Introduction to
Deep Learning

산업경영공학과 22학번 박채원
컴퓨터공학과 23학번 이현수

Massachusetts Institute of Technology. *Introduction to Deep Learning.*
Retrieved from https://introtodeeplearning.com

Progress of deep learning



2015



2018



2020



2020

creating this 2 min video required

- 2h of professional audio
- 50h of HD video
- Static, pre-defined script
- Over 15K USD of compute

https://www.youtube.com/watch?v=l82PxsKHxYc



2025

creating this 2 min video required .....?

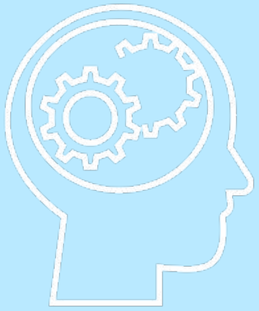
https://app.heygen.com/videos/bbffea1361784fb6afa00b35f8d20281

What is Deep learning?

Artificial Intelligence

정보를 처리하는 능력

컴퓨터가 인간의 행동을 따라할 수 있도록 만드는 모든 기술



Machine Learning

데이터로부터 패턴을 스스로 학습해 예측·분류·판단 학습하는 능력

(정보 자체로 패턴 알아내고 결정을 내림)



Deep Learning

(Deep) neural network를 사용하여 데이터의 패턴을 추출해냄

(스스로 학습해서 Feature 정의)

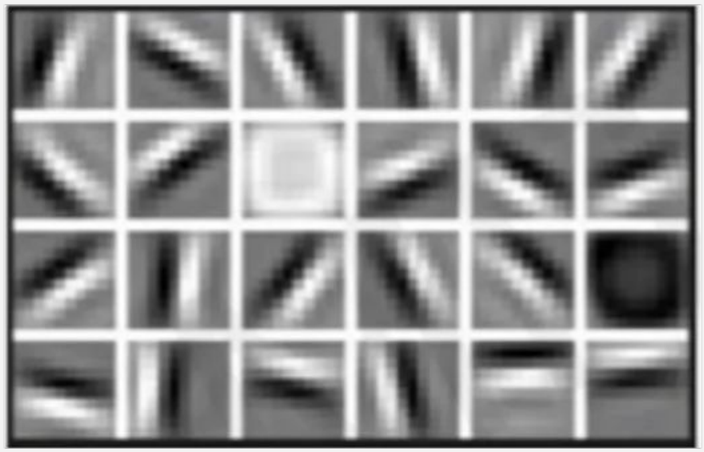


Teaching computers how to **learn a task** directly from **raw data (obeservation)**

Why Deep learning?

직접 feature 만드는 것은 시간이 걸리고, 다루기 힘들며, 실제 수치화가 어려움.

Low level features



Lines & Edges

Mid level features



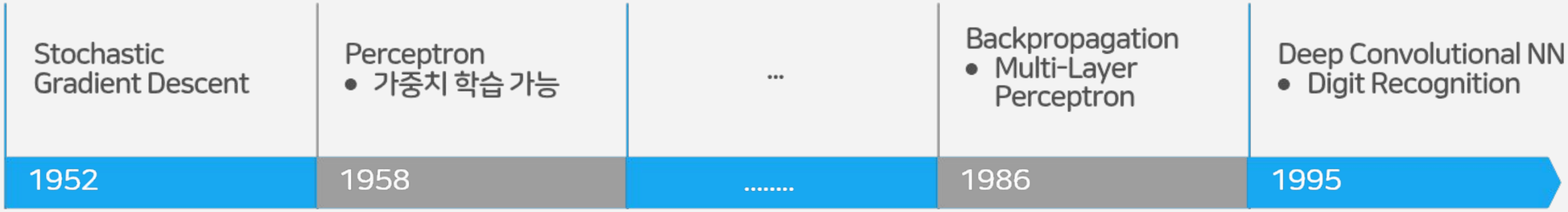
Eyes & Nose & Ears

High level features



Facial Structure

Why Now?



Neural Networks date back decades, so why the dominance?

1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET



2. Hardware (Compute)

- Graphics Processing Units (GPU)
- Massively Parallelizable

3. Software

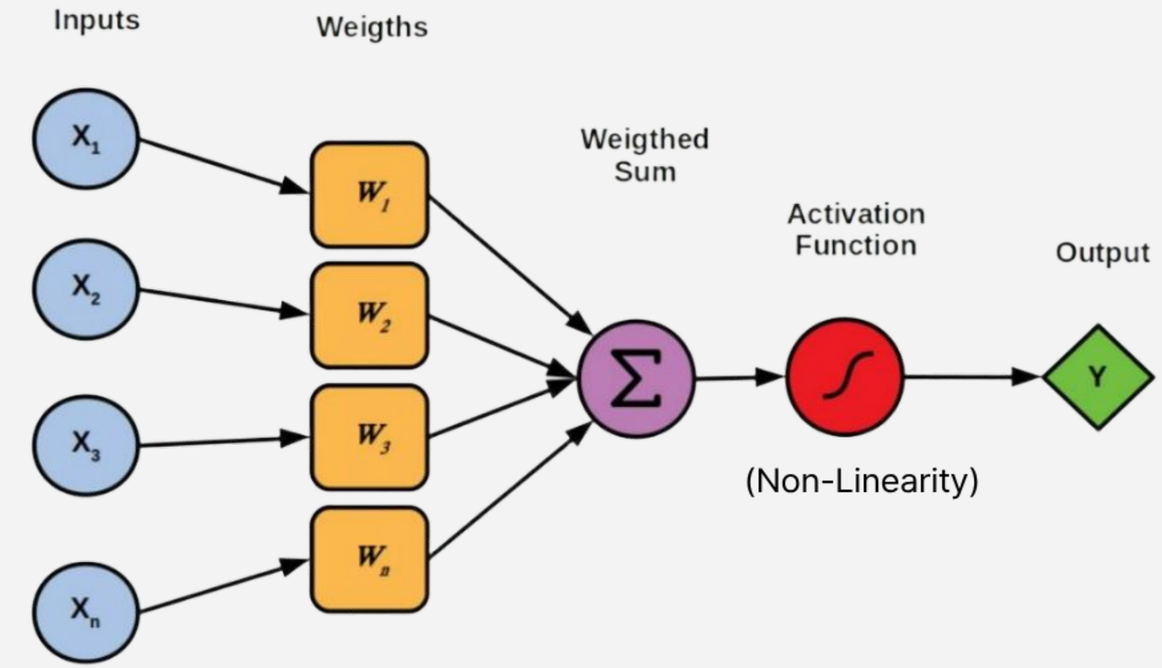
- Improved Techniques
- New Models
- Toolboxes



The Perceptron

- The structural building block of deep learning
- Single neuron
- 초기 형태의 인공 신경망 모델

The Perceptron: Forward Propagation



Features

(Feature의 특성 반영.
합이 1일 필요 없으며 음수도 가능.)

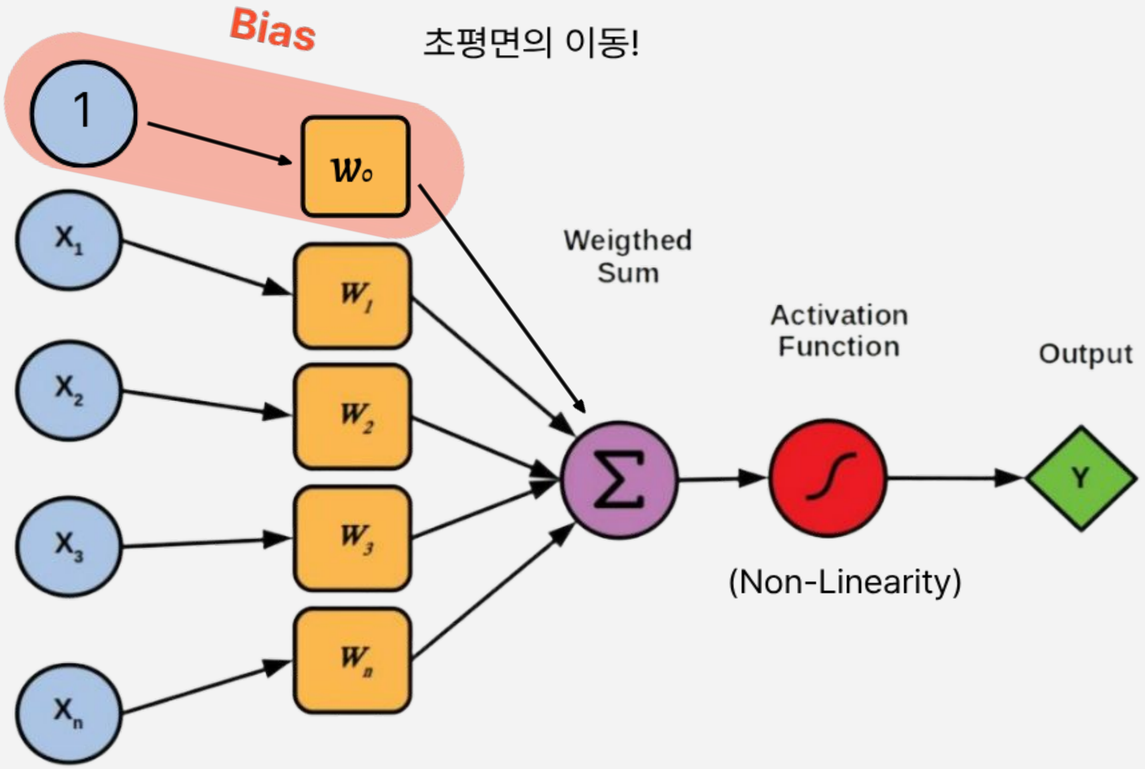
Output

Linear combination of inputs

$$\hat{y} = g(\sum_{i=1}^{m} x_i w_i)$$

Non-linear activation function

The Perceptron: Forward Propagation



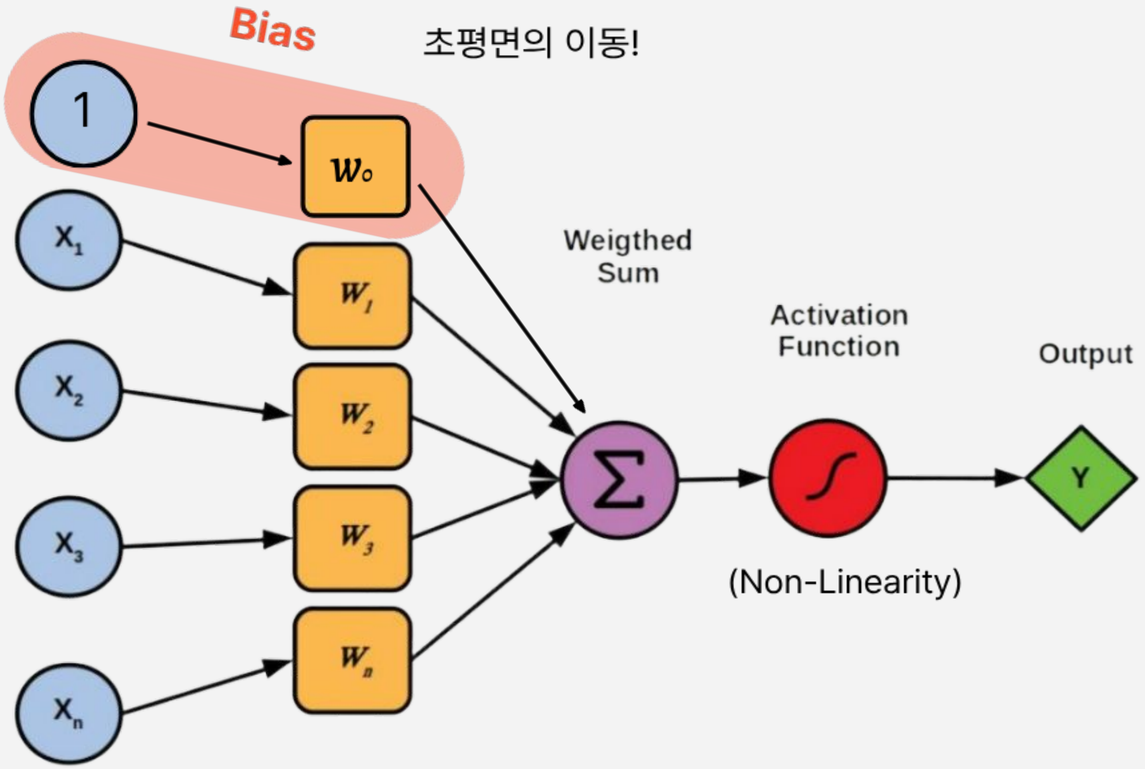
동일 형태로 넣는 이유:
다른 가중치들과 같이 학습되게 하기 위해

Output Bias Linear combination of inputs

$$\hat{y} = g(w_0 + \sum_{i=1}^{m} x_i w_i)$$

Non-linear activation function

The Perceptron: Forward Propagation



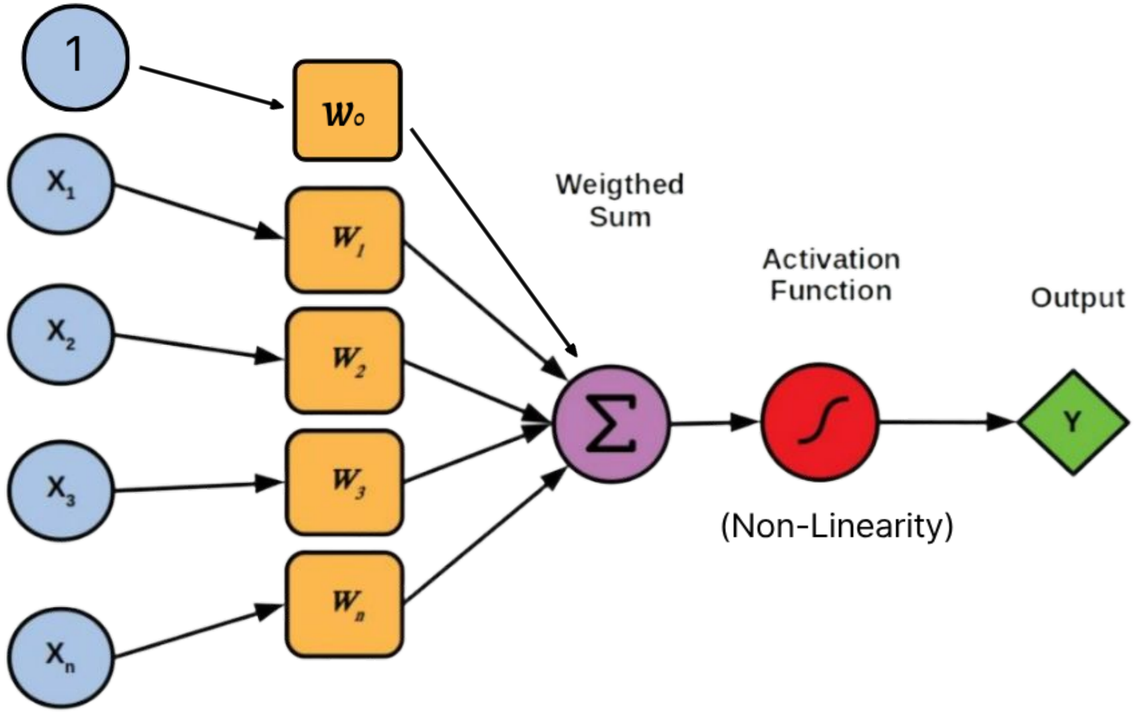
동일 형태로 넣는 이유:
다른 가중치들과 같이 학습되게 하기 위해

$$\hat{y} = g(w_0 + \sum_{i=1}^{m} x_i w_i)$$

$$\hat{y} = g(w_0 + X^T W)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

The Perceptron: Forward Propagation

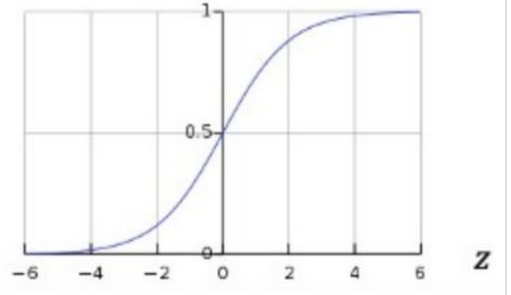


Activation Functions

$$\hat{y} = g(w_0 + X^T W)$$

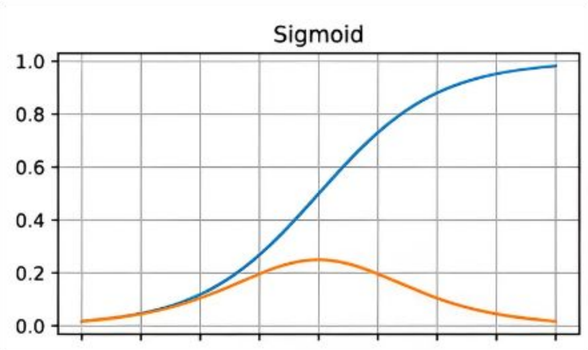
Example: sigmoid function

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



Common Activation Functions

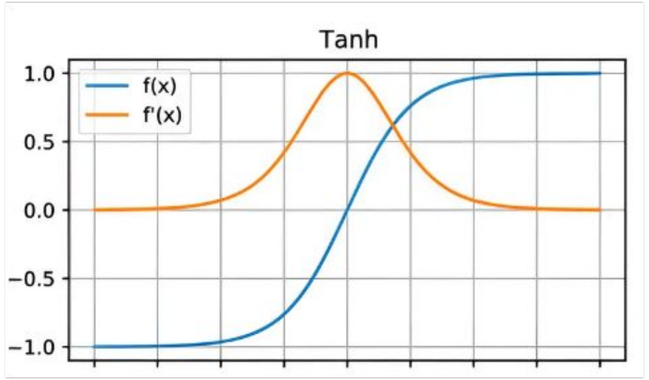
Sigmoid Function



$$\varphi(x) = \frac{1}{1 + exp^{-ax}} = \frac{exp^{ax}}{1 + exp^{ax}}$$

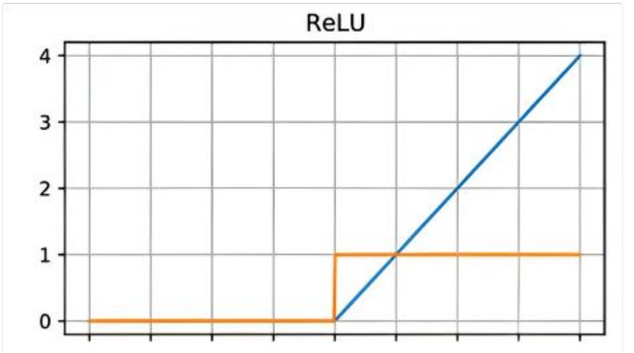
$$\varphi'(v) = a\varphi(v)(1 - \varphi(v))$$

Hyperbolic Tangent



$$\varphi(x) = \frac{e^{ax} - e^{-ax}}{e^{ax} + e^{-ax}}$$

Rectified Linear Unit (ReLU)



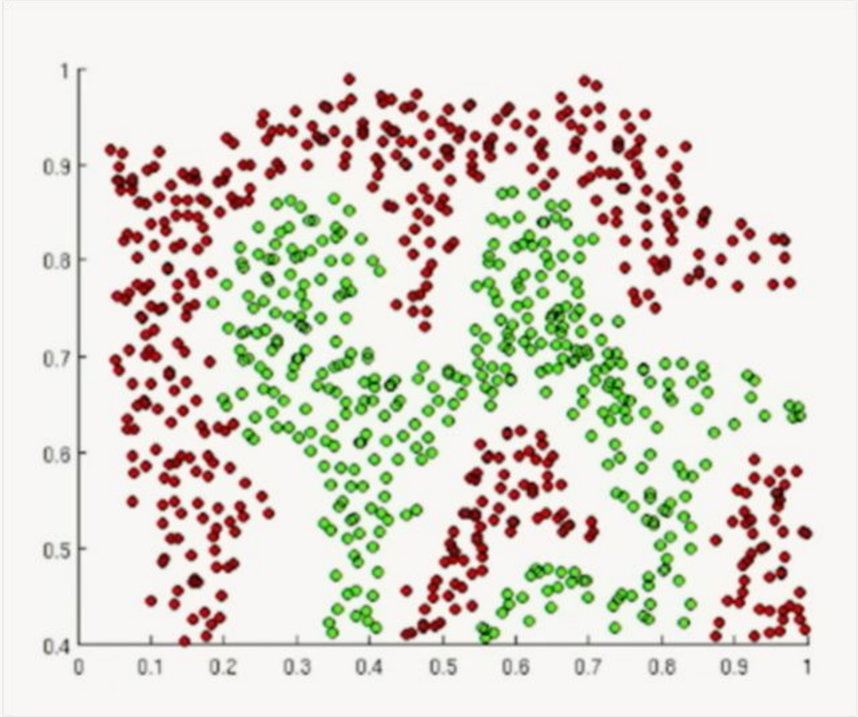
$$\varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} = \max(0, x)$$

$$\varphi'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

NOTE: Allactivation functions are non-linear

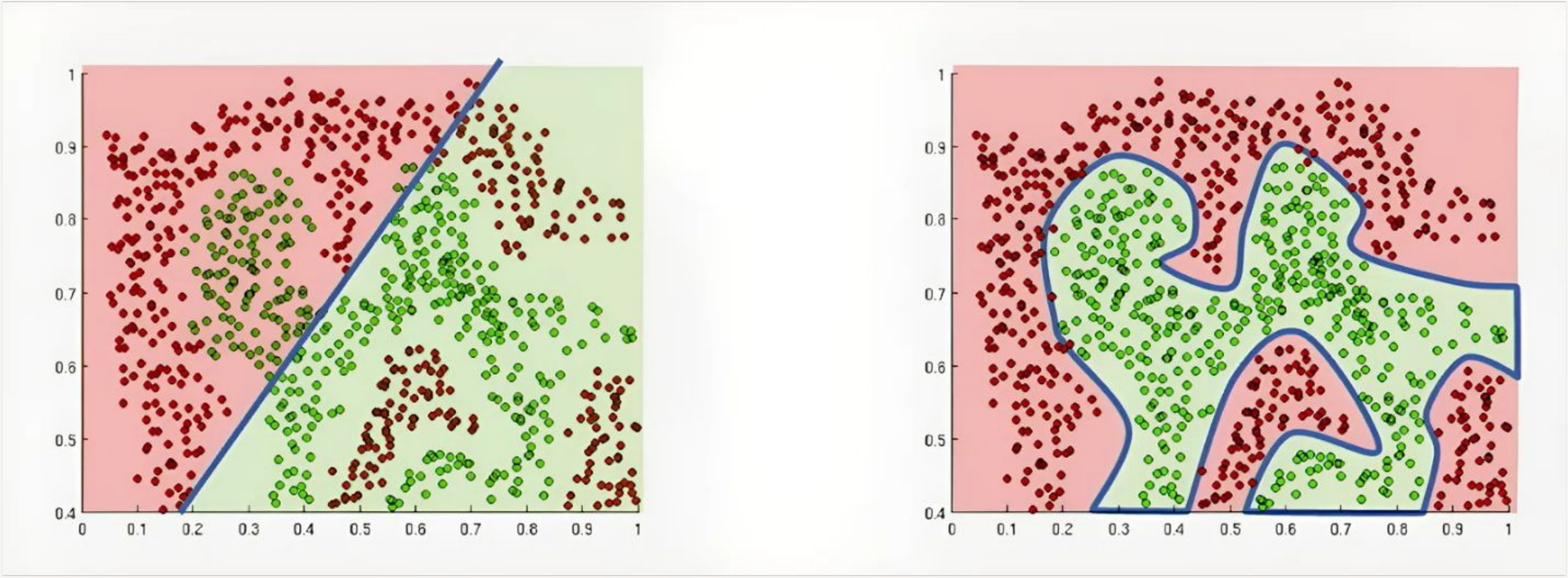
Importance of Activation Functions

The purpose of activation functions :
to introduce non-linearities into the network



What if we wanted to build a neural network to distinguish green vs red points?

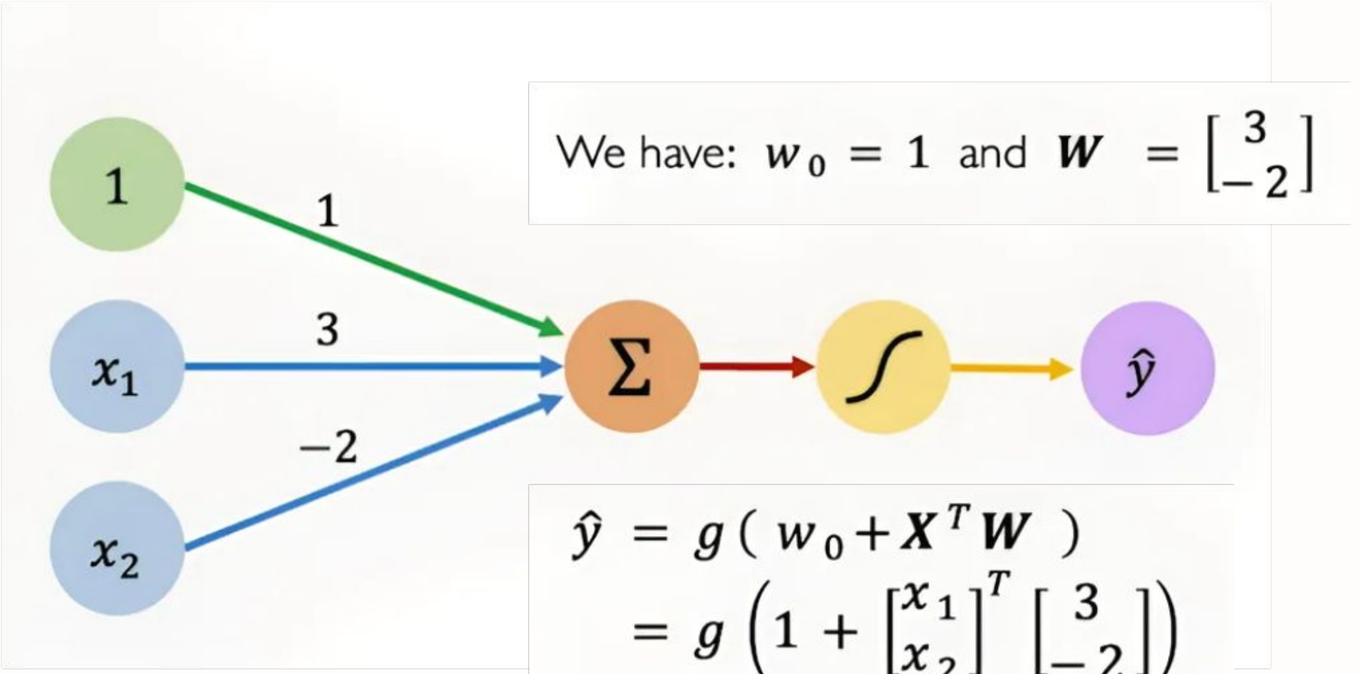
Common Activation Functions



Linear activation functions produce linear decisions no matter the network size

Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example



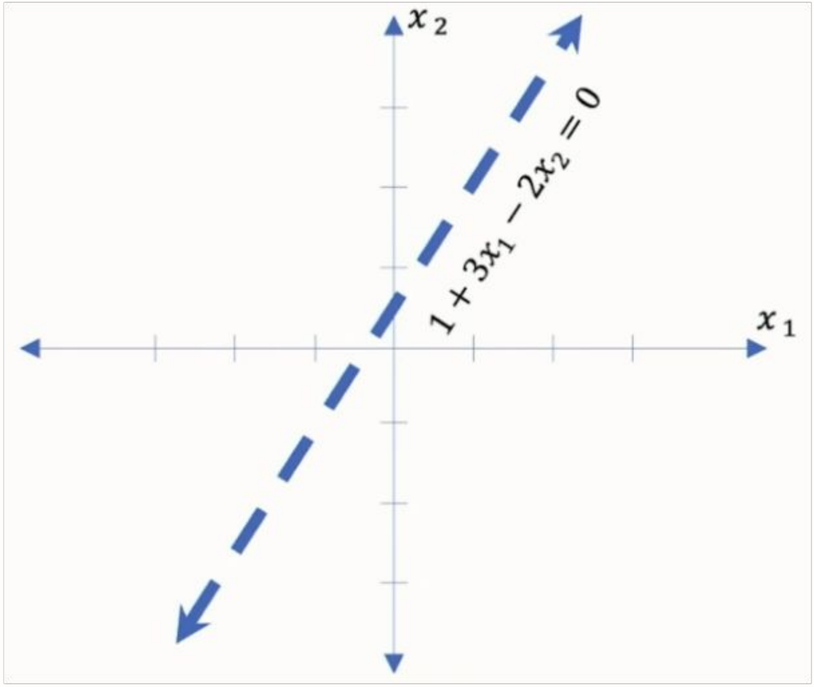
We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

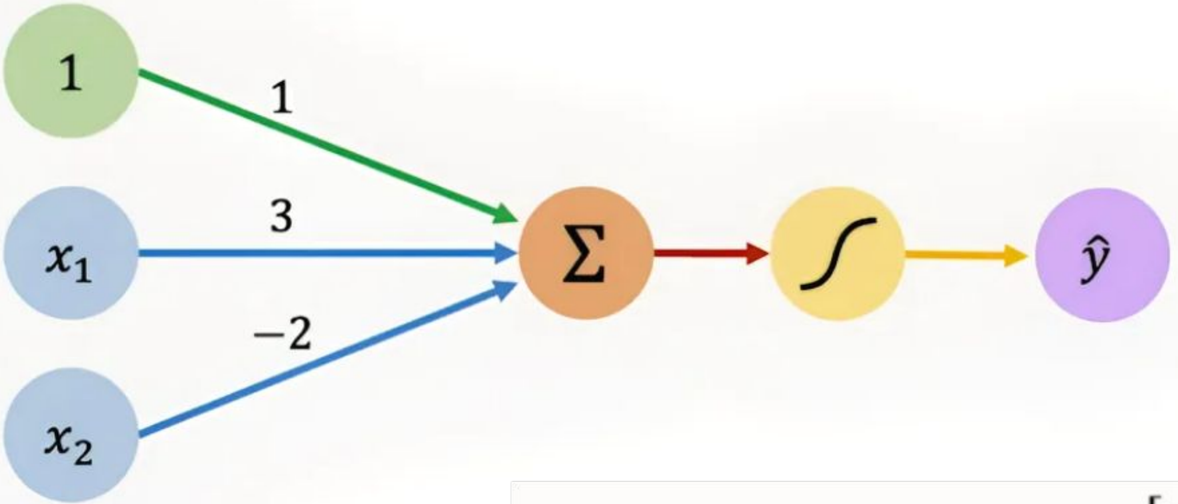
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

This is just a line in 2D!

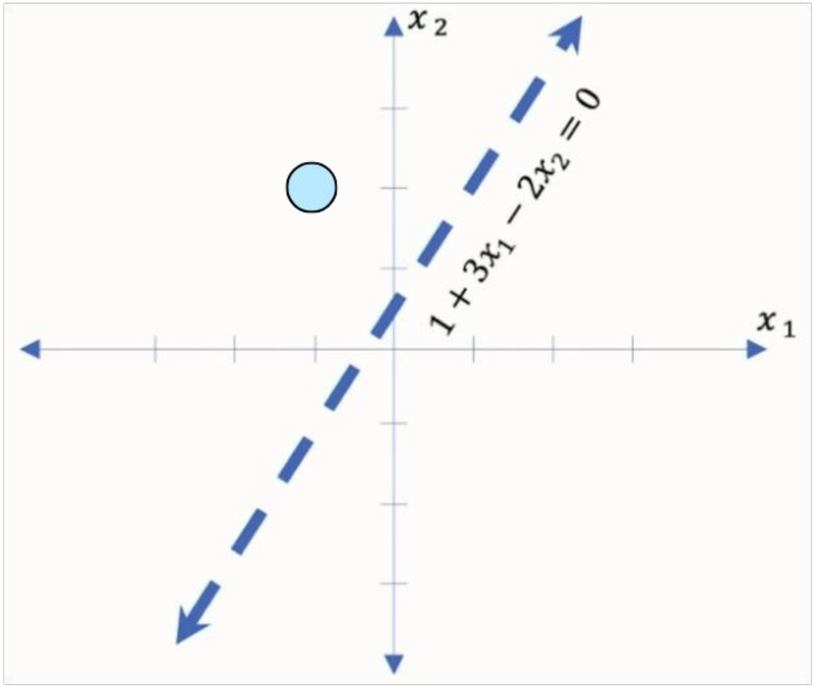


The Perceptron: Example

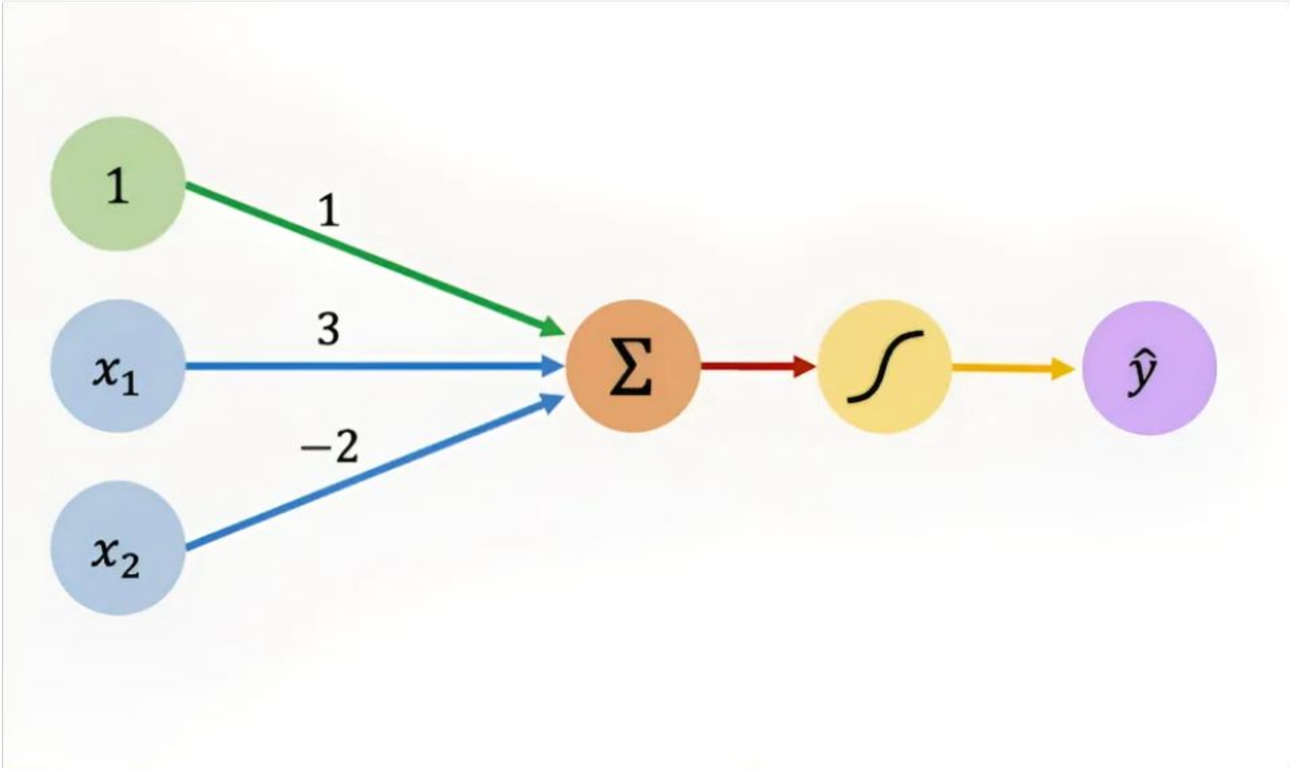


Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

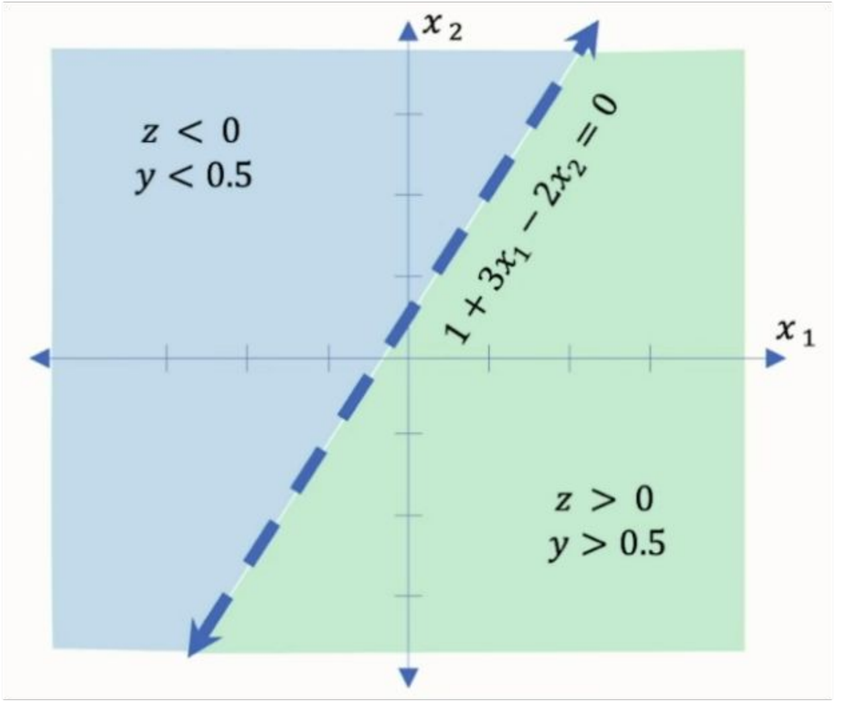
$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$



The Perceptron: Example

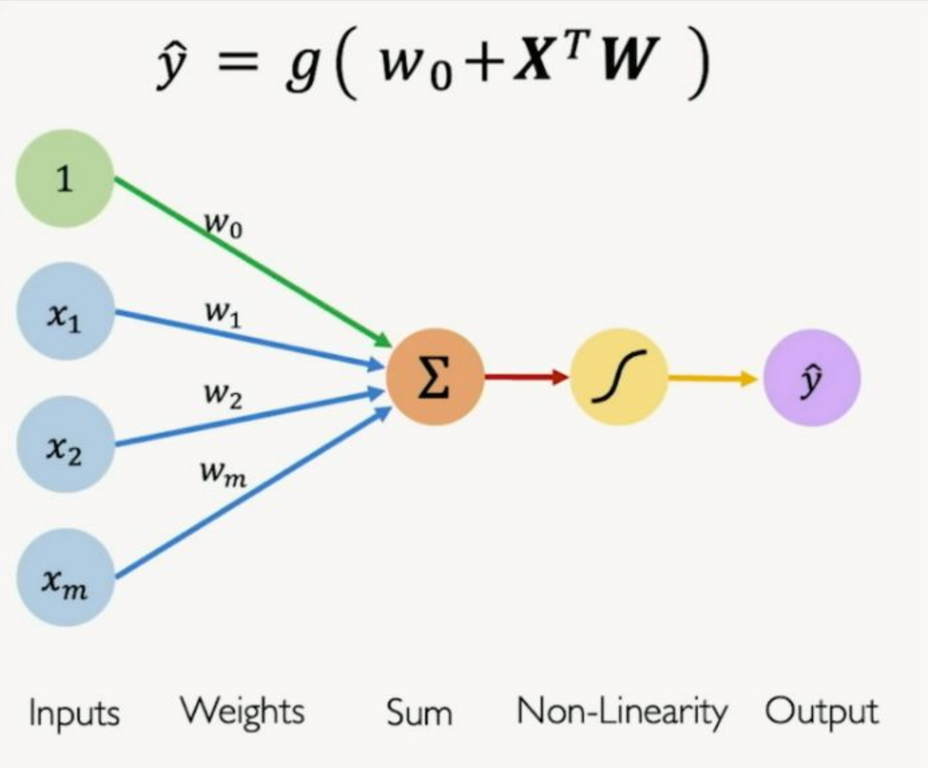


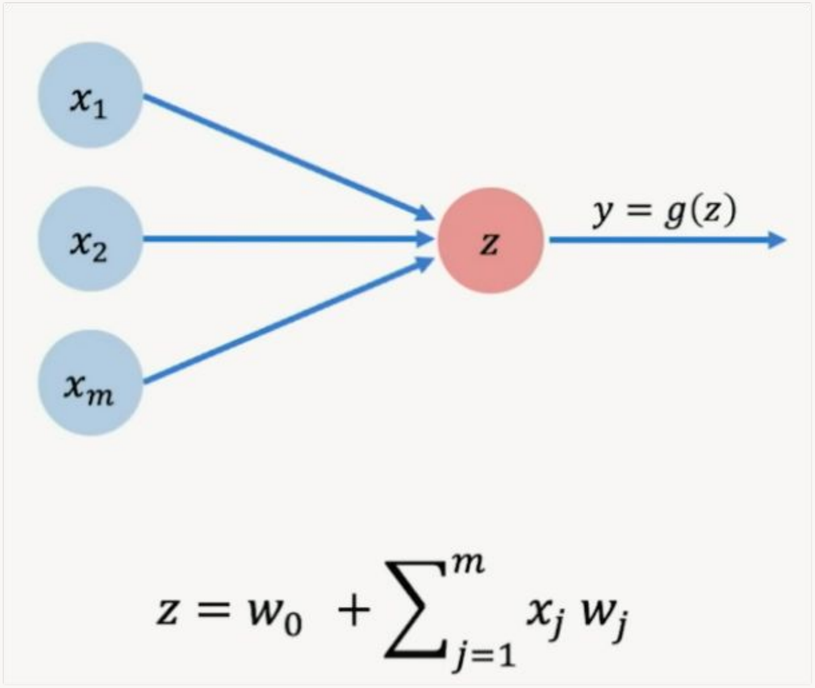
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



Building Neural Networks with Perceptrons

The Perceptron: Simplified

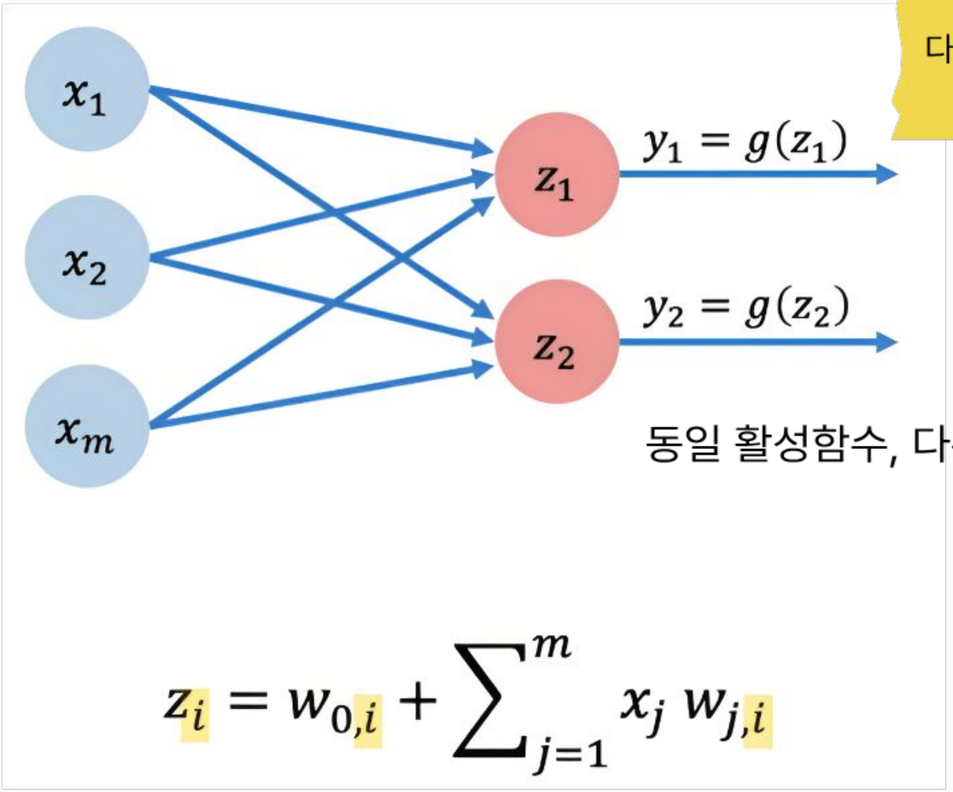




Multi Output Perceptron

모든 입력이 모든 출력에 밀접하게 연결되므로, Dense layer 라 불림

Layer: 퍼셉트론을 기능 단위로 묶은것.
다시 말해, 신경망에서 동일한 역할을 하는 뉴런들의 집합
(관례적으로) 동일한 활성 함수를 사용



동일 활성함수, 다른 가중치!

$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Dense layer from scratch

tensorflow

```
class MyDenseLayer(tf.keras.layers.Layer):
  def __init__(self, input_dim, output_dim):
    super(MyDenseLayer, self).__init__()

    # Initialize weights and bias
    self.W = self.add_weight([input_dim, output_dim])
    self.b = self.add_weight([1, output_dim])

  def call(self, inputs):
    # Forward propagate the inputs
    z = tf.matmul(inputs, self.W) + self.b

    # Feed through a non-linear activation
    output = tf.math.sigmoid(z)
    return output
```

Pytorch

```
class MyDenseLayer(nn.Module):
  def __init__(self, input_dim, output_dim):
    super(MyDenseLayer, self).__init__()

    # Initialize weights and bias
    self.W = nn.Parameter(torch.randn(input_dim,
        output_dim, requires_grad=True)
    self.b = nn.Parameter(torch.randn(1, output_dim,
        requires_grad=True)

  def forward(self, inputs):
    # Forward propagate the inputs
    z = torch.matmul(inputs, self.W) + self.b

    # Feed through a non-linear activation
    output = torch.sigmoid(z)
    return output
```

Dense layer from scratch

Tensorflow

```tensorflow
import tensorflow as tf

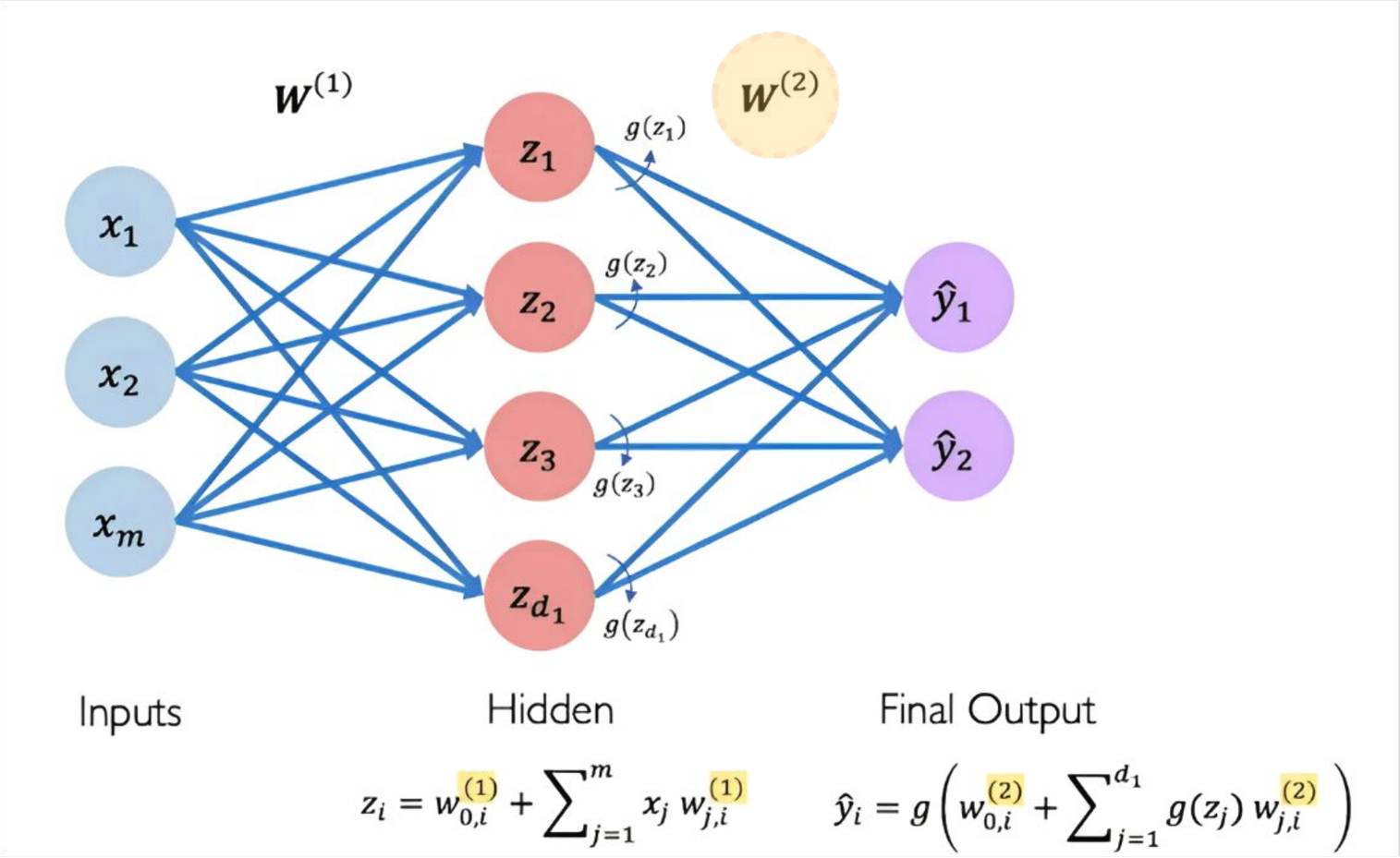
layer = tf.keras.layers.Dense(
    units=2, activation='relu')
```

Pytorch

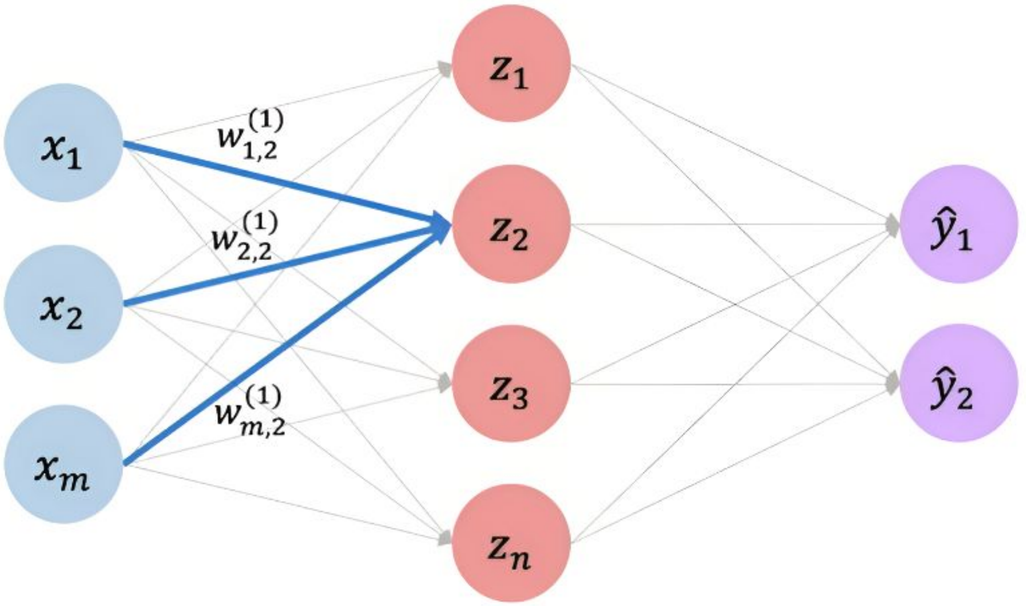
```pytorch
import torch.nn as nn

layer = nn.Sequential(
    nn.Linear(in_features=10,
              out_features=2),
    nn.ReLU()
)
```

Single (Hidden) Layer Neural Network



Single (Hidden) Layer Neural Network



$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,2}^{(1)}$$

$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

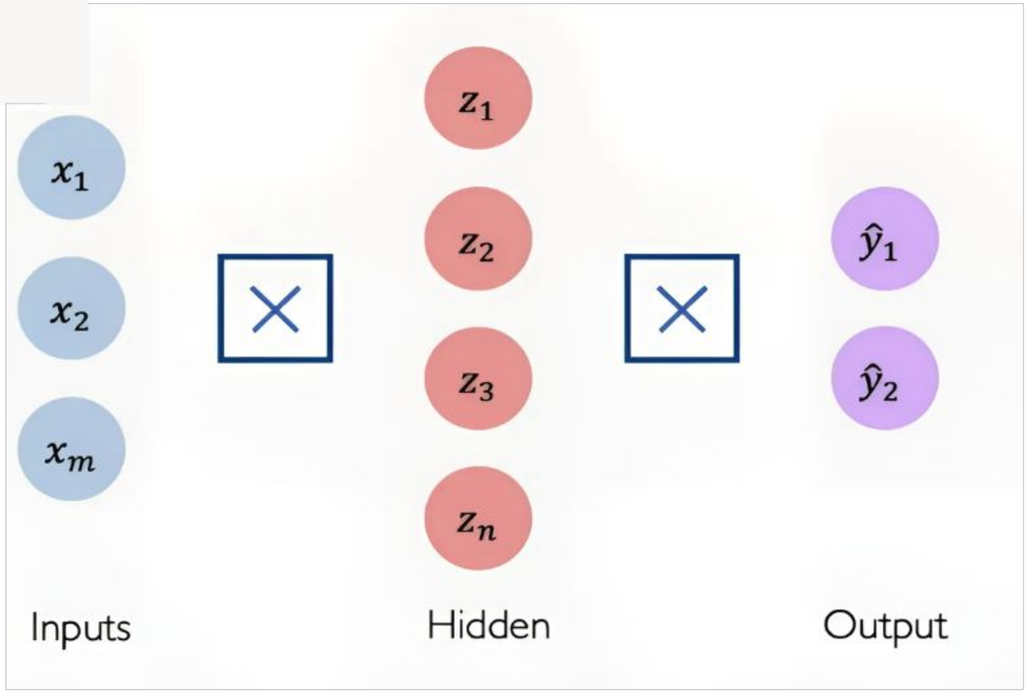
Multi Output Perceptron

```tensorflow
import tensorflow as tf

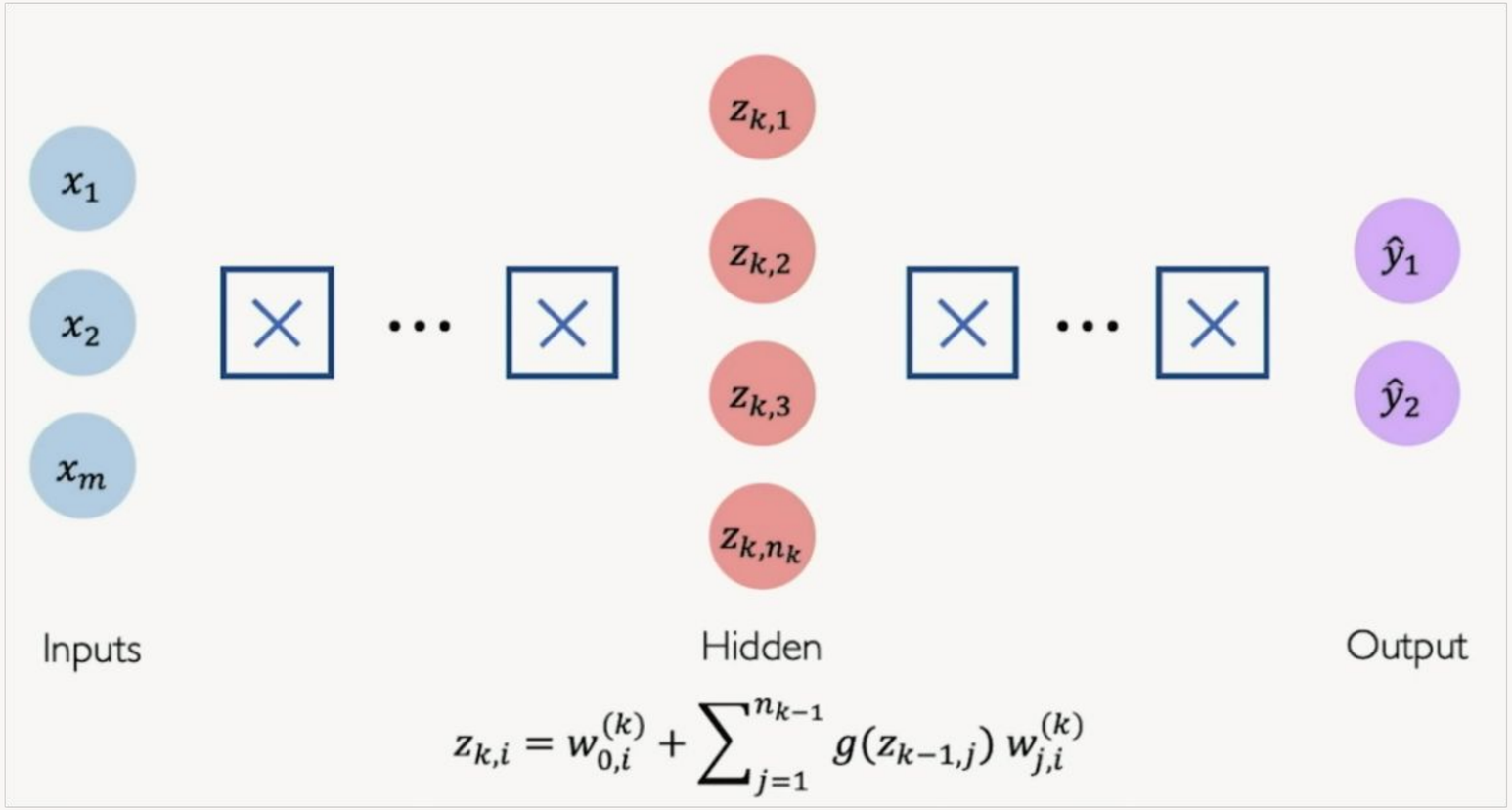
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

```pytorch
from torch import nn

model = nn.Sequential(
    nn.Linear(m, n),
    nn.ReLU(),
    nn.Linear(n, 2)
)
```



Deep Neural Network




```tensorflow
import tensorflow as tf

model = tf.keras.Sequential([
  tf.keras.layers.Dense(n_1),
  tf.keras.layers.Dense(n_2),
  ⋮
  tf.keras.layers.Dense(2)
])
```

```pytorch
from torch import nn

model = nn.Sequential(
  nn.Linear(m, n1),
  nn.ReLU(),
⋮
  nn.ReLU(),
  nn.Linear(nK, 2)
)
```

Example Problem

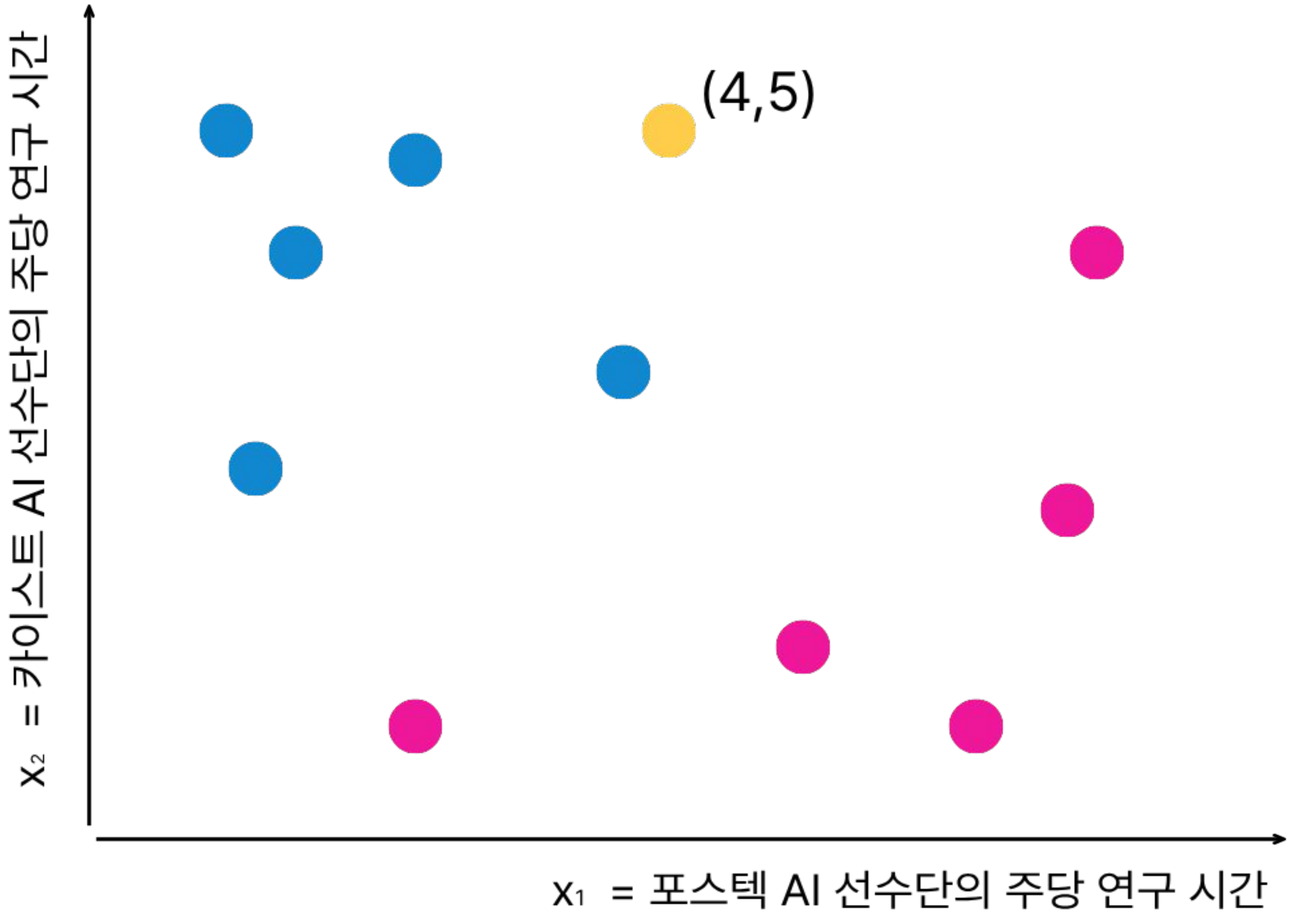
포카전 AI종목에서 올해도 포스텍이 승리할까?

를 간단한 two feature 모델로 설계

x_1 = 포스텍 AI 선수단의 주당 연구 시간
x_2 = 카이스트 AI 선수단의 주당 연구 시간

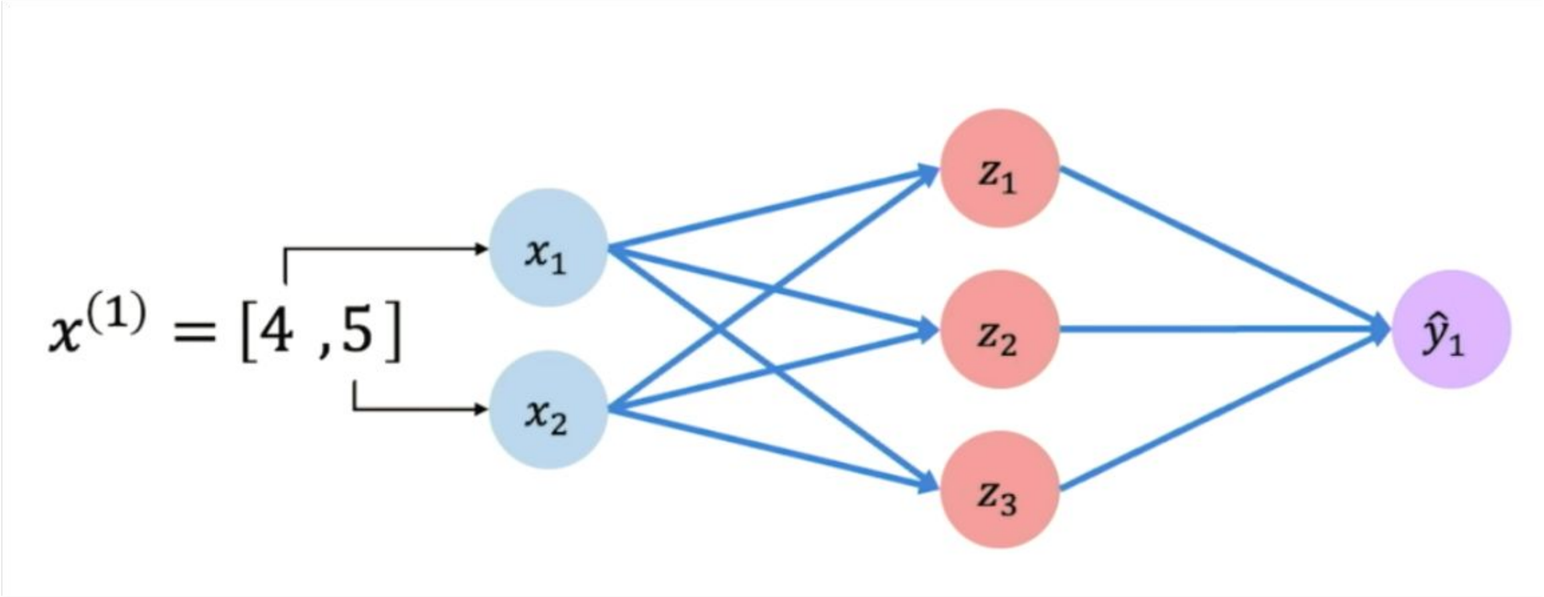
Example Problem

포카전 AI종목에서 올해도 포스텍이 승리할까?



Example Problem

포카전 AI종목에서 올해도 포스텍이 승리할까?



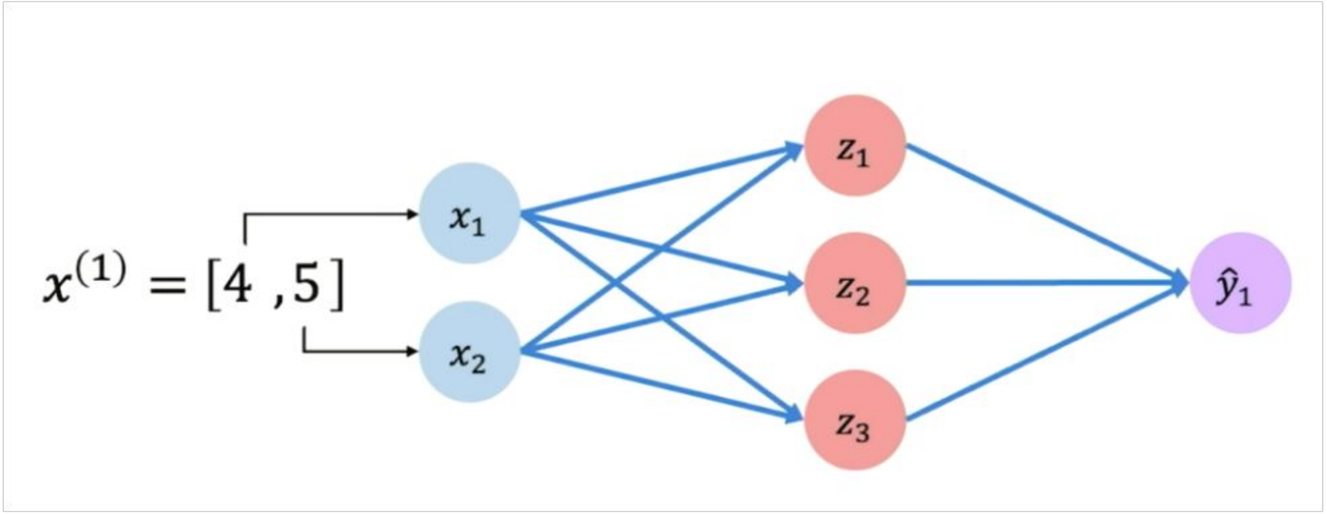
Predicted: 0.1

Actual: 1

모델의 성능을 어떻게 나타낼 것인가?
-> 예측과 실제 답 사이의 간극= Loss

Quantifying Loss

Loss를 통해 잘못된 예측으로 인해 발생하는 비용을 측정할 수 있다.



Predicted: 0.1

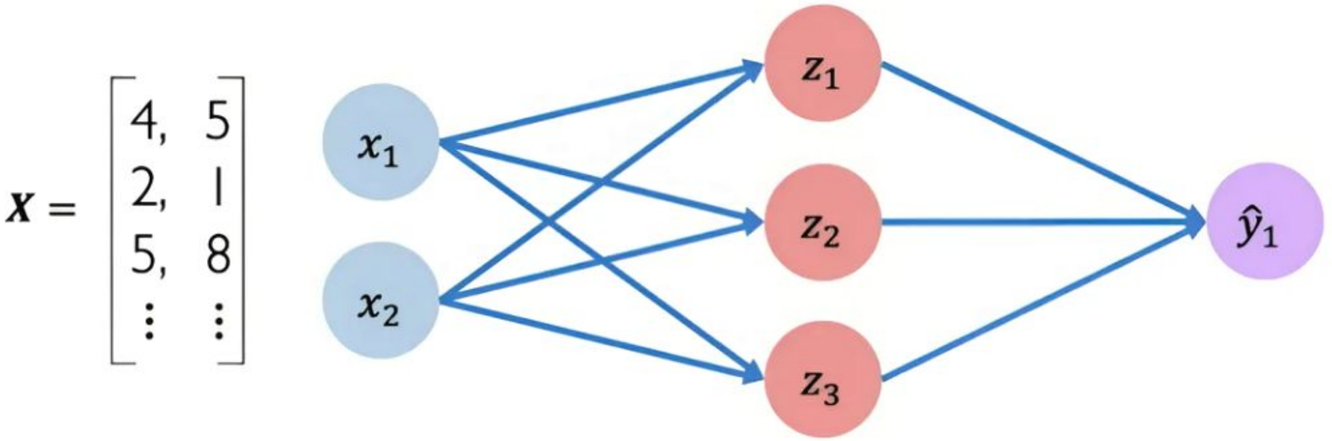
Actual: 1

$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

전체 dataset의 Loss를 평균내어 Empirical Loss를 구한다.



$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$

$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \begin{matrix} \times \\ \times \\ \checkmark \\ \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

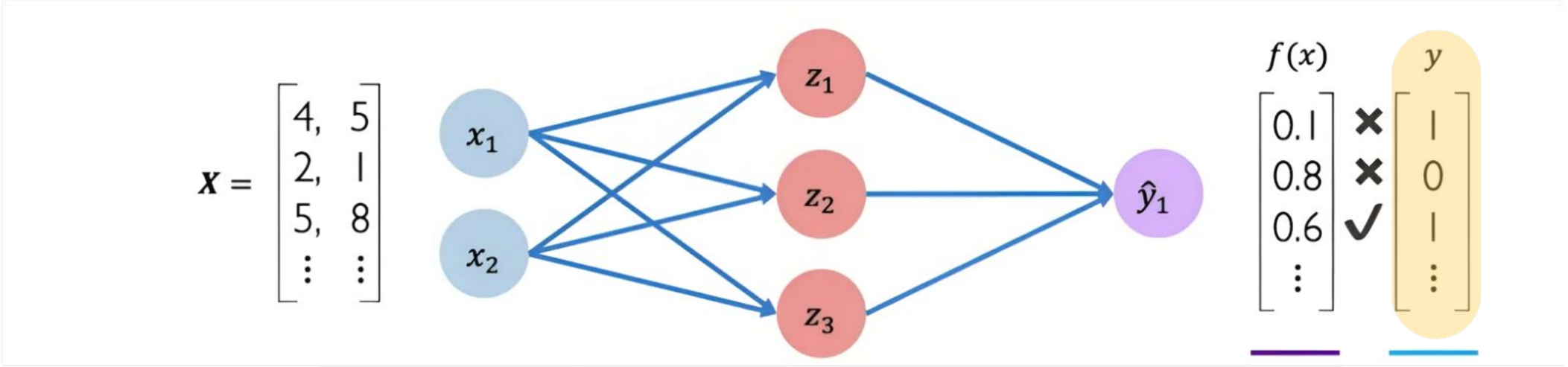
Also known as:
- Objective function
- Cost function
- Empirical Risk

$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Binary Cross Entropy Loss

이진 분류 문제에서 자주 쓰이는 Loss 계산 방식으로, log를 사용함.



$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f\left(x^{(i)}; W\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - f\left(x^{(i)}; W\right)\right)$$

Actual Predicted Actual Predicted

Tensorflow

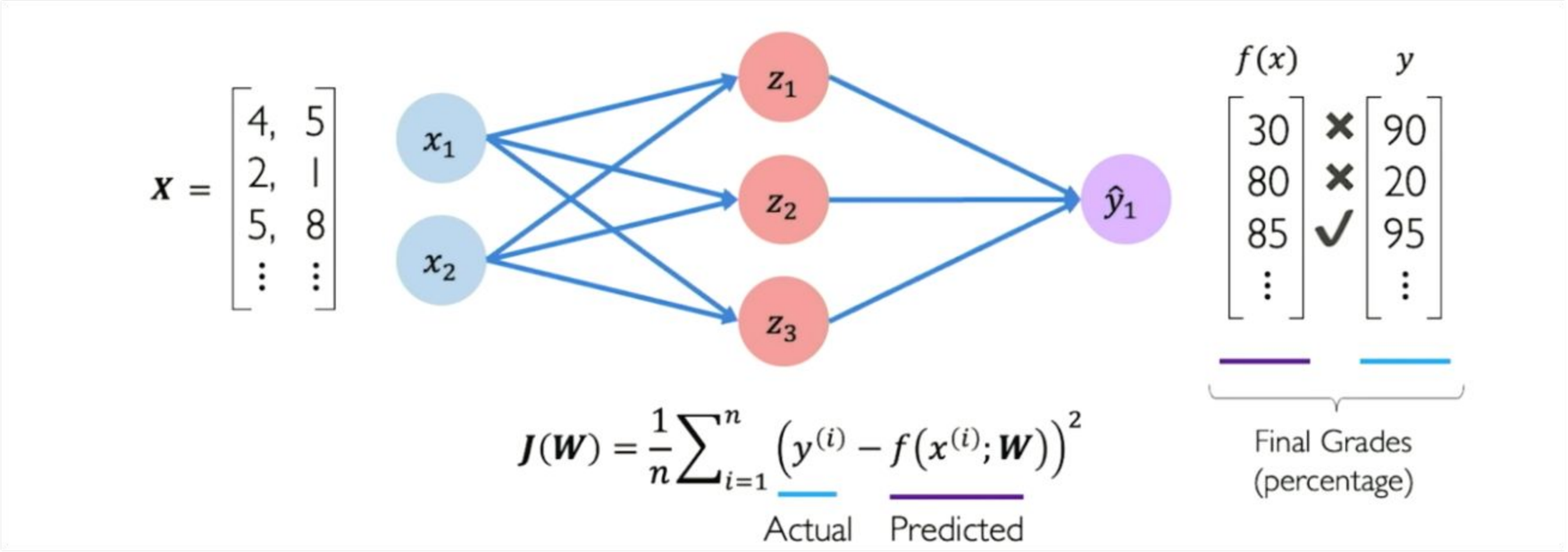
```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

Pytorch

```
loss = torch.nn.functional.cross_entropy( predicted, y )
```

Mean Squared Error Loss

출력이 연속적인 실수인 경우에 자주 쓰이는 Loss 계산 방식으로, 제곱수를 사용함.



Tensorflow

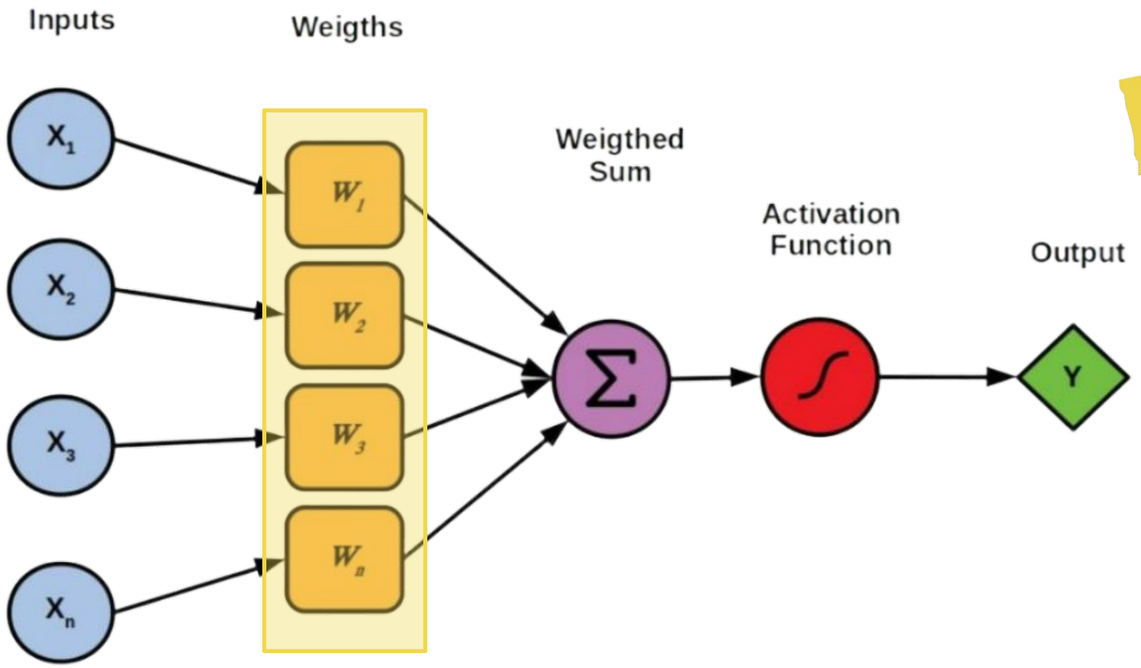
```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted))
loss = tf.keras.losses.MSE( y, predicted )
```

Pytorch

```
loss = torch.nn.functional.mse_loss(
        predicted, y )
```

일반적으로, 오차에 대한 손실은 선형으로 증가하지 않는다.
(log, 제곱수 등으로 이를 반영)

Loss Optimization



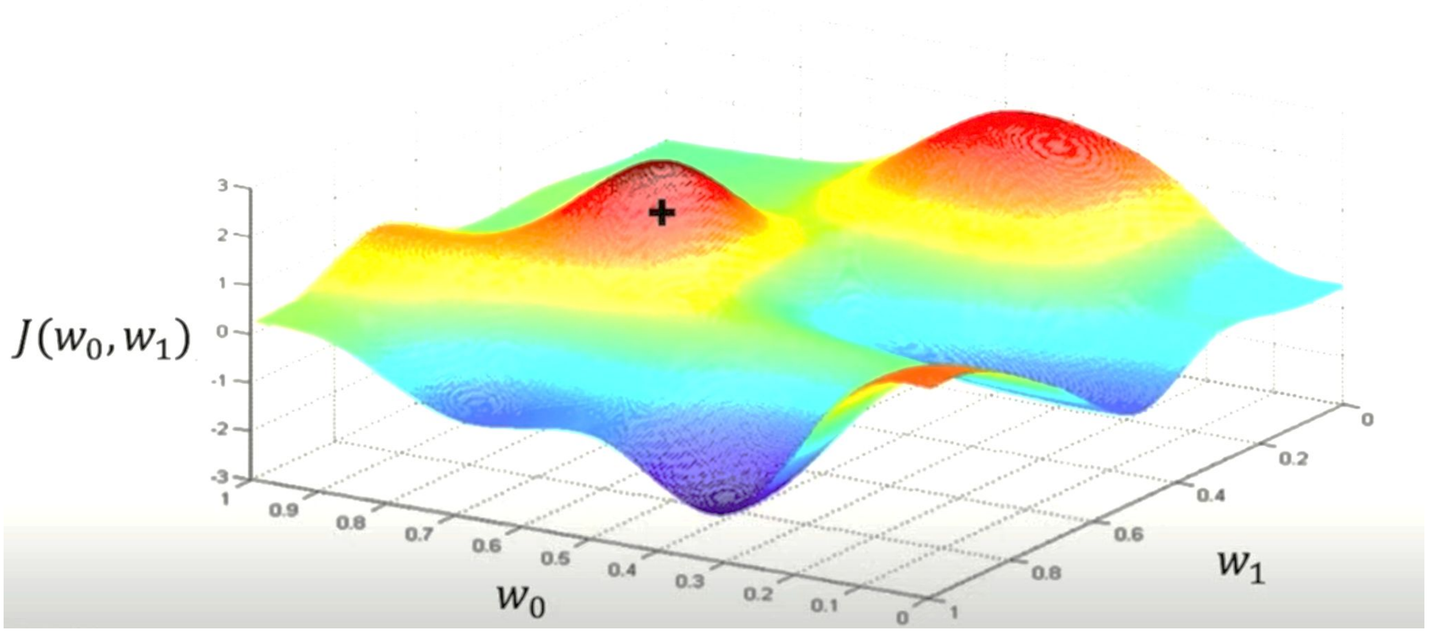
GOA

- loss를 최소화하는 weight를 찾는 것

$$W^* = argmin \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

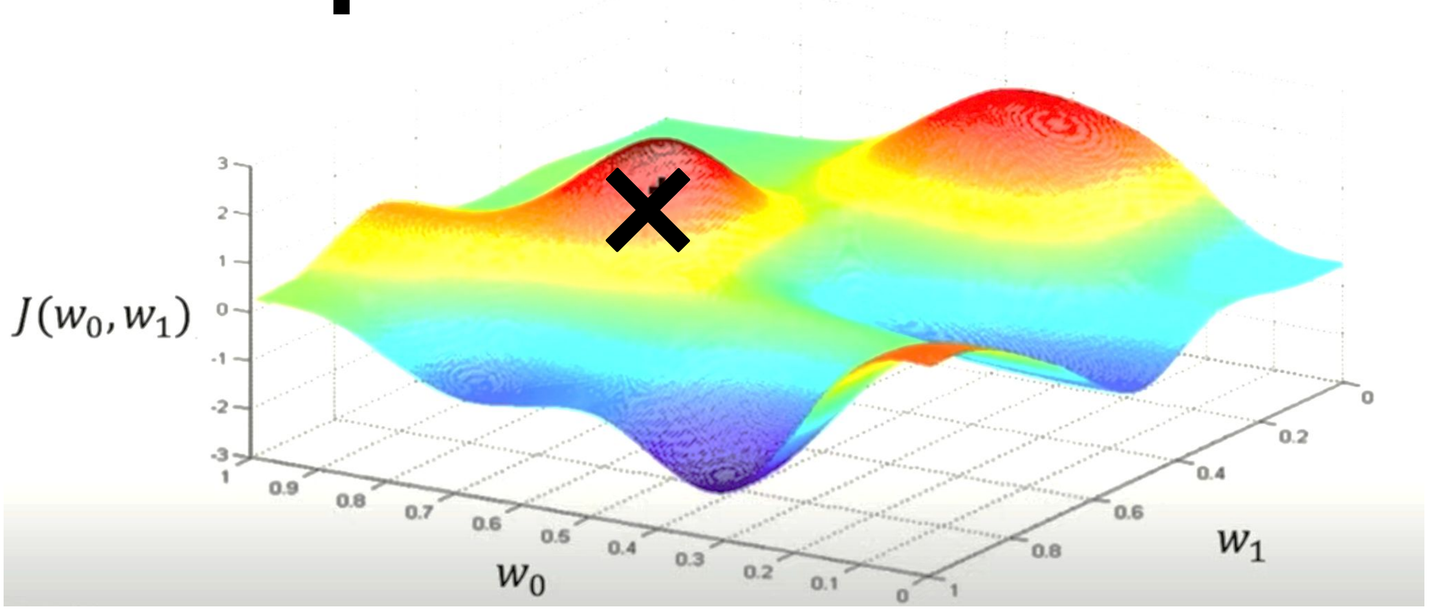
$$W^* = \underset{W}{argmin}\, J(W)$$

Loss Optimization



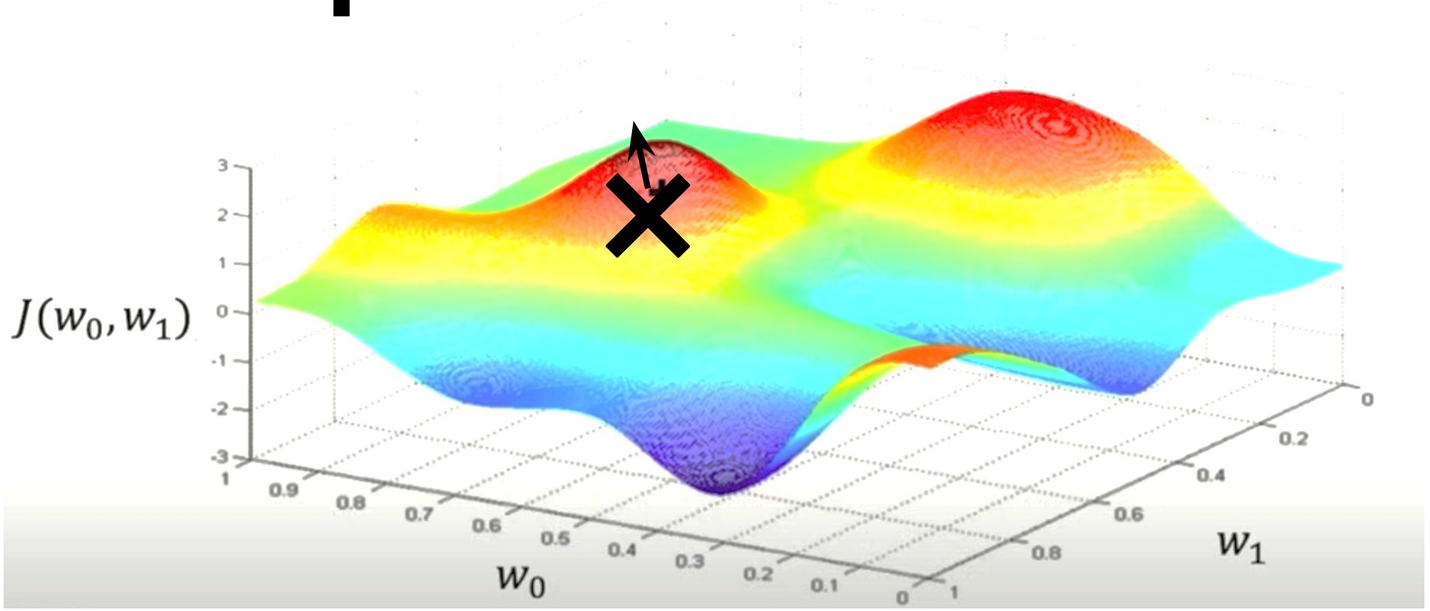
Loss function은 network weight에 대한 함수이다!

Loss Optimization



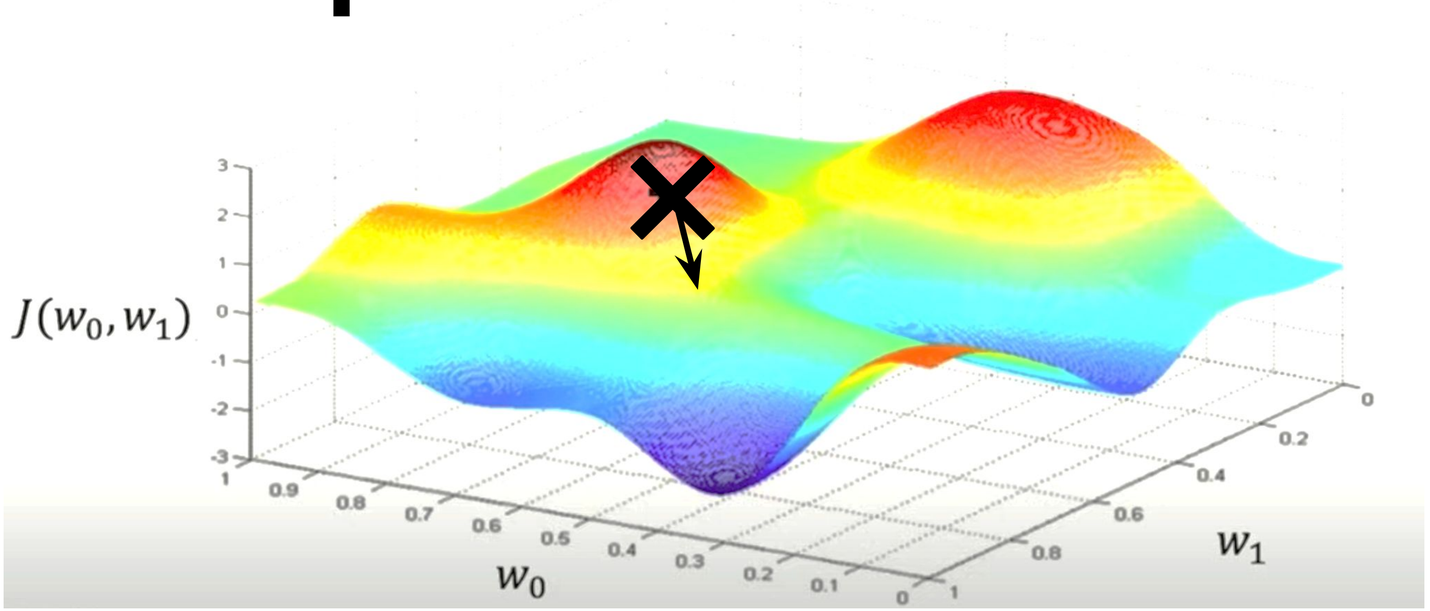
랜덤하게 **initial point** 정하기

Loss Optimization



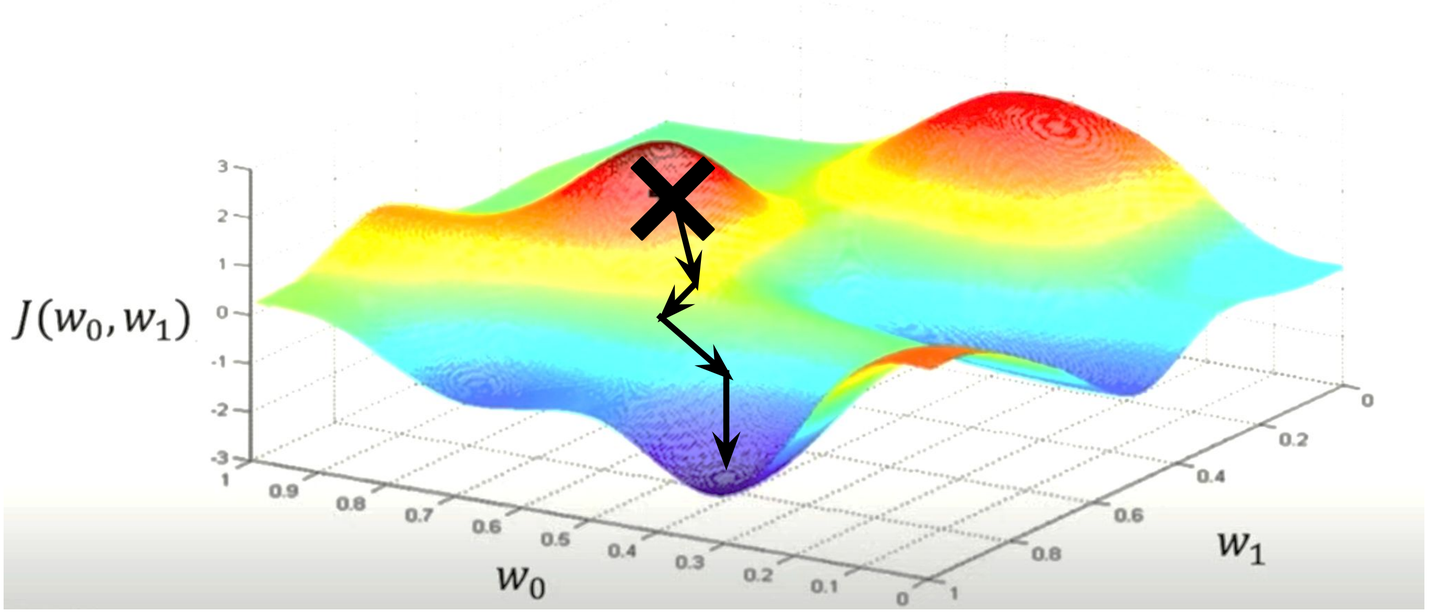
gradient $\frac{\partial J(W)}{\partial W}$ 계산하기

Loss Optimization



gradient 반대 방향으로 small step 이동하기

Gradient Descent



수렴할 때까지 반복

Gradient Descent

Algorithm

기본적으로 표준 정규 분포에서 뽑음

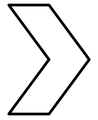
Initialize network weight

While (converge):

compute gradient, $\frac{\partial J(W)}{\partial W}$

update weight, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

return weights





```python
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

Gradient Descent

Algorithm

기본적으로 표준 정규 분포에서 뽑음

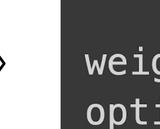
Initialize network weight

While (converge):

compute gradient, $\frac{\partial J(W)}{\partial W}$

update weight, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

return weights

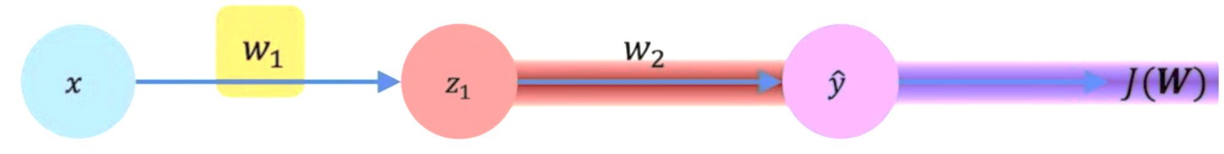


```python
import torch

weights = torch.randn(1, requires_grad=True)
optimizer = torch.optim.SGD([weights], lr=0.01)

while True:
    optimizer.zero_grad()
    loss = compute_loss(weights)
    loss.backward()
    optimizer.step()
```

Computing Gradients: Backpropagation



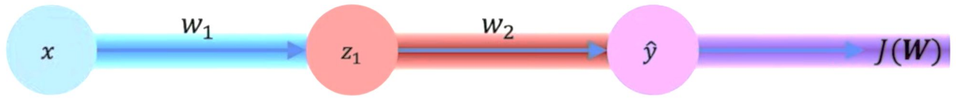
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

keyword: chain rule

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

```python
class Add(Op):
    @staticmethod
    def forward(ctx, a, b):
        ctx.save_for_backward(a.shape, b.shape)
        return a + b

    @staticmethod
    def backward(ctx, grad_output):
        a_shape, b_shape = ctx.saved_tensors
        grad_a = reduce_grad_to_shape(grad_output, a_shape)
        grad_b = reduce_grad_to_shape(grad_output, b_shape)
        return grad_a, grad_b
```

Forward propagation에서 back propagation을 위해 값들을 저장해놓음

Training Neural Networks is Difficult

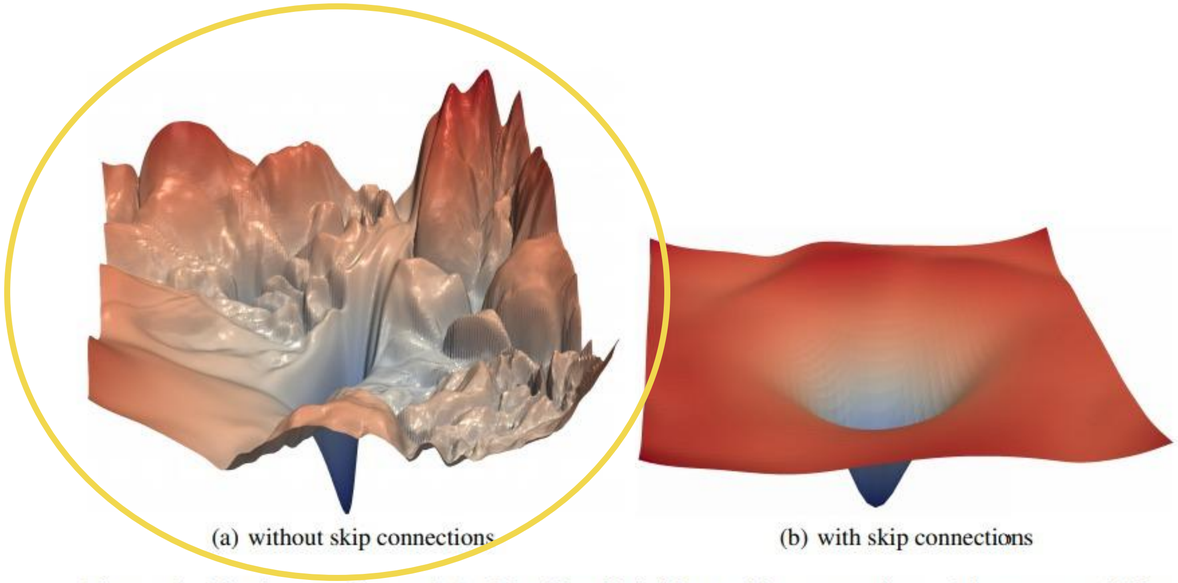


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

이런 평면에서는 Gradient Descent를 적용하는데 어려움이 있음

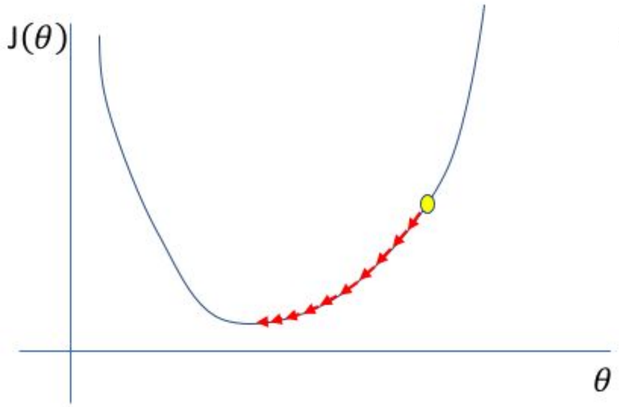
Setting the Learning Rate

Learning rate를 어떻게 설정해야 할까?

$$\mathbf{W} \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

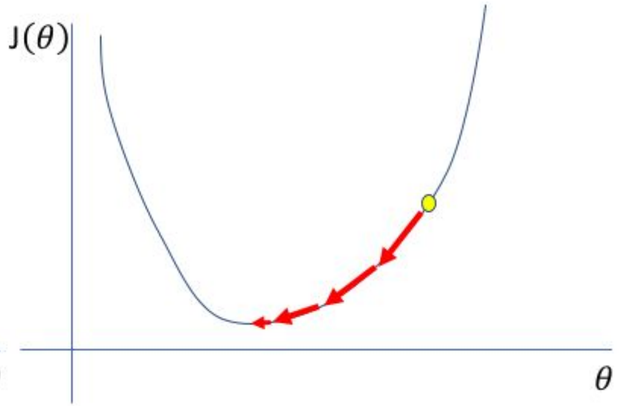
Setting the Learning Rate

Too low



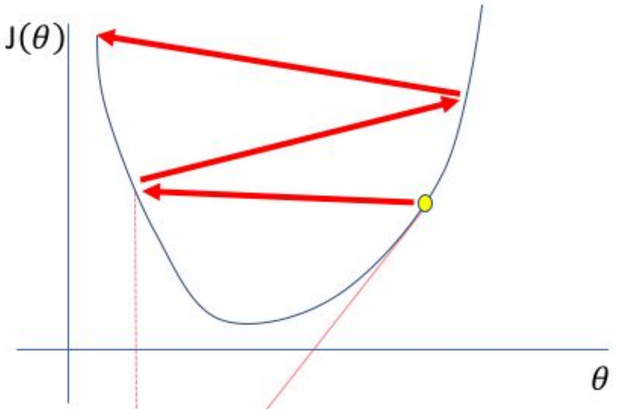
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

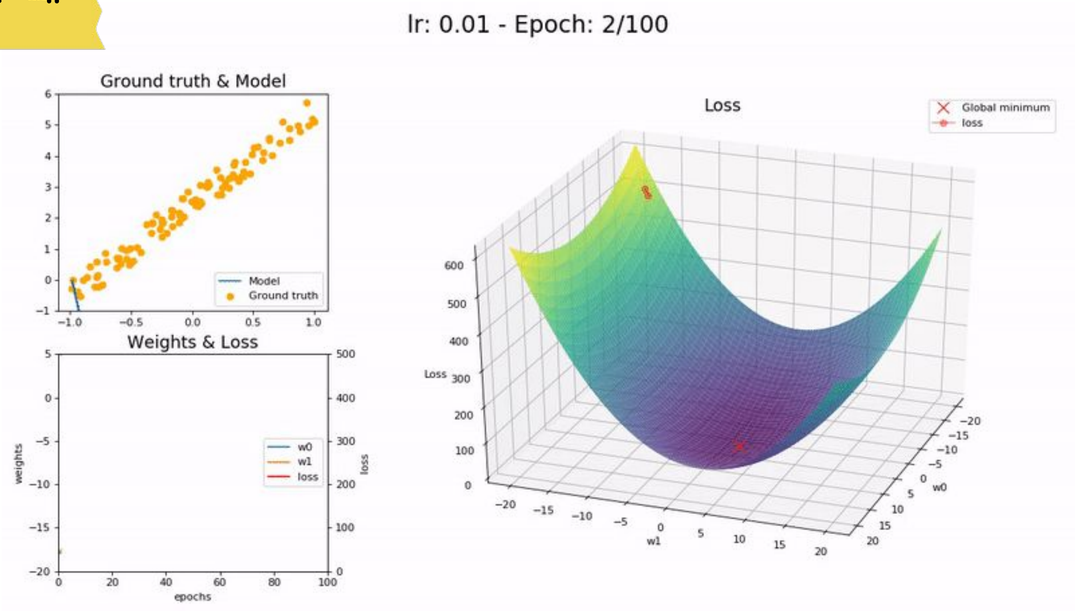
Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

Setting the Learning Rate

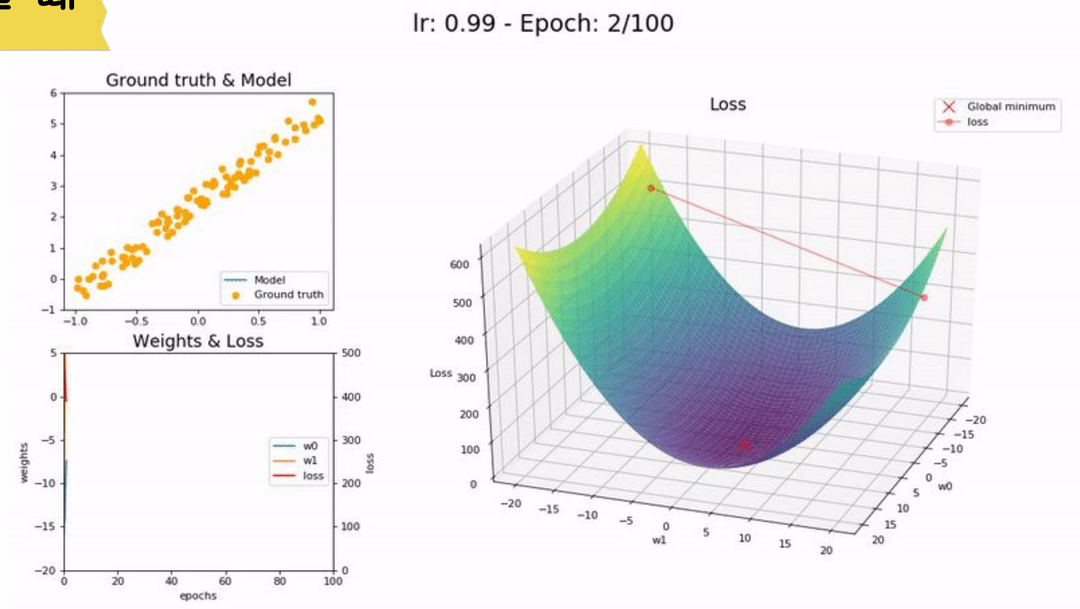
Learning rate가 너무 작을 때



1. 느리게 수렴
2. 목표한 **epoch** 안에 **local minima** 도달 실패
3. 만약 **false local minima**가 있을 때, **stuck**될 수 있음

Setting the Learning Rate

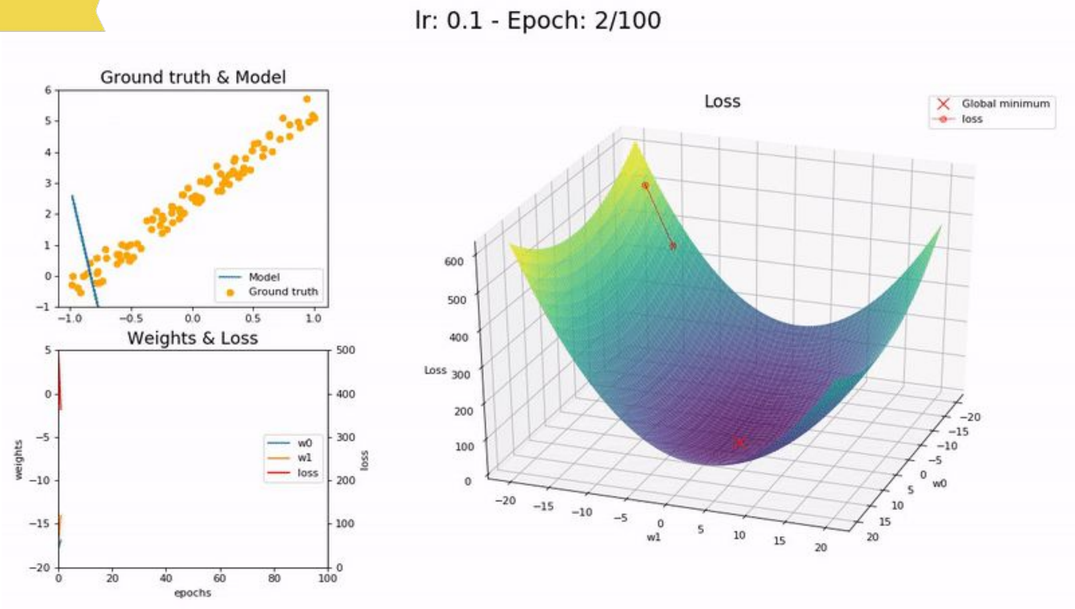
Learning rate가 너무 클 때



1. overshooting
2. Unstable
3. diverge

Setting the Learning Rate

Learning rate가 적절할 때



How to deal with this?

Option 1.

Learning rate를 바꿔가며 최적의 값 찾기

Option 2.

Adaptive learning rate

Adaptive Learning Rates

- How large gradient is $\propto \frac{1}{size\ of\ gradient}$
- How fast learning is happening $\propto \frac{1}{speed\ of\ learning}$
- Size of particular weights $\propto \frac{1}{size\ of\ particular\ weights}$

Gradient Descent Algorithms

Stochastic Gradient Descent

Algorithm	TF Implementation	Torch Implementation	Reference
• SGD	tf.keras.optimizers.SGD	torch.optim.SGD	Kiefer & Wolfowitz, 1952.
• Adam	tf.keras.optimizers.Adam	torch.optim.Adam	Kingma et al., 2014.
• Adadelta	tf.keras.optimizers.Adadelta	torch.optim.Adadelta	Zeiler et al., 2012.
• Adagrad	tf.keras.optimizers.Adagrad	torch.optim.Adagrad	Duchi et al., 2011.
• RMSProp	tf.keras.optimizers.RMSProp	torch.optim.RMSProp	

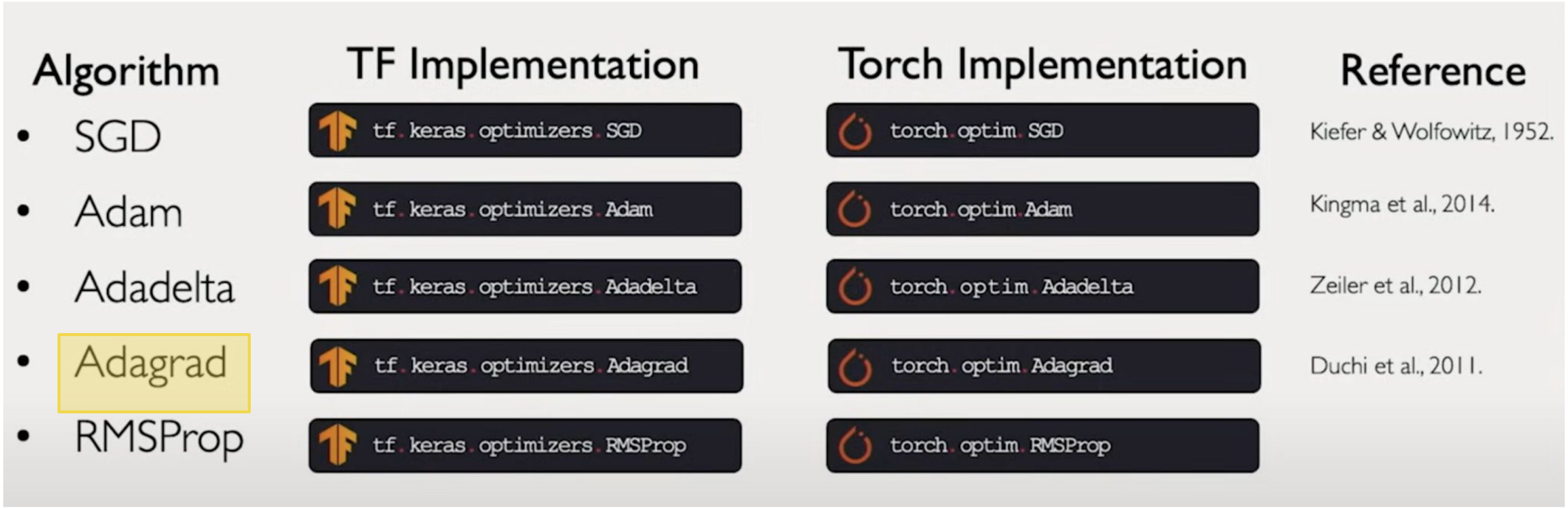
가장 기본적인 알고리즘
Learning rate 고정 & momentum 같은 옵션

Gradient Descent Algorithms

Algorithm	TF Implementation	Torch Implementation	Reference
• SGD	tf.keras.optimizers.SGD	torch.optim.SGD	Kiefer & Wolfowitz, 1952.
• Adam	tf.keras.optimizers.Adam	torch.optim.Adam	Kingma et al., 2014.
• Adadelta	tf.keras.optimizers.Adadelta	torch.optim.Adadelta	Zeiler et al., 2012.
• Adagrad	tf.keras.optimizers.Adagrad	torch.optim.Adagrad	Duchi et al., 2011.
• RMSProp	tf.keras.optimizers.RMSProp	torch.optim.RMSProp	

가장 널리 쓰이는 adaptive learning rate optimizer
빠르고 안정적

Gradient Descent Algorithms



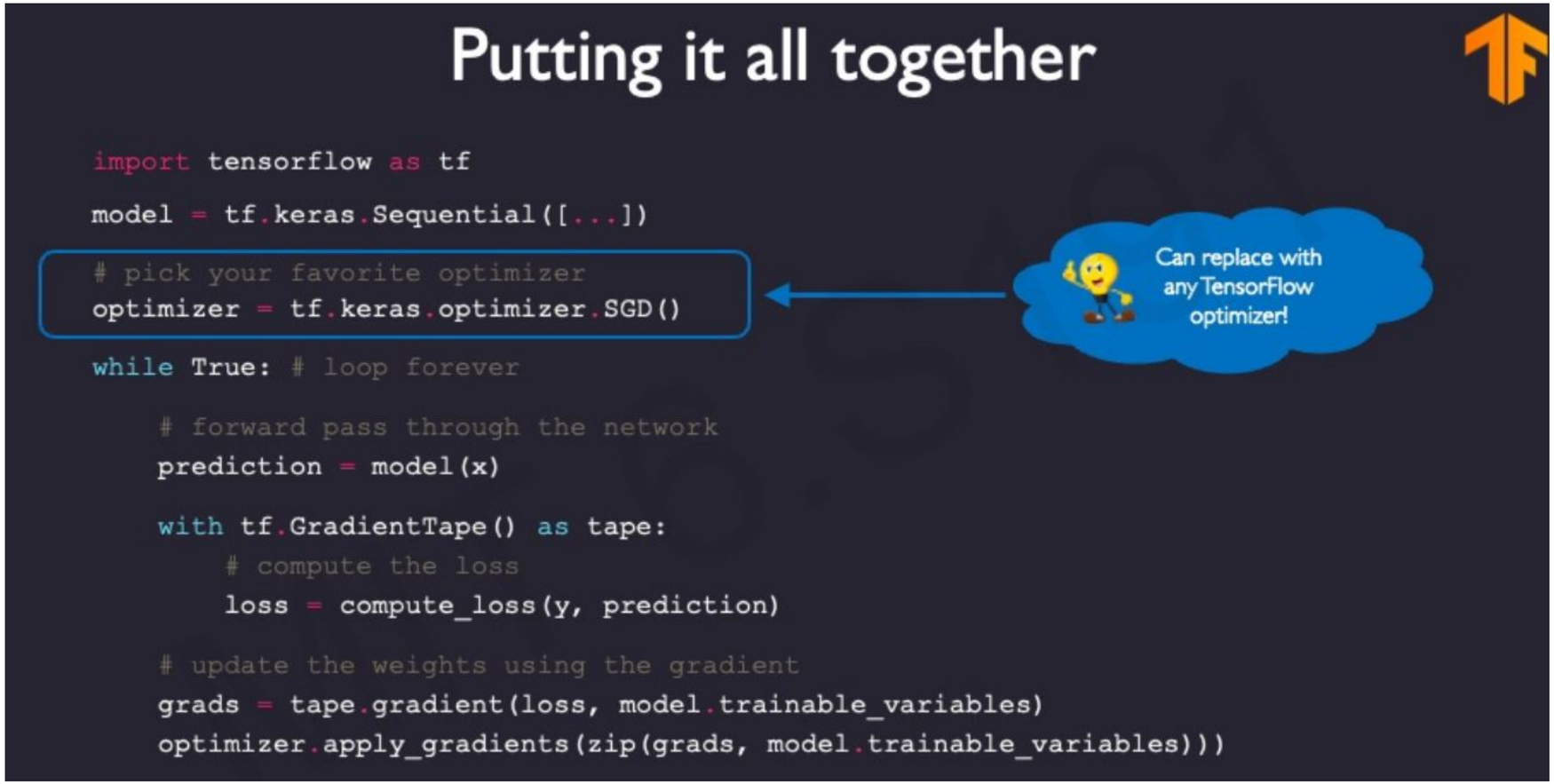
Gradient의 제곱 누적합을 이용해서 파라미터마다 다른 학습률 적용.
자주 업데이트되는 파라미터는 학습률 작게, 드물게 업데이트되는 파라미터는 크게

Gradient Descent Algorithms

Algorithm	TF Implementation	Torch Implementation	Reference
• SGD	tf.keras.optimizers.SGD	torch.optim.SGD	Kiefer & Wolfowitz, 1952.
• Adam	tf.keras.optimizers.Adam	torch.optim.Adam	Kingma et al., 2014.
• Adadelta	tf.keras.optimizers.Adadelta	torch.optim.Adadelta	Zeiler et al., 2012.
• Adagrad	tf.keras.optimizers.Adagrad	torch.optim.Adagrad	Duchi et al., 2011.
• RMSProp	tf.keras.optimizers.RMSProp	torch.optim.RMSProp	

Adagrad의 learning rate이 너무 빨리 줄어드는 문제를 보완

Putting it all together



Differences between GD and SGD



Gradient Descent

Initialize network weight

While (converge):

compute gradient, $\frac{\partial J(W)}{\partial W}$

update weight, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

return weights

Computationally intensive

Stochastic Gradient Descent

Initialize network weight

While (converge):

pick single data point i

compute gradient, $\frac{\partial J_i(W)}{\partial W}$

update weight, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

return weights

Easy to compute But very noisy

Mini-batches

SGD

Initialize network weight

While (converge):

pick batch of B data points

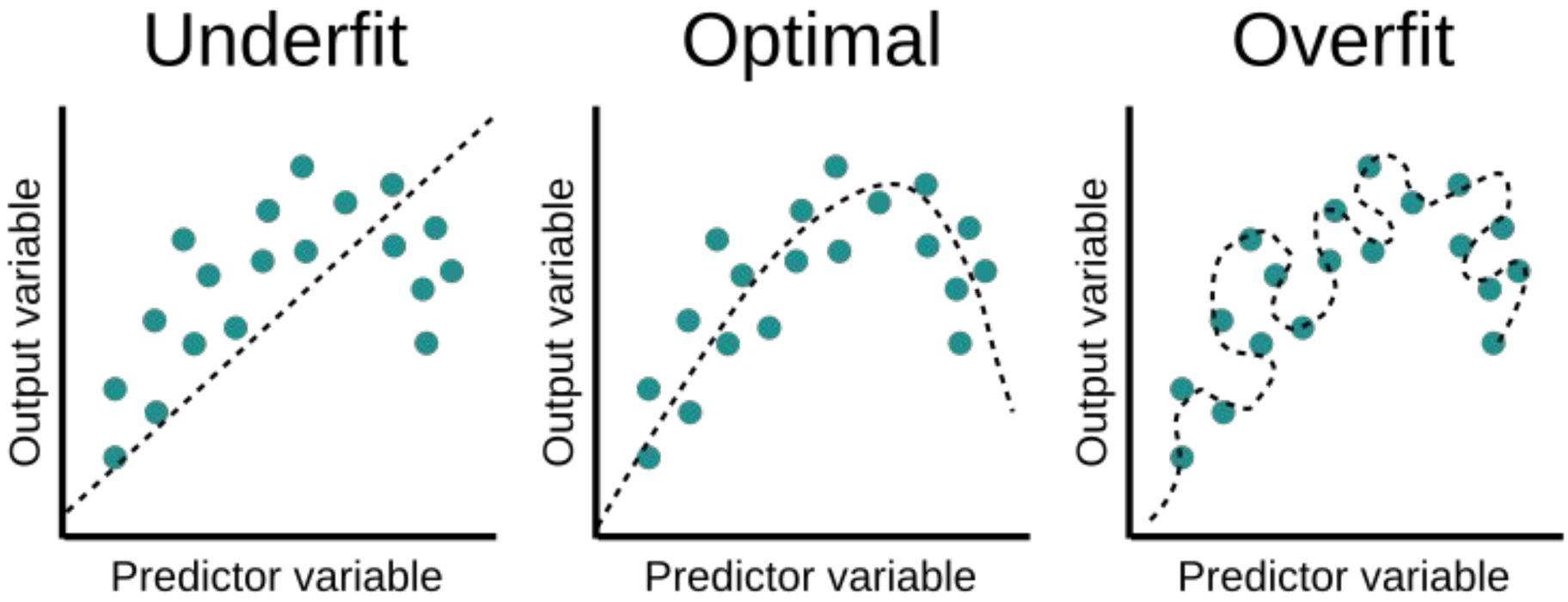
compute gradient, $\frac{\partial J(W)}{\partial W} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(W)}{\partial W}$

update weight, $W \leftarrow W - \eta\frac{\partial J(W)}{\partial W}$

return weights

Gradient Descent 보단 빠르고, Stochastic Gradient Descent 보다는 덜 noisy

The Problem of Overfitting



Regularization

- 가중치에 제약을 가해 **complex model**이 되는 것을 **discourage**
- **improve generality!**

Option 1.

dropout

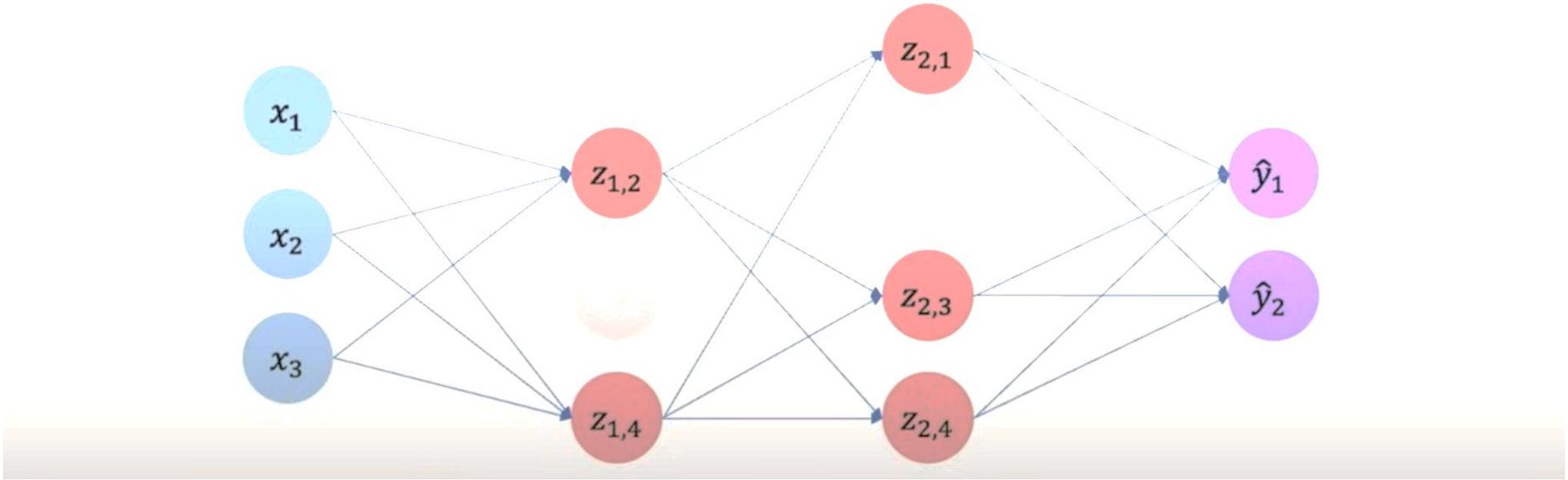
Option 2.

Early stopping

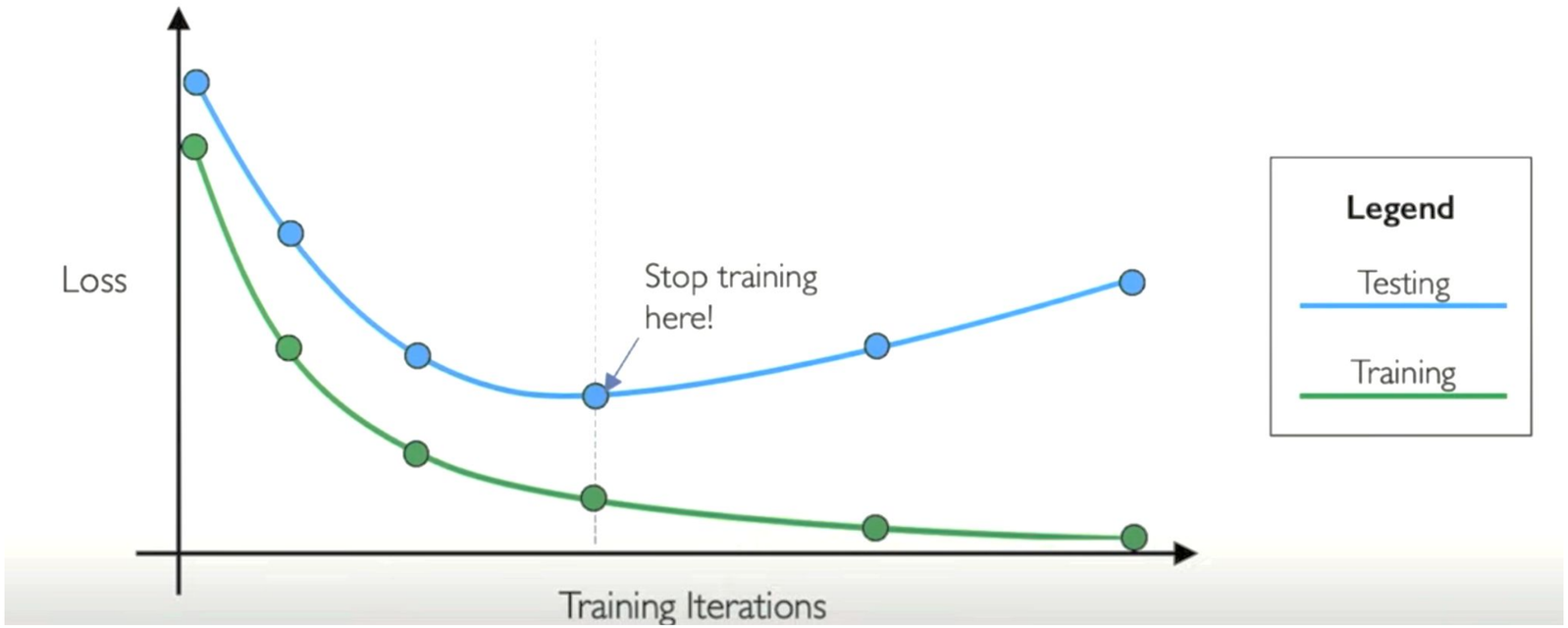
Regularization I: Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
tf.keras.layers.Dropout(p=0.5)
torch.nn.Dropout(p=0.5)
```



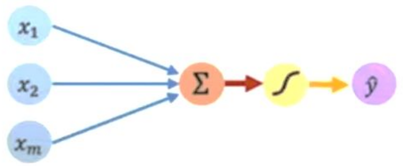
Regularization II: Early Stopping



Core Foundation Review

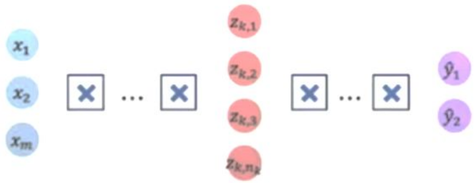
The Perceptron

- Structural building blocks
- Nonlinear activation functions



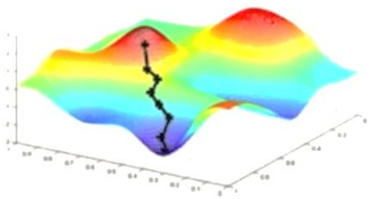
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



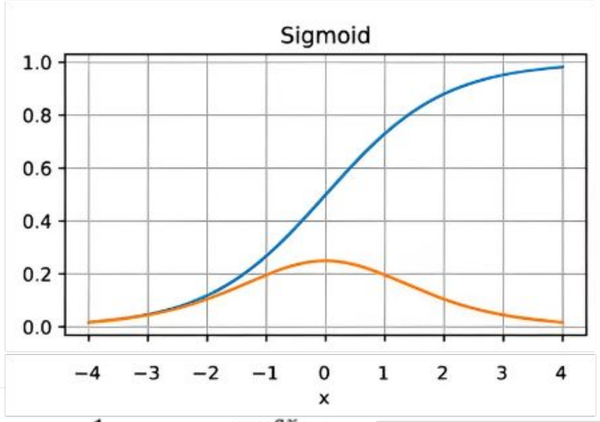
Training in Practice

- Adaptive learning
- Batching
- Regularization



Appendix

Sigmoid Function



$$\varphi(x) = \frac{1}{1 + exp^{-ax}} = \frac{exp^{ax}}{1 + exp^{ax}}$$

$$\varphi'(v) = a\varphi(v)(1 - \varphi(v))$$

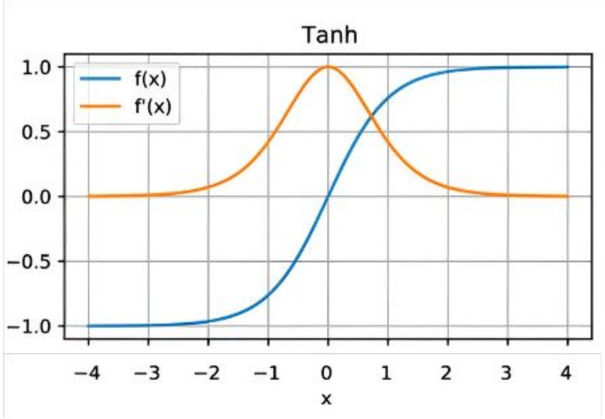
- 출력 범위가 0~1 사이 이므로, 확률로 해석하기 쉬움
- 극단의 입력값은 Vanishing gradient problem 일으킬 수 있음
- 확률적 특성으로 인해 이진 분류의 출력층에 자주 사용
- 은닉층에서는 Gradient 소실 문제로 인해 잘 사용하지 않음

Vanishing gradient problem:

역전파에서 각 층의 gradient가 계속 곱해지기 때문에, 0에 가까운 gradient가 많은 층에서 나올수록 총 곱이 0에 수렴한다. 따라서 가중치가 바뀌지 않고, 학습이 멈춘다.

Appendix

Hyperbolic Tangent

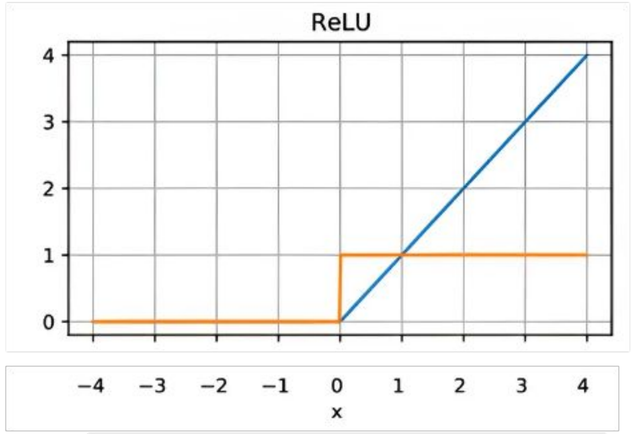


$$\varphi(x) = \frac{e^{ax} - e^{-ax}}{e^{ax} + e^{-ax}}$$

- 출력 범위가 -1~1 사이 이므로, 음/양수를 균형있게 다루는 문제에서 유용
- 여전히 gradient 소실 가능
- 그러나 출력이 음수와 양수 둘 다 존재해 편향이 적게 일어나고,
  훨씬 더 빠른 학습이 가능
- 일부 은닉층에 사용

Appendix

Rectified Linear Unit (ReLU)



$$\varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} = \max(0, x)$$

$$\varphi'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

- 출력 범위: [0, ∞)
- 계산이 단순하고 gradient 소실 문제를 완화
- 여전히 gradient 소실 가능
- 그러나 x가 0 이하일 때 Dying ReLU 현상 발생 가능
- 대부분의 은닉층에서 기본적으로 사용
- Deep Neural Network에 적합

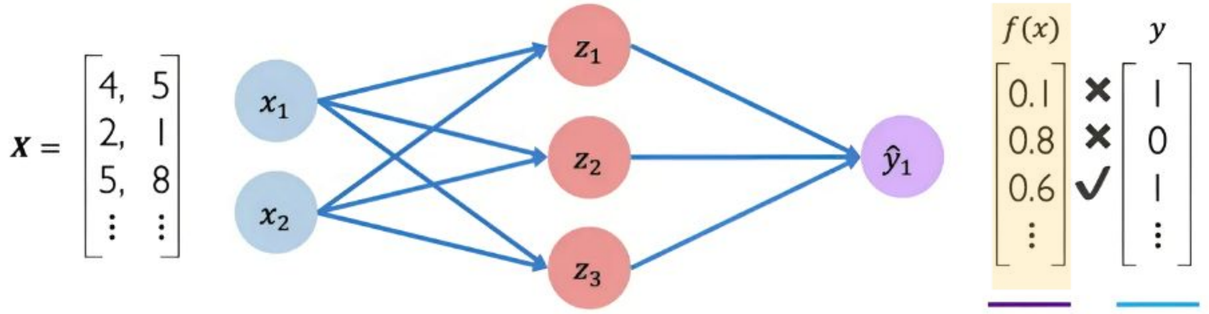
Dying ReLU:

음수만을 출력하는 가중치가 설정되어, 뉴런이 비활성화되고 모든 입력에 대해 0을 출력하는 문제

Appendix

Binary Cross Entropy Loss

이진 출력의 예측값은
일종의 확률처럼 작동한다



$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f\left(x^{(i)}; W\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - f\left(x^{(i)}; W\right)\right)$$

Actual Predicted Actual Predicted

Entropy,h(x): "Degree of Surprise"

x 값을 알게 되었을 때 얻을 수 있는 정보의 양

*(p(x)가 작을수록 x발생시 얻는 정보의 양이 큼 = 엔트로피 높음)**

Why log?

x,y 가 서로 독립일 때의 엔트로피를 생각해보자.

두 변수를 각각 관찰했을 때 얻는 정보의 양

= 동시에 관찰했을 때 얻는 정보의 양

이므로, h(x,y)=h(x)+h(y)여야 한다.

또한, *를 고려했을 때, h(x)는 p(x)에 대한 하강 monotonic 함수

여야 하므로, $h(x)=-\log_2\{p(x)\}$