Deep Learning

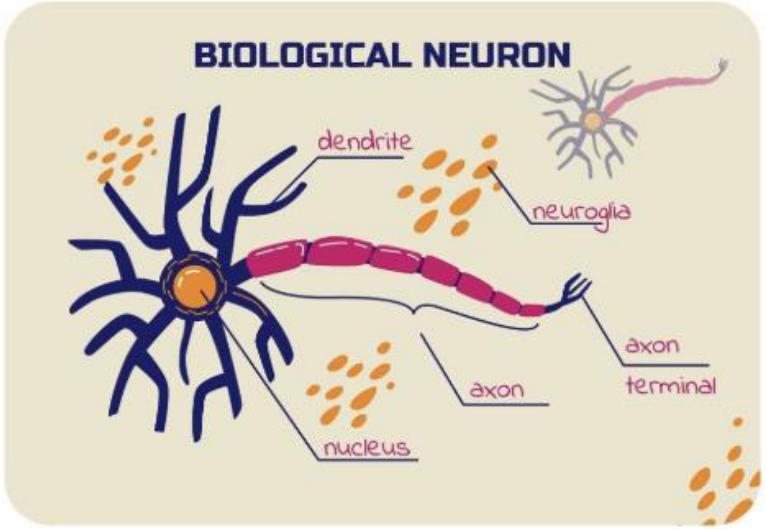
2025. 8

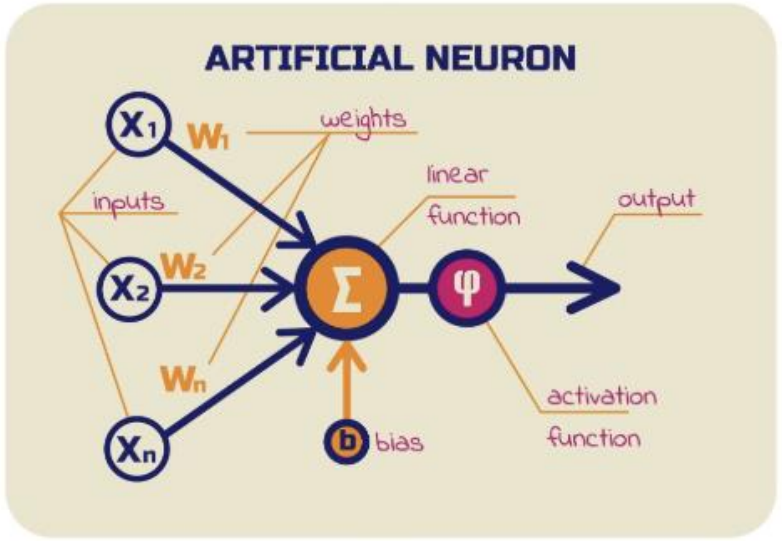
Yongjin Jeong, KwangWoon University

Introduction to Deep Learning
(딥러닝 소개)

딥러닝의 영감, 뇌 신경망

- **뇌 신경망 (Neural Negwork)**
  - 딥러닝(deep Learning)은 인간의 뇌를 구성하는 뉴런 (Neuron)의 연결 구조에서 영감을 얻은 인공지능 기술의 한 분야.
  - 생물학적 뉴런은 다른 뉴런으로부터 수상돌기(dendrite)를 통해 신호를 받아 세포체(soma)에서 처리한 후, 축삭(axon)을 통해 다음 뉴런으로 신호를 전달한다.
  - 이러한 구조를 수학적으로 모방하여 입력(inputs)에 가중치(weights)를 곱하고, 그 합을 활성화 함수(activation function)에 통과시켜 출력(output)을 내는 **인공 뉴런 (Artificial Neuron)** 모델을 사용.



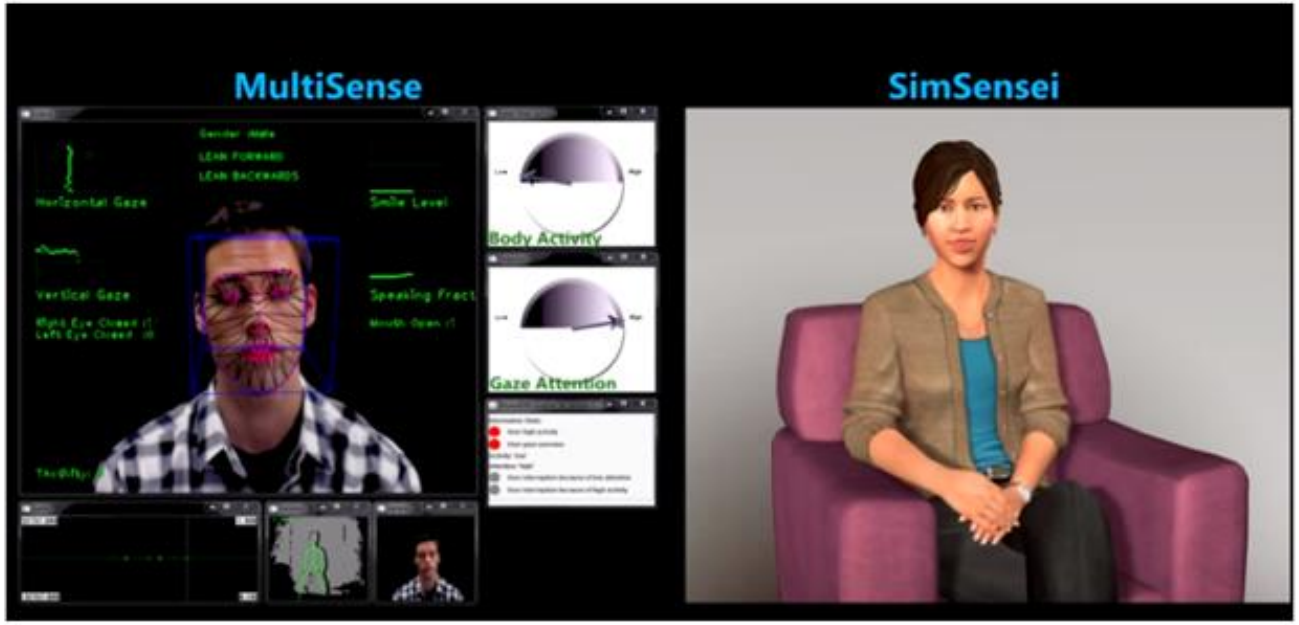


적용 사례: 알파고 (AlphaGo)

- **적용 사례 1 - 알파고 (AlphaGo)**
  - 2016년, 구글 딥마인드(Google DeepMind)가 개발한 딥러닝 기반 바둑 프로그램 **알파고**가 이세돌 9단과의 대국에서 4승 1패로 승리하며 전 세계에 충격을 주었음.
  - 알파고는 기계학습과 병렬처리 기술을 활용했으며, 수십 개의 TPU(Tensor Processing Unit)를 사용한 분산 컴퓨팅 시스템으로 구현되었다. 이후 단 4개의 TPU를 사용하는 AlphaGo Zero로 발전하며 더 뛰어난 성능을 보였다.

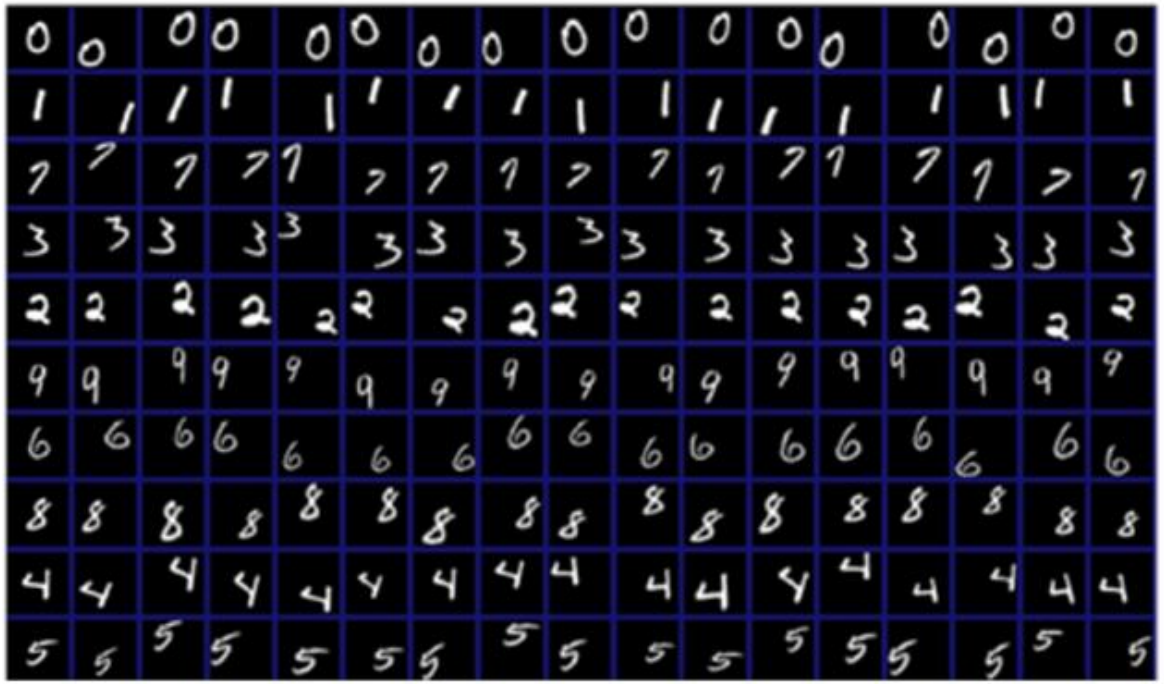
적용 사례: SimSensei (감성 컴퓨팅)

- **적용 사례 2 - SimSensei (감성 컴퓨팅)**
  - 미국 USC 에서 개발한 **SimSensei**는 딥러닝의 안면인식 및 동작인식 기술을 활용해 사람의 우울증 징후를 감지.
  - 가상 아바타가 사용자와 대화하는 동안, 소프트웨어는 사용자의 얼굴 근육, 음성 패턴, 자세 등 66개의 특징을 분석하여 기존의 설문지 방식보다 더 정확하게 심리 상태를 파악.



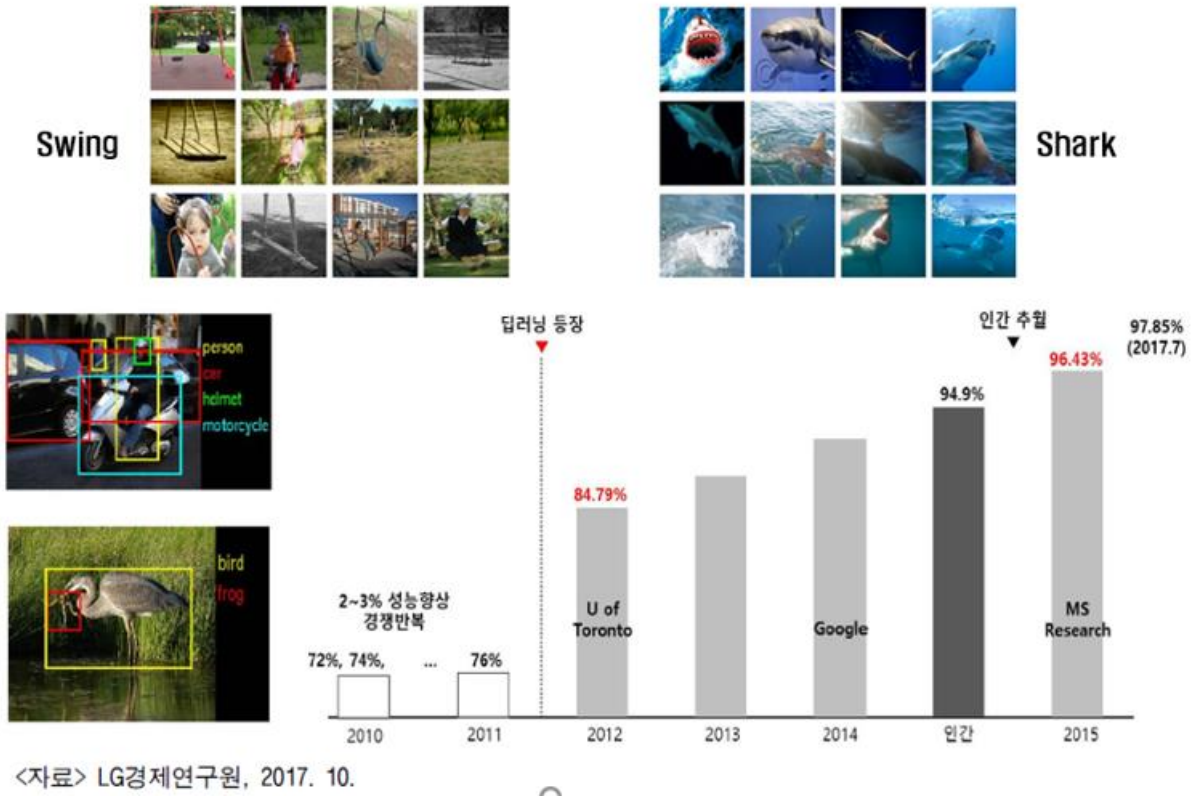
적용 사례: MNIST (필기체 숫자 인식)

- **적용 사례 3 - MNIST (필기체 숫자 인식)**
  - **MNIST**는 손으로 쓴 숫자 이미지 데이터 집합으로, 딥러닝 모델의 성능을 측정하는 대표적인 기준(benchmark)으로 사용됨.
  - 2012년, 딥러닝 기술을 통해 필기체 숫자 인식 오류율이 0.23%까지 떨어지면서, 복잡한 패턴 인식 분야에서 딥러닝의 뛰어난 성능이 입증.



적용 사례: ImageNet Challenge (이미지 인식)

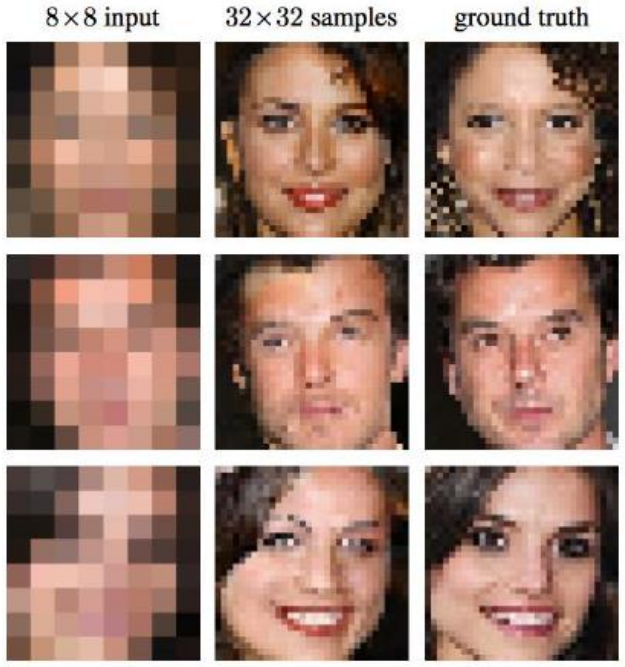
- **적용 사례 4 - ImageNet Challenge (이미지 인식)**
  - ImageNet 경진대회(ILSVRC)는 1000개의 카테고리로 이미지를 분류하는 세계 최대 규모의 이미지 인식 대회.
  - 2012년, 딥러닝 모델이 처음 등장하여 오류율을 획기적으로 낮추면서 이미지 인식 분야의 판도를 바꾸게 됨.
  - 이후 딥러닝 기술은 발전을 거듭하여 2015년에는 인간의 인식 능력을 뛰어넘는 정확도를 달성.

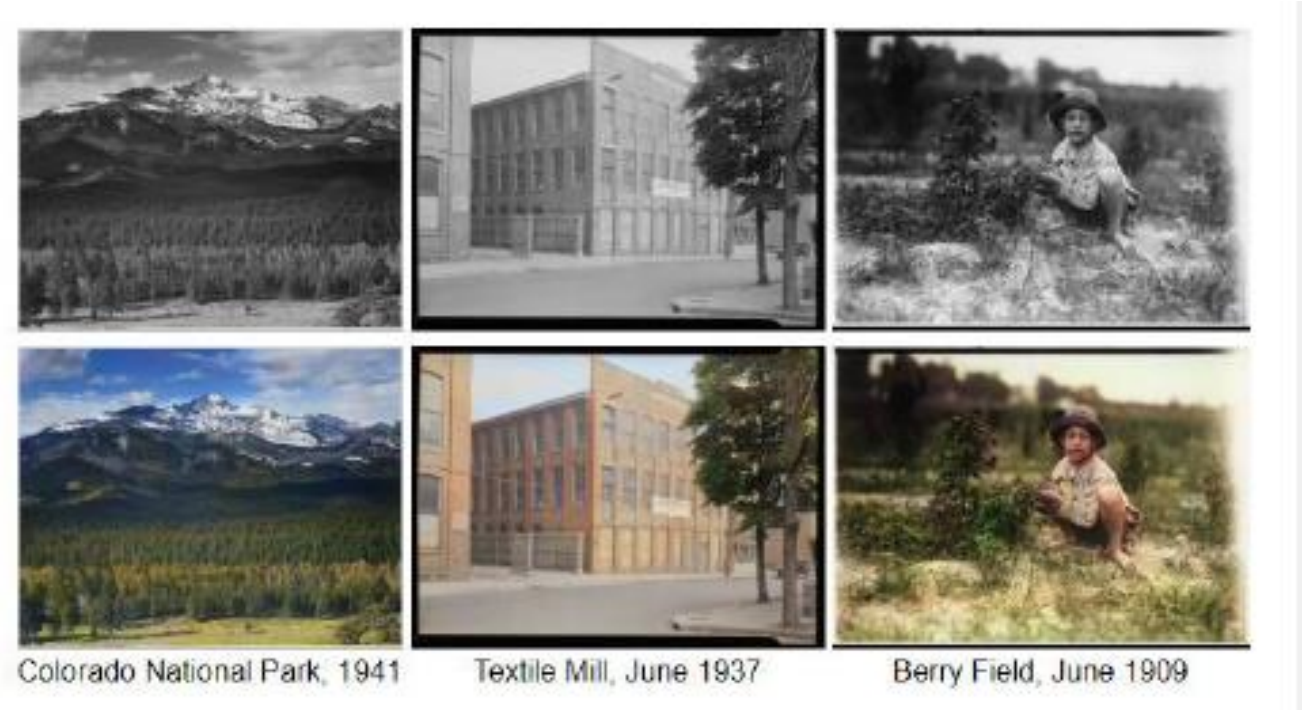


<자료> LG경제연구원, 2017. 10.

적용 사례: 이미지 복원 및 생성

- **적용 사례 5 - 이미지 복원 및 생성**
  - 손상되거나 정보가 부족한 이미지를 복원하는 데에도 탁월한 능력.
  - 이미지 화소 복원(Image Pixel Recovery): 저해상도(8×8)의 얼굴 이미지를 고해상도(32×32) 복원.
  - 이미지 색상 복원(Image Color Recovery): 흑백 사진을 보고 원래의 색을 추정하여 자동 채색.





적용 사례: 이미지 캡셔닝 (Image Captioning)

- **적용 사례 6 - 이미지 캡셔닝 (Image Captioning)**
  - 이미지 캡셔닝: 이미지의 내용을 이해하고, 그 내용을 자연어 문장으로 설명하는 기술.
  - "a man throwing a frisbee in a green field." (녹색 들판에서 프리스비를 던지는 남자) 와 같이 사진 속 객체와 상황을 정확히 묘사하는 문장을 자동으로 생성.

a man wearing a blue shirt with his arms on the grass.
a man holding a frisbee bat in front of a green field.
a man throwing a frisbee in a green field.
a boy playing ball with a disc in a field.
a young man playing in the grass with a green ball.

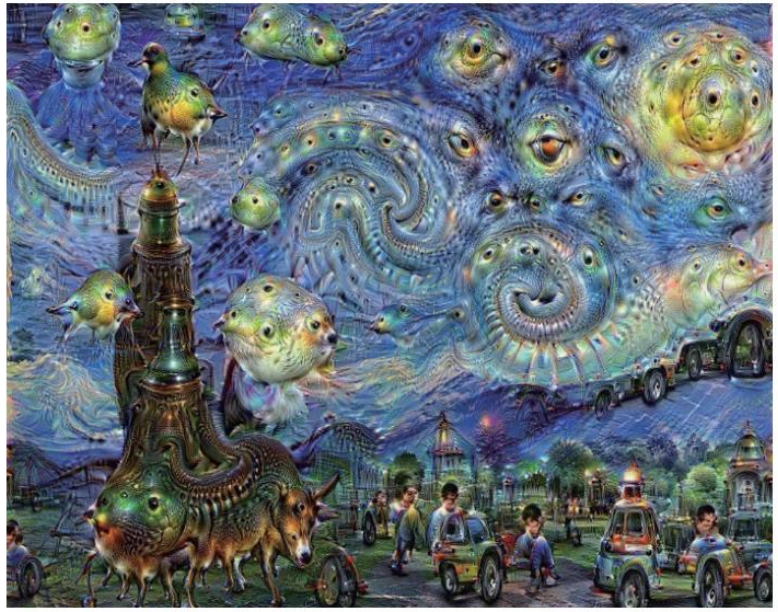
a red car on the side of the road in the small race.
a truck driving uphill on the side of the road.
a person driving a truck on the road.
a small car driving down a dirt and water.
a truck in a field of car is pulled up to the back.

a group of birds standing next to each other.
a group of ducks that are standing in a row.
a group of ducks that are standing on each other.
a group of sheep next to each other on sand.
a group of small birds is standing in the grass.

a kite flying over the ocean on a sunny day.
a person flying over the ocean on a sunny day.
a person flying over the ocean on a cloudy day.
a kite on the beach on the water in the sky.
a large flying over the water and rocks.

적용 사례: 딥드림 (Deep Dream)과 예술 창작

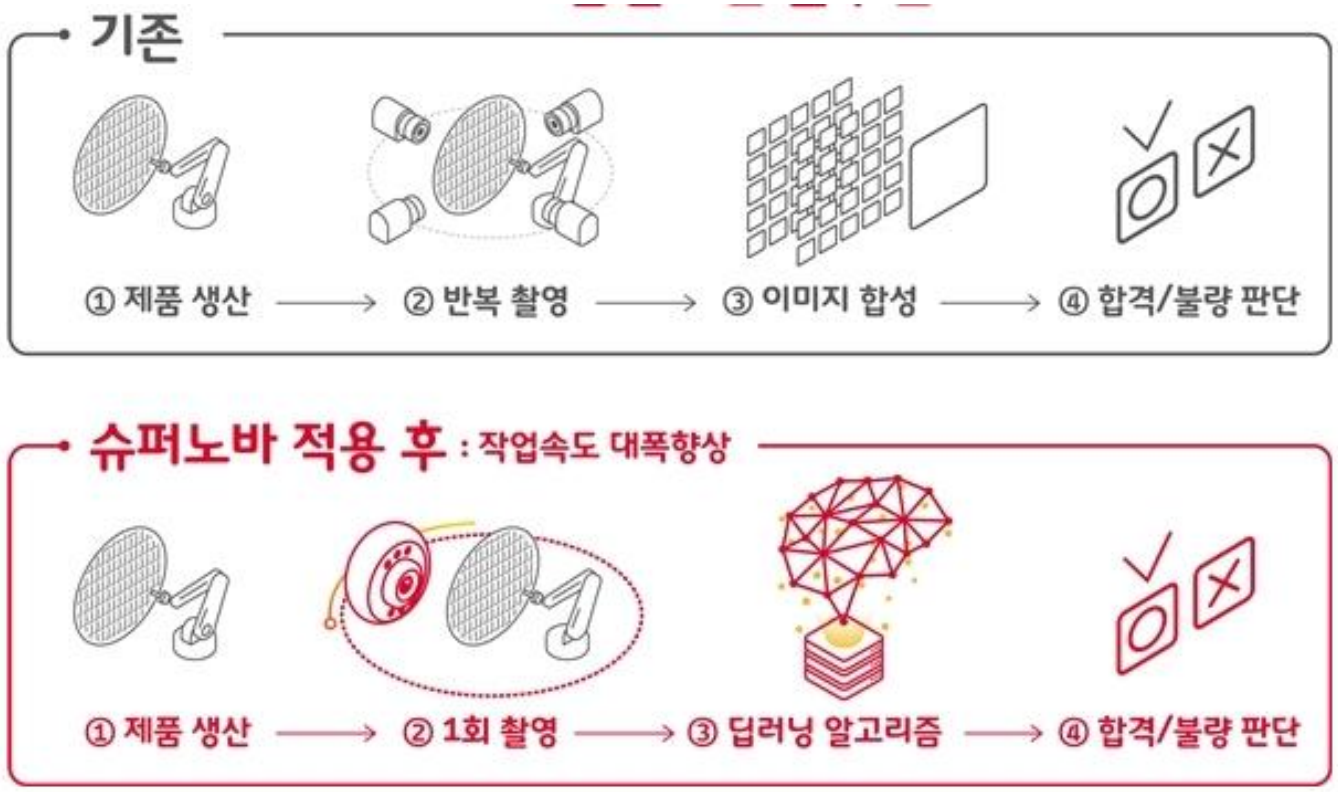
- **적용 사례 7 - 딥드림 (Deep Dream)과 예술 창작**
  - 구글의 딥드림(Deep Dream): 딥러닝 모델이 이미지에서 특정 패턴을 반복적으로 찾아 강화하면서 초현실적이고 몽환적인 이미지를 만들어내는 기술.
  - 딥러닝으로 생성된 예술 작품이 경매에서 높은 가격에 판매되는 등, 딥러닝은 이제 창작의 영역까지 확장.



구글 딥드림의 최고가 8,000달러에 낙찰된 작품 @월스트리트저널

적용 사례: 제조업 혁신 (SKT 슈퍼노바)

- **적용 사례 8 - 제조업 혁신 (SKT 슈퍼노바)**
  - SK텔레콤의 AI 품질개선 솔루션 '슈퍼노바(Supernova)'는 딥러닝을 반도체 제조 공정에 적용한 사례.
  - 기존에는 여러 번 반복 촬영하고 이미지를 합성하여 불량을 판별했지만, 슈퍼노바는 **단 1회 촬영** 후 딥러닝 알고리즘을 통해 합격/불량을 판단하여 작업 속도와 정확도를 대폭 향상.

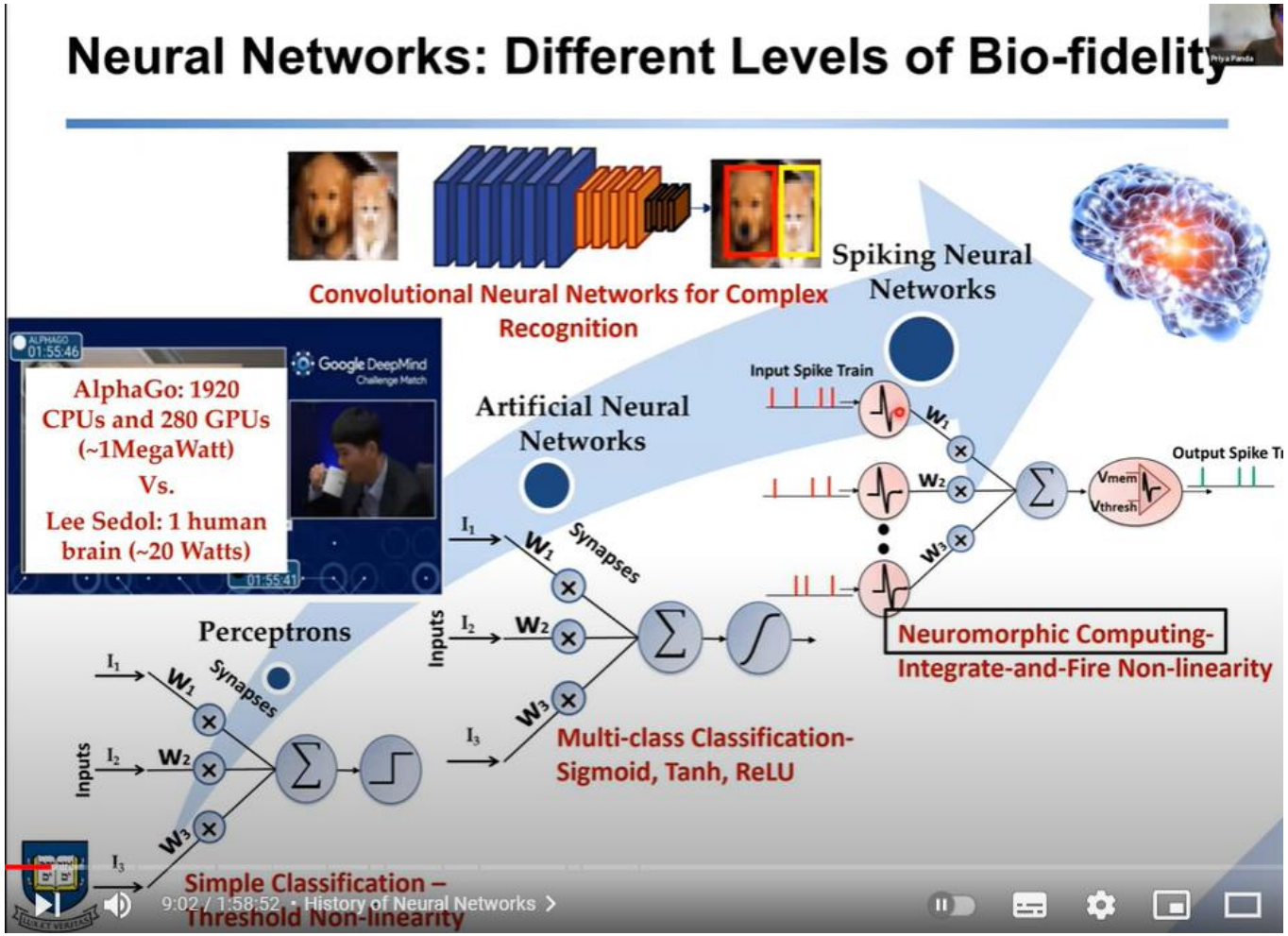


인포그래픽은 슈퍼노바 기술 개념도. SK텔레콤은 자사 전시관 전시관 5G 커넥티드 팩토리 부스에 '슈퍼노바'를 활용한 반도체 제조공정 혁신 모델을 전시한다. <SK텔레콤>

Artificial Neural Network
(인공신경망)

인공 신경망

- **신경망의 종류와 발전**
  - 가장 단순한 형태인 퍼셉트론(Perceptron)에서 시작하여, 여러 층을 쌓은 심층 신경망(Deep Neural Network), 이미지 처리에 특화된 합성곱 신경망(CNN), 그리고 인간의 뇌를 더 유사하게 모방한 스파이킹 신경망(Spiking Neural Network) 등으로 발전.
  - 신경망은 점차 복잡하고 생물학적으로 유사한 형태로 진화 중.



https://www.youtube.com/watch?v=7TybETlCslM (Professor Priya Panda at Yale University)

Perceptron (퍼셉트론)

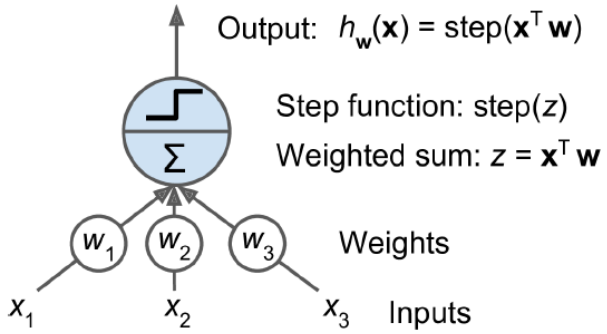
- **퍼셉트론 (Perceptron)**
  - 1957년에 고안된 가장 단순한 형태의 인공 신경망으로, 다수의 입력을 받아 하나의 결과를 출력.
  - 각 입력 신호(x1,x2,...)에 고유한 가중치(w1,w2,...)가 곱해진다. 이 신호들의 가중합(Weighted Sum)이 특정 임계값(Threshold)을 넘으면 1을 출력하고, 넘지 못하면 0(또는 -1)을 출력하는 구조.
  - 이는 뉴런이 특정 세기 이상의 자극을 받아야만 활성화되는 모습을 모방한 것임.

Perceptron

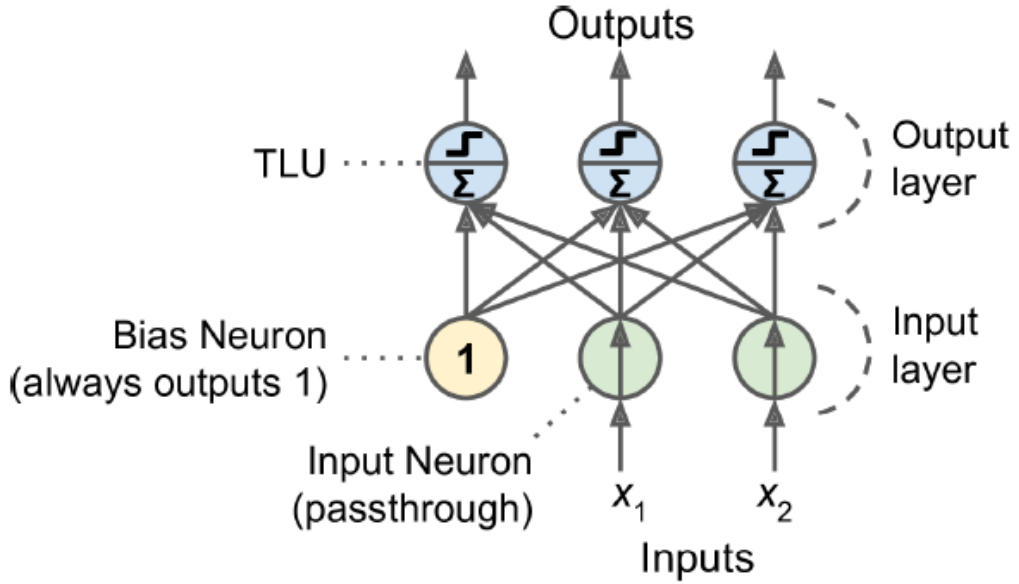
- **Perceptron**
  - A simplest ANN architecture based on threshold logic unit (TLU) or linear threshold unit (LTU) (1957)
  - A single layer of TLUs (fully connected or a dense layer)

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

Threshold logic unit

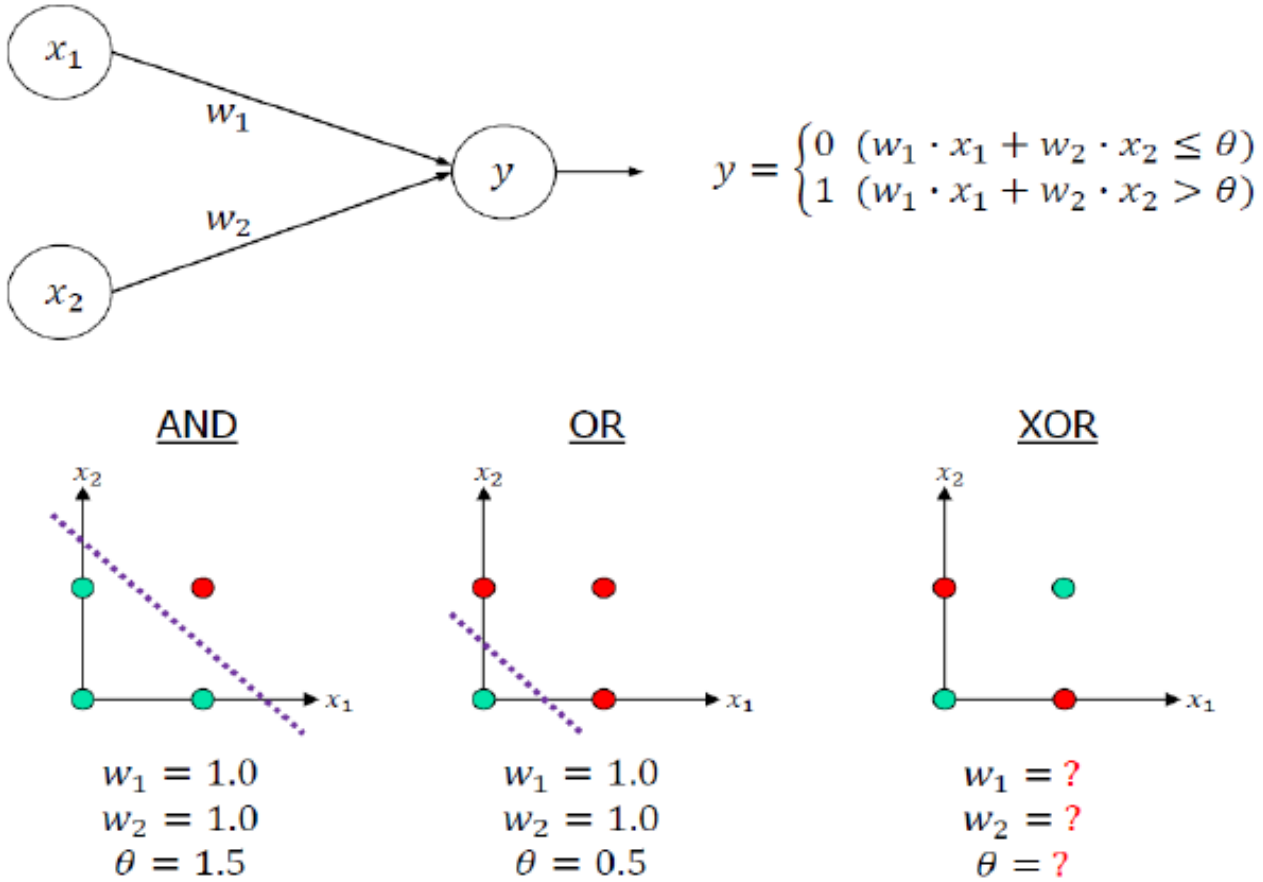


Perceptron diagram



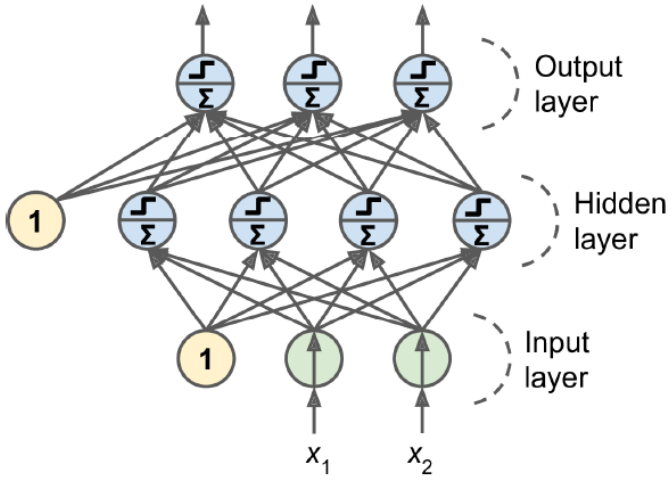
퍼셉트론의 한계: XOR 문제

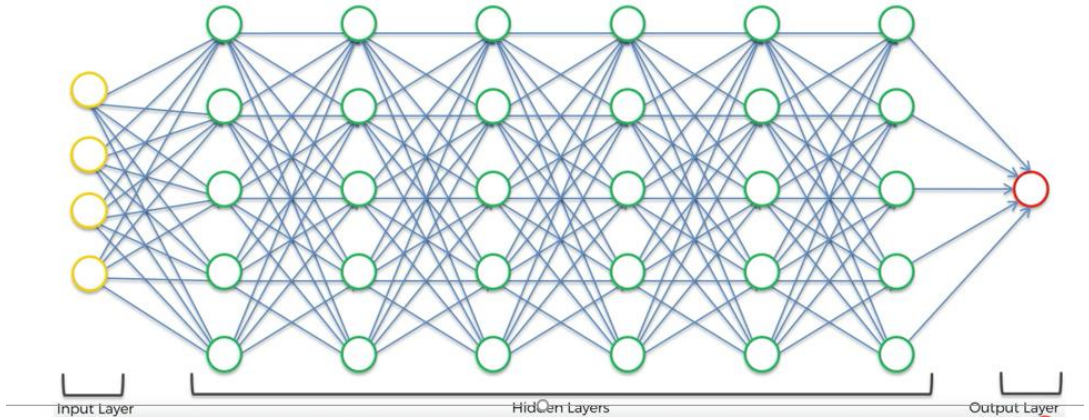
- 퍼셉트론은 선형(linear) 경계선으로 데이터를 분류하는 모델이다. 따라서, AND, OR 게이트처럼 선 하나로 두 그룹을 나눌 수 있는 문제는 해결할 수 있다.
- 하지만, XOR 문제와 같이 직선 하나로는 분류가 불가능한, 즉 비선형(non-linear) 문제에 대해서는 해결할 수 없는 명확한 한계를 가짐. (인공 신경망 연구 한동안 침체기 겪은 원인)
- But, later proved that Multilayer Perceptron (MLP) can solve the problem.



다층 퍼셉트론(MLP)과 XOR 문제 해결

- **다층 퍼셉트론(MLP)과 XOR 문제 해결**
  - XOR 문제는 퍼셉트론을 여러 개 쌓은 다층 퍼셉트론(Multi-Layer Perceptron, MLP)로 해결 가능.
  - MLP는 입력층(Input Layer)과 출력층(Output Layer) 사이에 하나 이상의 은닉층(Hidden Layer)을 추가한 구조.
  - 은닉층을 통해 입력 데이터를 비선형적인 공간으로 변환함으로써, 단층 퍼셉트론이 풀지 못했던 XOR와 같은 비선형 문제를 해결할 수 있게 됨.

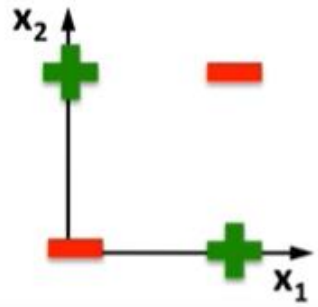


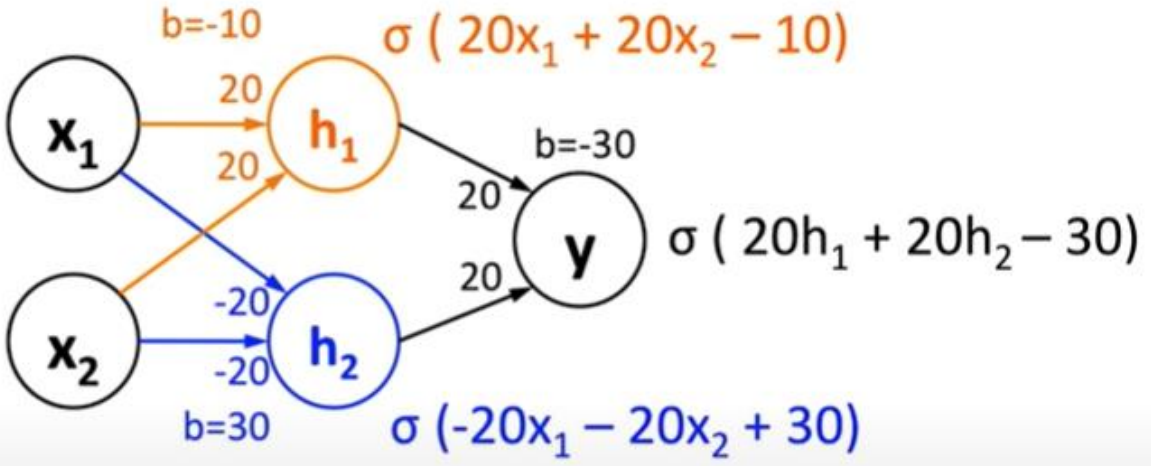


MLP and XOR Problem

- **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)

Activation Functions (활성화 함수)

- **활성화 함수 (Activation Function)**
  - 뉴런의 가중합 결과를 출력 신호로 변환하는 함수
  - 초기 퍼셉트론은 계단 함수(Step Function)를 사용했는데, 이는 미분이 불가능하여 가중치를 점진적으로 학습시키기 어려웠음.
  - 이 문제를 해결하기 위해, 미분 가능한 다양한 비선형 함수들이 활성화 함수로 제안됨.

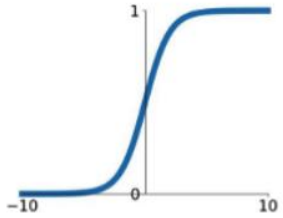
- **대표적인 활성화 함수**
  - 시그모이드(Sigmoid): 출력을 0과 1 사이로 압축한다. 주로 이진 분류 문제의 출력층에서 확률을 나타낼 때 사용.
  - 하이퍼볼릭 탄젠트(Tanh): 출력을 -1과 1 사이로 압축한다. 시그모이드보다 중심값이 0이라 더 효율적인 학습이 가능할 때가 많다.
  - ReLU (Rectified Linear Unit): 입력이 0보다 크면 그대로 출력하고, 0보다 작으면 0을 출력한다. 계산이 간단하고 학습 속도가 빨라 현재 은닉층에서 가장 널리 사용되는 활성화 함수.

Activation Functions

- **Changed Activation function**
  - step function is flat, so there is no gradient.
  - There are several activation functions.

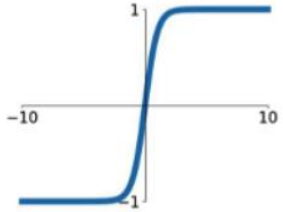
Sigmoid

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



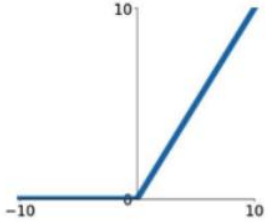
tanh

$$\tanh(x)$$



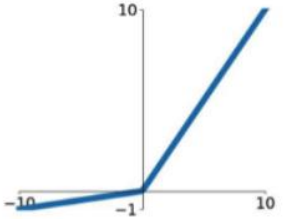
ReLU

$$\max(0, x)$$



Leaky ReLU

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

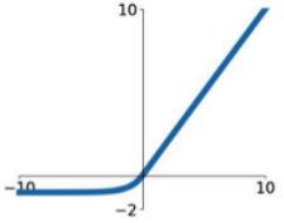


Maxout

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

ELU

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



활성화 함수와 Gradient(도함수)

- **활성화 함수 (Activation Function)**
  - 신경망이 가중치를 학습하는 과정(역전파)에서는 활성화 함수의 미분값(그래디언트) 사용.
  - 계단 함수는 대부분의 구간에서 미분값이 0이므로 학습이 진행되지 않는다.
  - 시그모이드(Sigmoid)와 Tanh 함수 역시 입력값의 절댓값이 커지면 그래프의 기울기가 0에 가까워지는 그래디언트 소실 문제(Vanishing Gradient Problem)가 발생한다. 이 때문에 은닉층이 깊어질수록 앞쪽 레이어까지 그래디언트가 잘 전달되지 않아 학습 효율이 떨어질 수 있다.
  - 반면, ReLU 함수는 입력값이 양수일 때 미분값이 항상 1이므로, 그래디언트 소실 문제에서 비교적 자유로워 깊은 신경망의 은닉층에서 훨씬 효율적인 학습이 가능하다.
    - Relu 의 음수 영역: 뉴런의 활성화를 억제하는 반대 방향의 자극으로 이해할 수 있음. (실제 뇌의 뉴런은 흥분성(Excitatory) 신호와 억제성(Inhibitory) 신호 두 종류의 신호를 받는다고 함. 가중합 (XW+b) 이 양수는 흥분성, 음수는 억제성으로 해석)
    - 실제 뉴런도 마이너스 강도로 신호를 보내지 않고, 뉴런은 **발화하거나(fire) 발화하지 않거나(not fire)** 둘 중 하나의 행동을 한다. 여기서 '발화하지 않는 상태'가 바로 억제성 신호가 흥분성 신호보다 강했거나, 흥분성 신호가 활성화 임계값을 넘기에 충분히 강하지 않았다는 '결과'를 의미한다.

Activation Functions and Derivatives

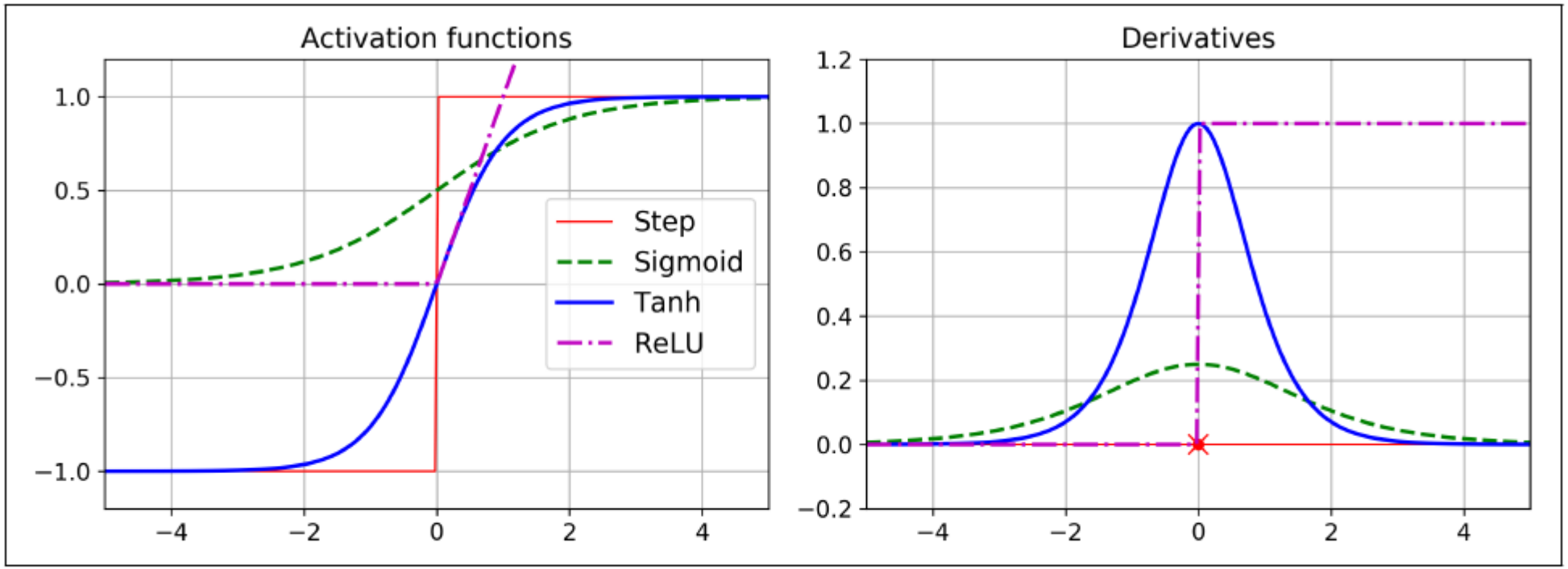


Figure 10-8. Activation functions and their derivatives

MLP 응용: Regression

- **MLP 응용 1 - 회귀 (Regression)**
  - 회귀는 주가, 온도, 점수 등 **연속적인 값**을 예측하는 문제
  - 입력층: 예측에 사용할 특징(feature)의 수만큼 뉴런을 둔다.
  - 은닉층: 보통 1~5개의 층을 사용하며, 각 층의 뉴런은 10~100개 사이로 설정한다. 활성화 함수로는 주로 ReLU를 사용한다.
  - 출력층: 예측하려는 값의 차원 수만큼 뉴런을 둔다 (보통 1개). 예측값이 특정 범위에 있을 필요가 없다면 활성화 함수를 사용하지 않는다.
  - 손실 함수: 예측값과 실제값의 차이를 측정하기 위해 주로 평균 제곱 오차(MSE)를 사용한다.

MLP for regression

Table 10-1. Typical Regression MLP Architecture

Hyperparameter	Typical Value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

MLP 응용: Classification

- **MLP 응용 2 - 분류 (Classification)**
  - 분류는 스팸 메일 여부, 숫자 종류 등 주어진 데이터를 정해진 카테고리로 나누는 문제.
  - 입력층과 은닉층의 구성은 회귀 문제와 유사하다.
  - 출력층의 활성화 함수와 손실 함수가 문제 유형에 따라 달라진다.
    - 이진 분류(Binary Classification): 두 개 중 하나를 선택하는 문제. 출력 뉴런 1개와 시그모이드(로지스틱) 함수를 사용.
    - 다중 클래스 분류(Multi-class Classification): 여러 개 중 단 하나를 선택하는 문제. 각 클래스마다 뉴런을 하나씩 두고, 출력 확률의 총합이 1이 되도록 하는 소프트맥스(Softmax) 함수를 사용.
  - 손실 함수: 모든 분류 문제에서 일반적으로 **교차 엔트로피(Cross-Entropy)**를 사용한다.

Multi-Class Classification and Softmax

- **Multi-Class Classification and Softmax**
  - 3개 이상의 클래스를 분류할 때, 출력층은 각 클래스에 대한 **점수(score)**를 계산한다.
  - **소프트맥스(Softmax)** 함수는 이 점수들을 입력받아, 모든 클래스에 대한 확률 값으로 변환해준다.
  - 변환된 값들은 0과 1 사이이며, 모든 값을 더하면 항상 1이 된다.
  - 모델은 이 확률 값들 중 가장 높은 값을 가진 클래스를 최종 예측 결과로 선택한다.

MLP for classification

MLP (including ReLU and softmax) for classification

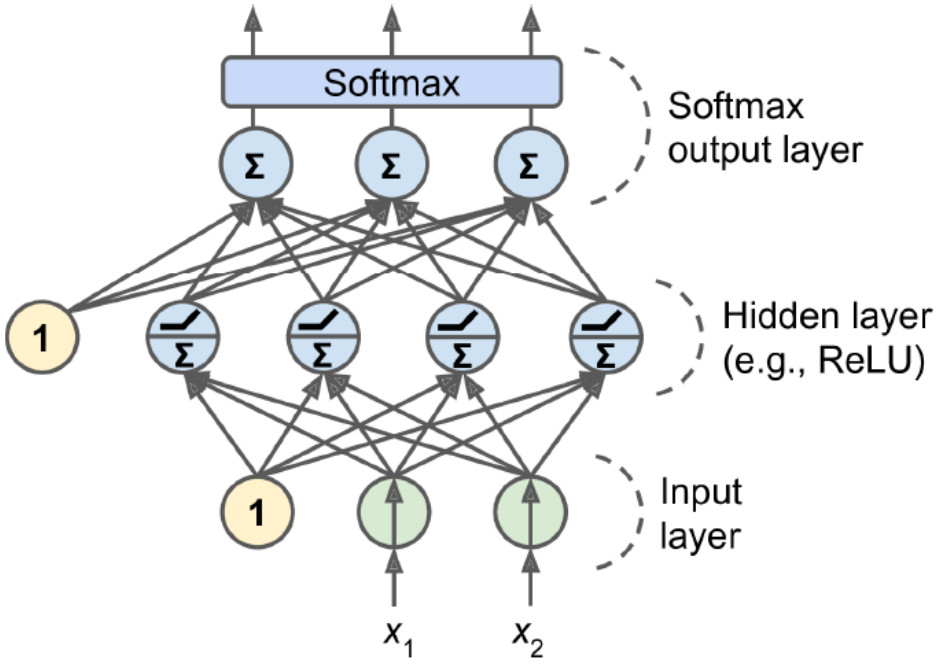


Table 10-2. Typical Classification MLP Architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross-Entropy	Cross-Entropy	Cross-Entropy

multi-class: single label multi-class, (e.g. [0] or [1] or [2])
multi-label: may have more than one class label
(e.g. [1,0,1], [1,1,0])

Deep Learning Training

Training(학습) and Loss Function(손실함수)

- **딥러닝 모델의 학습:**
  - 초기 신경망 모델은 아직 훈련되지 않은 초기 상태이다. 학습(Training)은 이 모델이 정답을 잘 맞힐 수 있도록 내부의 **가중치(Weight)**와 **편향(Bias)**을 점진적으로 업데이트하는 과정이다. 이 학습 과정은 세 가지 핵심 요소로 이루어진다.
    - **손실 함수 (Loss Function)**: 모델이 얼마나 틀렸는지 '오차'를 측정
    - **역전파 (Backpropagation)**: 이 오차에 각 가중치가 얼마나 기여했는지('책임')를 계산
    - **최적화 알고리즘 (Optimizer)**: 계산된 책임을 바탕으로 가중치를 업데이트

- **손실 함수 (Loss Function)**
  - 손실 함수는 모델의 예측 값과 실제 정답이 얼마나 다른지를 측정하는 '오답 노트'와 같다. 손실 함수의 결과값(Loss)이 크면 모델이 많이 틀렸다는 의미이고, 작으면 정답에 가깝게 예측했다는 의미이다.
  - 딥러닝 학습의 목표는 이 손실 함수의 값을 가능한 0에 가깝게 만드는 것이다. 이를 위해 모델은 손실 값을 줄이는 방향으로 자신의 가중치를 계속해서 수정해 나간다.

분류 문제의 손실 함수: 교차 엔트로피 (Cross-Entropy)

- **분류 문제의 손실 함수: 교차 엔트로피 (Cross-Entropy):**
  - 분류 문제에서는 주로 교차 엔트로피(Cross-Entropy) 손실 함수를 사용한다.
    - 소프트맥스(Softmax) 함수가 모델의 예측을 '확률 분포'로 만들어준다. (예: [고양이일 확률: 70%, 개일 확률: 20%, 토끼일 확률: 10%])
    - 정답 또한 확률 분포로 표현할 수 있다. (예: [고양이일 확률: 100%, 개일 확률: 0%, 토끼일 확률: 0%])
    - 교차 엔트로피는 모델이 예측한 확률 분포와 정답의 확률 분포가 얼마나 다른지를 측정하는 지표이다. 두 분포가 비슷할수록 값은 작아지고, 다를수록 값은 커진다.

Softmax and Cross-Entropy

- **소프트맥스 + 교차 엔트로피 조합:**
  - 다중 클래스 분류 모델에서 소프트맥스 활성화 함수와 교차 엔트로피 손실 함수는 거의 항상 함께 사용되는 표준적인 조합.
    - 입력 데이터가 신경망을 통과하여 각 클래스에 대한 **점수(Score)**를 출력한다.
    - 출력층의 소프트맥스 함수가 이 점수를 확률로 변환한다.
    - 교차 엔트로피 함수가 이 예측 확률과 실제 정답을 비교하여 최종 손실(Loss) 값을 계산한다.

Backpropagation and Optimizer

- **역전파 알고리즘 (Backpropagation):**
  - 역전파: 모델이 얼마나 틀렸는지를 손실함수로 표현하고, 이 오차의 원인을 거꾸로 추적하여 각 가중치가 오차에 얼마나 기여했는지, 즉 책임을 계산하는 과정.
  - 수학적으로는 미분의 연쇄 법칙(Chain Rule)을 사용하여, 손실 함수에 대한 모든 가중치의 그래디언트(Gradient)를 효율적으로 계산하는 알고리즘.
  - 계산된 그래디언트 값은 '손실을 줄이려면 이 가중치를 어느 방향으로 얼마나 조절해야 하는지'에 대한 정보를 담고 있다.

- **최적화 알고리즘 (Optimizer):**
  - 역전파를 통해 계산된 그래디언트 정보를 사용하여, 실제로 모델의 가중치를 업데이트하는 역할
  - 즉, 손실 함수라는 산의 가장 낮은 지점(최솟값)을 찾아가는 등산객의 발걸음과 같다.

Gradient Descent, GD (경사 하강법)

- **경사 하강법 (Gradient Descent, GD):**
  - 가장 기본적인 최적화 알고리즘.
  - 현재 위치에서 손실 함수 값(산의 높이)이 가장 가파르게 감소하는 방향, 즉 **그래디언트의 반대 방향**으로 한 걸음씩 이동하는 방식.
  - 이 과정을 반복하면 결국 손실이 가장 낮은 지점(Global Minimum 또는 Local Minimum)에 도달하게 된다.
  - **업데이트 공식:** $W_{new} = W_{old} - \eta \times \nabla J(W)$
    - (η: 학습률, $\nabla J(W)$: 손실 함수에 대한 가중치의 그래디언트)

Learning Rate, η (학습률)

- **학습률 (Learning Rate, η):**
  - 경사 하강법에서 한 번에 얼마나 큰 걸음으로 이동할지를 결정하는 매우 중요한 하이퍼파라미터
  - 학습률이 너무 크면 (high): 최적점을 그냥 지나쳐 버려 값이 발산(diverge)할 수 있다.
  - 학습률이 너무 크면 (high): 최적점을 그냥 지나쳐 버려 값이 발산(diverge)할 수 있다.
  - 따라서, 적절한 학습률을 찾는 것이 중요하다.

Stochastic GD and Mini-Batch

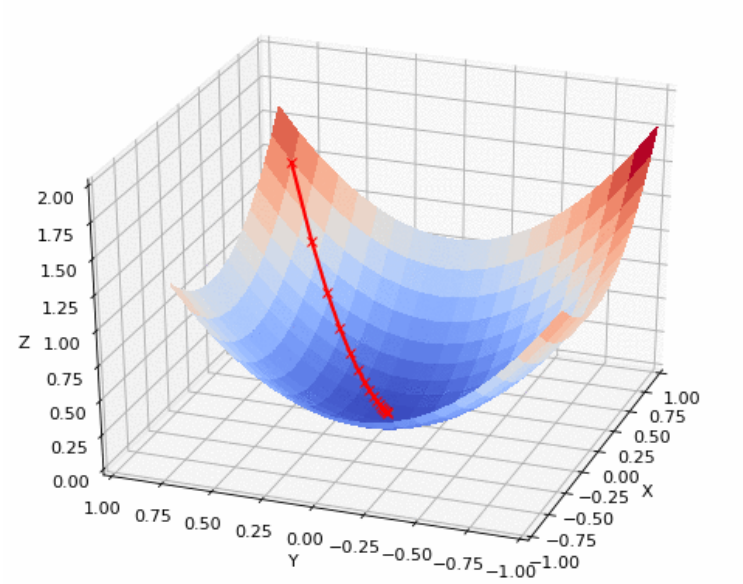
- **경사 하강법의 한계:**
  - 기본 경사 하강법(배치 경사 하강법)은 한 걸음을 내딛기 위해 **전체 학습 데이터**를 모두 계산에 사용해야 한다. 데이터가 수백만 개 이상인 딥러닝에서는 이 방식이 비효율적이고 너무 느리다는 단점이 있다.

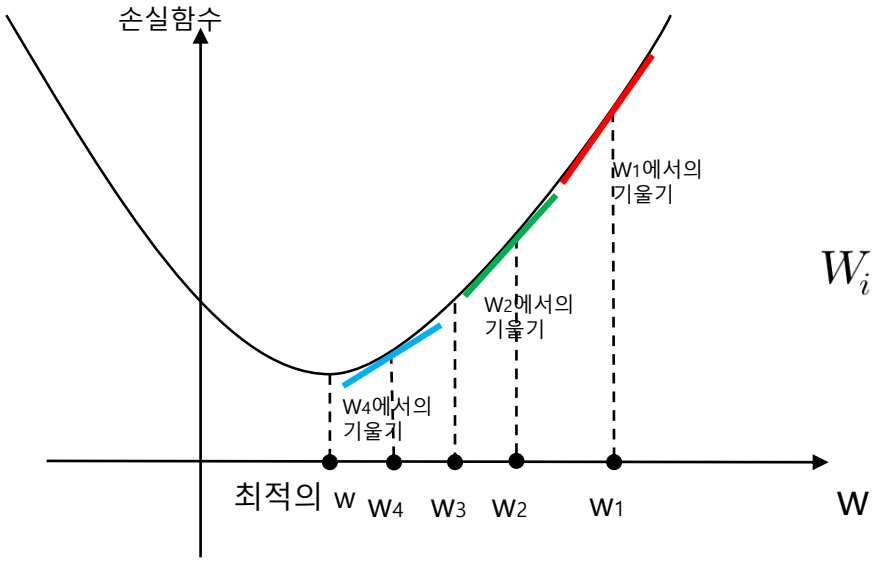
- **확률적 경사 하강법 (SGD)과 미니배치:**
  - 확률적 경사 하강법 (Stochastic Gradient Descent, SGD): 전체 데이터가 아닌, 랜덤하게 뽑은 데이터 1개만 보고 가중치를 업데이트한다. 속도는 매우 빠르지만, 업데이트 방향이 불안정하여 손실 값이 심하게 요동칠 수 있다.
  - 미니배치 경사 하강법 (Mini-batch GD): 전체 데이터와 1개 데이터의 절충안으로, 작은 묶음(mini-batch, 보통 32~256개)의 데이터를 보고 가중치를 업데이트한다. 안정적이면서도 속도가 빨라, 현재 딥러닝에서 가장 표준적으로 사용되는 방식이다.

Optimization Algorithms

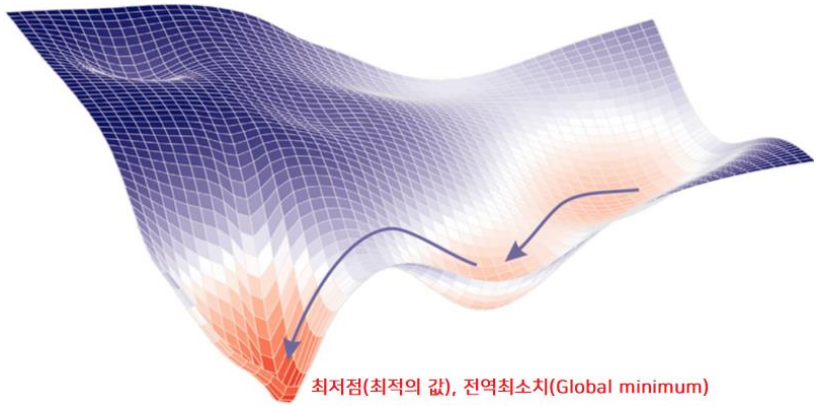
Optimization algorithms

- **Gradient Descent (경사하강법)**
  - General optimization algorithm
  - take repeated steps in the opposite direction of the **gradient** (or approximate **gradient**) of the function at the current point



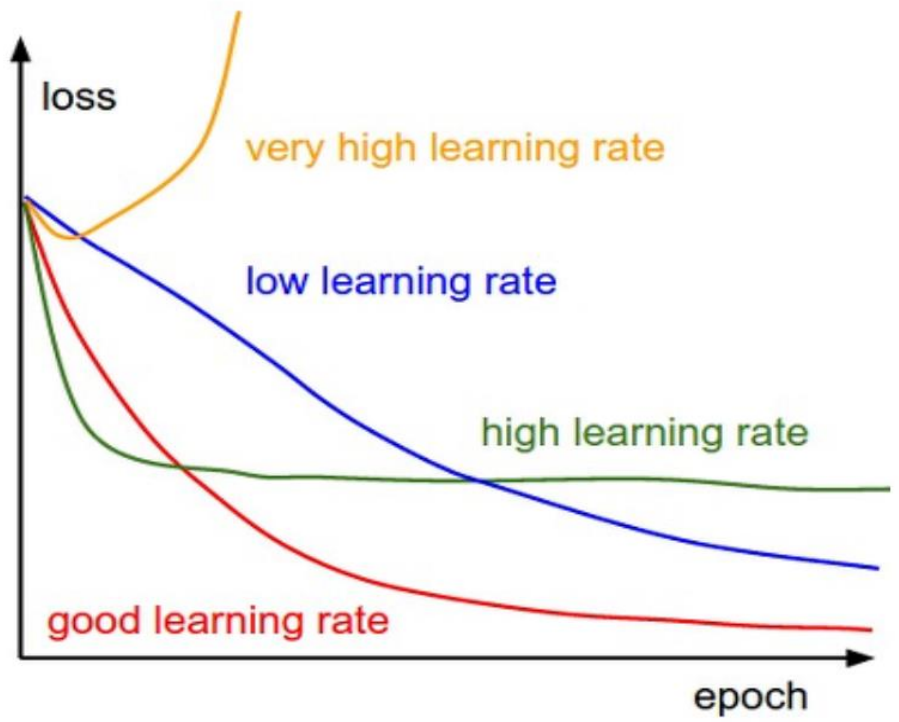


$$W_i = W_{i-1} - \eta Grad(i)$$



Optimization algorithms

- **Learning rate: η (eta)**
  - low: takes time to converge, and may get stuck in an undesirable local minimum
  - high: may jump over minima
  - too high: may diverge
  - Need adaptive adjustment

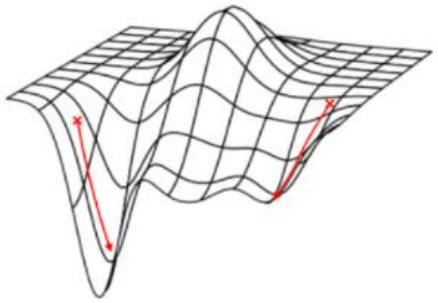


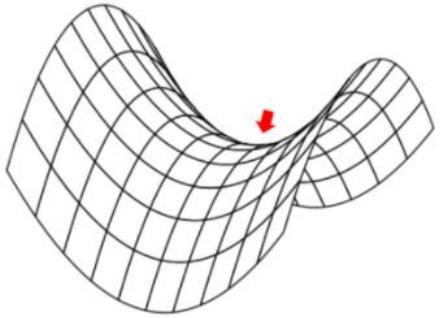
Optimization algorithms

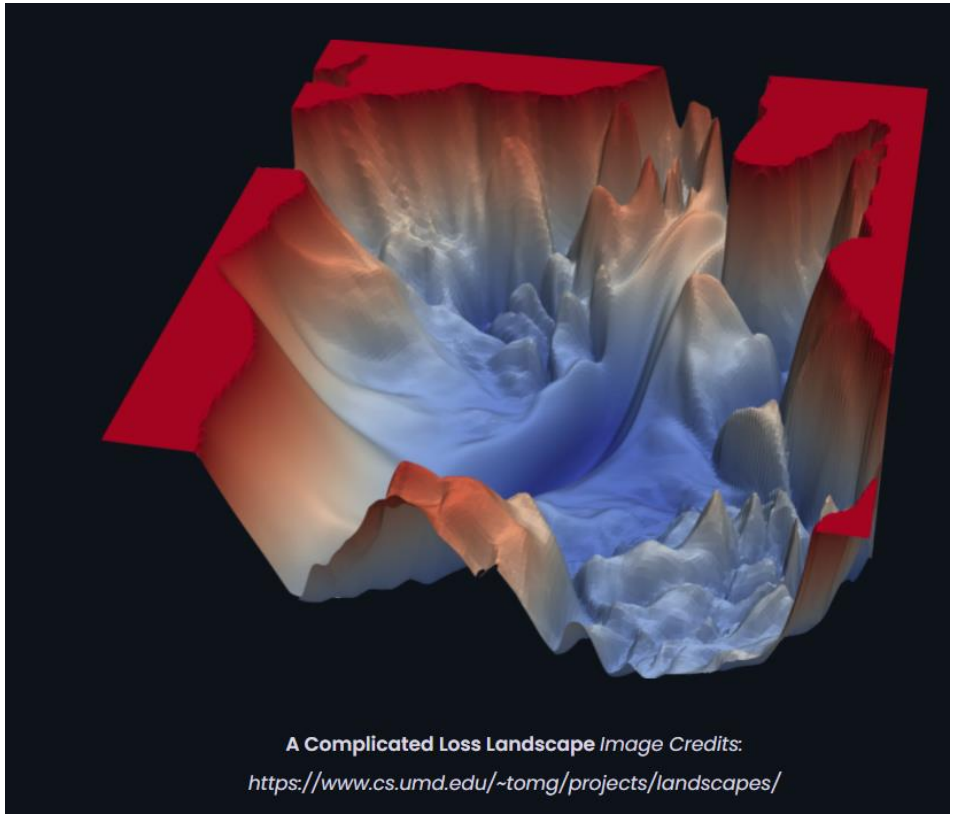
- **Gradient descent**
  - computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset for each update (too much computation)

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

  - Local minimum
  - Saddle Point





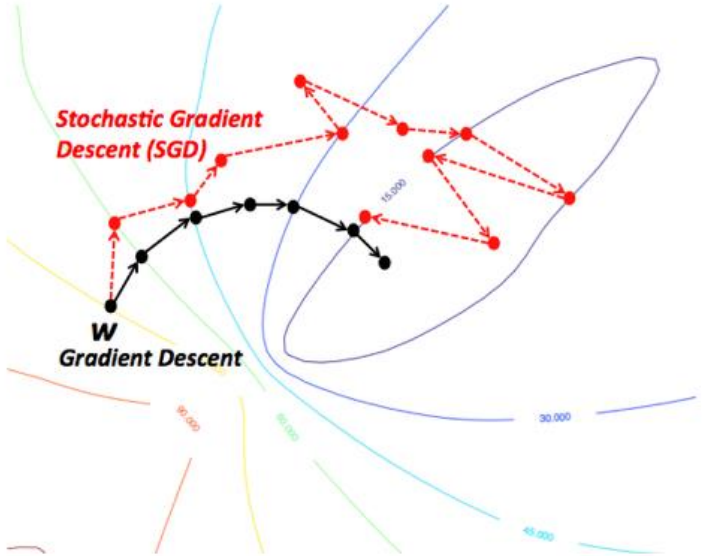


A Complicated Loss Landscape *Image Credits:*
https://www.cs.umd.edu/~tomg/projects/landscapes/

Optimization algorithms

- **Stochastic gradient descent (SGD)**
  - performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$
  - usually much faster and can also be used to learn online
  - It performs frequent updates with a high variance that cause the objective function to fluctuate heavily.
  - higher probability not to fall in local minima

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

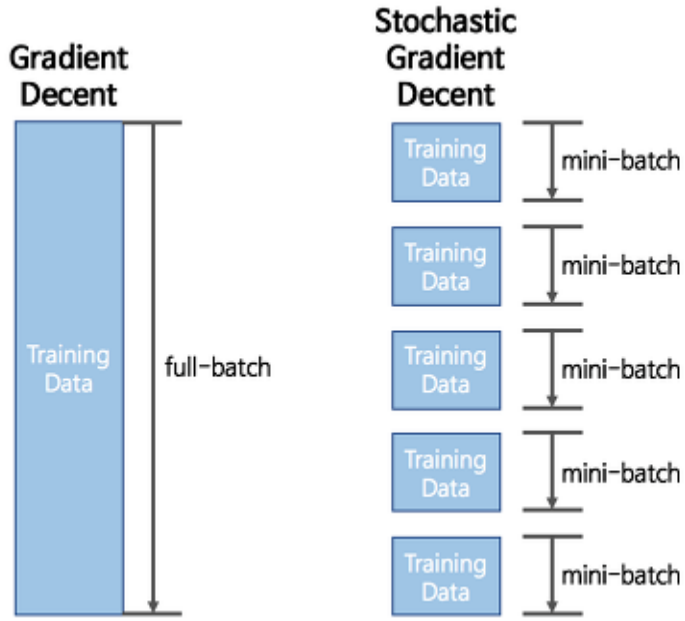


Optimization algorithms

- **Mini-batch gradient descent**
  - performs an update for every **mini-batch** of *n* training examples
  - common mini-batch sizes range between 50 and 256, but can vary for different applications

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

- **Challenges**
  - GD does **not guarantee good convergence**, but offers few challenges.
  - How to find optimal learning rates? -> decaying (step size)
  - How often should the parameters be updated? -> batch size
  - How to escape from local minima? -> momentum (step direction)



Problems in Optimization

- 미니배치 경사 하강법의 문제점:
  - 손실 함수의 표면이 길고 좁은 골짜기 형태일 때, 골짜기 양옆으로 진동하며 비효율적으로 탐색할 수 있다.
  - 최적점이 아닌 움푹 팬 지점(local minimum)이나 평평한 안장점(saddle point)에 빠져 학습이 멈출 수 있다.
- 이 문제들을 해결하기 위해, 단순히 기울기 반대 방향으로만 움직이는 것을 넘어 더 현명하게 움직이는 방법들이 고안되었다. 핵심 아이디어는 걸음의 방향과 걸음의 크기를 **동적으로 조절**하는 것이다.

Optimization algorithms

- **Momentum (batch GD with Momentum)**
  - movement(update) vector (*v*) introduced
  - Also use the results from previous batch (momentum term ***m*** is close to 0.9)
  - Converge faster, reduced vibration, and helps to escape from local minimums.

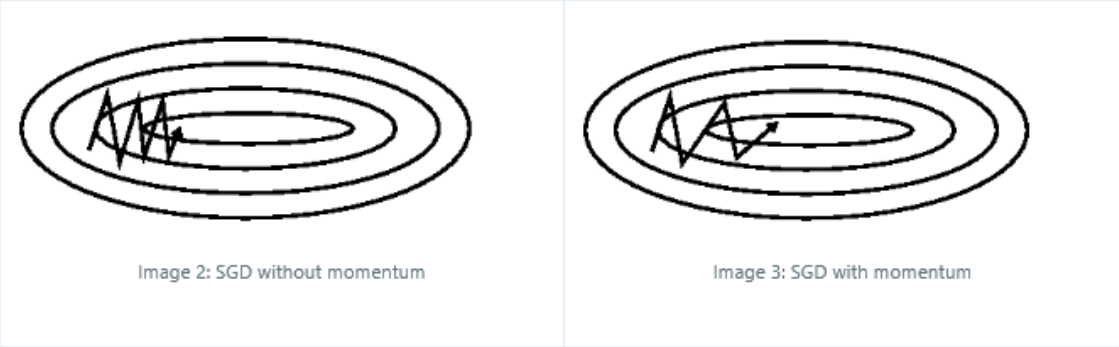
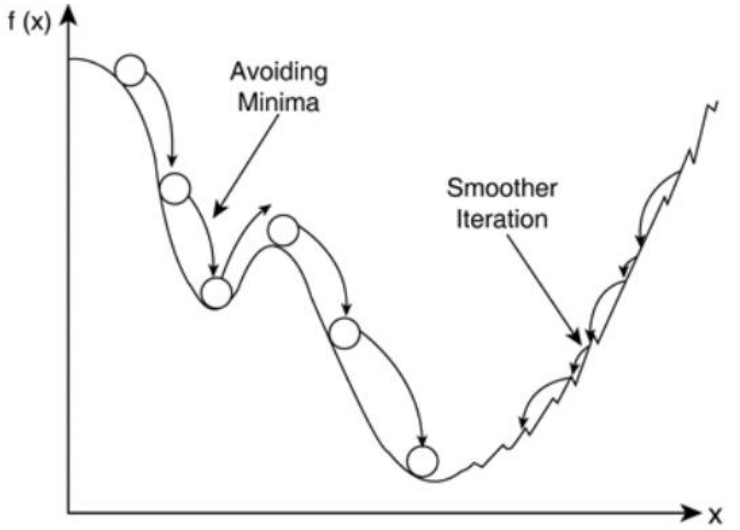


Image 2: SGD without momentum

Image 3: SGD with momentum

$$V_t = m \times V_{t-1} - \eta \nabla_\omega J(\omega_t)$$

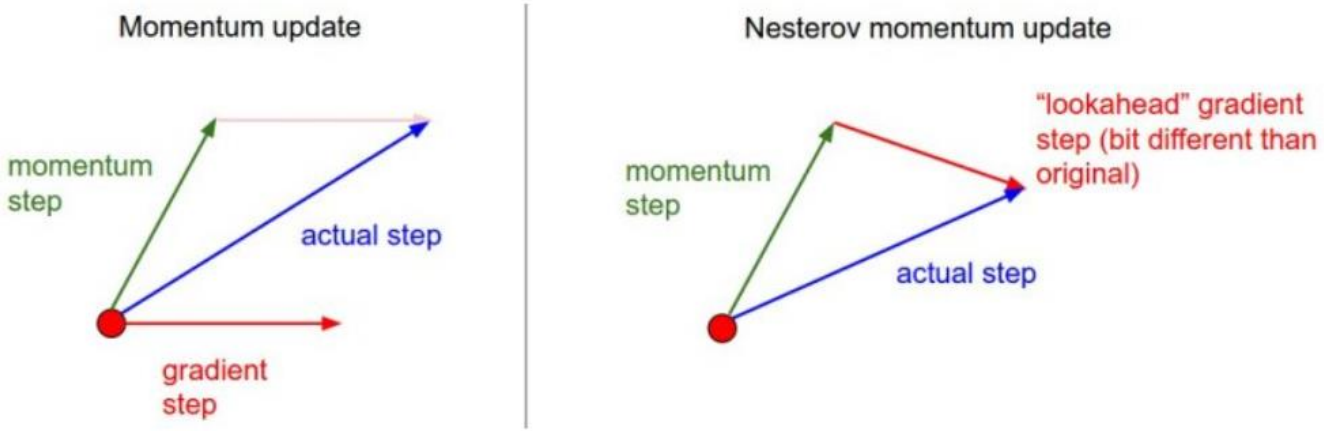
$$\omega_{t+1} = \omega_t + V_t$$



Avoiding Local Minima. Picture from http://www.yaldex.com.

Optimization algorithms

- **Improved Momentum (Nesterov accelerated gradient (NAG))**
  - Momentum 방법과 동일하지만 Gradient 계산이 약간 다름
  - Momentum 에서는 현재 위치의 기울기와 모멘텀을 따로 계산하고 나중에 더하지만, 여기서는 먼저 모멘텀을 계산 후 기울기 를 계산



Difference between Momentum and NAG. Picture from CS231.

$$V_t = m \times V_{t-1} - \eta \nabla_\omega J(\omega_t - m \times V_{t-1})$$

$$\omega_{t+1} = \omega_t + V_t$$

적응형 학습률 (Adaptive Learning Rates)

- Separate adaptive learning rates: 학습률을 weight 에 따라 다르게 함.
- **Adagrad** (Adaptive gradient)
  - 파라미터 별 update (different learning rates for ωi's)
  - 과거에 많이 변경되지 않은 매개 변수에 더 큰 learning rate 적용 -> step-size 감소
  - 작동 방식: 각 파라미터마다 '지금까지 변화해온 양(그래디언트 제곱의 합)'을 기록한다. 그리고 이 값이 클수록 학습률을 작게, 작을수록 학습률을 크게 조절한다.
  - 아래 식에서 squaring 과 dot(.) 연산은 element-wise 연산

$$G_t = G_{t-1} + (\nabla_\omega J(\omega_t))^2 = \sum_{i=1}^{k} (\nabla_{\omega_i} J(\omega_i))^2$$

$$\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_\omega J(\omega_t)$$

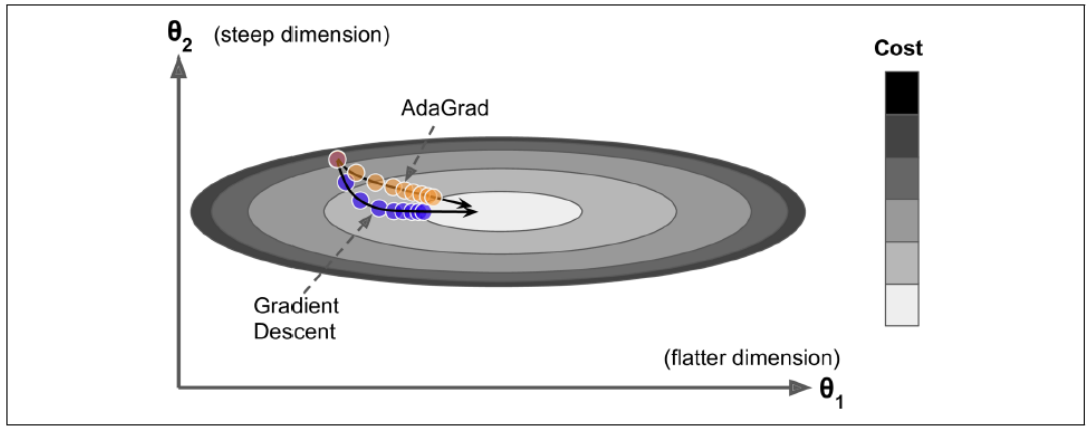


Figure 11-7. AdaGrad versus Gradient Descent

ref: Hands-on-Machine Learning (2nd)

Optimization algorithms

- **RMSProp (**Root Mean Square Propagation**)**
  - Adagrad 알고리즘은 너무 급격히 감소하여 global optimum 에 도달하지 못하는 경우 발생.
  - 처음부터 모든 gradient Gt를 합산하는 대신 지수 이동 평균 (exponential moving average) 사용하여 최근 것 사용 (more weights on the recent gradient): typical decay rate γ = 0.9~0.999
  - Always performs much better than Adagrad, and most preferred until Adam came around.

$$G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_\omega J(\omega_t))^2$$

$$\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_\omega J(\omega_t)$$

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

gamma

Optimization algorithms

- **Adam** (Adaptive Moment Estimation)
  - RMSProp + Momentum
  - Momentum 과 유사하게 기울기의 지수 평균 반영 (과거 기울기 (past gradient) 의 지수적으로 감소하는 평균을 유지)
  - RMSProp 과 유사하게 기울기 제곱 값의 지수 평균 반영 (각 매개 변수에 대한 적응형 학습률 (adaptive learning rate))
  - Recommended values: $\varepsilon=10^{-8}$, $\beta_1=0.9$, $\beta_2=0.999$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\omega J(\omega_t)$$ **first moment (관성)**

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\omega J(\omega_t))^2$$ **second moment(학습률)**

$$\omega_{t+1} = \omega_t - m_t \frac{\eta}{\sqrt{v_t + \epsilon}}$$

$$M(t) = \beta_1 M(t-1) + (1-\beta_1)\frac{\partial}{\partial w(t)} Cost(w(t))$$

$$V(t) = \beta_2 V(t-1) + (1-\beta_2)\left(\frac{\partial}{\partial w(i)} Cost(w(i))\right)^2$$

$$\hat{M}(t) = \frac{M(t)}{1-\beta_1^{\,t}} \qquad \hat{V}(t) = \frac{V(t)}{1-\beta_2^{\,t}}$$

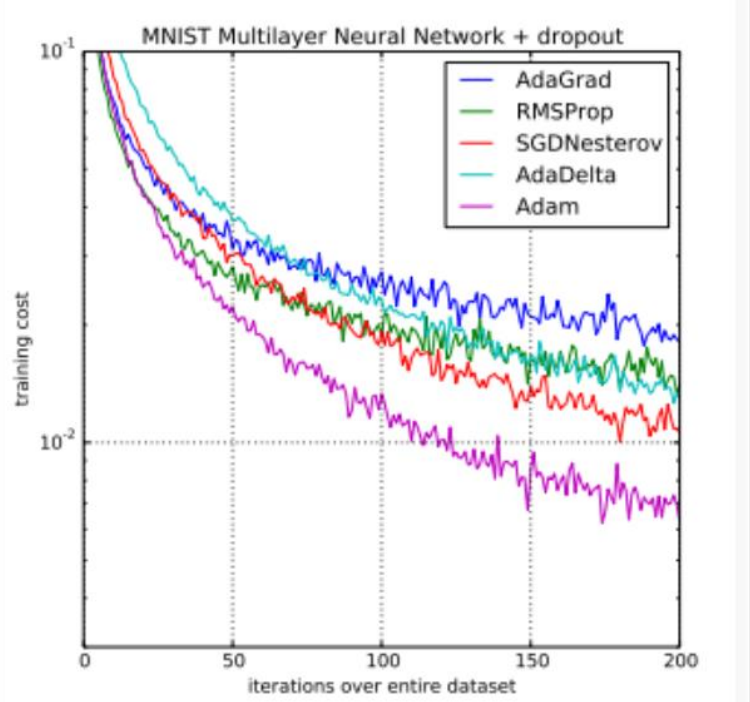
$$W(t+1) = W(t) - \alpha * \frac{\hat{M}(t)}{\sqrt{\hat{V}(t) + \epsilon}}$$

- bias correction terms (exponential decay)
- 일반적으로 m, v 가 0 으로 초기화되어 있기 때문에 훈련 초기에 크게 bias 되는데 이를 교정하기 위해 bias correction term 이 필요함.
- 훈련 iteration 이 진행 되면서 bias correction term 은 1 로 접근하게 됨.

Optimization algorithms

- **Adam** (Adaptive Moment Estimation)

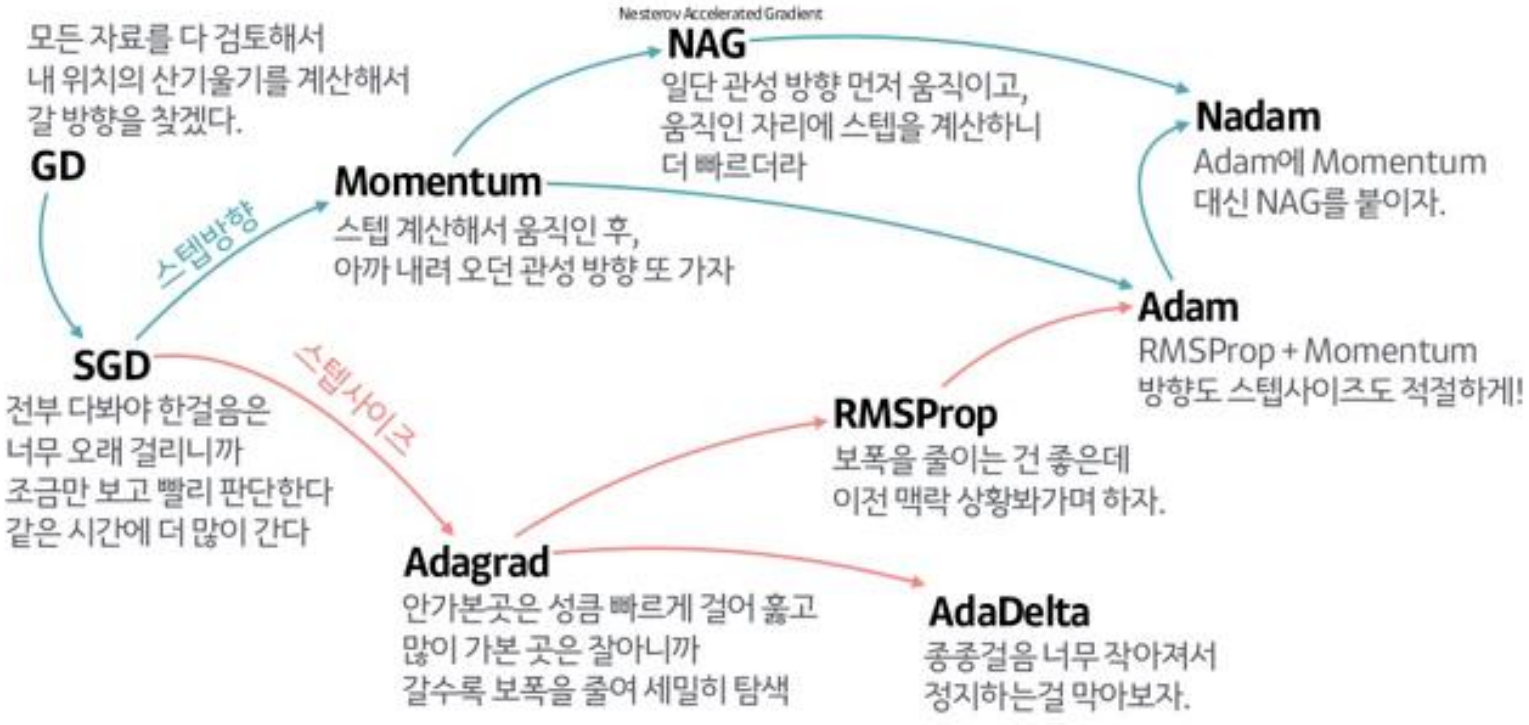
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)



Comparison of Adam to Other Optimization Algorithms Training a Multilayer Perceptron
Taken from Adam: A Method for Stochastic Optimization, 2015.

Optimization algorithms

- **Summary**



- **More:**
  - Adamax
  - Nadam
  - AMSGrad
  - more …
- **Good animation**
  - https://towardsdatascience.com/complete-guide-to-adam-optimization-1e5f29532c3d

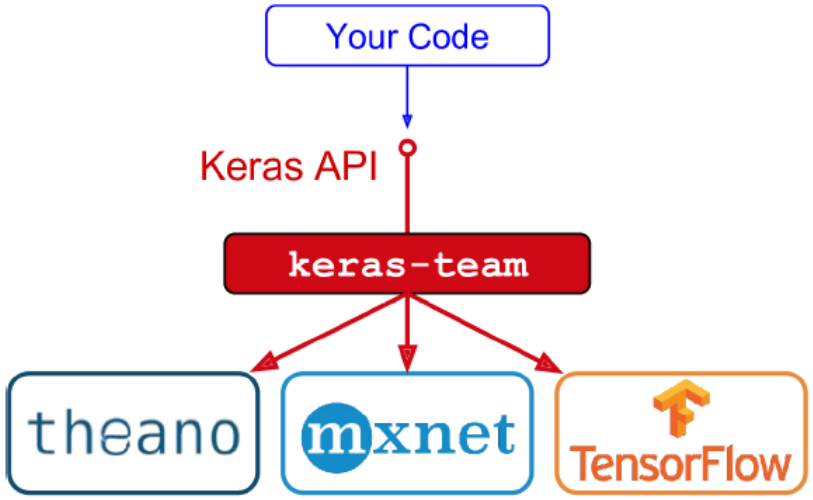
Deep Learning using Keras

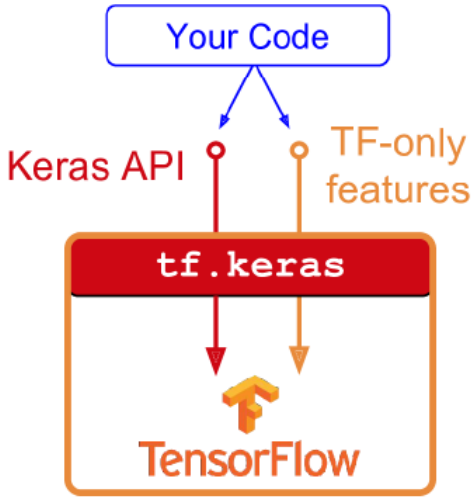
Tensorflow and Keras

- **딥러닝 구현을 위한 도구: 텐서플로우와 케라스:**
  - 텐서플로우(TensorFlow): 구글이 개발한 가장 강력하고 대중적인 딥러닝 프레임워크다. 저수준(low-level)의 복잡한 연산부터 고수준(high-level) 모델링까지 모두 지원하는 핵심 엔진이다.
  - 케라스(Keras): 텐서플로우 위에서 작동하는 사용자 친화적인 고수준 API다. 마치 자동차의 '자동 변속기'처럼, 복잡한 내부 구현을 몰라도 쉽고 빠르게 딥러닝 모델을 만들 수 있게 해준다.
  - 우리는 Trensorflow 에 내장된 tf.keras 를 사용한다.

Implementing MLP with Keras

- **Original Keras (https://keras.io) by Francois Chollet**
  - Can choose Backend from Tensorflow, Microsoft (CNTK), or Theano
- **Tensorflow Keras**
  - Tensorflow 2.0 comes bundled with its own Keras (tf.keras)
  - Only support Tensorflow as the backend






```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

Building a Model with Keras

- **Sequential API**
  - Create the model

(모델 생성 방법 1: model.add() 사용)

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

(모델 생성 방법 2: 리스트로 한 번에 전달)

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

  - Compile the Model (specify loss function and optimizer, [extra metrics])
  - Training and evaluating
  - Make prediction

Building a Model with Keras

- **Model compile**
  - Compile() 메쏘드를 통해 학습 과정을 설정한다.

(*) sparse_categorical_crossentropy: when the target is integer (sparse) labeled (not one-hot encoded)

```
model.compile(loss="sparse_categorical_crossentropy",   # 손실함수
              optimizer="sgd",          # 최적화 알고리즘 (예: 'sgd', 'adam')
              metrics=["accuracy"])     # 훈련 중 모니터링할 지표 (예:정확도)
```

Building a Model with Keras

- **Model Train, Evaluate, and Predict**

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
```

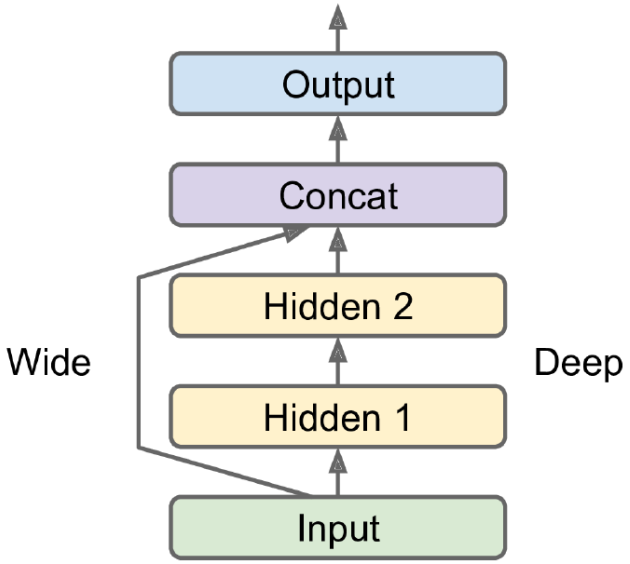
```
>>> model.evaluate(X_test, y_test)
8832/10000 [=========================] - ETA: 0s - loss: 0.4074 - acc: 0.8540
[0.40738476498126985, 0.854]
```

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.09, 0.  , 0.12, 0.  , 0.79],
       [0.  , 0.  , 0.94, 0.  , 0.02, 0.  , 0.04, 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

Building a Model with Keras

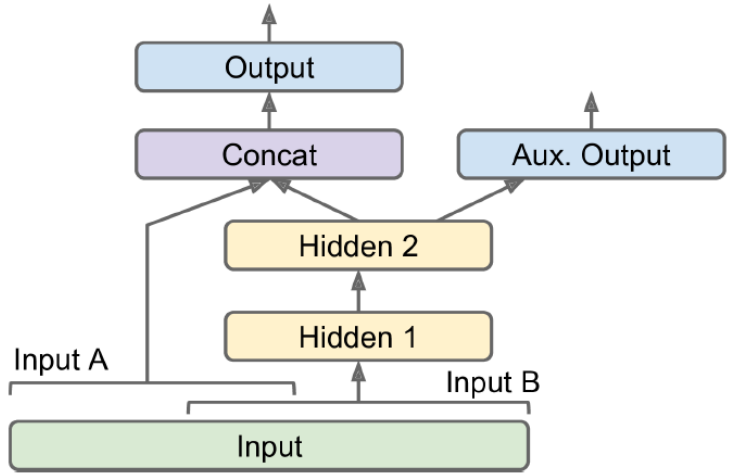
- **Functional API: to build complex models**

Wide and deep Neural network




```
input = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()[input, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input], outputs=[output])
```

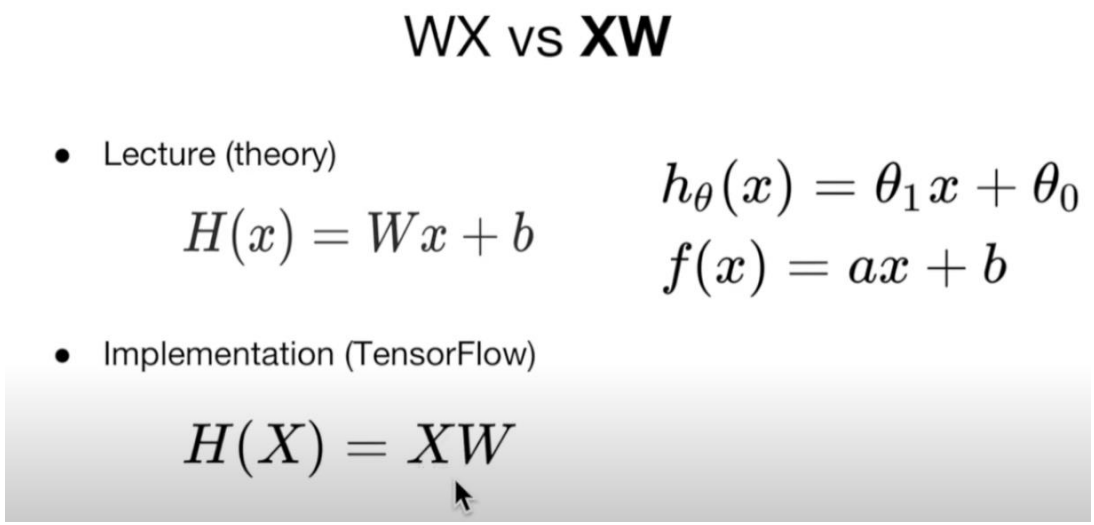
Handling multiple outputs

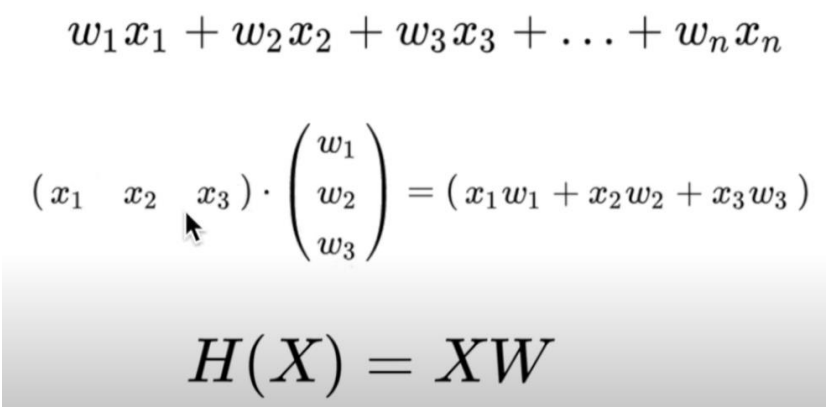



```
[...] # Same as above, up to the main output layer
output = keras.layers.Dense(1)(concat)
aux_output = keras.layers.Dense(1)(hidden2)
model = keras.models.Model(inputs=[input_A, input_B],
                           outputs=[output, aux_output])
```

구현의 이면: 행렬 연산

- 신경망의 모든 계산은 사실 행렬(Matrix) 연산으로 이루어진다.
- 입력 벡터 X와 가중치 행렬 W의 곱셈인 H(X) = XW 로 표현.
- 수많은 데이터를 한 번에 처리하기 위해, 여러 데이터의 입력 벡터를 행으로 쌓은 행렬 X를 만들어 가중치 행렬 W와 곱셈한다. 이는 GPU를 이용한 병렬 처리에 매우 효율





From https://www.youtube.com/watch?v=9l1YHo-pbf8&list=PLQ28Nx3M4Jrguyuwg4xe9d9t2XE639e5C&index=7

구현의 이면: 행렬 연산

x_1	x_2	x_3	y
73	80	75	152
93	88	93	185
89	91	90	180
96	98	100	196
73	66	70	142

Test Scores for General Psychology

$$H(x_1, x_2, x_3) = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} x_1 w_1 + x_2 w_2 + x_3 w_3 \end{pmatrix}$$

$$H(X) = XW$$

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_n x_n$$

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} x_{11} w_1 + x_{12} w_2 + x_{13} w_3 \\ x_{21} w_1 + x_{22} w_2 + x_{23} w_3 \\ x_{31} w_1 + x_{32} w_2 + x_{33} w_3 \\ x_{41} w_1 + x_{42} w_2 + x_{43} w_3 \\ x_{51} w_1 + x_{52} w_2 + x_{53} w_3 \end{pmatrix}$$

$$H(X) = XW$$

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} x_{11} w_1 + x_{12} w_2 + x_{13} w_3 \\ x_{21} w_1 + x_{22} w_2 + x_{23} w_3 \\ x_{31} w_1 + x_{32} w_2 + x_{33} w_3 \\ x_{41} w_1 + x_{42} w_2 + x_{43} w_3 \\ x_{51} w_1 + x_{52} w_2 + x_{53} w_3 \end{pmatrix}$$

[n, 3] [3, 1] [n, 1]

$$H(X) = XW$$

Hypothesis using Matrices

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{pmatrix} \cdot ? = \begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} + x_{13}w_{31} & x_{11}w_{12} + x_{12}w_{22} + x_{13}w_{32} \\ x_{21}w_{11} + x_{22}w_{21} + x_{23}w_{31} & x_{21}w_{12} + x_{22}w_{22} + x_{23}w_{32} \\ x_{31}w_{11} + x_{32}w_{21} + x_{33}w_{31} & x_{31}w_{12} + x_{32}w_{22} + x_{33}w_{32} \\ x_{41}w_{11} + x_{42}w_{21} + x_{43}w_{31} & x_{41}w_{12} + x_{42}w_{22} + x_{43}w_{32} \\ x_{51}w_{11} + x_{52}w_{21} + x_{53}w_{31} & x_{51}w_{12} + x_{52}w_{22} + x_{53}w_{32} \end{pmatrix}$$

[n, 3] [?, ?] [n, 2]

$$H(X) = XW$$

Hypothesis using Matrices (Example)

- **Easier and simpler notation (Using Matrices)**

$$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} x_1 w_1 + x_2 w_2 + x_3 w_3 \end{pmatrix}$$

$$H(X) = XW$$

```
# initialize W
w1 = tf.Variable(tf.random_normal([1]))
w2 = tf.Variable(tf.random_normal([1]))
w3 = tf.Variable(tf.random_normal([1]))

# hypothesis, prediction function
w1 * x1 +  w2 * x2 + w3 * x3 + b

# update w1,w2,w3
w1.assign_sub(learning_rate * w1_grad)
w2.assign_sub(learning_rate * w2_grad)
w3.assign_sub(learning_rate * w3_grad)
```

```
# initialize W
W = tf.Variable(tf.random_normal([3, 1]))

# hypothesis, prediction function
tf.matmul(X, W) + b

# updates parameters (W and b)
W.assign_sub(learning_rate * W_grad)
```

Hypothesis using Matrices (Example)

- Low-level Tensorflow 구현 (Keras의 `.fit()` 메쏘드 내부)

$$H(X) = XW$$

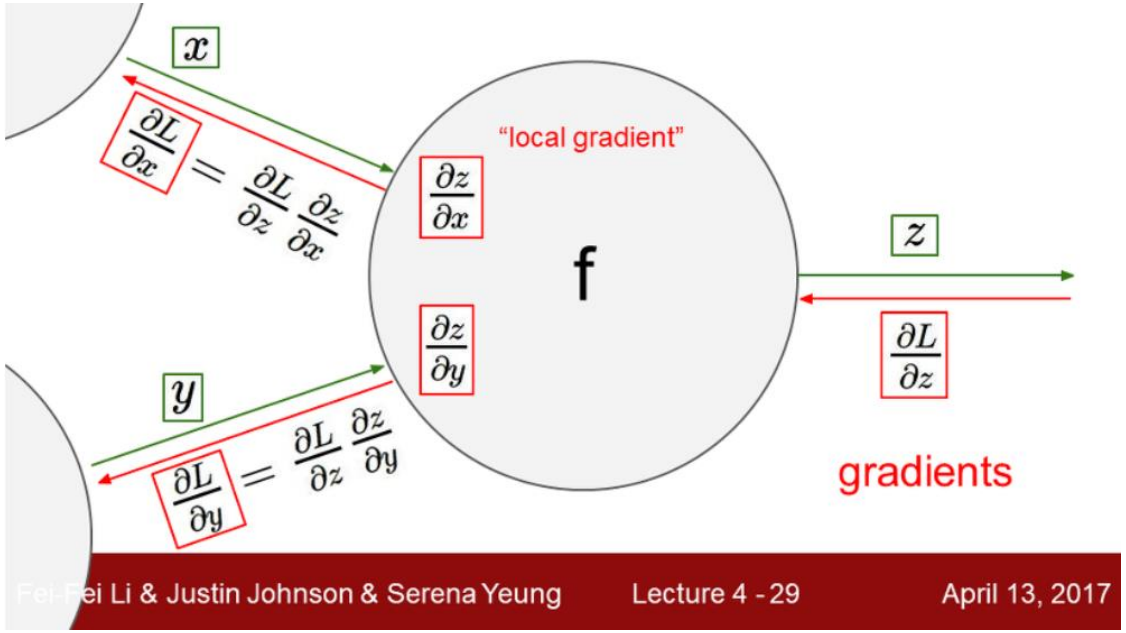
```
data = np.array([
    # X1,   X2,    X3,   y
    [ 73.,  80.,  75., 152. ],
    [ 93.,  88.,  93., 185. ],
    [ 89.,  91.,  90., 180. ],
    [ 96.,  98., 100., 196. ],
    [ 73.,  66.,  70., 142. ]
], dtype=np.float32)

# slice data
X = data[:, :-1]
y = data[:, [-1]]
```

```
# 가중치 W와 편향 b를 변수로 직접 선언
W = tf.Variable(tf.random.normal([3, 1]))
b = tf.Variable(tf.random.normal([1]))

# 학습 루프
for i in range(2001):
    # GradientTape로 그래디언트를 기록
    with tf.GradientTape() as tape:
        # 가설(예측)과 비용(손실) 계산
        hypothesis = tf.matmul(X, W) + b
        cost = tf.reduce_mean(tf.square(hypothesis - y))

    # 그래디언트 계산 및 가중치 업데이트
    W_grad, b_grad = tape.gradient(cost, [W, b])
    W.assign_sub(learning_rate * W_grad)
    b.assign_sub(learning_rate * b_grad)
```

여기까지

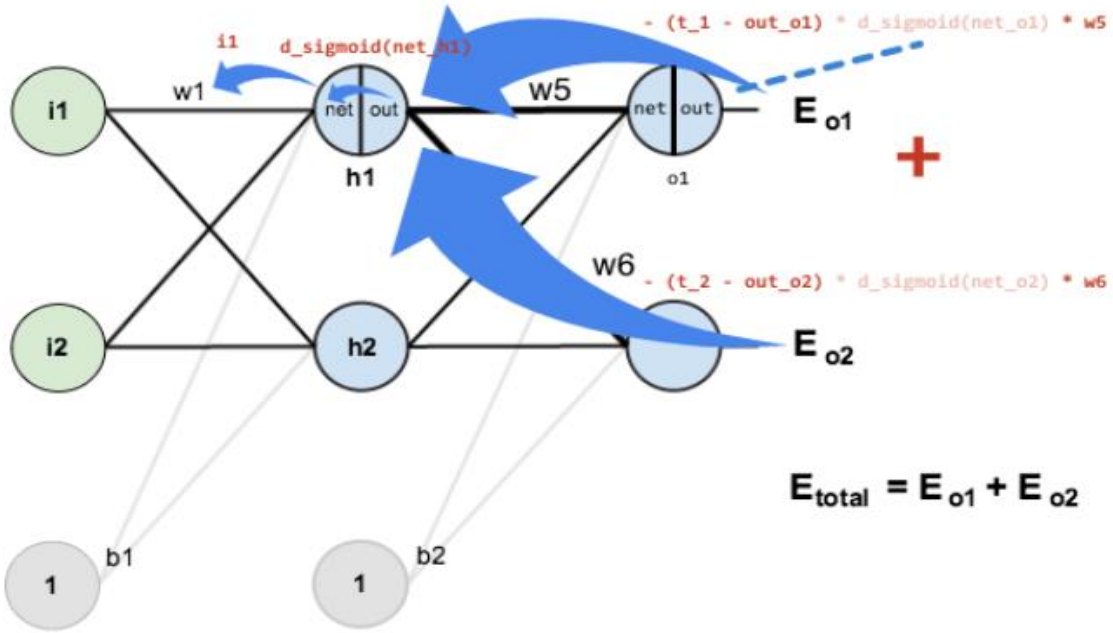
Back-Propagation

- **Use Chain-rule**

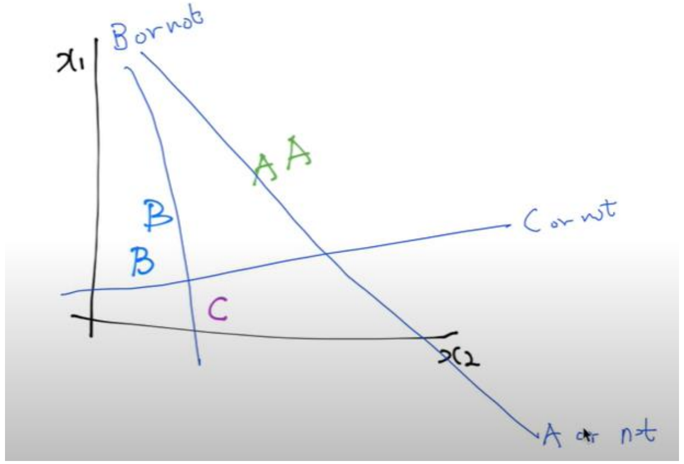


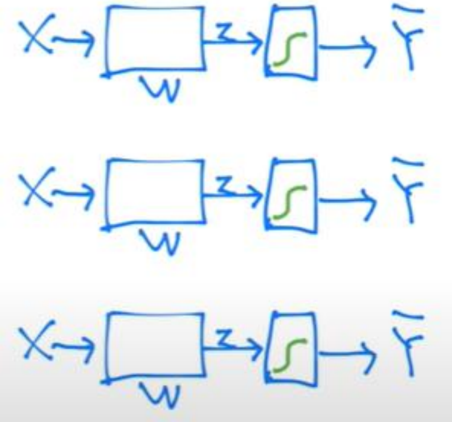
$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

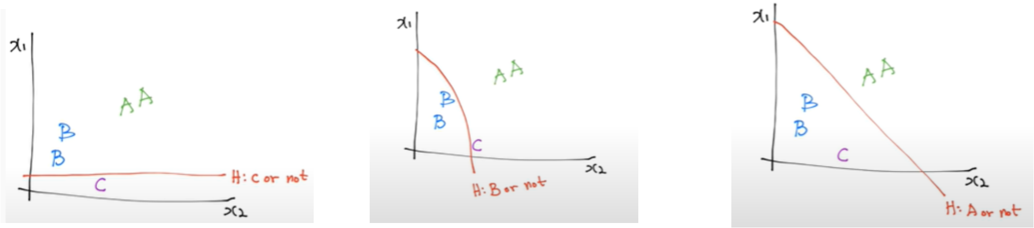
$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

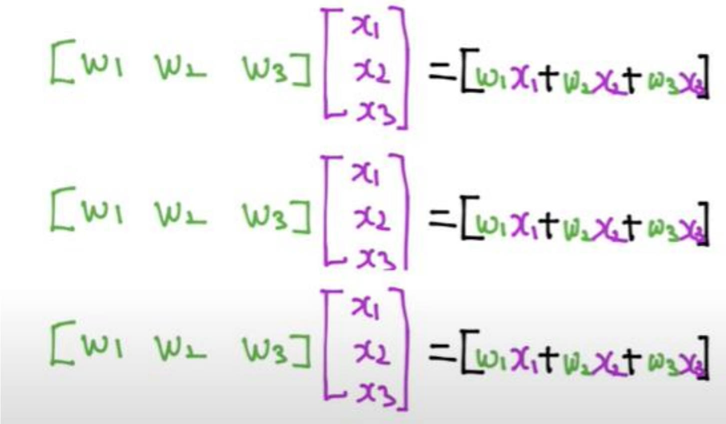


Multinomial (Multi-class) Classification

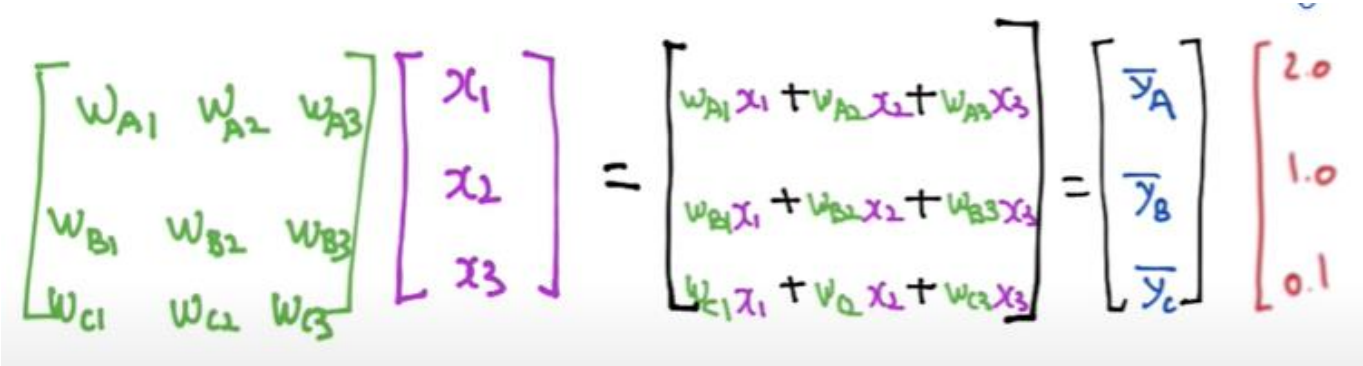




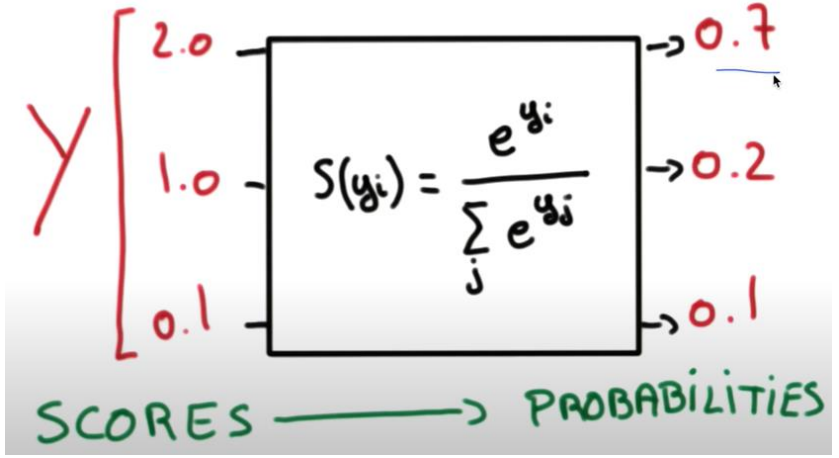




Multinomial (Multi-class) Classification

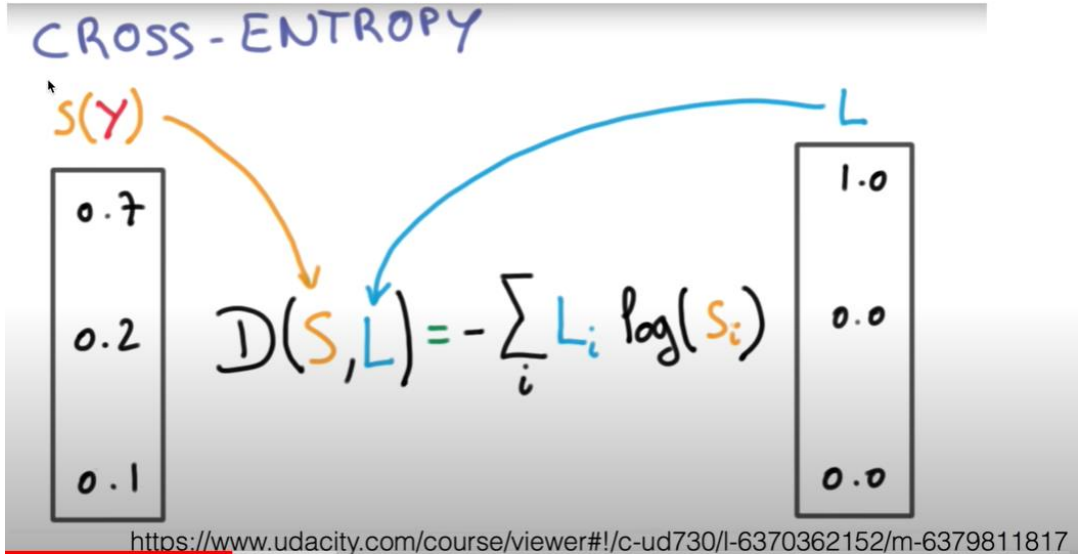


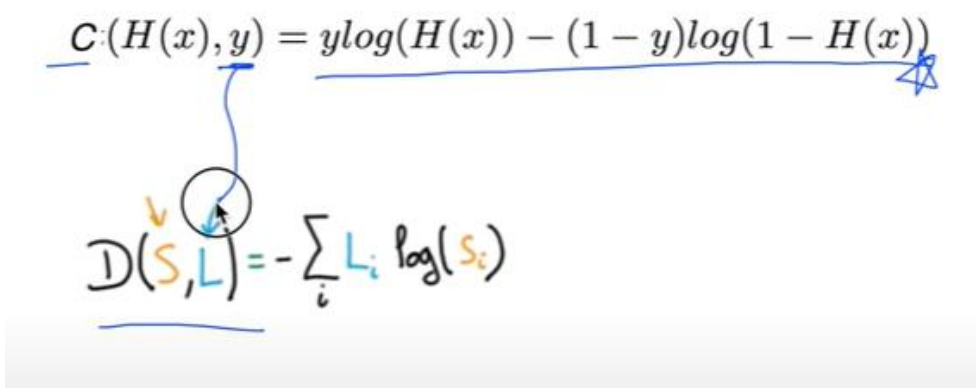
- **Softmax function**



Multinomial (Multi-class) Classification

- **Loss function for Multinomial classification**





Logistic cost (binary cross-entropy)

Cross-entropy cost

Multinomial (Multi-class) Classification

- **Cross Entropy (교차 엔트로피)**

$$E = -\sum_{k} t_k \log y_k$$

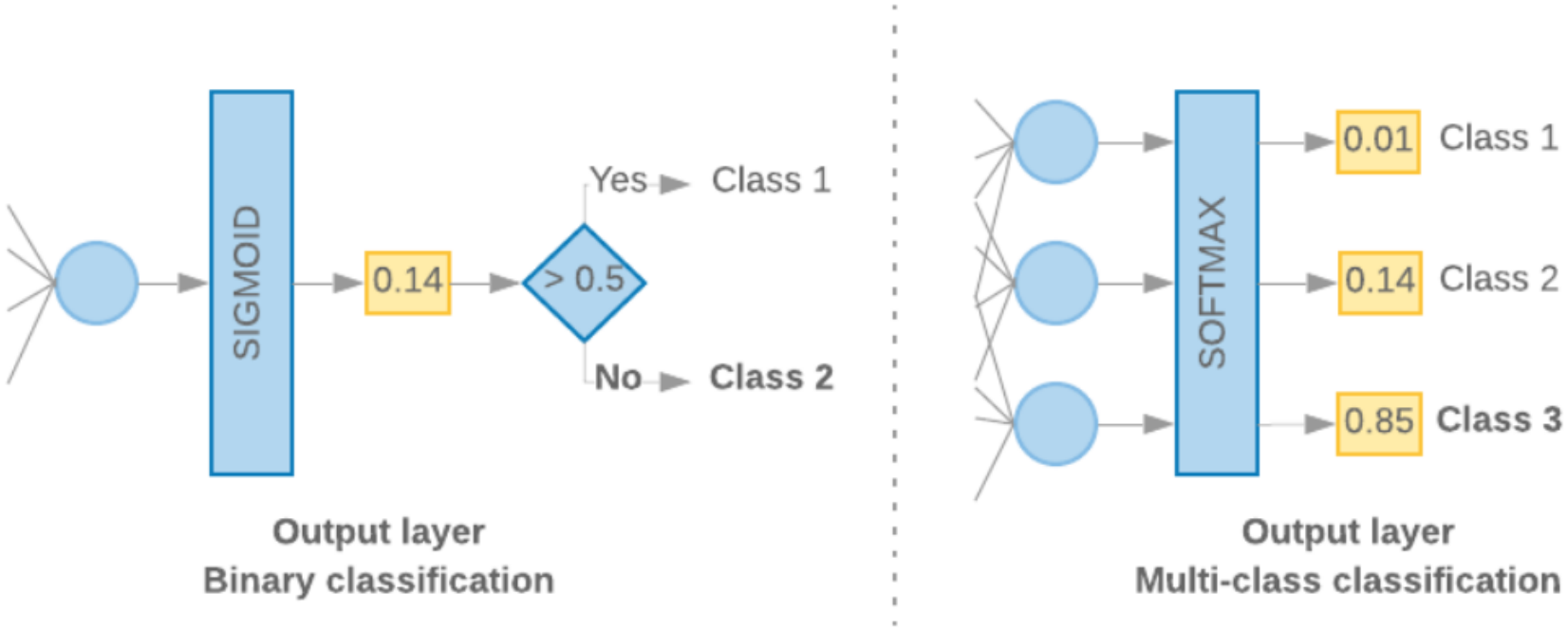
(y: predict, t: target, k: dimension, log: natural log)

- **Softmax (소프트맥스)**

$$\sigma(j) = \frac{\exp(\mathbf{w}_j^\top \mathbf{x})}{\sum_{k=1}^{K} \exp(\mathbf{w}_k^\top \mathbf{x})} = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}$$

Cross Entropy - Softmax

- **Output layer of the Classifier in Deep Learning**



출처: https://developers.google.com/machine-learning/guides/text-classification/step-4

Cross Entropy - Softmax

- 상대적인 점수 비교 : 확률처럼 0~1 사이 값으로 매핑