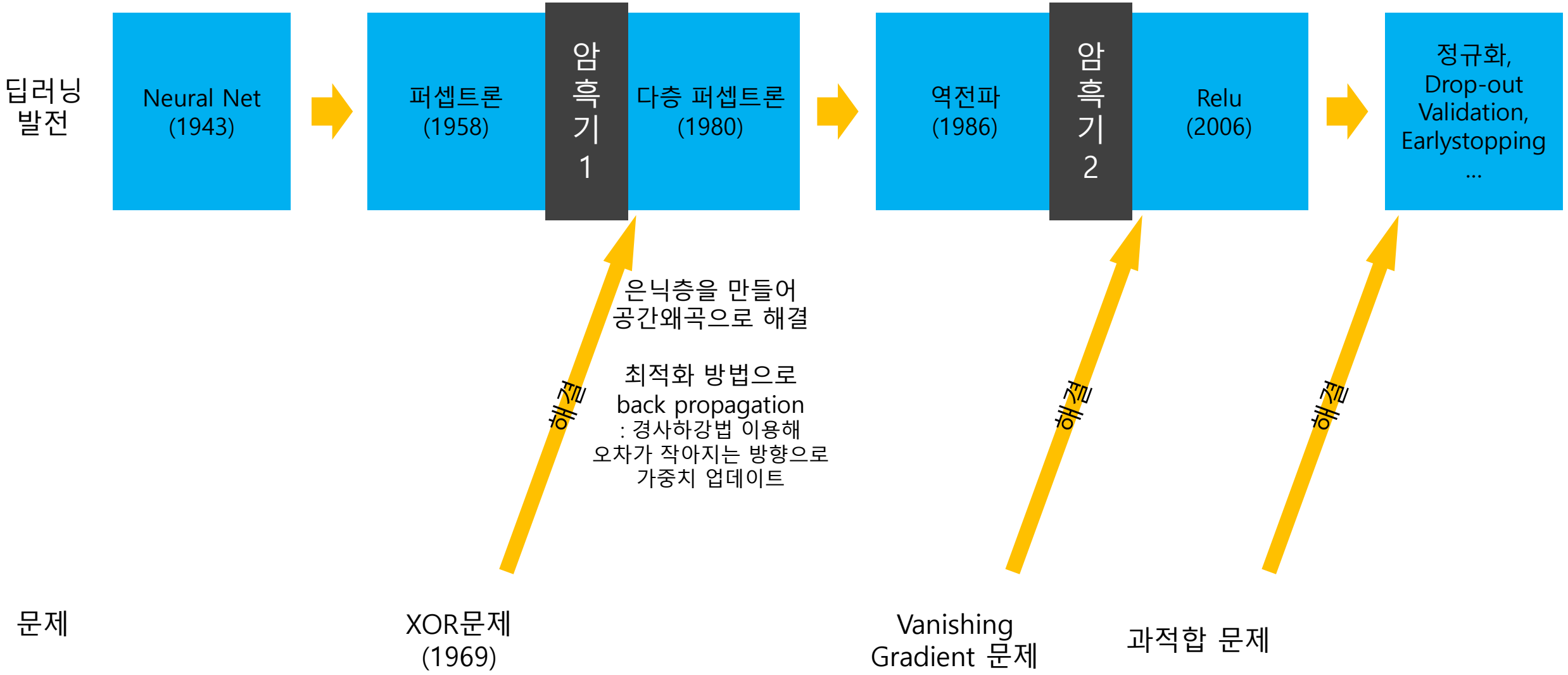



많은 사람들은 힌튼이 우승했던 이유가 CNN(Convolutional neural network) 덕분이라고 알고 있습니다. CNN은 이전의 신경망처럼 모든 노드들이 완전히 연결되어 있지 않고 컨볼루션과 풀링을 사용해 이미지의 특징을 뽑아냅니다. 여러 층을 지날 수록 단순한 특징에서 복잡한 특징으로 추상화됩니다.



# 딥러닝 발전 과정

딥러닝은 뉴럴 넷으로부터 시작해서 여러가지 위기를 넘기며 발전함



# 딥러닝 기초

## 폐암환자 생존예측

# 딥러닝 모델 적용 프로세스 정리

머신러닝(딥러닝 포함)은 일반화(generalization)와 최적화(optimization)의 줄다리기이다.


<보유 데이터>
- DB
- 클라우드

<크롤링>
- 셀레늄
- Request, 뷰티풀숲
- Rest API

<데이터 벡터화>
- 텍스트 -> 정수화
- 이미지 -> 원-핫

<정규화>
- 0~1 값
- 평균0, 표준편차1
- 특성 균일화

<누락 값 처리>
- 누락값 0으로

<특성 공학>
- 딥러닝은 특성공학 X
- NLP n-그램, BOW

<activation>
- Sigmoid, ReLU, Softmax, etc.

<과대적합 방지>
- 가중치 규제(L1, L2)
- Dropout

<하이퍼 파라미터 튜닝>
- 층의 노드 수
- optimizer 학습률

<optimizer>
- Adam, RMSProp 등

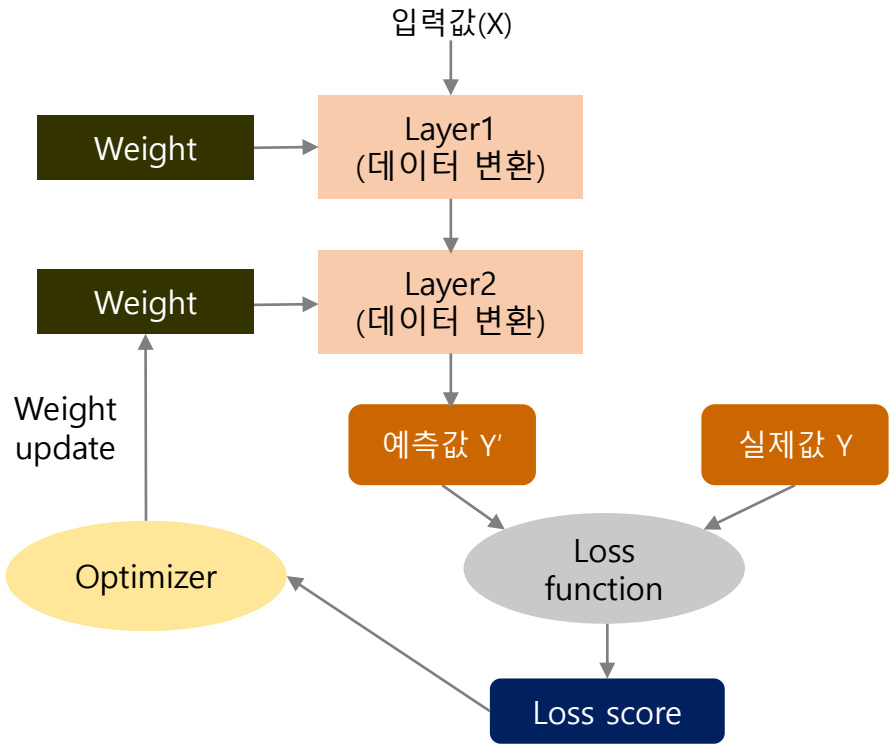
<loss 함수>
- 회귀: MSE
- 분류: crossentropy (ROC, AUC과 반비례)

<학습 파라미터>
- epoch
- batch_size

<모니터링>
- accuracy
- (train) loss
- val_loss

# 딥러닝 학습 프로세스

입력값이 네트워크 층을 거치면 예측값이 나오고, 이를 실제값과 비교해서 Loss score를 계산한 후에 Optimizer를 통해 Weight를 업데이트 한다.



# 폐암 환자의 생존율 예측하기

폐암 환자 470명 데이터 살펴보기(2013년 폴란드 브로츠와프 의과대학)

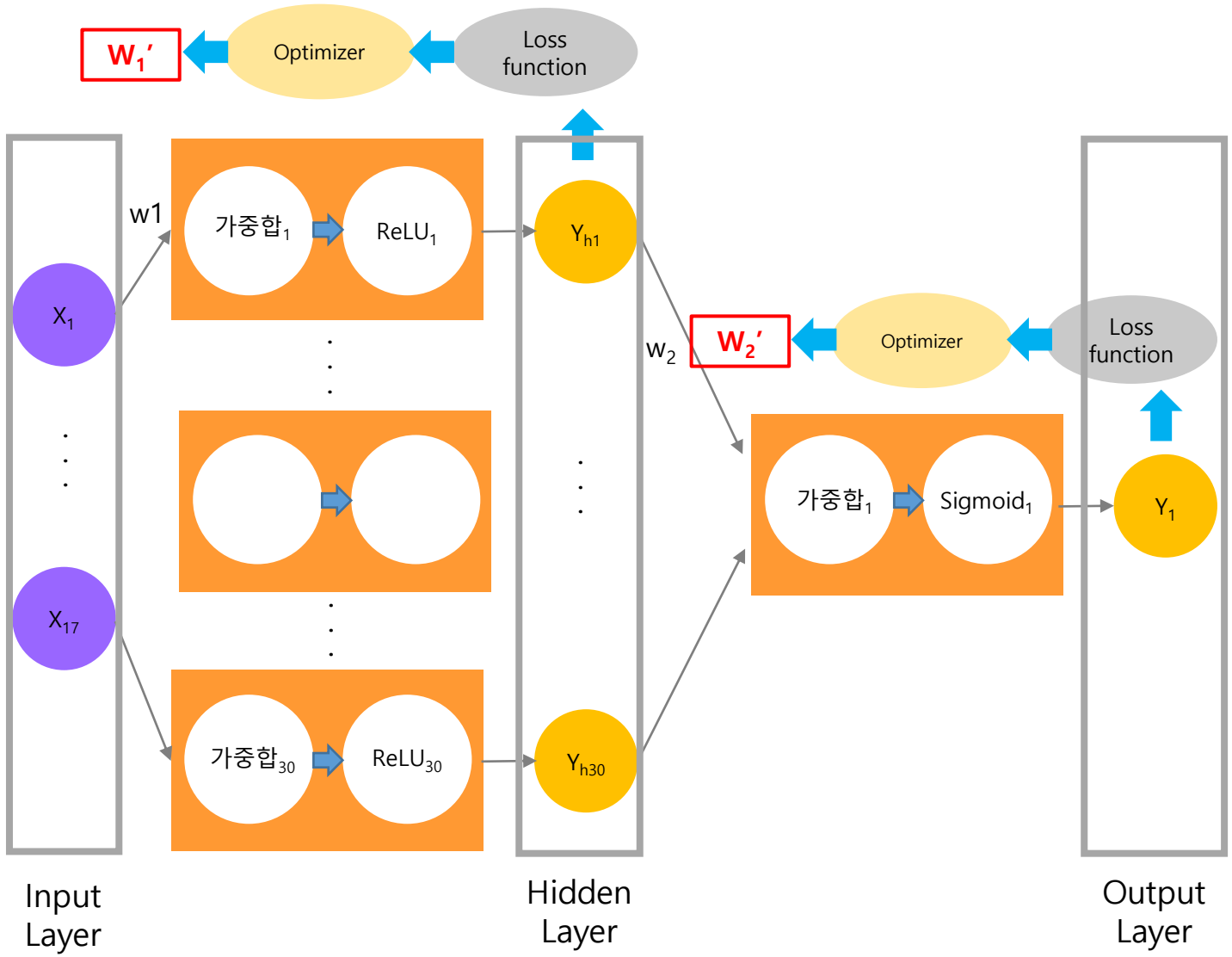
**Feature(수술 전 진단데이터)**
(종양 유형, 폐활량, 호흡곤란 여부, 고통 정도, 기침, 흡연, 천식 여부 등 17가지)

**Class(생존 결과)**
(생존:1, 사망:0)

| 줄 / 항목 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 환자1 | 1 | 293 | 1 | 3.8 | 2.8 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 1 | 0 | 62 | 0 |
| 환자2 | 2 | 1 | 2 | 2.88 | 2.16 | 1 | 0 | 0 | 0 | 1 | 1 | 14 | 0 | 0 | 0 | 1 | 0 | 60 | 0 |
| 환자3 | 3 | 8 | 2 | 3.19 | 2.5 | 1 | 0 | 0 | 0 | 1 | 0 | 11 | 0 | 0 | 1 | 1 | 0 | 66 | 1 |
| … | … | … | … | … | … | … | … | … | … | … | … | … | … | … | … | … | … | … | … |
| 환자470 | 470 | 447 | 8 | 5.2 | 4.1 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 49 | 0 |

feature: X = Data_set[ :, 0:17]
Class: Y = Data_set[ :, 17]

# 폐암 환자의 생존율 예측하기



# 폐암환자 생존예측 딥러닝 코드

```
# 딥러닝에 필요한 라이브러리 임포트
from keras.models import Sequential
from keras.layers import Dense

(중략)

# 딥러닝 구조 결정
model = Sequential()
model.add(Dense(30, input_dim=17, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# 딥러닝 실행
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
model.fit(X, Y, epochs=30, batch_size=10)

print(model.evaluate(X, Y)[1])
```

Sequential 함수 : 딥러닝 모델 구조를 한 층 한 층 쉽게 쌓을 수 있게 해줌
(전체 구조를 담는 그릇)
Dense 함수 : 각 층이 가질 특성을 각각 다르게 지정할 수 있게 해줌

model.add 함수 : 층 추가를 쉽게 해줌 -> 케라스의 가장 큰 장점
인자 : node수, (Input 층에는)input_dim, 활성화함수

model.compile 함수 : 딥러닝을 실행시켜 줌
인자 : Loss fn, optimizer, metrics

model.fit 함수 : 딥러닝을 학습 시킴
인자 : epochs, batch_size

model.evaluate 함수 : 딥러닝 예측 Accuracy 확인

# 폐암환자 생존예측 딥러닝 코드

In [1]:
```
# 딥러닝으로 폐암 수술 환자의 생존을 예측하기
from keras.models import Sequential
from keras.layers import Dense
import numpy
import tensorflow as tf
```

```
Using TensorFlow backend.
```

In [2]:
```
# 실행할 때마다 같은 결과를 출력하기 위해 설정하는 부분
seed = 0
numpy.random.seed(seed)
tf.random.set_seed(seed)
```

In [3]:
```
## ★단계1. 데이터 셋 생성(train, validation, test set 나누기) # 470명 환자의 기록과 수술 결과를 X와 Y로 구분하여 저장
Data_set = numpy.loadtxt("./dataset/ThoracicSurgery.csv", delimiter=",")
X = Data_set[:,0:17] # 수술기록
Y = Data_set[:,17]   # 수술결과
```

In [4]:
```
## ★단계2. 딥러닝 구조&모델 구성(sequential, layer, activation function)
model = Sequential()
model.add(Dense(30, input_dim=17, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

In [5]:
```
## ★단계3. 학습과정 설정(loss function, optimizer 지정)
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
```

In [6]:
```
## ★단계4. 학습시키기(train set를 fit()하기)
model.fit(X, Y, epochs=30, batch_size=10)
```

```
Epoch 1/30
470/470 [==============================] - 0s 195us/step - loss: 0.6655 - accuracy: 0.3085
Epoch 2/30
470/470 [==============================] - 0s 79us/step - loss: 0.1488 - accuracy: 0.8511
Epoch 3/30
470/470 [==============================] - 0s 81us/step - loss: 0.1488 - accuracy: 0.8511
Epoch 4/30
470/470 [==============================] - 0s 79us/step - loss: 0.1488 - accuracy: 0.8511
Epoch 5/30
470/470 [==============================] - 0s 79us/step - loss: 0.1488 - accuracy: 0.8511
Epoch 6/30
470/470 [==============================] - 0s 79us/step - loss: 0.1487 - accuracy: 0.8511
Epoch 7/30
470/470 [==============================] - 0s 76us/step - loss: 0.1487 - accuracy: 0.8511
Epoch 8/30
470/470 [==============================] - 0s 74us/step - loss: 0.1487 - accuracy: 0.8511
Epoch 9/30
470/470 [==============================] - 0s 70us/step - loss: 0.1487 - accuracy: 0.8511
Epoch 10/30
470/470 [==============================] - 0s 70us/step - loss: 0.1486 - accuracy: 0.8511
Epoch 11/30
470/470 [==============================] - 0s 68us/step - loss: 0.1499 - accuracy: 0.8447
Epoch 12/30
470/470 [==============================] - 0s 66us/step - loss: 0.1487 - accuracy: 0.8511
Epoch 13/30
470/470 [==============================] - 0s 68us/step - loss: 0.1485 - accuracy: 0.8511
Epoch 14/30
470/470 [==============================] - 0s 68us/step - loss: 0.1483 - accuracy: 0.8511
Epoch 15/30
470/470 [==============================] - 0s 68us/step - loss: 0.1485 - accuracy: 0.8511
Epoch 16/30
470/470 [==============================] - 0s 70us/step - loss: 0.1491 - accuracy: 0.8447
Epoch 17/30
Epoch 18/30
470/470 [==============================] - 0s 68us/step - loss: 0.1482 - accuracy: 0.8468
Epoch 19/30
470/470 [==============================] - 0s 70us/step - loss: 0.1477 - accuracy: 0.8511
Epoch 20/30
470/470 [==============================] - 0s 68us/step - loss: 0.1480 - accuracy: 0.8511
Epoch 21/30
470/470 [==============================] - 0s 74us/step - loss: 0.1475 - accuracy: 0.8511
Epoch 22/30
470/470 [==============================] - 0s 93us/step - loss: 0.1469 - accuracy: 0.8489
Epoch 23/30
470/470 [==============================] - 0s 76us/step - loss: 0.1467 - accuracy: 0.8511
Epoch 24/30
470/470 [==============================] - 0s 79us/step - loss: 0.1476 - accuracy: 0.8489
Epoch 25/30
470/470 [==============================] - 0s 59us/step - loss: 0.1471 - accuracy: 0.8511
Epoch 26/30
470/470 [==============================] - 0s 62us/step - loss: 0.1466 - accuracy: 0.8511
Epoch 27/30
470/470 [==============================] - 0s 64us/step - loss: 0.1473 - accuracy: 0.8511
Epoch 28/30
470/470 [==============================] - 0s 70us/step - loss: 0.1471 - accuracy: 0.8489
Epoch 29/30
470/470 [==============================] - 0s 64us/step - loss: 0.1471 - accuracy: 0.8489
Epoch 30/30
470/470 [==============================] - 0s 79us/step - loss: 0.1462 - accuracy: 0.8532
```

Out[6]:
```
<keras.callbacks.callbacks.History at 0x2ab8d456b00>
```

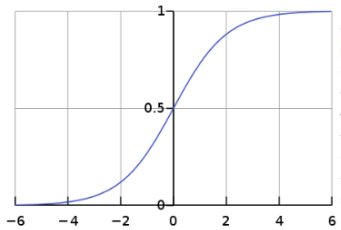
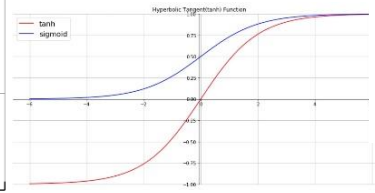
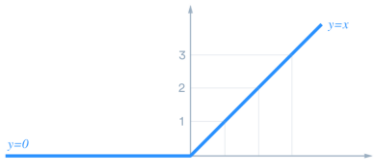
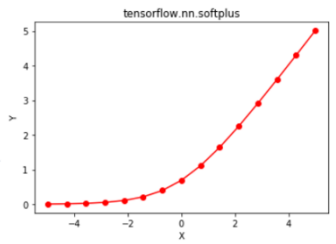
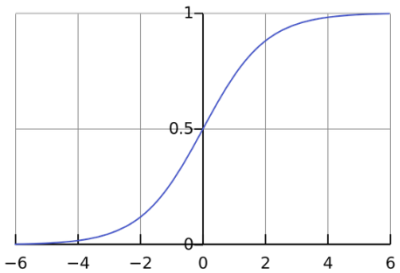
In [7]:
```
# 결과 출력
print("₩n Accuracy: %.4f" % (model.evaluate(X, Y)[1]))
```

```
470/470 [==============================] - 0s 38us/step

 Accuracy: 0.8511
```

# Activation function 종류

Sigmoid에서 시작된 활성화 함수는 ReLU를 비롯해 다양한 종류가 있다.

| 활성화 함수 | sigmoid | hyperbolic tangent | relu | softplus | softmax |
|---|---|---|---|---|---|
| |  |  |  |  |  |
| | $S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$ | $tanh(x) = \frac{1-e^{-x}}{1+e^{-x}}$ | $f(x) = max(0, x)$ | $f(x) = ln(1+e^x)$ | $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ |
| 설명 | 0 또는 1의 값이 출력됨 | 위 아래로 늘리기 | 0보다 작을 땐 0<br>0보다 클 땐 X | 0을 만드는 기준을 완화시키기 | 0과 1 사이 값이 여러 개 출력됨 |
| 출력값 | 0 or 1 | -1 ~ 1 | O or X | O or X | 0 ~ 1 |
| 사용처 | 이진 분류 출력층 | | Vanishing 해결<br>은닉층에 사용. | | 다중 분류 출력층 |

# Loss function(오차 함수) 종류

| 계열 | 오차함수 종류 | 의미 | 사용처 | 특징 |
|---|---|---|---|---|
| 평균제곱 계열<br>(Mean squared) | MSE<br>(Mean squared error) | 평균제곱 오차 | 회귀 문제 | 속도가 느리다는 단점 |
| | MAE | 평균절대 오차 | | |
| | MAPE | 평균절대백분율 오차 | | |
| | MSLE | 평균제곱로그 오차 | | |
| 교차엔트로피 계열<br>(Cross-entropy) | categorical_<br>crossentropy | 범주형<br>교차 엔트로피 | 다중 클래스 분류 | 장점 :<br>출력 값에 로그를 취해서,<br>오차가 커지면 수렴 속도 증가,<br>오차가 작아지면 수렴 속도 감소 |
| | binary_<br>crossentropy | 이항<br>교차 엔트로피 | 이진 클래스 분류 | |

# Optimizer(경사하강법) 종류

lr : learning rate(학습률)

| Optimizer 종류 | 개요 | 장점 | 케라스 사용법 |
|---|---|---|---|
| 1. SGD<br>(확률적 경사 하강법) | 랜덤하게 추출한 데이터를 활용해<br>더 빨리, 더 자주 업데이트 | 속도 | keras.optimizers.SGD(lr=0.1) |
| 2. Momentum | 관성의 방향을 고려해 진동과 폭을 줄이는 효과 | 정확도 | keras.optimizers.SGD(lr=0.1,<br>momentum=0.9) |
| 3. NAG<br>(네스테로프 모멘텀) | 모멘텀이 이동시킬 방향으로 미리 이동해서<br>그래디언트 계산 -> 불필요한 이동 줄임 | 정확도 | keras.optimizers.SGD(lr=0.1,<br>momentum=0.9, nesterov=True) |
| 4. Adagrad | 변수의 업데이트가 잦으면 학습률을 적게하여,<br>이동 보폭을 조절하는 방법 | 보폭 크기 | keras.optimizers.Adagrad(lr=0.01,<br>epsilon=1e-6) |
| 5. RMSProp<br>(알엠에스프롭) | 아다그라드의 보폭 민감도를 보완 | 보폭 크기 | keras.optimizers.RMSprop(lr=0.001,<br>rho=0.9, epsilon=1e-08, decay=0.0) |
| 6. Adam | 모멘텀+RMSProp<br>(최근 것. 이 중 가장 좋다) | 보폭 크기,<br>정확도 | keras.optimizers.Adam(lr=0.001,<br>beta_1=0.9, beta_2=0.999,<br>epsilon=1e-08, decay=0.0) |