Artificial Intelligence and Mobility Lab

GAN and Reinforcement Learning: Theory and Implementation

Prof. Joongheon Kim
Korea University, School of Electrical Engineering
Artificial Intelligence and Mobility Laboratory
https://joongheon.github.io
joongheon@korea.ac.kr

- **Introduction**
- Generative Adversarial Network
- Reinforcement Learning
- Deep Reinforcement Learning

GAN and Reinforcement Learning
Introduction
Introduction

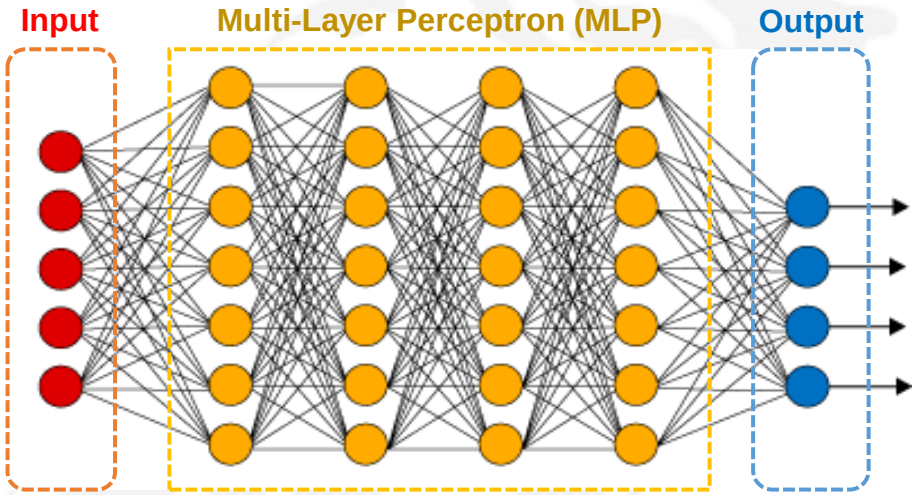
- **Deep Learning Overview**
- Linear Regression
- Binary Classification
- Neural Network Basics

- How Deep Learning Works?
  - Deep Learning Computation Procedure

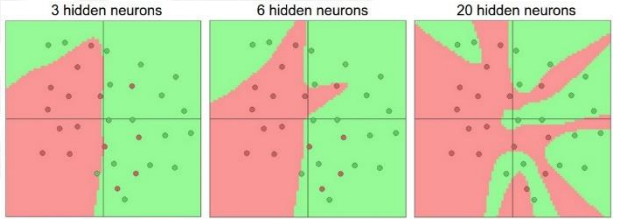
Deep Learning Model Setup
- MLP, CNN, RNN, GAN, or Customized
- # Hidden Layers, # Units, Input/Output, …
- Cost Function / Optimizer Selection

Training (with Large-Scale Dataset)
- Input: Data, Output: Labels
- Learning → Weights Updates for Cost Function Minimization

Inference / Testing (Real-Word Execution)
- Input: Real-World Input Data
- Output: Interference Results based on Updated Weights in Deep Neural Networks



Non-Linear Training (Weights Updates) for Cost Minimization: GD, SGD, Adam, etc.



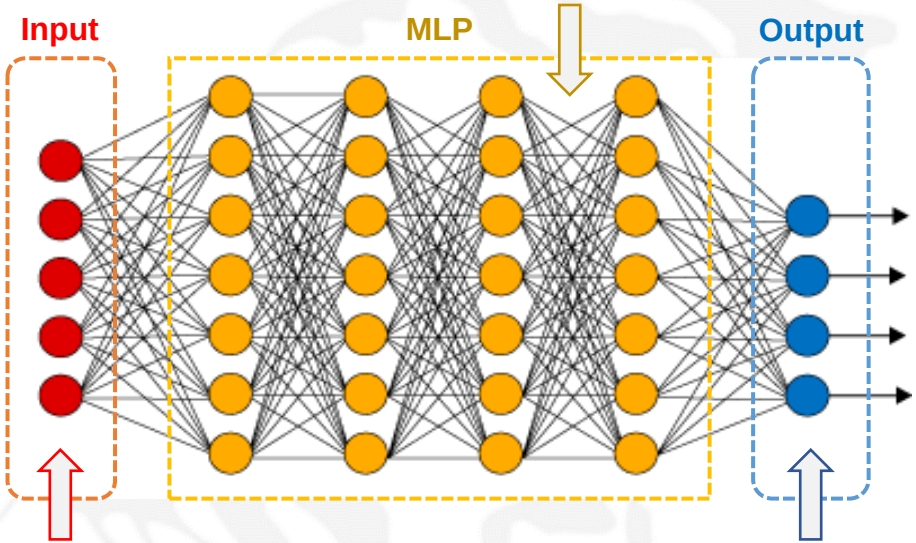
- How Deep Learning Works?
  - Deep Learning Computation Procedure

Deep Learning Model Setup
- MLP, CNN, RNN, GAN, or Customized
- # Hidden Layers, # Units, Input/Output, …
- Cost Function / Optimizer Selection

Training (with Large-Scale Dataset)
- Input: Data, Output: Labels
- Learning → Weights Updates for Cost Function Minimization

Inference / Testing (Real-Word Execution)
- Input: Real-World Input Data
- Output: Interference Results based on Updated Weights in Deep Neural Networks

All weights in units are trained/set (under cost minimization)



INPUT: Data
- One-Dimension Vector

OUTPUT: Labels
- One-Hot Encoding

We need a lot of training data for generality (otherwise, we will suffer from overfitting problem).

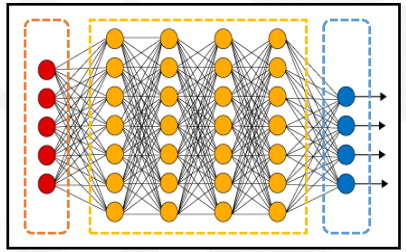
- How Deep Learning Works?
  - Deep Learning Computation Procedure

Deep Learning Model Setup
- MLP, CNN, RNN, GAN, or Customized
- # Hidden Layers, # Units, Input/Output, …
- Cost Function / Optimizer Selection

Training (with Large-Scale Dataset)
- Input: Data, Output: Labels
- Learning → Weights Updates for Cost Function Minimization

Inference / Testing (Real-Word Execution)
- Input: Real-World Input Data
- Output: Interference Results based on Updated Weights in Deep Neural Networks

Trained Model



Intelligent Surveillance Platforms

INPUT: Real-Time Arrivals

OUTPUT: Inference
- Computation Results based on (i) INPUT and (ii) trained weights in units (trained model).

- How Deep Learning Works?
  - Issue - **Overfitting**

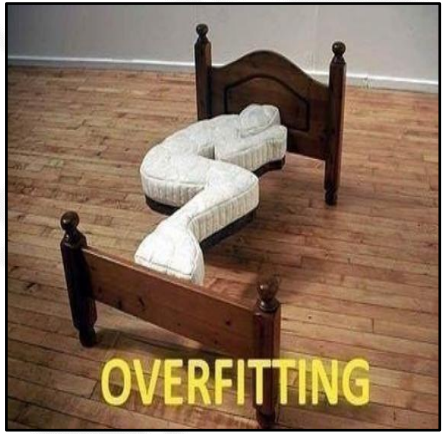
What if we do not have enough data for training (not enough to derive Gaussian/normal distribution)?

Deep Learning Model Setup
- MLP, CNN, RNN, GAN, or Custom
- # Hidden Layers, # Units, Input/Output, …
- Cost Function / Optimizer Selection

Training (with Large-Scale Dataset)
- Input: Data, Output: Labels
- Learning → Weights Updates for Cost Function Minimization

Inference / Testing (Real-Word Execution)
- Input: Real-World Input Data
- Output: Interference Results based on Updated Weights in Deep Neural Networks

Situation becomes worse when the model (with insufficient training data) accurately fits on training data.



To Combat the Overfitting
- More training data
- Autoencoding (or variational auto-encoder (VAE))
- Droupout
- Regularization

- Quick Start Example

ex_add.py

```
1  import tensorflow as tf
2  # Create nodes in computation graph
3  node1 = tf.constant(3, dtype=tf.int32)
4  node2 = tf.constant(5, dtype=tf.int32)
5  node3 = tf.add(node1, node2)
6
7  # Create session object
8  sess = tf.Session()
9  print("node1 + node2 = ", sess.run(node3))
10 # Close the session
11 sess.close()
```

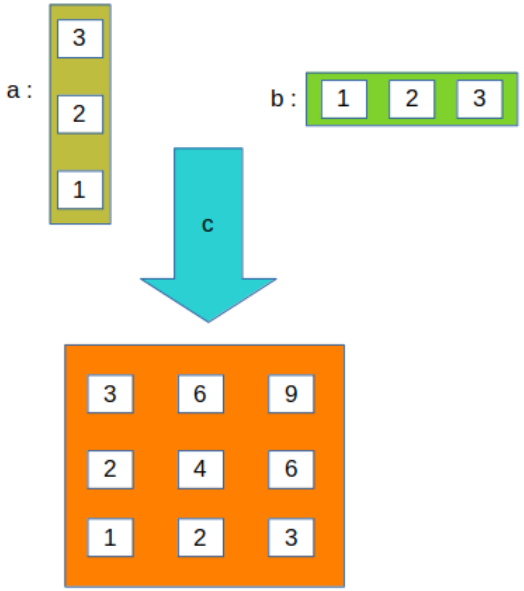
ex_add2.py

```
1  import tensorflow as tf
2  # Create nodes in computation graph
3  node1 = tf.constant(3, dtype=tf.int32)
4  node2 = tf.constant(5, dtype=tf.int32)
5  node3 = tf.add(node1, node2)
6
7  # Create session object
8  with tf.Session() as sess:
9      print("node1 + node2 = ", sess.run(node3))
```

- Example: Placeholder

ex_placeholder.py

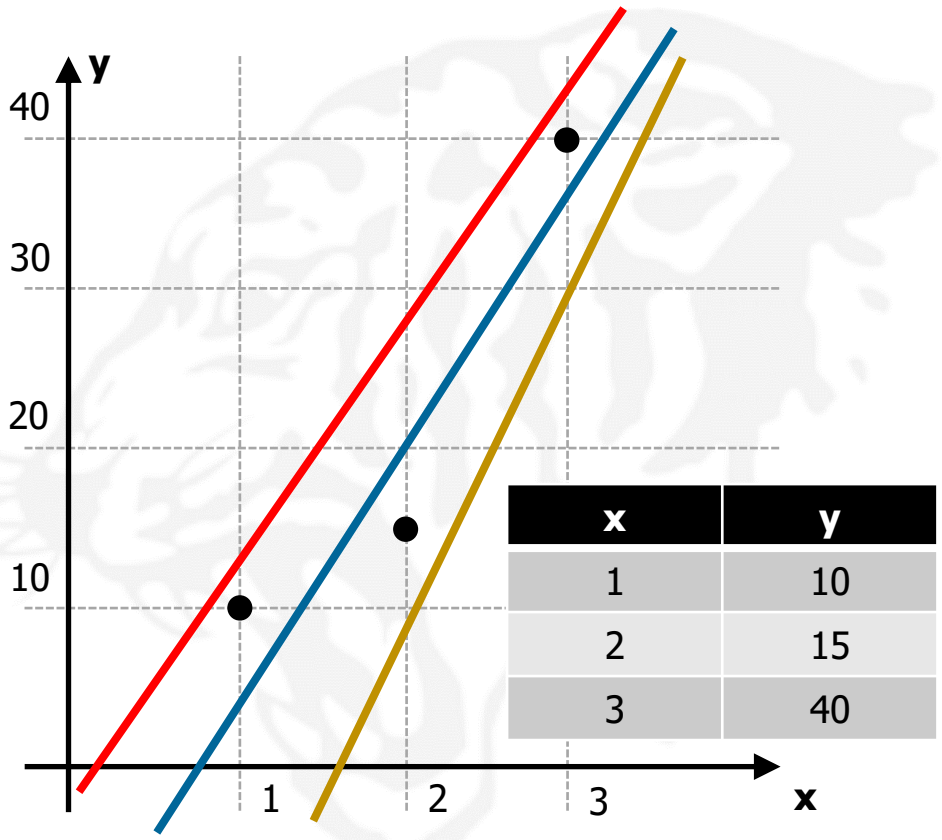
```
1  import tensorflow as tf
2
3  # Create nodes in computation graph
4  a = tf.placeholder(tf.int32, shape=(3,1))
5  b = tf.placeholder(tf.int32, shape=(1,3))
6  c = tf.matmul(a, b)
7
8  # Run computation graph
9  with tf.Session() as sess:
10     print(sess.run(c, feed_dict={a: [[3], [2], [1]], b: [[1, 2, 3]]}))
```



GAN and Reinforcement Learning
Introduction
Linear Regression

- Deep Learning Overview
- **Linear Regression**
- Binary Classification
- Neural Network Basics

- Linear model: $H(x) = Wx + b$
- Which model is the best among the given three?



x	y
1	10
2	15
3	40

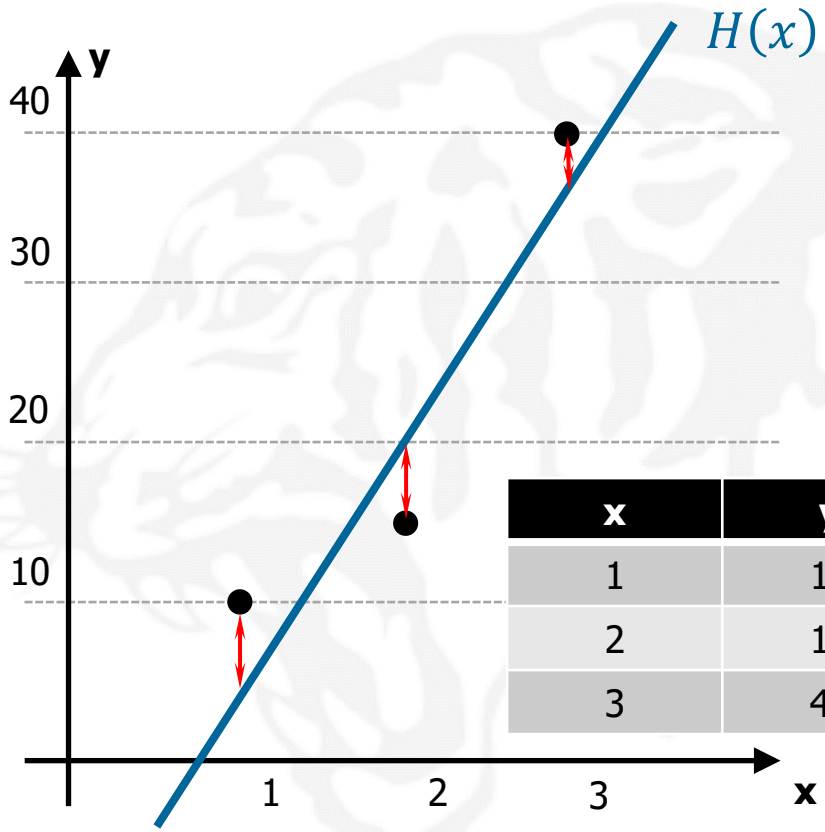
- Cost Function (or Loss Function)
  - How to fit the line to training data
  - The difference between model values and real measurements:

m: The number of training data

$$\frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^i\right)-y^i\right)^2$$

$$H(x)=Wx+b$$

$$\text{Cost}(W,b)=\frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^i\right)-y^i\right)^2$$

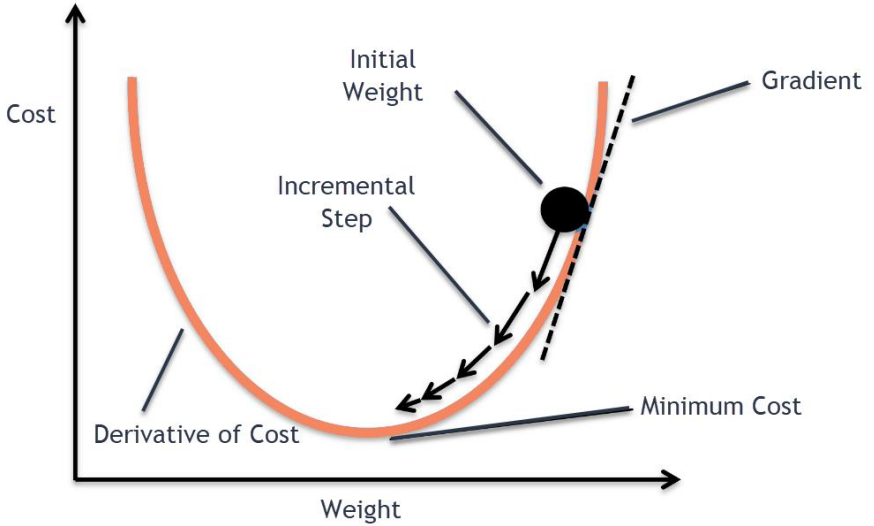


x	y
1	10
2	15
3	40

- Cost Function Minimization
  - Model: $H(x) = Wx + b$
  - Cost Function: $Cost(W, b) = \frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^i\right) - y^i\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(Wx^i + b - y^i\right)^2$
  - How to Minimize this Function? → **Gradient Descent Method**
    - Angle → Differentiation

$$W \leftarrow W - \alpha \frac{\partial}{\partial W} Cost(W)$$

α: Learning rate



- Multi-Variable Linear Regression
  - Model:

$$H(x_1, x_2, \dots, x_n) = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

  - Cost:

$$Cost(W, b) = \frac{1}{m} \sum_{i=1}^{m} \left(H\left(x_1^i, x_2^i, \dots, x_n^i\right) - y^i \right)^2$$

- Multi-Variable Linear Regression
  - Model:

$$H(x_1, x_2, \dots, x_n) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b \Rightarrow H(X) = XW + b$$

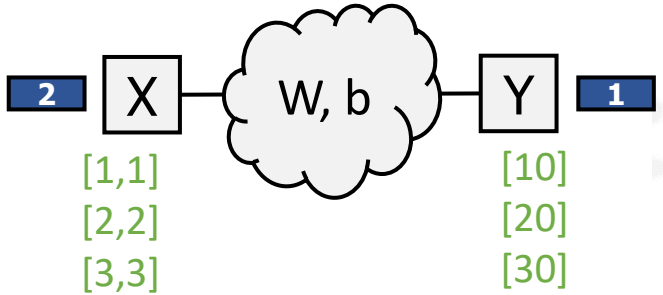
$$\underbrace{\begin{pmatrix} x_1 & x_2 \dots & x_n \end{pmatrix}}_{X} \cdot \underbrace{\begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix}}_{W} = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

$$\Rightarrow H(X) = XW^T + b$$

when $W = \begin{pmatrix} w_1 & w_2 \dots & w_n \end{pmatrix}$

Linear Regression Implementation (TensorFlow)


```python
import tensorflow as tf

x_data = [[1,1], [2,2], [3,3]]
y_data = [[10], [20], [30]]
X = tf.placeholder(tf.float32, shape=[None, 2])
Y = tf.placeholder(tf.float32, shape=[None, 1])

W=tf.Variable(tf.random_normal([2,1]))
b=tf.Variable(tf.random_normal([1]))

model = tf.matmul(X,W)+b
cost = tf.reduce_mean(tf.square(model - Y))
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Training
    for step in range(2001):
        c, W_, b_, _ = sess.run([cost, W, b, train], feed_dict={X: x_data, Y: y_data})
        print(step, c, W_, b_)
    # Testing
    print(sess.run(model, feed_dict={X: [[4,4]]}))
```



Model, Cost, Train

Linear Regression Implementation (TensorFlow)

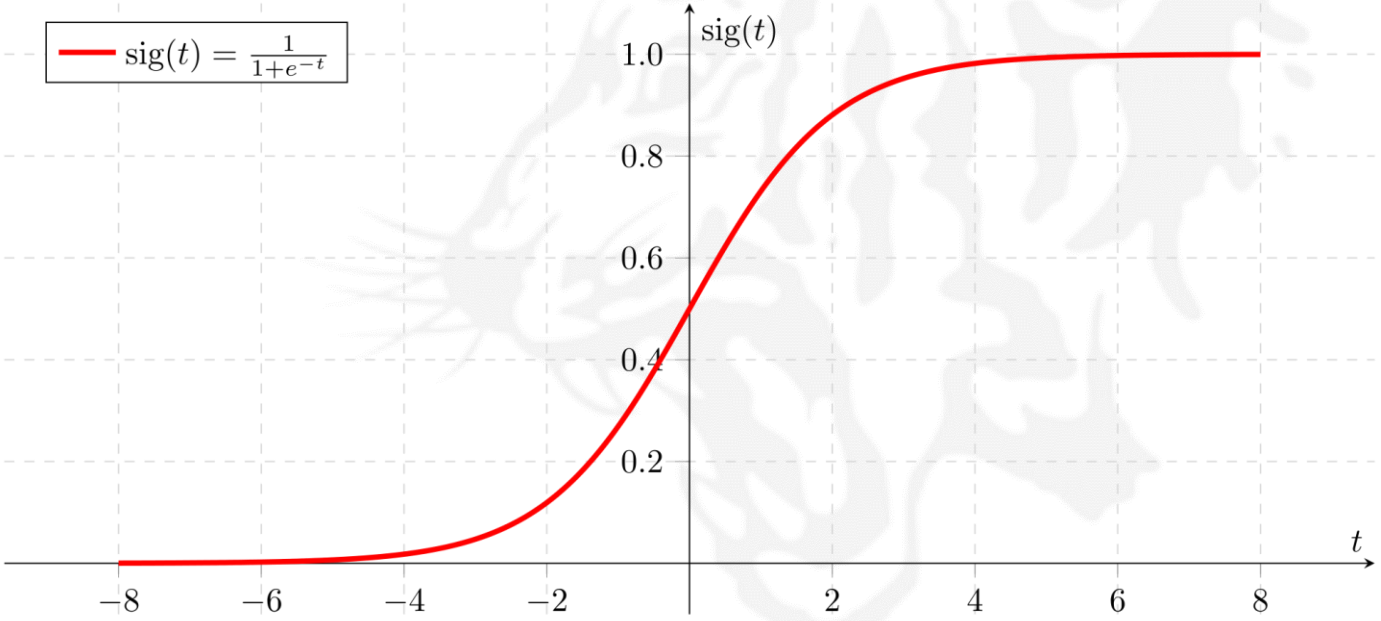

```
 [4.667964 ]] [0.014943]
1991 3.1940912e-05 [[5.325548 ]
 [4.6679726]] [0.01490401]
1992 3.1772186e-05 [[5.3255568]
 [4.667981 ]] [0.0148651]
1993 3.1603915e-05 [[5.3255653]
 [4.6679897]] [0.01482627]
1994 3.143936e-05 [[5.325574 ]
 [4.6679983]] [0.01478756]
1995 3.1277286e-05 [[5.3255825]
 [4.668007 ]] [0.01474891]
1996 3.1110336e-05 [[5.325591 ]
 [4.6680155]] [0.01471035]
1997 3.095236e-05 [[5.325599 ]
 [4.6680236]] [0.01467189]
1998 3.079518e-05 [[5.325608]
 [4.668032]] [0.01463356]
1999 3.062387e-05 [[5.325616 ]
 [4.6680403]] [0.01459529]
2000 3.0467529e-05 [[5.325624 ]
 [4.6680484]] [0.01455714]
[[39.989246]]
```

GAN and Reinforcement Learning
Introduction
Binary Classification

- Deep Learning Overview
- Linear Regression
- **Binary Classification**
- Neural Network Basics

- Binary Classification Examples
  - **Spam Detection:** Spam [1] or Ham [0]
  - **Facebook Feed:** Show [1] or Hide [0]
    - Facebook learns with your like-articles; and shows your favors.
  - **Credit Card Fraudulent Transaction Detection:** Fraud [1] or Legitimate [0]
  - **Tumor Image Detection in Radiology:** Malignant [1] or Benign [0]

- Binary Classification Basic Idea
  - Step 1) Linear regression with $H(x) = Wx + b$
  - Step 2) **Logistic/sigmoid function ($sig(t)$)** based on the result of Step 1.



Binary Classification

Linear Regression Model

$$H(x) = Wx + b \text{ or } H(X) = W^T X$$

Logistic/Sigmoid Function

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

Binary Classification Model

$$g(X) = \frac{1}{1 + e^{-W^T X}}$$

$$\text{Cost}(W,b)=\frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^i\right)-y^i\right)^2$$

Linear Regression Model

$$H(x)=Wx+b \text{ or } H(X)=W^TX$$

Gradient Descent Method can be used because $Cost(W,b)$ is convex (local minimum is global minimum).

Binary Classification Model

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

Gradient Descent Method can not be used because $Cost(W,b)$ is non-convex. **New Cost Function is required.**

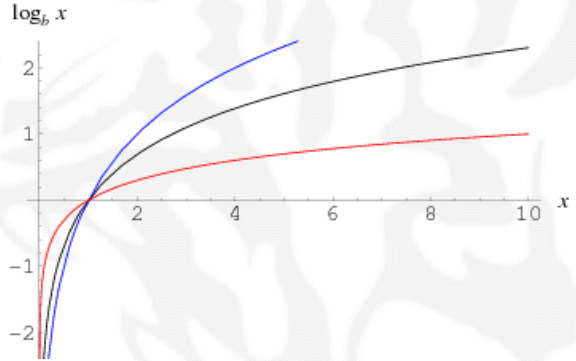
$$\text{Cost}(W)=\frac{1}{m}\sum c(H(x),y)$$

$$c(H(x),y)=\begin{cases}-\log\left(H(x)\right), & y=1\\ -\log\left(1-H(x)\right), & y=0\end{cases}$$

Understanding this Cost Function

Cost	0	∞	∞	0
$H(x)$	0	0	1	1
y	0	1	0	1

Log Function



$$\text{Cost}(W)=\frac{1}{m}\sum c(H(x),y)$$

$$c(H(x),y)=\begin{cases}-\log\left(H(x)\right), & y=1\\ -\log\left(1-H(x)\right), & y=0\end{cases}$$

$$c(H(x),y)=-y\log\left(H(x)\right)-(1-y)\log\left(1-H(x)\right)$$

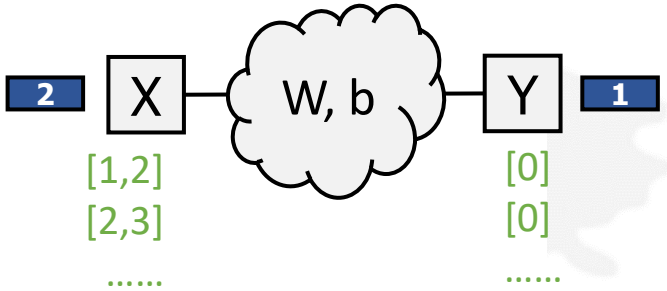
$$Cost(W)=-\frac{1}{m}\sum y\log\left(H(x)\right)+(1-y)\log\left(1-H(x)\right)$$

Gradient Descent Method

$$W\leftarrow W-\alpha\frac{\partial}{\partial W}Cost(W)$$

Binary Classification Implementation (TensorFlow)


```python
import tensorflow as tf

x_data = [[1,2], [2,3], [3,1], [4,3], [5,3], [6,2]]
y_data = [[0], [0], [0], [1], [1], [1]]
X = tf.placeholder(tf.float32, shape=[None, 2])
Y = tf.placeholder(tf.float32, shape=[None, 1])
W = tf.Variable(tf.random_normal([2,1]))
b = tf.Variable(tf.random_normal([1]))

model = tf.sigmoid(tf.add(tf.matmul(X,W),b))
cost = tf.reduce_mean((-1)*Y*tf.log(model) + (-1)*(1-Y)*tf.log(1-model))
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

prediction = tf.cast(model > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, Y), dtype=tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Training
    for step in range(10001):
        cost_val, train_val = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
        print(step, cost_val)
    # Testing
    h, c, a = sess.run([model, prediction, accuracy], feed_dict={X: x_data, Y: y_data})
    print("\nModel: ", h,"\nCorrect: ", c, "\nAccuracy: ", a)
```



Model, Cost, Train

Binary Classification Implementation (TensorFlow)

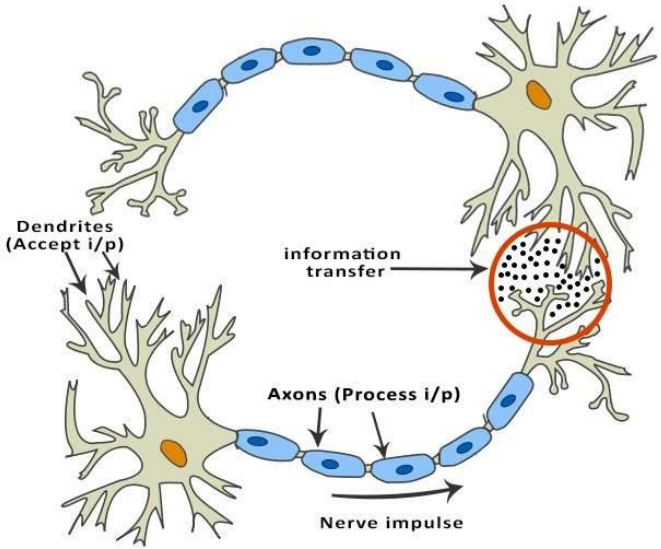
```python
import tensorflow as tf

x_data = [[1,2], [2,3], [3,1], [4,3], [5,3], [6,2]]
y_data = [[0], [0], [0], [1], [1], [1]]
X = tf.placeholder(tf.float32, shape=[None, 2])
Y = tf.placeholder(tf.float32, shape=[None, 1])
W = tf.Variable(tf.random_normal([2,1]))
b = tf.Variable(tf.random_normal([1]))

model = tf.sigmoid(tf.add(tf.matmul(X,W),b))
cost = tf.reduce_mean((-1)*Y*tf.log(model) + (-1)*(1-Y)*tf.log(1-model))
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

prediction = tf.cast(model > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, Y), dtype=tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Training
    for step in range(10001):
        cost_val, train_val = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
        print(step, cost_val)
    # Testing
    h, c, a = sess.run([model, prediction, accuracy], feed_dict={X: x_data, Y: y_data})
    print("\nModel: ", h,"\nCorrect: ", c, "\nAccuracy: ", a)
```

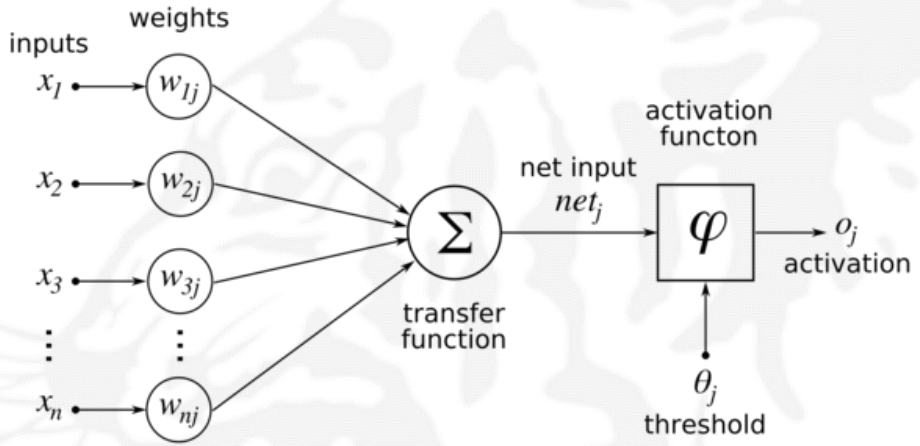
```
9985 0.1493395
9986 0.14932828
9987 0.14931704
9988 0.14930585
9989 0.14929464
9990 0.1492834
9991 0.14927219
9992 0.14926098
9993 0.14924978
9994 0.14923854
9995 0.14922734
9996 0.14921615
9997 0.14920495
9998 0.14919376
9999 0.1491826
10000 0.1491714

Model:  [[0.0306041 ]
 [0.15866211]
 [0.30421484]
 [0.7816807 ]
 [0.93976283]
 [0.9802319 ]]
Correct:  [[0.]
 [0.]
 [0.]
 [1.]
 [1.]
 [1.]]
Accuracy:  1.0
```

GAN and Reinforcement Learning
Introduction
Neural Network Basics

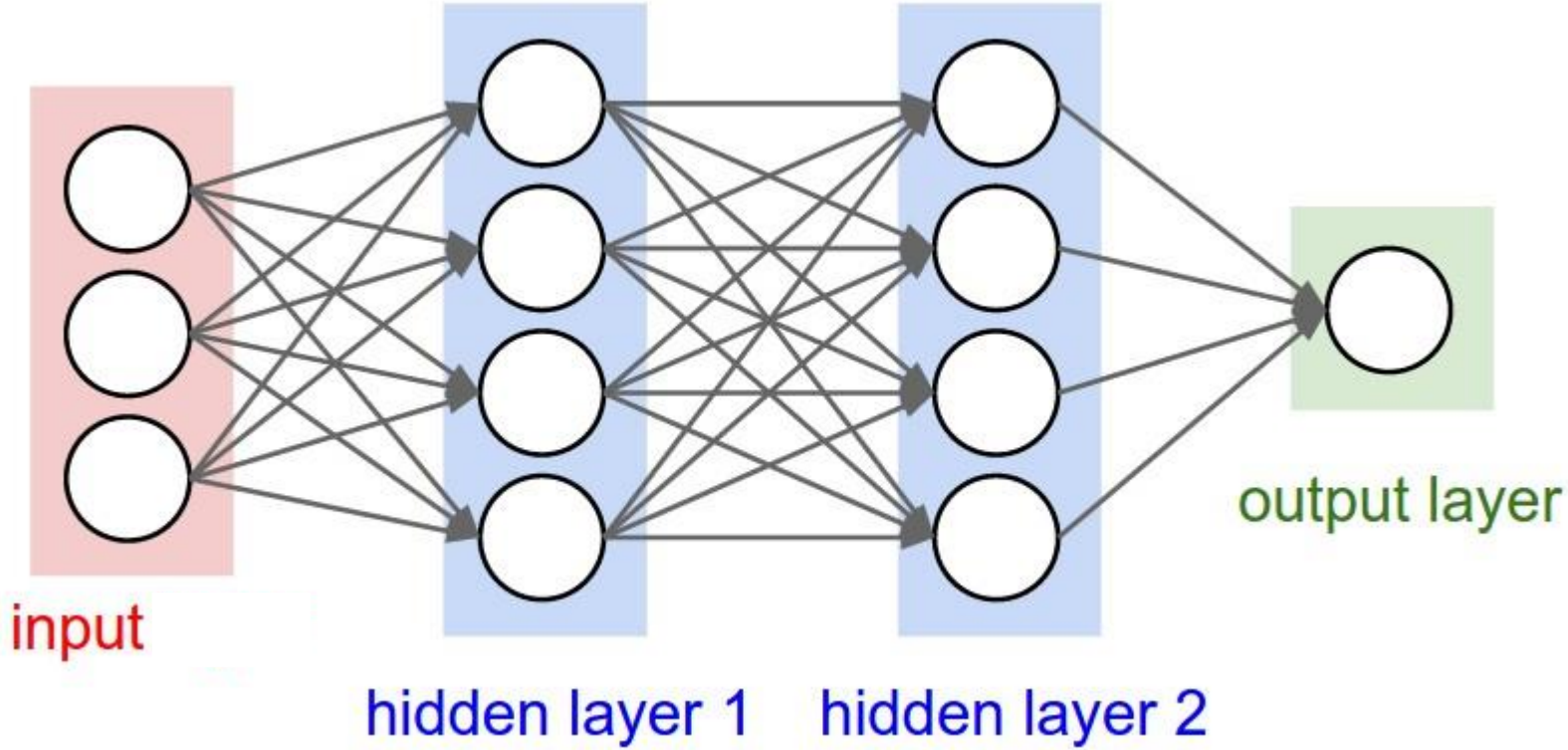
- Deep Learning Overview
- Linear Regression
- **Binary Classification**
- Neural Network Basics

- Human Brain (Neuron)



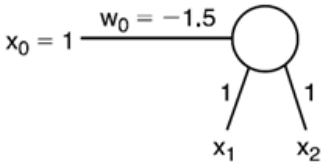


Binary Classification

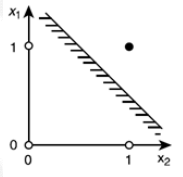

input
hidden layer 1
hidden layer 2
output layer

- Application to Logic Gate Design

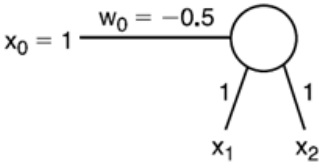
AND gate



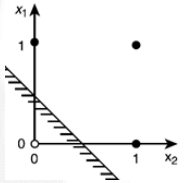
x_1	x_2	$W \cdot X$	y
0	0	−1.5	0
0	1	−0.5	0
1	0	−0.5	0
1	1	0.5	1



OR gate

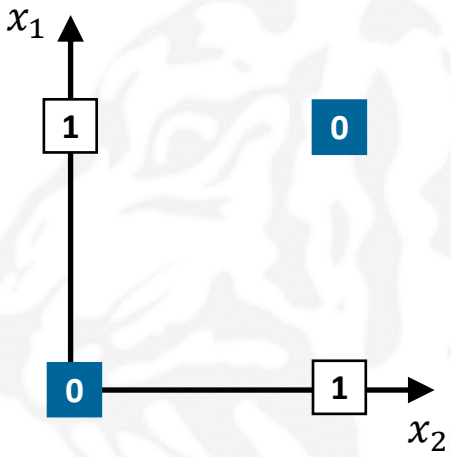


x_1	x_2	$W \cdot X$	y
0	0	−0.5	0
0	1	0.5	1
1	0	0.5	1
1	1	1.5	1



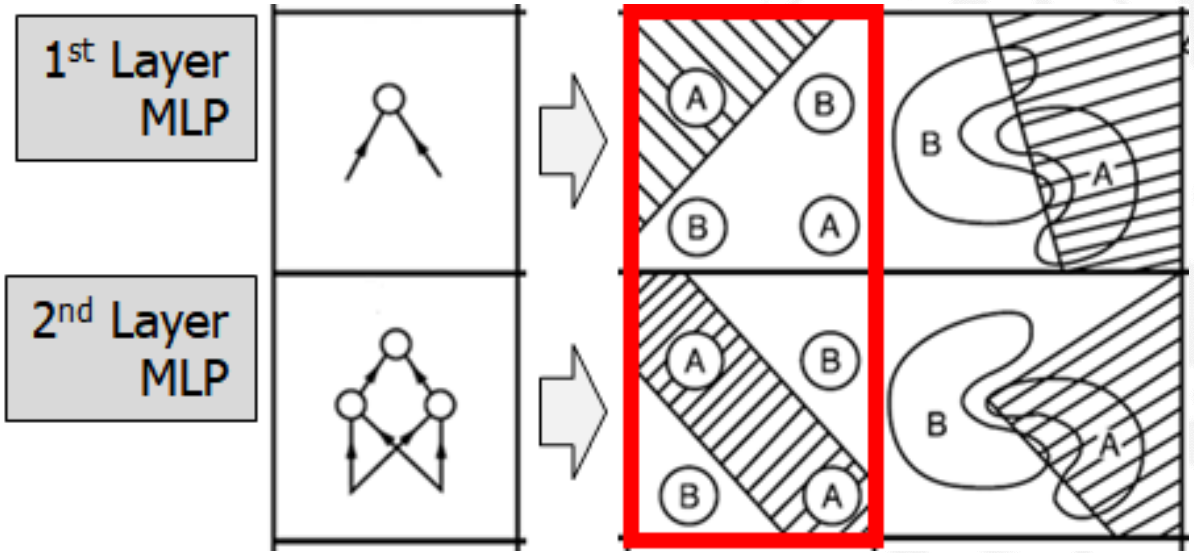
- **What about XOR?**

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

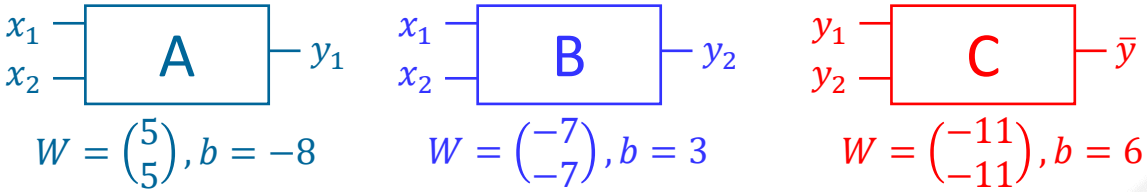


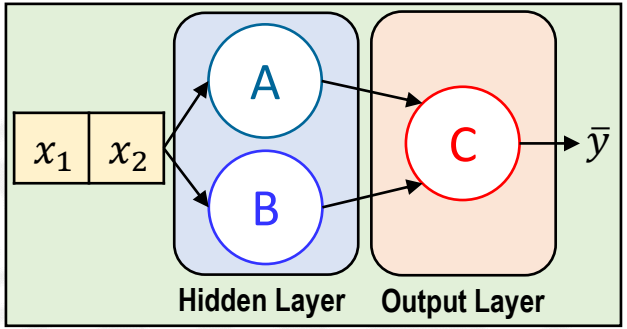
Mathematically proven by
Prof. Marvin Minsky at MIT (1969)

- Multilayer Perceptron (MLP)
  - Proposed by Prof. Marvin Minsky at MIT (1969)
  - Can solve XOR Problem

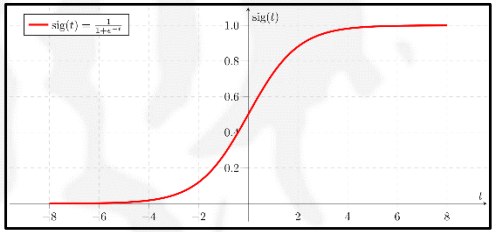


ANN: Solving XOR with MLP



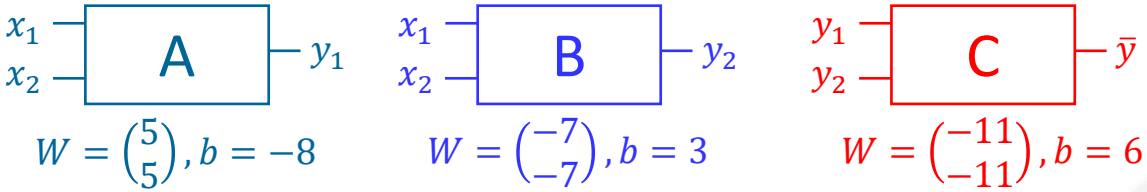


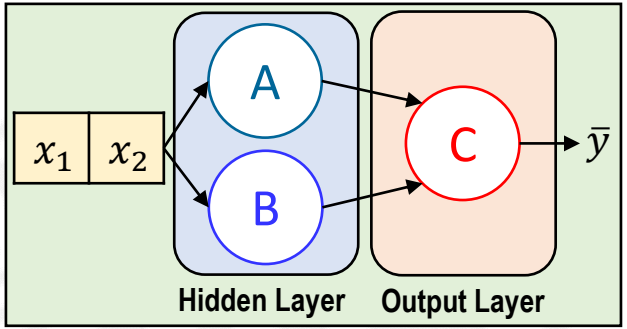
- $(x_1\ x_2) = (0\ 0)$
  - $(0\ 0)\begin{pmatrix}5\\5\end{pmatrix} + (-8) = -8$, i.e., $y_1 = Sigmoid(-8) \cong 0$
  - $(0\ 0)\begin{pmatrix}-7\\-7\end{pmatrix} + (3) = 3$, i.e., $y_2 = Sigmoid(3) \cong 1$
  - $(y_1\ y_2)\begin{pmatrix}-11\\-11\end{pmatrix} + (6) = -11 + 6 = -5$, i.e., $\bar{y} = Sigmoid(-5) \cong 0$



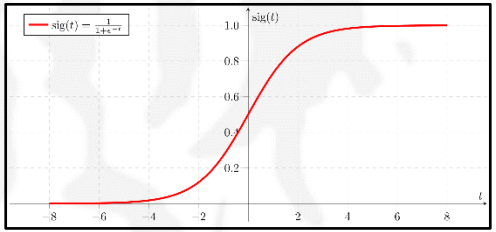
x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	**0**	**1**	**0**	**0**
0	1				**1**
1	0				**1**
1	1				**0**

ANN: Solving XOR with MLP



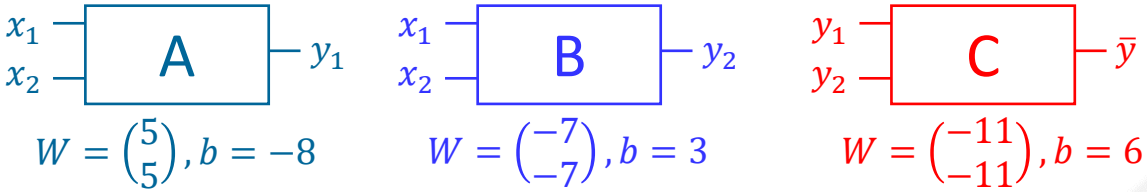


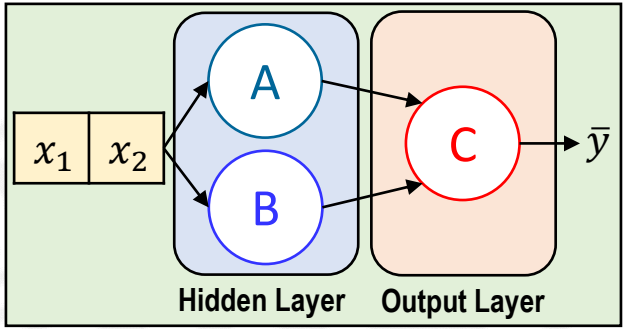
- $(x_1\ x_2) = (0\ 1)$
  - $(0\ 1)\begin{pmatrix}5\\5\end{pmatrix} + (-8) = -3$, i.e., $y_1 = Sigmoid(-3) \cong 0$
  - $(0\ 1)\begin{pmatrix}-7\\-7\end{pmatrix} + (3) = -4$, i.e., $y_2 = Sigmoid(-4) \cong 0$
  - $(y_1\ y_2)\begin{pmatrix}-11\\-11\end{pmatrix} + (6) = 6$, i.e., $\bar{y} = Sigmoid(6) \cong 1$



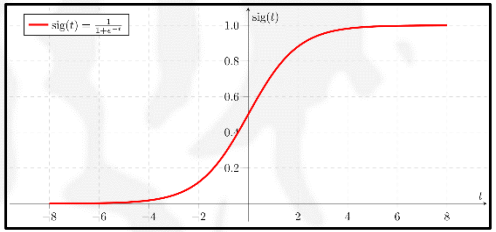
x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	0	1	0	0
0	1	0	0	1	1
1	0				1
1	1				0

ANN: Solving XOR with MLP



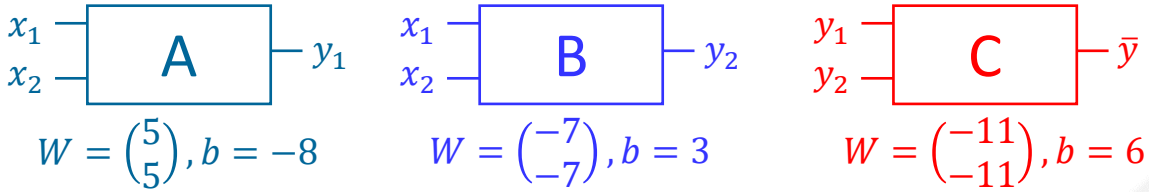


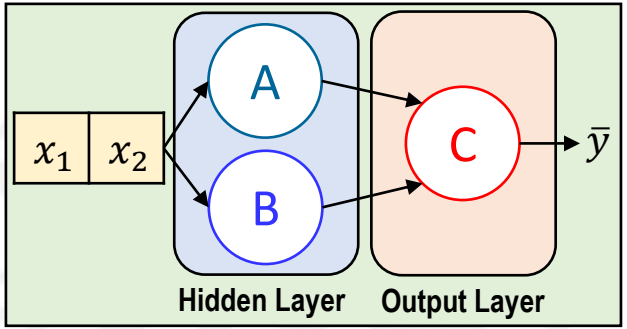
- $(x_1\ x_2) = (1\ 0)$
  - $(1\ 0)\begin{pmatrix}5\\5\end{pmatrix} + (-8) = -3$, i.e., $y_1 = Sigmoid(-3) \cong 0$
  - $(1\ 0)\begin{pmatrix}-7\\-7\end{pmatrix} + (3) = -4$, i.e., $y_2 = Sigmoid(-4) \cong 0$
  - $(y_1\ y_2)\begin{pmatrix}-11\\-11\end{pmatrix} + (6) = 6$, i.e., $\bar{y} = Sigmoid(6) \cong 1$



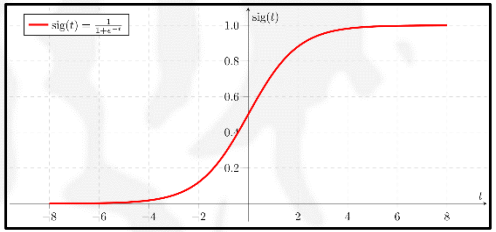
x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	**0**	**1**	**0**	**0**
0	1	**0**	**0**	**1**	**1**
1	0	**0**	**0**	**1**	**1**
1	1				**0**

ANN: Solving XOR with MLP



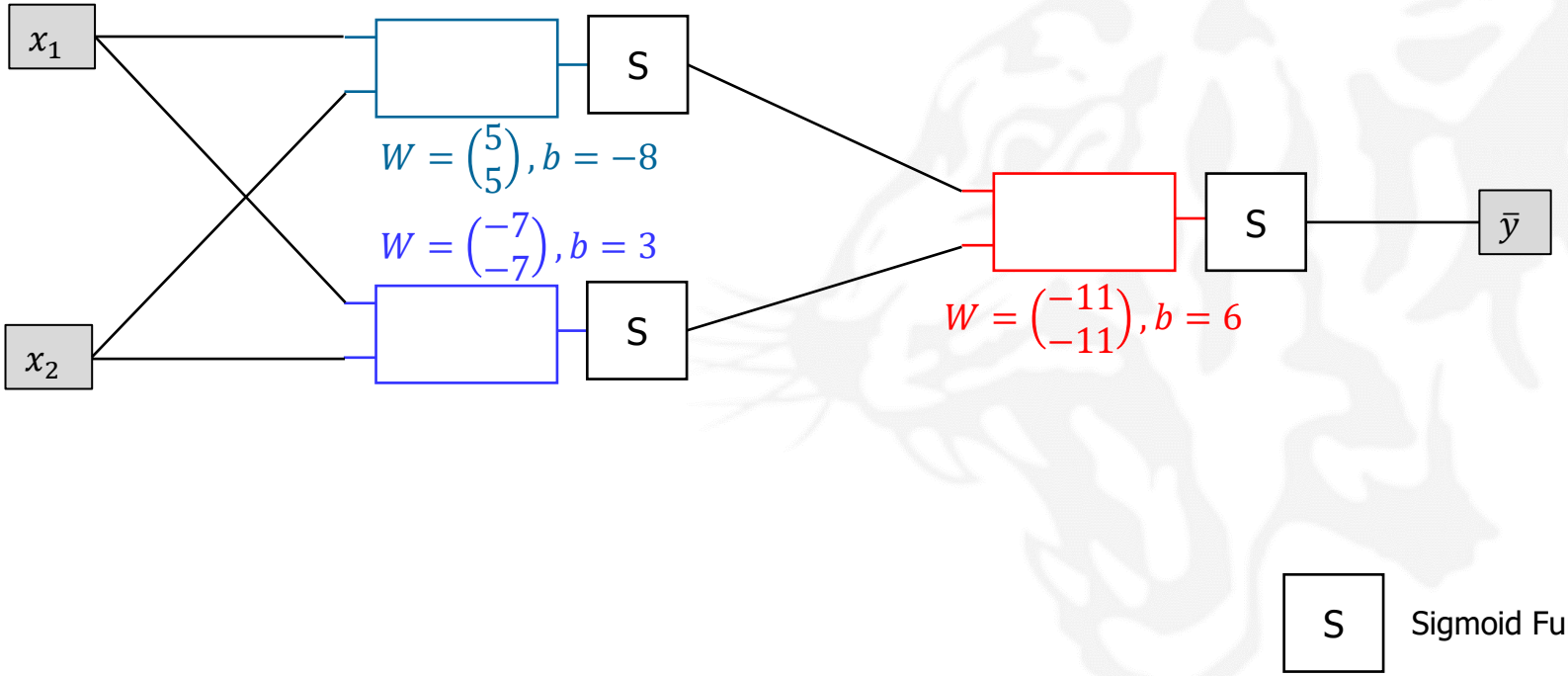


- $(x_1\ x_2) = (1\ 1)$
  - $(1\ 1)\begin{pmatrix}5\\5\end{pmatrix} + (-8) = 2$, i.e., $y_1 = Sigmoid(2) \cong 1$
  - $(1\ 1)\begin{pmatrix}-7\\-7\end{pmatrix} + (3) = -11$, i.e., $y_2 = Sigmoid(-11) \cong 0$
  - $(y_1\ y_2)\begin{pmatrix}-11\\-11\end{pmatrix} + (6) = -5$, i.e., $\bar{y} = Sigmoid(-5) \cong 0$

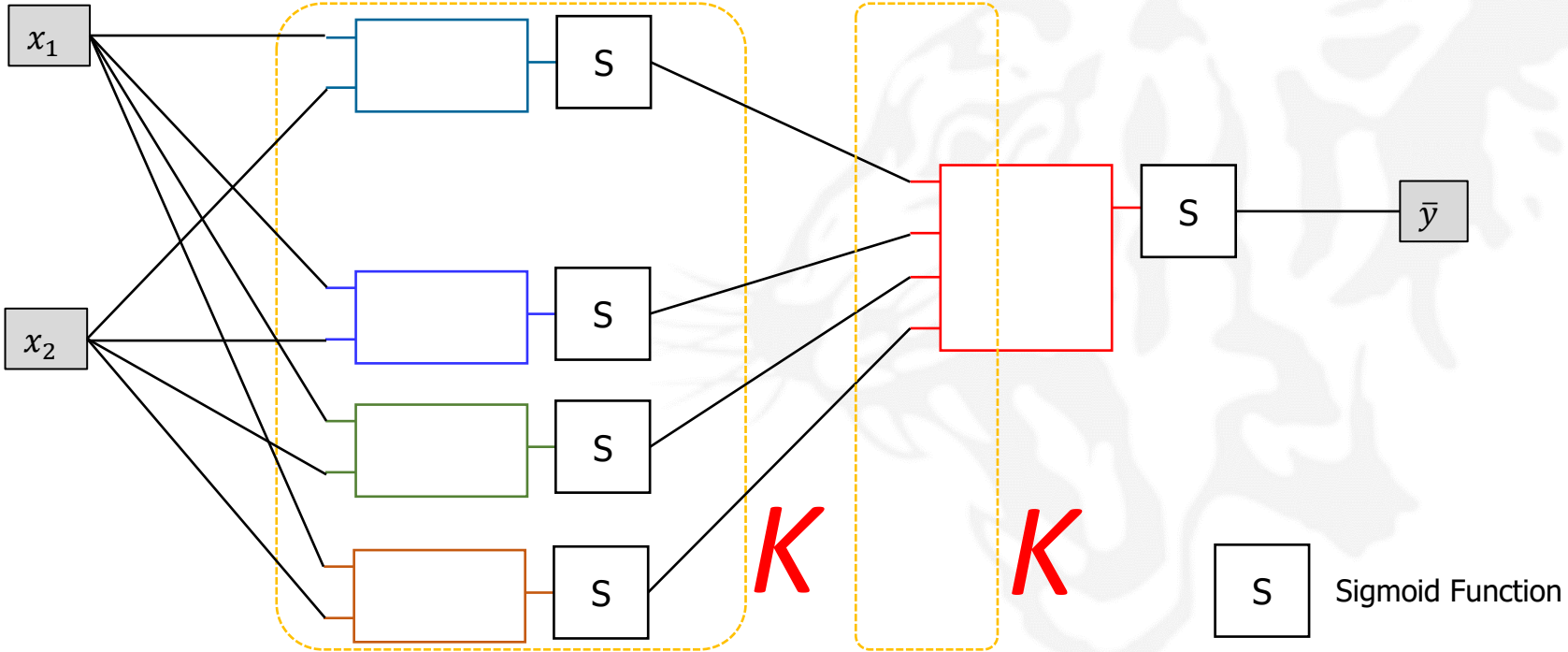


x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	0	1	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	0	0	0

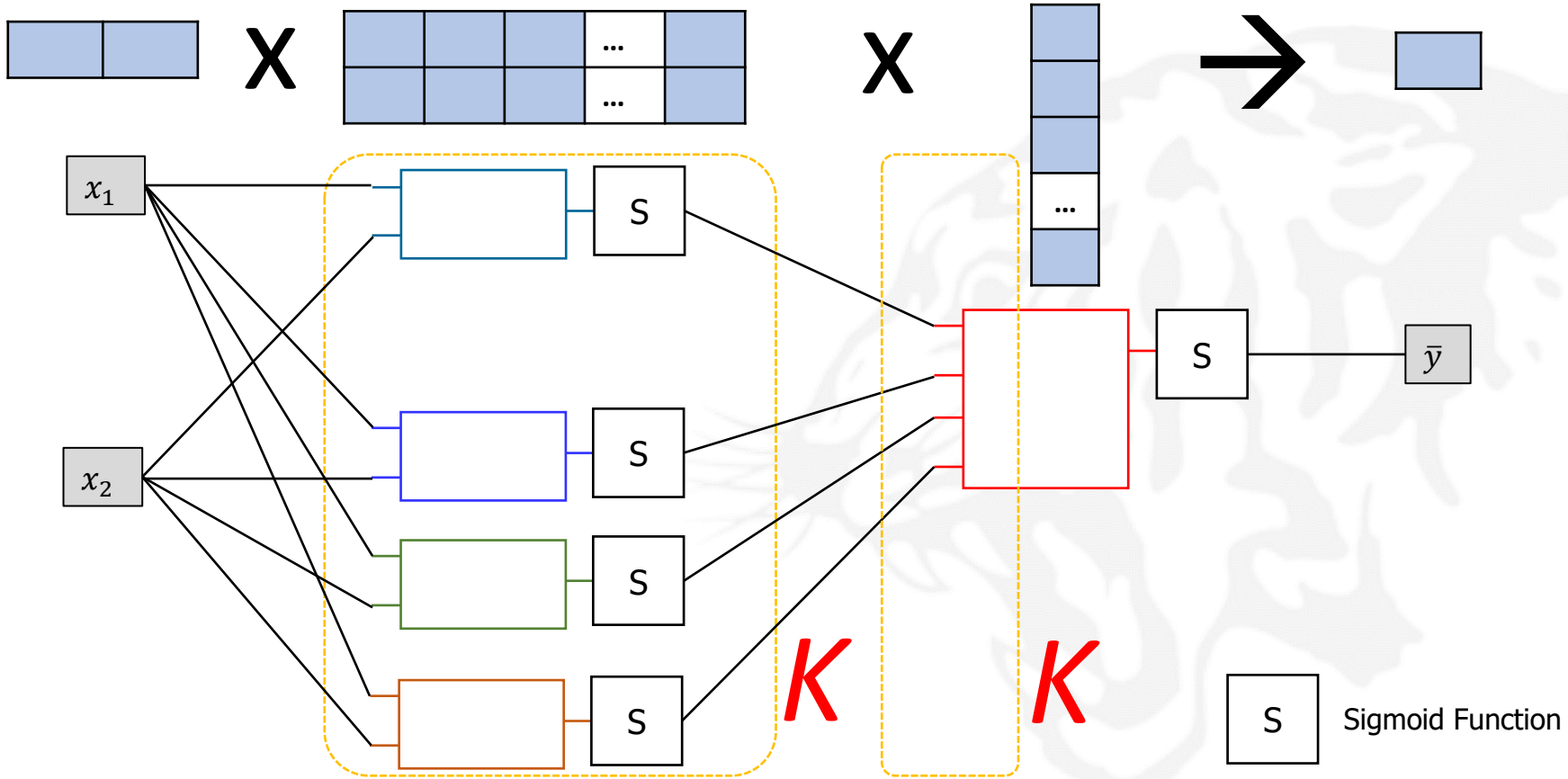
ANN: Solving XOR with MLP (Forward Propagation)



ANN: Solving XOR with MLP (Forward Propagation)

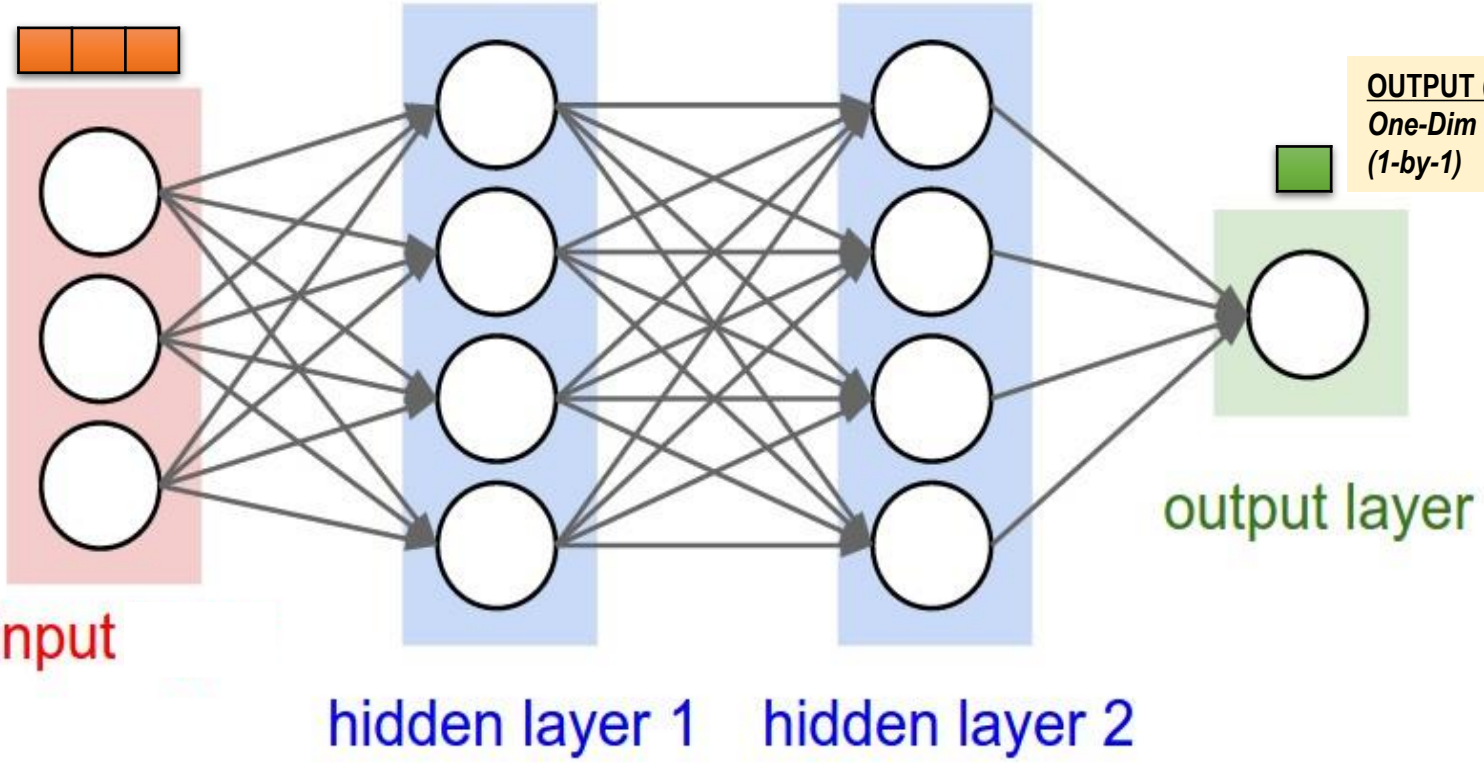


ANN: Solving XOR with MLP (Forward Propagation)



- Toy Model

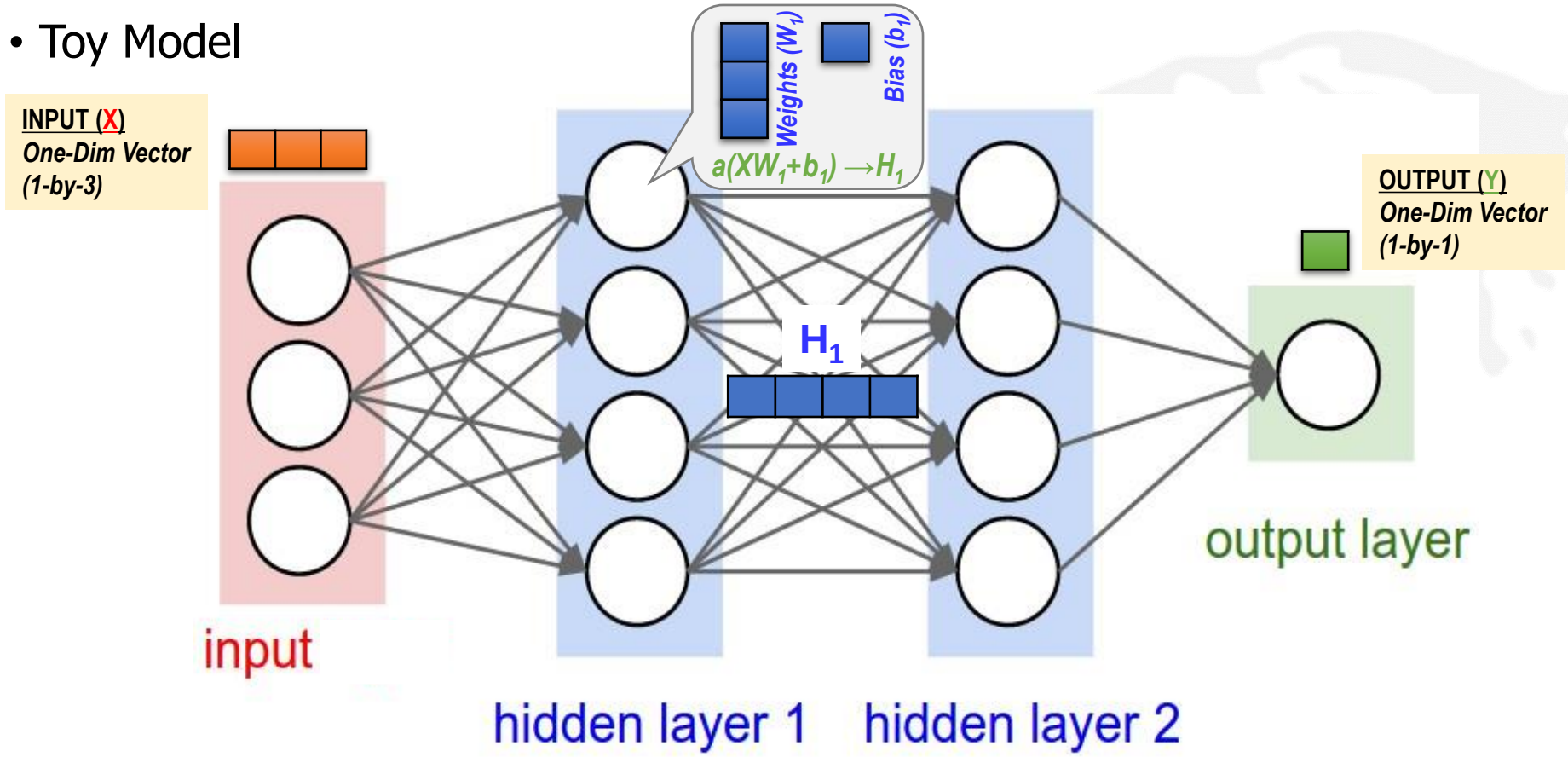
INPUT (X)
One-Dim Vector (1-by-3)



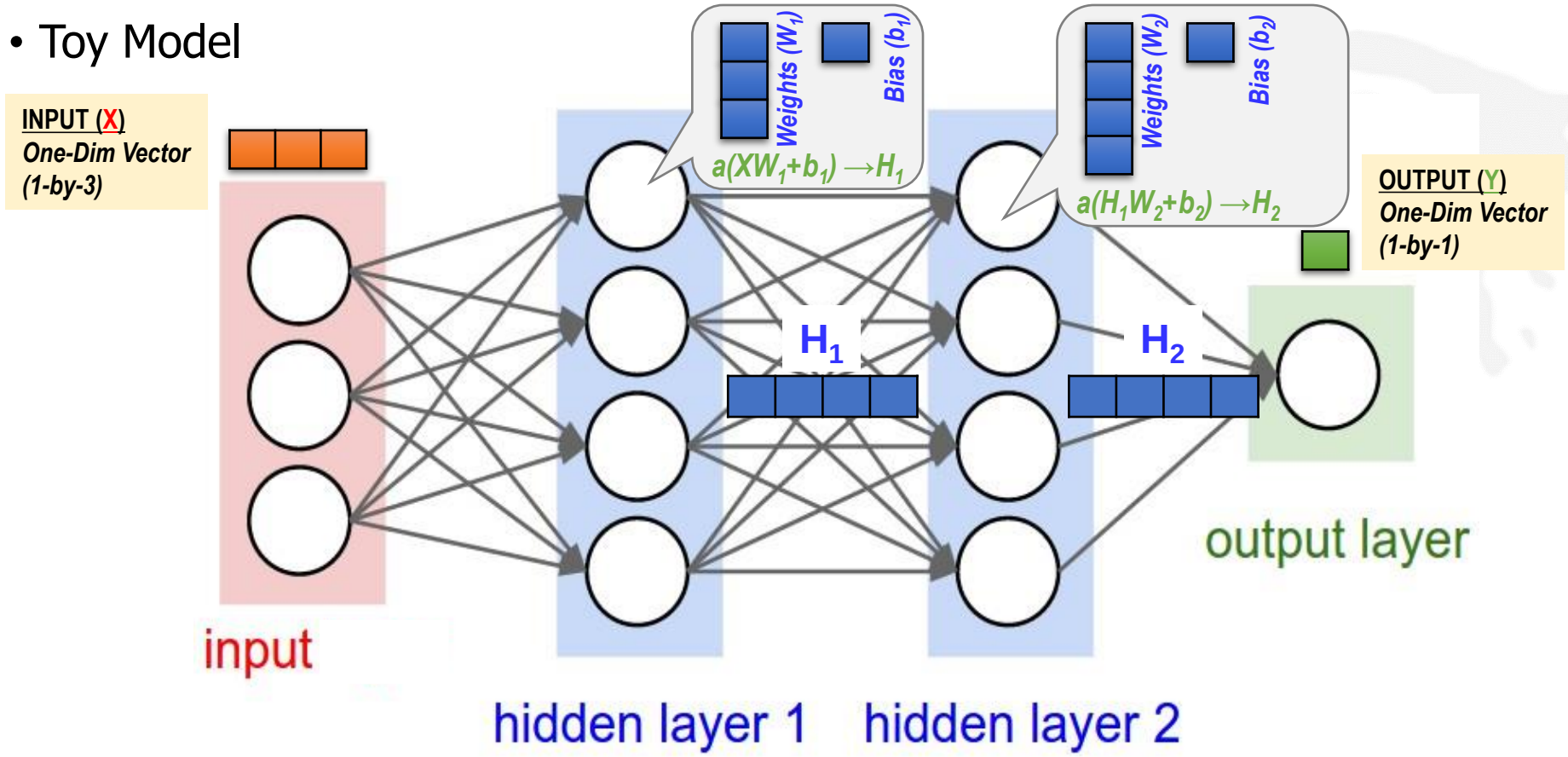
OUTPUT (Y)
One-Dim Vector (1-by-1)

Conventional Deep Neural Network Training and Inference

- Toy Model

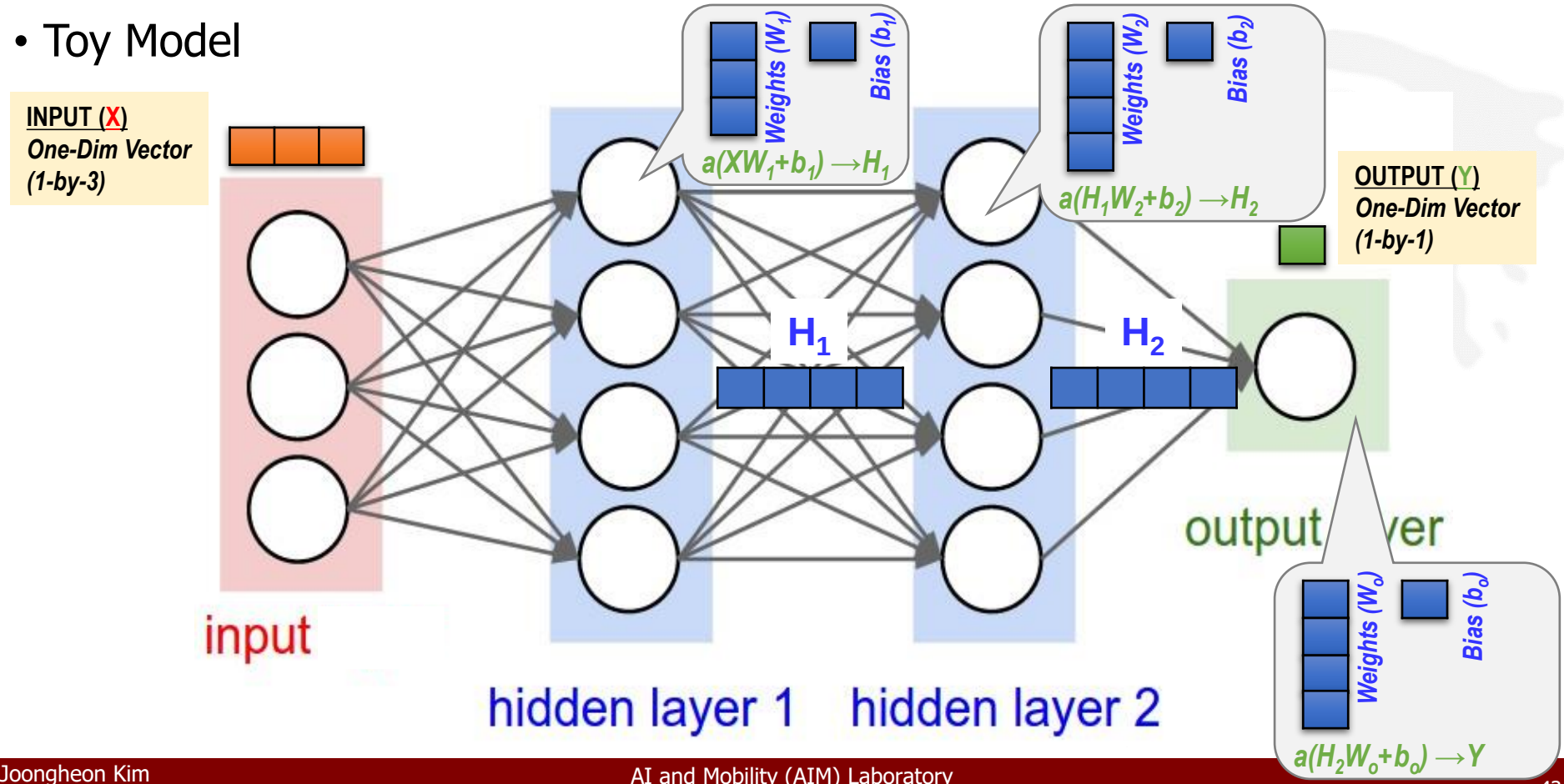


- Toy Model


INPUT (X)
One-Dim Vector
(1-by-3)
Weights (W_1)
Bias (b_1)
$a(XW_1+b_1) \rightarrow H_1$
Weights (W_2)
Bias (b_2)
$a(H_1W_2+b_2) \rightarrow H_2$
OUTPUT (Y)
One-Dim Vector
(1-by-1)
H_1
H_2
input
hidden layer 1
hidden layer 2
output layer

Conventional Deep Neural Network Training and Inference

- Toy Model

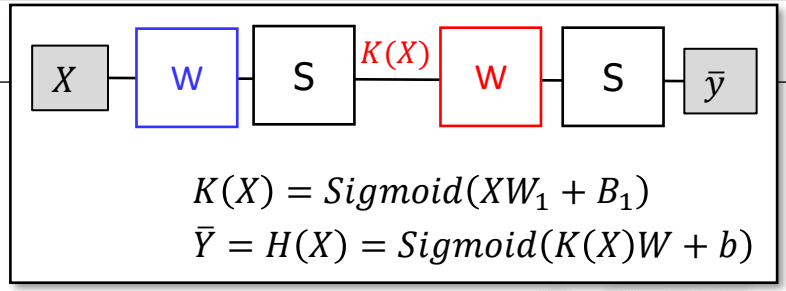


TensorFlow for ANN (XOR with Binary Classification)

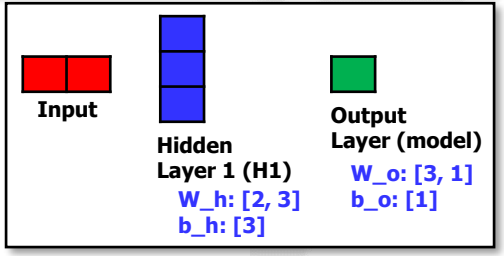
```
0 1.0826586
2000 0.6931472
4000 0.6931472
6000 0.6931472
8000 0.6931472
10000 0.6931472
12000 0.6931472
14000 0.6931472
16000 0.6931472
18000 0.6931472
20000 0.6931472
[[0.5]
 [0.5]
 [0.5]
 [0.5]] [[0.]
 [0.]
 [0.]
 [0.]] 0.5
```

```python
import tensorflow as tf

x_data = [[0,0], [0,1], [1,0], [1,1]]
y_data = [[0], [1], [1], [0]]
X = tf.placeholder(tf.float32, shape=[None, 2])
Y = tf.placeholder(tf.float32, shape=[None, 1])
W = tf.Variable(tf.random_normal([2,1]))
b = tf.Variable(tf.random_normal([1]))
model = tf.sigmoid(tf.matmul(X,W)+b)
cost = tf.reduce_mean((-1)*Y*tf.log(model) + (-1)*(1-Y)*tf.log(1-model))
train = tf.train.GradientDescentOptimizer(0.1).minimize(cost)

prediction = tf.cast(model > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, Y), dtype=tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Training
    for step in range(20001):
        c, _ = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
        if step % 2000 == 0:
            print(step, c)
    # Testing
    m, p, a = sess.run([model, prediction, accuracy], feed_dict={X: x_data, Y: y_data})
    print(m,p,a)
```

TensorFlow for ANN (XOR with ANN)



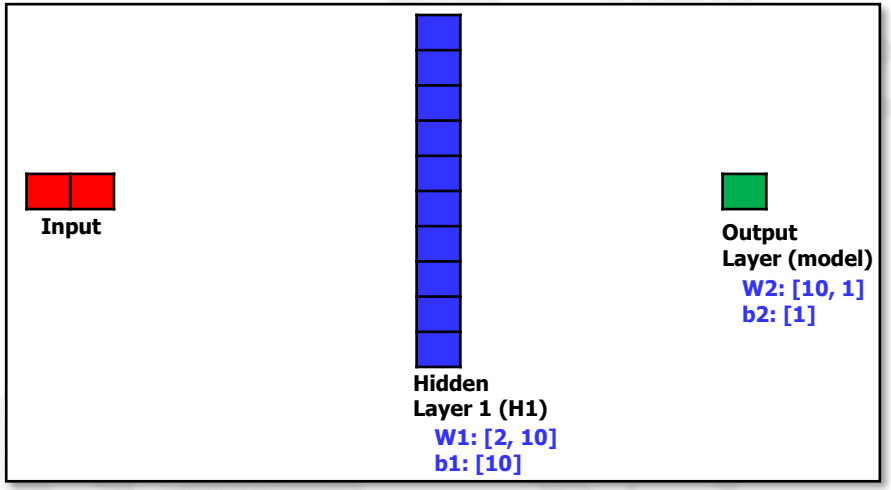

```python
import tensorflow as tf

x_data = [[0,0], [0,1], [1,0], [1,1]]
y_data = [[0], [1], [1], [0]]
X = tf.placeholder(tf.float32, shape=[None, 2])
Y = tf.placeholder(tf.float32, shape=[None, 1])
W_h = tf.Variable(tf.random_normal([2,3]))
b_h = tf.Variable(tf.random_normal([3]))
H1 = tf.sigmoid(tf.matmul(X,W_h)+b_h)
W_o = tf.Variable(tf.random_normal([3,1]))
b_o = tf.Variable(tf.random_normal([1]))
model = tf.sigmoid(tf.matmul(H1,W_o)+b_o)
cost = tf.reduce_mean((-1)*Y*tf.log(model) + (-1)*(1-Y)*tf.log(1-model))
train = tf.train.GradientDescentOptimizer(0.1).minimize(cost)

prediction = tf.cast(model > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, Y), dtype=tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Training
    for step in range(20001):
        c, _ = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
        if step % 2000 == 0:
            print(step, c)
    # Testing
    m, p, a = sess.run([model, prediction, accuracy], feed_dict={X: x_data, Y: y_data})
    print(m,p,a)
```




```
0 0.86801255
2000 0.27334553
4000 0.046823796
6000 0.02246793
8000 0.0143708335
10000 0.010443565
12000 0.008153362
14000 0.0066633224
16000 0.0056207716
18000 0.0048525333
20000 0.004264111
[[0.00480547]
 [0.99502313]
 [0.9966924 ]
 [0.00392833]] [[0.]
 [1.]
 [1.]
 [0.]] 1.0
```

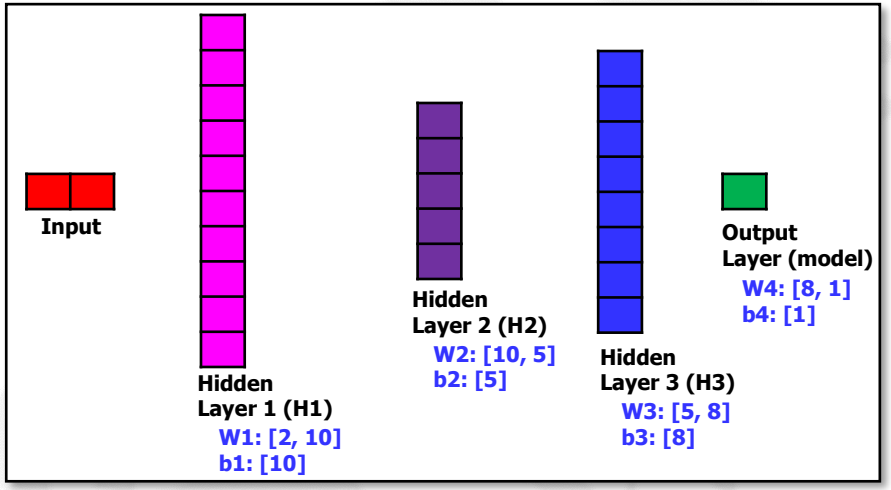
- **Wide** ANN for XOR

```
W1 = tf.Variable(tf.random_normal([2, 10]))
b1 = tf.Variable(tf.random_normal([10]))
H1 = tf.sigmoid(tf.matmul(X, W1) + b1)

W2 = tf.Variable(tf.random_normal([10, 1]))
b2 = tf.Variable(tf.random_normal([1]))
model = tf.sigmoid(tf.matmul(H1, W2) + b2)
```

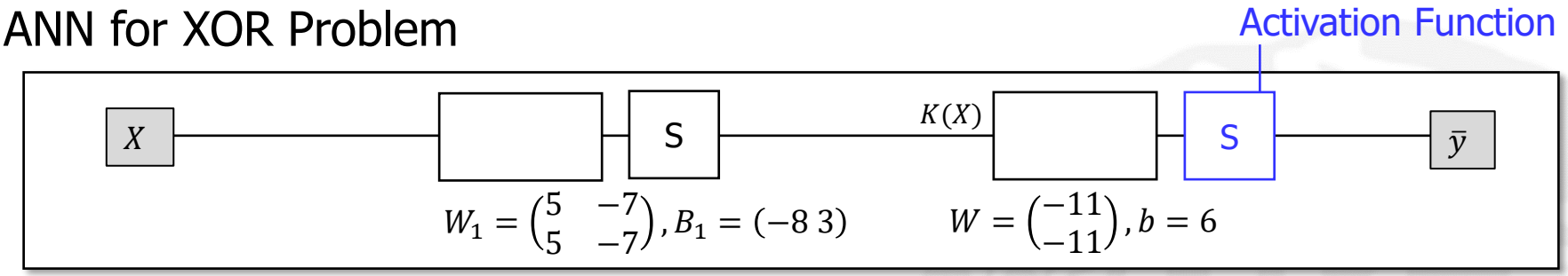


- **Deep** ANN for XOR

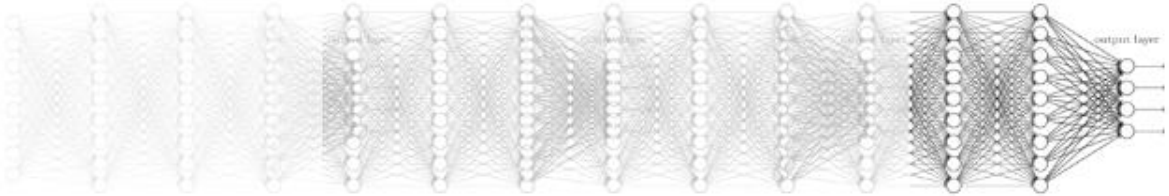
```
W1 = tf.Variable(tf.random_normal([2, 10]))
b1 = tf.Variable(tf.random_normal([10]))
H1 = tf.sigmoid(tf.matmul(X, W1) + b1)

W2 = tf.Variable(tf.random_normal([10, 5]))
b2 = tf.Variable(tf.random_normal([5]))
H2 = tf.sigmoid(tf.matmul(H1, W2) + b2)

W3 = tf.Variable(tf.random_normal([5, 8]))
b3 = tf.Variable(tf.random_normal([8]))
H3 = tf.sigmoid(tf.matmul(H2, W3) + b3)

W4 = tf.Variable(tf.random_normal([8, 1]))
b4 = tf.Variable(tf.random_normal([1]))
model = tf.sigmoid(tf.matmul(H3, W4) + b4)
```



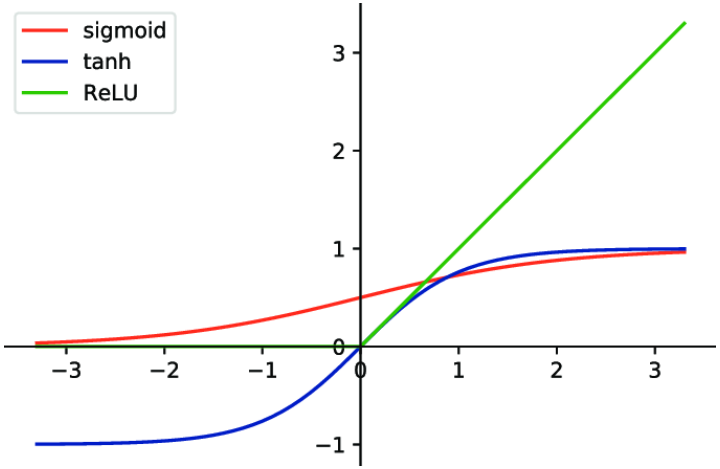
- ANN for XOR Problem

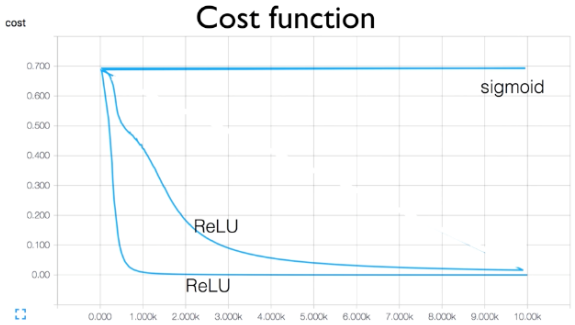


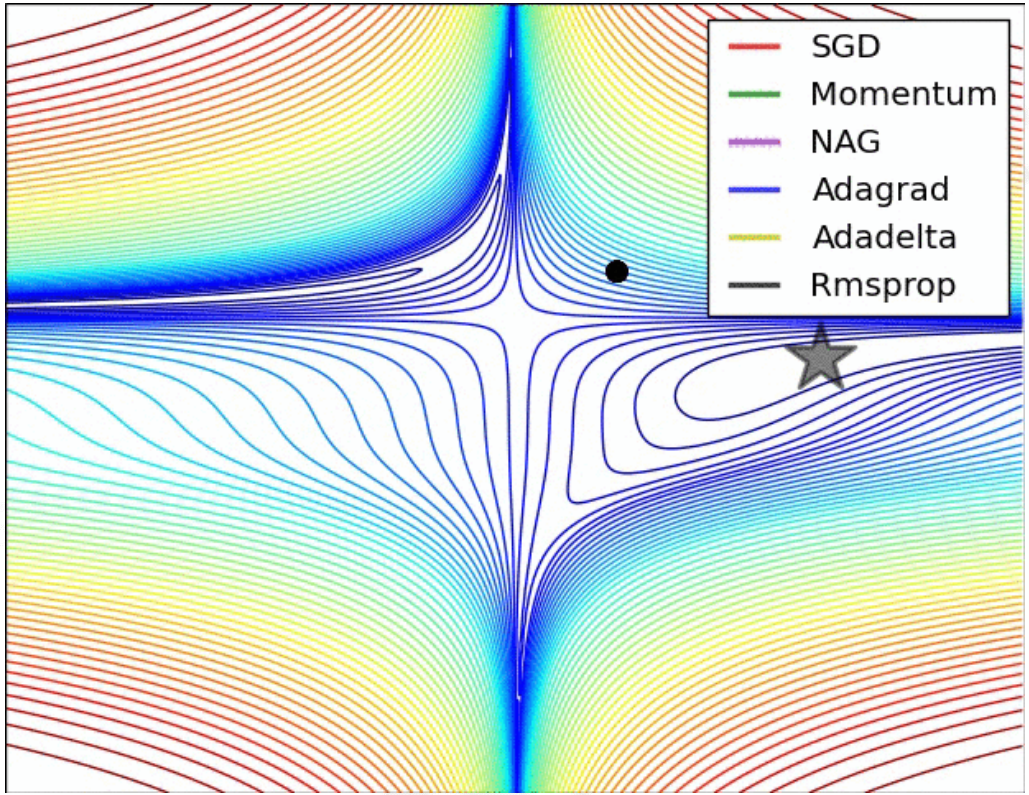
- Observation)
  - There exists cases when the accuracy is low even if the # layers is high. Why?
  - Answer)
    - The result of one ANN is the result of sigmoid function (**between 0 and 1**).
    - The numerous multiplication of this result converges to near zero.
      → **Gradient Vanishing Problem**



ANN: ReLU (Rectified Linear Unit)







- Introduction
  - Theory
  - TensorFlow
- **Generative Adversarial Network**
- Reinforcement Learning
- Deep Reinforcement Learning

GAN and Reinforcement Learning
Generative Adversarial Networks (GAN)
GAN Theory

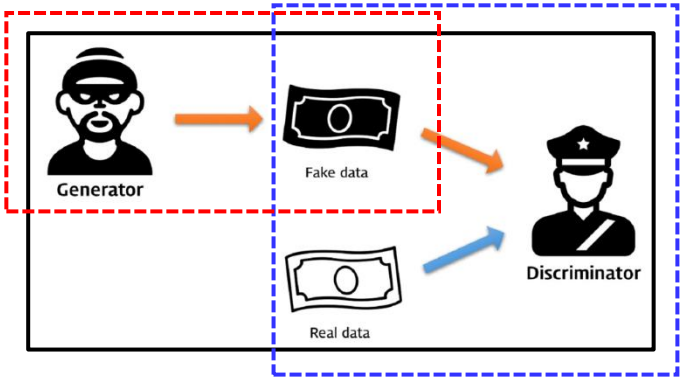
- **GAN Theory**
- GAN Implementation

- GAN: Generative Adversarial Network
- Training both of **generator** and **discriminator**; and then generates samples which are similar to the original samples

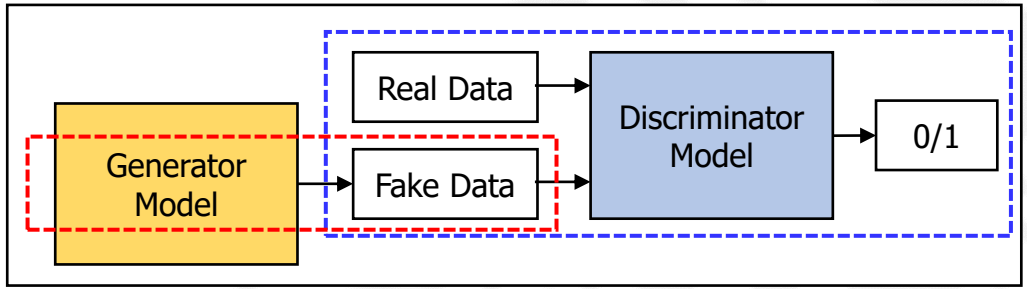
Generators

Performance Improvements via Competition

Discriminator



GAN architecture



Discriminator Model

- The discriminative model learns **how to classify** input to its class (fake → fake class, real → real class).
- Binary classifier.

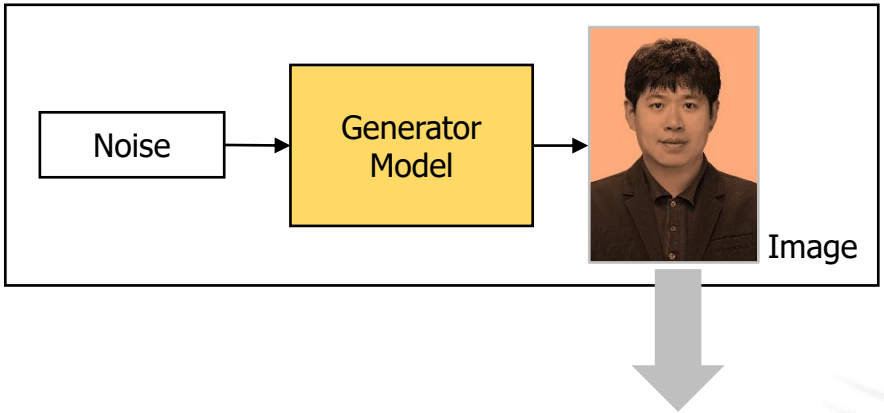
Supervised Learning

Generator Model

- The generative model learns **the distribution of training data**.

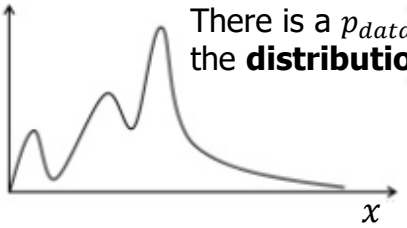
Unsupervised Learning

- Generative Model



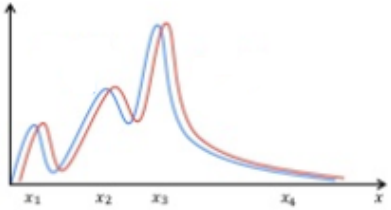
The generator model learns **the distribution of training data**.

$p_{data}(x)$
Probabilistic Density Function

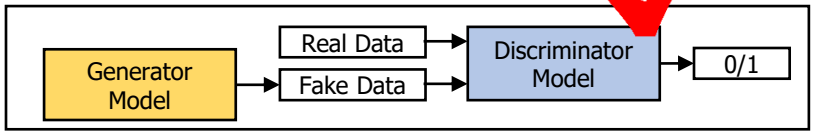


There is a $p_{data}(x)$ that represents the **distribution of actual images (training data)**.

The goal of the generative model is to find a $p_{model}(x)$ that approximates $p_{data}(x)$ well.



GAN architecture



Objective of D

Sample x from real data distribution

Sample latent code x from Gaussian distribution

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[log\ D(x)] + E_{z\sim p_z(z)}[log(1 - D(G(z))]$$

D should maximize V(D,G)

Maximize when D(x)=1

Maximize when D(G(z))=0

Training with REAL

x

Real Image

NN

D

D(x)

It should be high (close to 1).

The probability of that x came from the real data (0~1).

This D should classify a real image as real.

Training with FAKE

Fake image generated by G.

Random Noise (Latent Code)

z

NN

G

Fake Image

NN

D

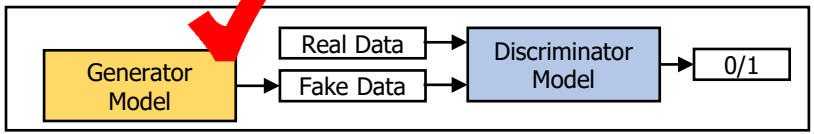
D(G(z))

Maybe low.

The G should create an image that is indistinguishable from real to receive D.

This D should classify a fake image as fake.

GAN architecture



Objective of G

G is independent to this part

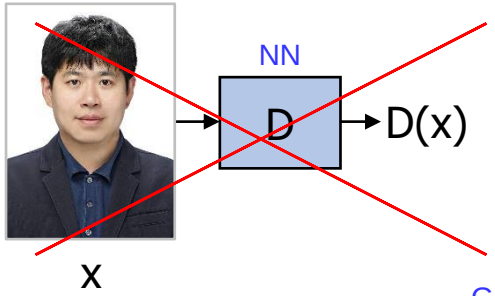
Sample latent code x from Gaussian distribution

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[log\ D(x)] + E_{z\sim p_z(z)}[log(1-D(G(z))]$$

G should minimize V(D,G)

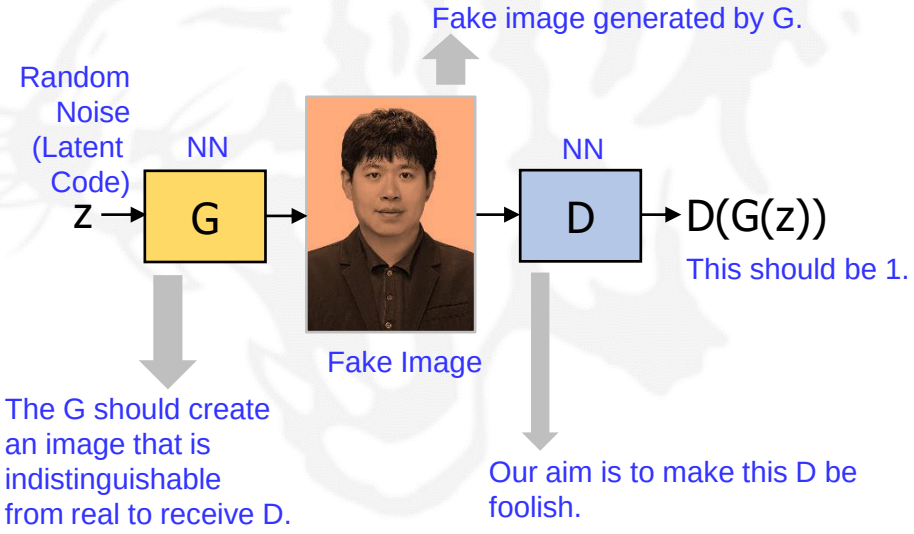
Minimum when D(G(z))=1

Training with REAL

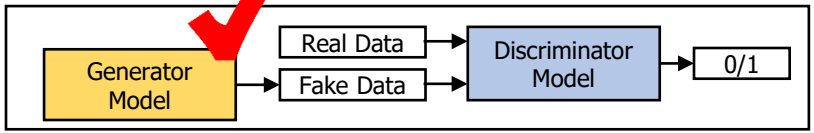


G has no inputs.

Training with FAKE



GAN architecture



Objective of G

G is independent to this part

Sample latent code x from Gaussian distribution

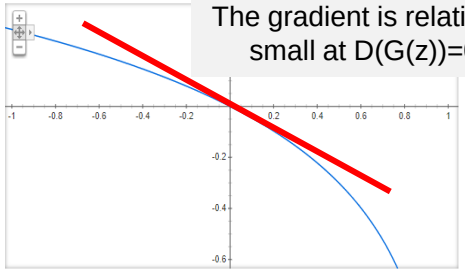
$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[log\ D(x)] + E_{z\sim p_z(z)}[log(1-D(G(z))]$$

G should minimize V(D,G)

Minimum when D(G(z))=1

- At the beginning of training, the D can clearly classify the generated image as fake because the quality of the image is very low.
- This means D(G(z)) is almost zero at early stage of training.

Graph for log(1-x)



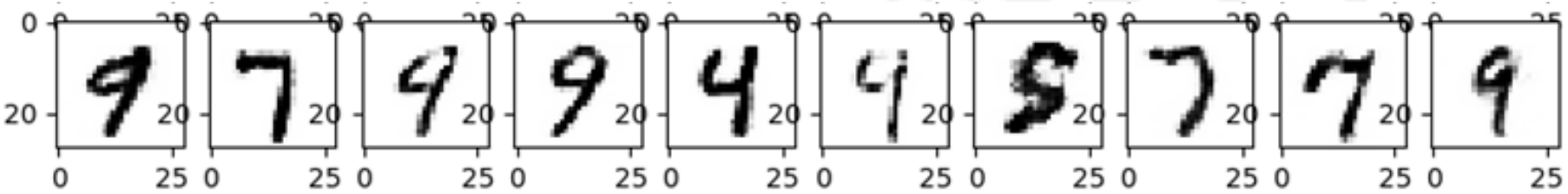
The gradient is relatively small at D(G(z))=0.

$$\min_G E_{z\sim p_z(z)}[\log(1-D(G(z))]$$

$$\max_G E_{z\sim p_z(z)}[\log(D(G(z))]$$

GAN and Reinforcement Learning
Generative Adversarial Networks (GAN)
GAN Implementation

- GAN Theory
- **GAN Implementation**


0
20
0
25

```python
# MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("data_MNIST", one_hot=True)

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

# Training Params
num_steps = 100000
batch_size = 128

# Network Params
dim_image = 784 # 28*28 pixels
nHL_G = 256
nHL_D = 256
dim_noise = 100 # Noise data points

# A custom initialization (Xavier Glorot init)
def glorot_init(shape):
    return tf.random_normal(shape=shape, stddev=1. / tf.sqrt(shape[0] / 2.))
```

```
W = {
    'HL_G' : tf.Variable(glorot_init([dim_noise, nHL_G])),
    'OL_G' : tf.Variable(glorot_init([nHL_G, dim_image])),
    'HL_D' : tf.Variable(glorot_init([dim_image, nHL_D])),
    'OL_D' : tf.Variable(glorot_init([nHL_D, 1])),
}
b = {
    'HL_G' : tf.Variable(tf.zeros([nHL_G])),
    'OL_G' : tf.Variable(tf.zeros([dim_image])),
    'HL_D' : tf.Variable(tf.zeros([nHL_D])),
    'OL_D' : tf.Variable(tf.zeros([1])),
}

# Neural Network: Generator
def nn_G(x):
    HL = tf.nn.relu(tf.add(tf.matmul(x, W['HL_G']), b['HL_G']))
    OL = tf.nn.sigmoid(tf.add(tf.matmul(HL, W['OL_G']), b['OL_G']))
    return OL

# Neural Network: Discriminator
def nn_D(x):
    HL = tf.nn.relu(tf.add(tf.matmul(x, W['HL_D']), b['HL_D']))
    OL = tf.nn.sigmoid(tf.add(tf.matmul(HL, W['OL_D']), b['OL_D']))
    return OL

# Network Inputs
IN_G = tf.placeholder(tf.float32, shape=[None, dim_noise])
IN_D = tf.placeholder(tf.float32, shape=[None, dim_image])
```

```
52  # Build Generator Neural Network
53  sample_G = nn_G(IN_G)
54
55  # Build Discriminator Neural Network (one from noise input, one from generated samples)
56  D_real = nn_D(IN_D)
57  D_fake = nn_D(sample_G)
58  vars_G = [W['HL_G'], W['OL_G'], b['HL_G'], b['OL_G']]
59  vars_D = [W['HL_D'], W['
60
61  # Cost, Train
62  cost_G = -tf.reduce_mean(tf.log(D_fake))
63  cost_D = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
64  train_G = tf.train.AdamOptimizer(0.0002).minimize(cost_G, var_list=vars_G)
65  train_D = tf.train.AdamOptimizer(0.0002).minimize(cost_D, var_list=vars_D)
```

G(z)

D(x)

D(G(z))

Objective of G

$$\max_G E_{z\sim p_z(z)}[\log(D(G(z))]$$

Objective of D

Sample x from real data distribution

Sample latent code x from Gaussian distribution

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[log\ D(x)] + E_{z\sim p_z(z)}[log(1-D(G(z))]$$

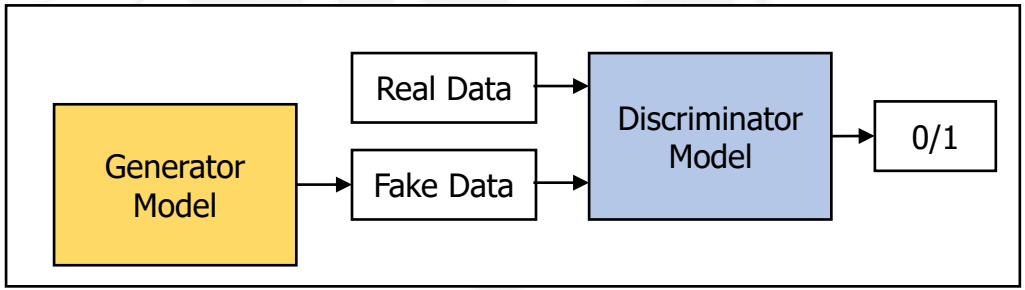
D should maximize V(D,G)

Maximize when D(x)=1

Maximize when D(G(z))=0

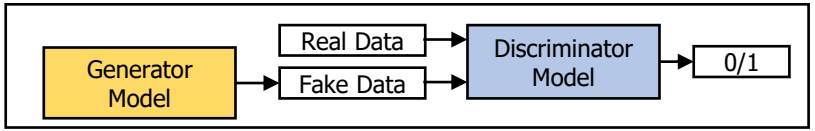
```
52  # Build Generator Neural Network
53  sample_G = nn_G(IN_G)
54
55  # Build Discriminator Neural Network (one from noise input, one from generated samples)
56  D_real = nn_D(IN_D)
57  D_fake = nn_D(sample_G)
58  vars_G = [W['HL_G'], W['OL_G'], b['HL_G'], b['OL_G']]
59  vars_D = [W['HL_D'], W['OL_D'], b['HL_D'], b['OL_D']]
60
61  # Cost, Train
62  cost_G = -tf.reduce_mean(tf.log(D_fake))
63  cost_D = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
64  train_G = tf.train.AdamOptimizer(0.0002).minimize(cost_G, var_list=vars_G)
65  train_D = tf.train.AdamOptimizer(0.0002).minimize(cost_D, var_list=vars_D)
```

GAN architecture

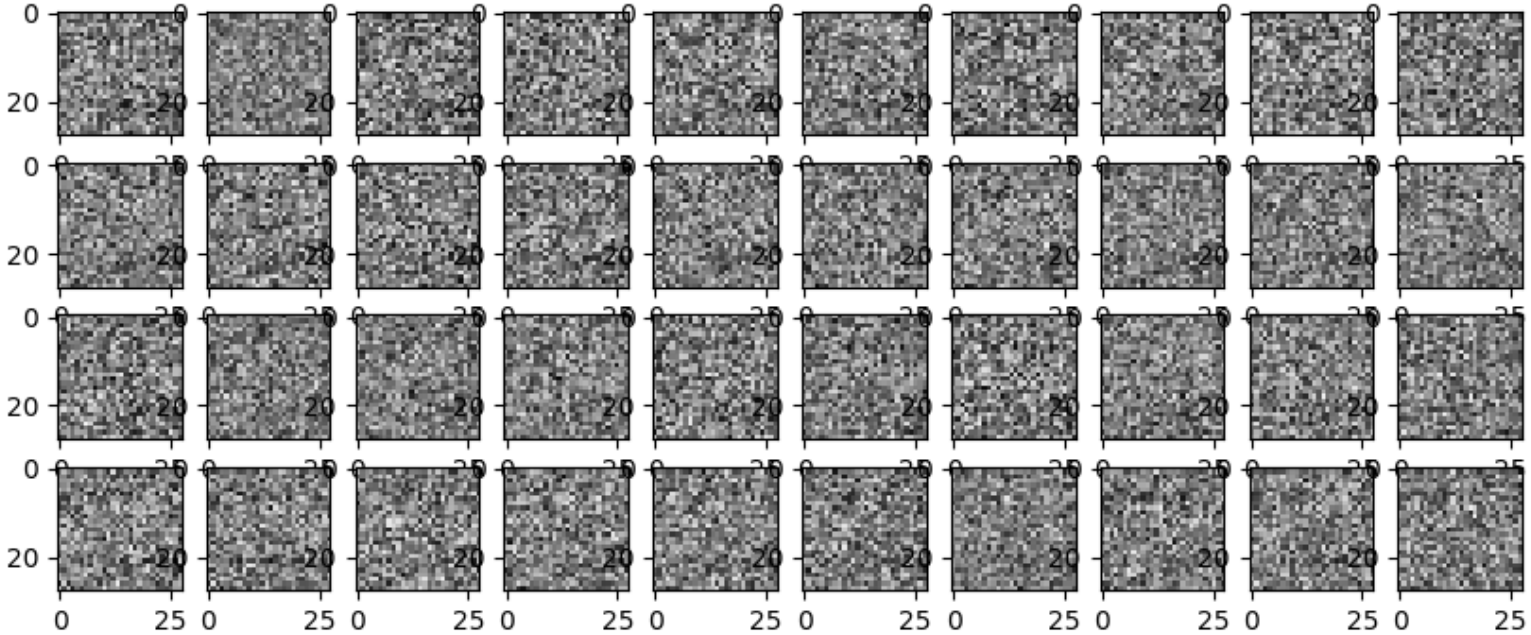


```
67  # Session
68  with tf.Session() as sess:
69      sess.run(tf.global_variables_initializer())
70      for i in range(1, num_steps+1):
71          # Get the next batch of MNIST data
72          batch_images, _ = mnist.train.next_batch(batch_size)
73          # Generate noise to feed to the generator G
74          z = np.random.uniform(-1., 1., size=[batch_size, dim_noise])
75          # Train
76          sess.run([train_G, train_D], feed_dict = {IN_D: batch_images, IN_G: z})
77          f, a = plt.subplots(4, 10, figsize=(10, 4))
78          for i in range(10):
79              z = np.random.uniform(-1., 1., size=[4, dim_noise])
80              g = sess.run([sample_G], feed_dict={IN_G: z})
81              g = np.reshape(g, newshape=(4, 28, 28, 1))
82              # Reverse colors for better display
83              g = -1 * (g - 1)
84              for j in range(4):
85                  # Generate image from noise. Extend to 3 channels for matplot figure.
86                  img = np.reshape(np.repeat(g[j][:, :, np.newaxis], 3, axis=2), newshape=(28, 28, 3)
87                  a[j][i].imshow(img)
88      f.show()
89      plt.draw()
90      plt.waitforbuttonpress()
```

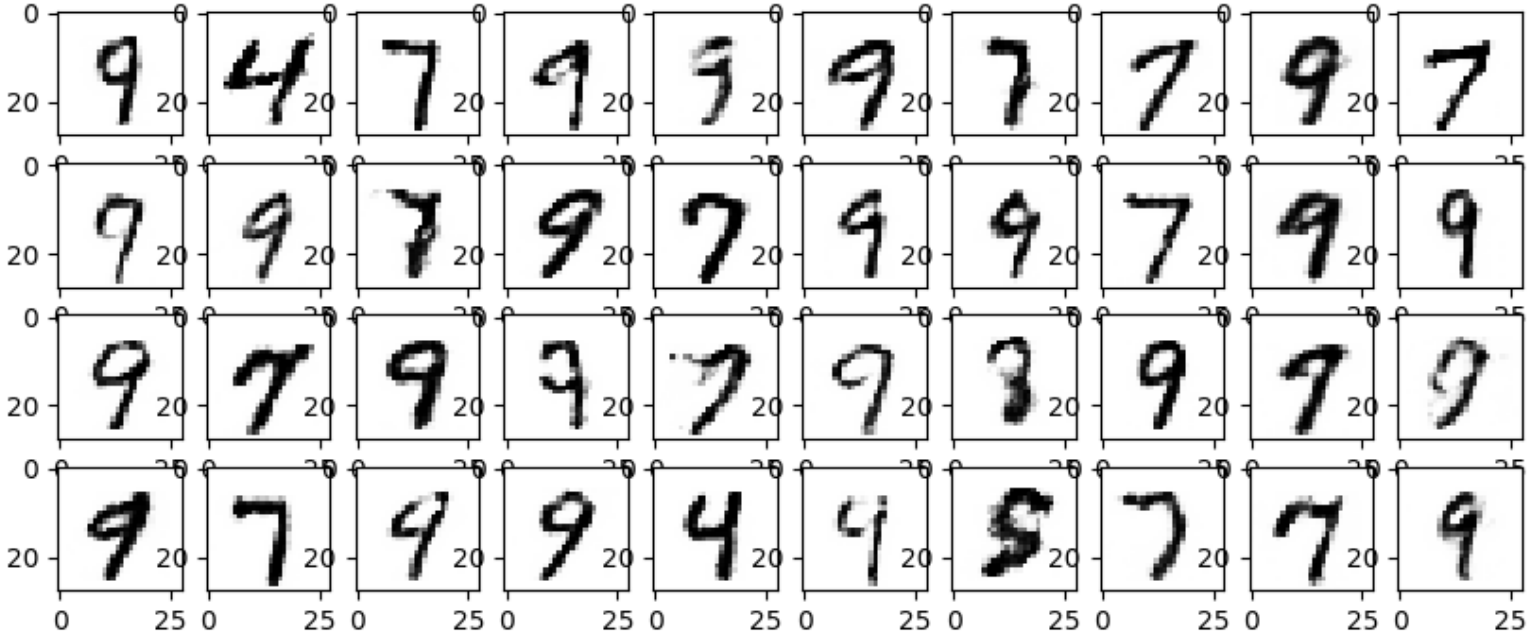
GAN architecture



num_steps: 1



num_steps: 100000



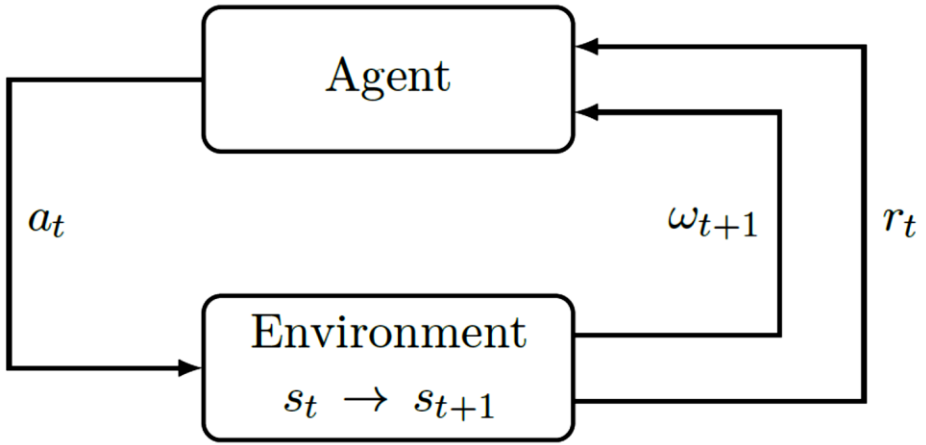
- Introduction
  - Theory
  - TensorFlow
- Generative Adversarial Network
- **Reinforcement Learning**
- Deep Reinforcement Learning

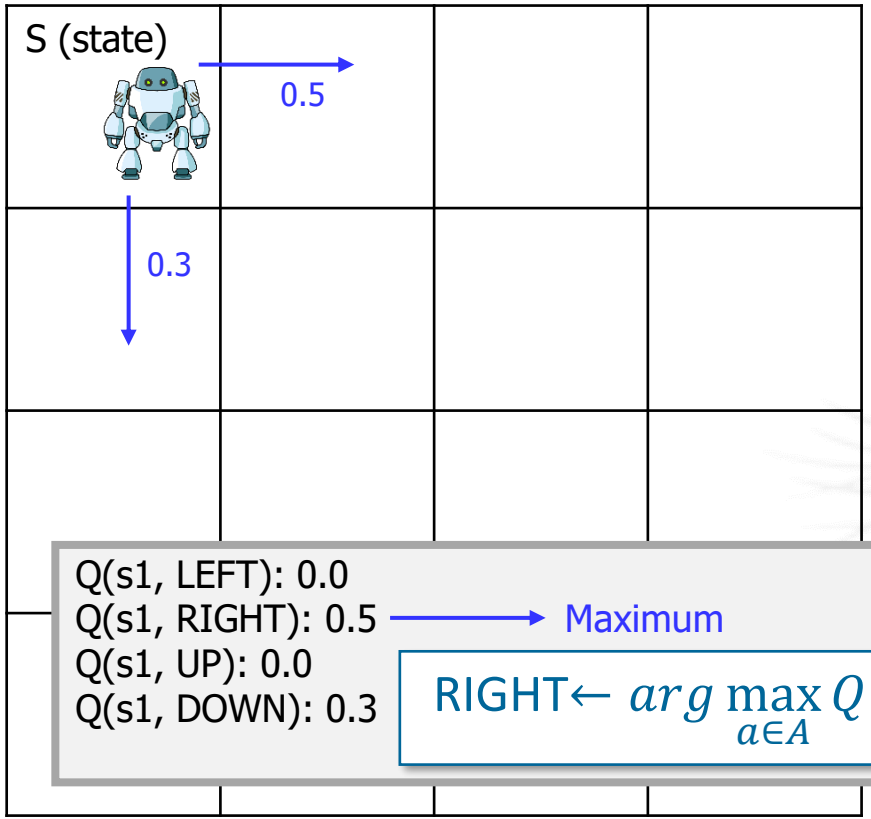
GAN and Reinforcement Learning
Reinforcement Learning (RL)
RL Theory

- **RL Theory**
- RL Implementation

- Action Sequence (also called **Policy**, later in this presentation)!

- **RL Setting**
  - The general RL problem is formalized as a **discrete time stochastic control process** where **an agent interacts with its environment** as follows:
    1. The agent starts
       in a given state within its environment $s_0 \in S$
       by gathering an initial observation $\omega_0 \in \Omega$.
    2. At each time step t,
       The agent has to take an action $a_t \in A$.
       It follows three consequences:
       1) Obtains a reward $r_t \in R$
       2) State transitions to $s_{t+1} \in S$
       3) Obtains an observation $\omega_{t+1} \in \Omega$

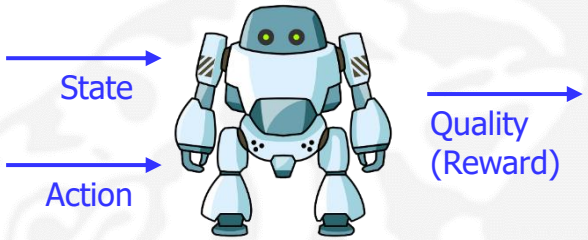




Q(s1, LEFT): 0.0
Q(s1, RIGHT): 0.5 → Maximum
Q(s1, UP): 0.0
Q(s1, DOWN): 0.3

RIGHT← $arg\max_{a \in A} Q(s_1, a)$

- Q-Function (State-action value)

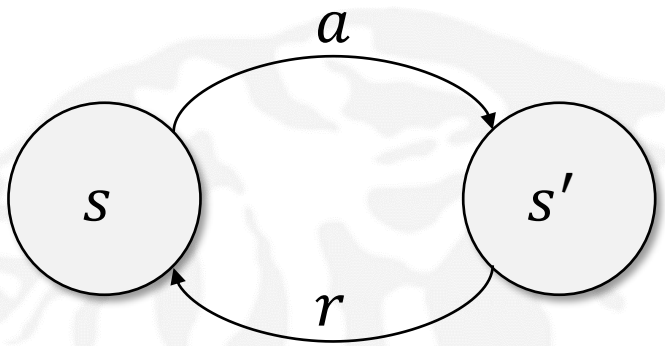


Q (state, action)

Optimal Policy π and Max Q

- Max Q = $\max_{a'} Q(s, a')$
- $\pi^*(s) = arg\max_{a} Q(s, a)$

- My condition
  - I am now in state s
  - When I do action a, I will go to s'.
  - When I do action a, I will get reward r
  - Q in s', it means $Q(s', a')$ exists.



- How can we express $Q(s, a)$ using $Q(s', a')$?

$$Q(s, a) = r + \max_{a'} Q(s', a')$$

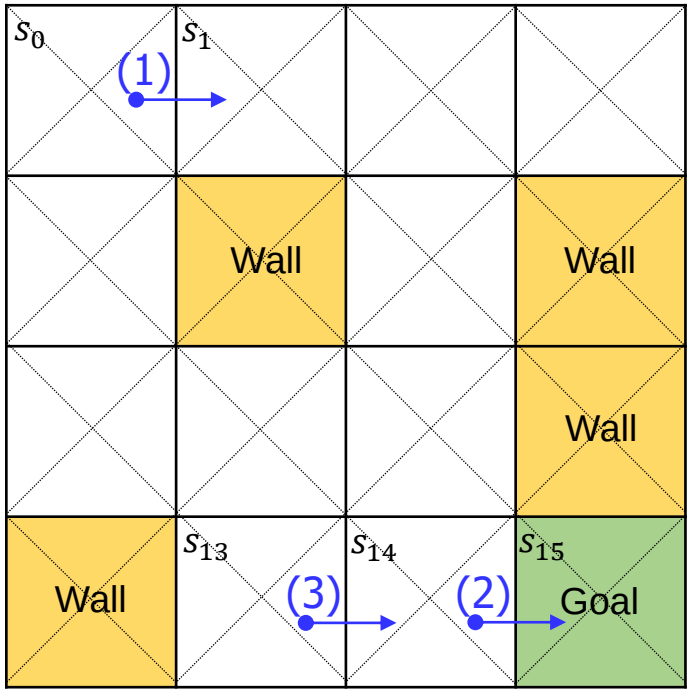
- **Solution with Dynamic Programming**
  - **Pseudo-polynomial complexity**

Recurrence (e.g., factorial)

```
F(x){
        if (x != 1){ x * F(x-1) }
        if (x == 1){ F(x) = 1 }
        }
}
```

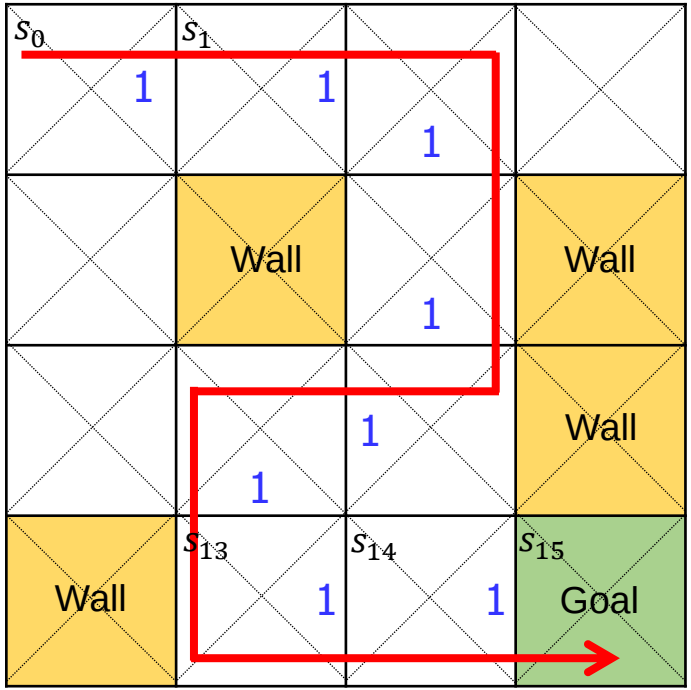
3! = F(3) = 3 * F(2)
= 3 * 2 * F(1)
= 3 * 2* 1 = 6

- 16 states and 4 actions (U, D, L, R)

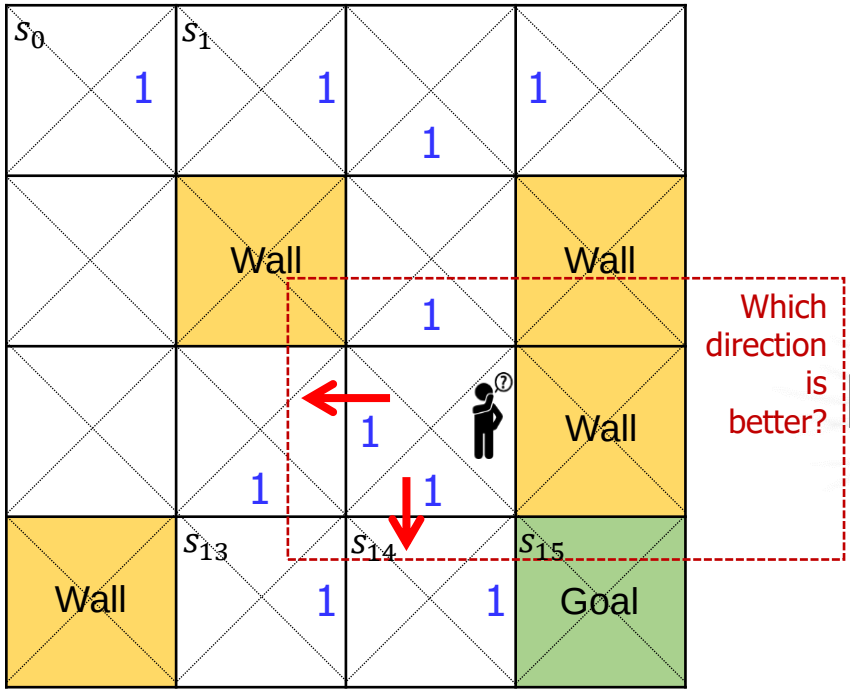


- Initial Status
  - All 64 Q values are 0,
  - Reward are all zero except $r_{s_{15},L} = 1$
- For (1), from s_0 to s_1
  - $Q(s_0, a_R) = r + \max_a Q(s_1, a) = 0 + \max\{0,0,0,0\} = 0$
- For (2), from s_{14} to s_{15} (goal)
  - $Q(s_{14}, a_R) = r + \max_a Q(s_{15}, a) = 1 + \max\{0,0,0,0\} = 1$
- For (3), from s_{13} to s_{14}
  - $Q(s_{13}, a_R) = r + \max_a Q(s_{14}, a) = 0 + \max\{0,0,1,0\} = 1$

- 16 states and 4 actions (U, D, L, R)



- 16 states and 4 actions (U, D, L, R)



Learning $Q(s,a)$ with Discounted Reward

$$Q(s,a) = r + \gamma \cdot arg\max_{a} Q(s',a')$$

$$0 < \gamma \leq 1$$

- For each s, a, initialize table entry $Q(s, a) \leftarrow 0$
- Observe current state s
- Do forever
  - Select an action a and execute it
  - Receive immediate reward r
  - Observe the new state s'
  - Update the table entry for $Q(s, a)$ as follows:

$$Q(s, a) \leftarrow r + \max_{a'} Q(s', a')$$

  - $s \leftarrow s'$

Q-Learning with Exploit and Exploration: ε-Greedy

Finding the Best Restaurant

- Try the best one during weekdays.
- Try new ones during weekends.

ε-Greedy

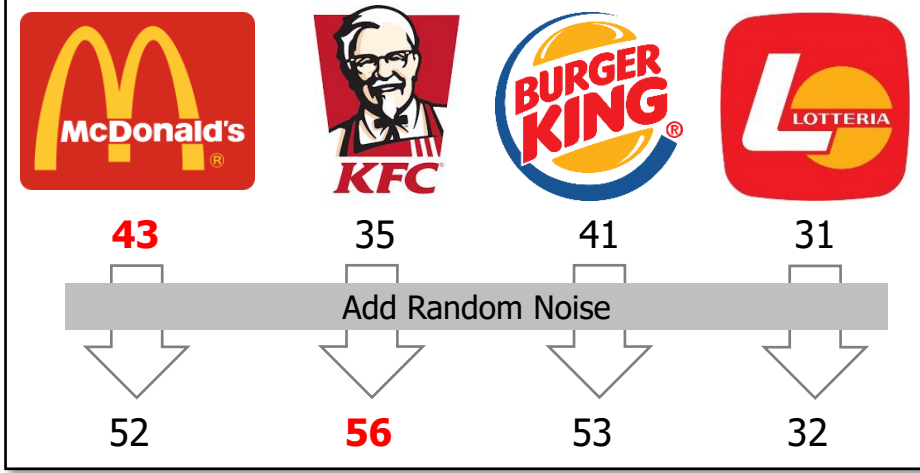
```
e=0.1
IF (random < e)
        a = random;
ELSE
        a = argmax(Q(s,a));
```

Decaying ε-Greedy

```
for i in range (1000); e=0.1 / (i+1);
IF (random < e)
        a = random;
ELSE
        a = argmax(Q(s,a));
```

Q-Learning with Exploit and Exploration: Add Random Noise

Finding the Best Restaurant



Add Random Noise

a = argmax(Q(s,a) + **random_values**);

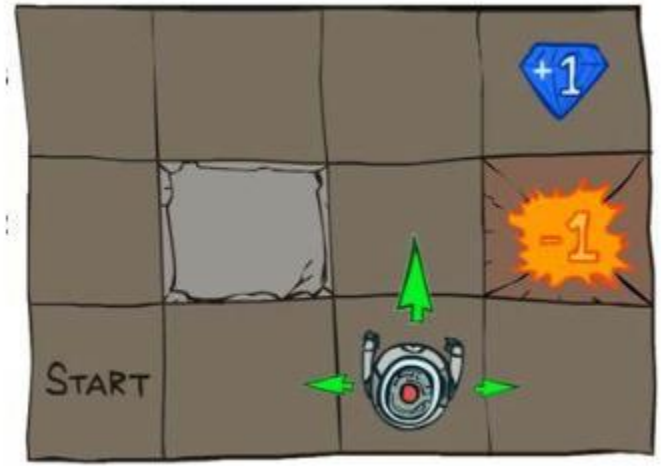
Add Decaying Random Noise

for i in range (1000);
a = argmax(Q(s,a) + random**/(i+1)**);

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - A: Set of actions
  - R: Reward function
  - T: Transition function
  - γ: Discount factor

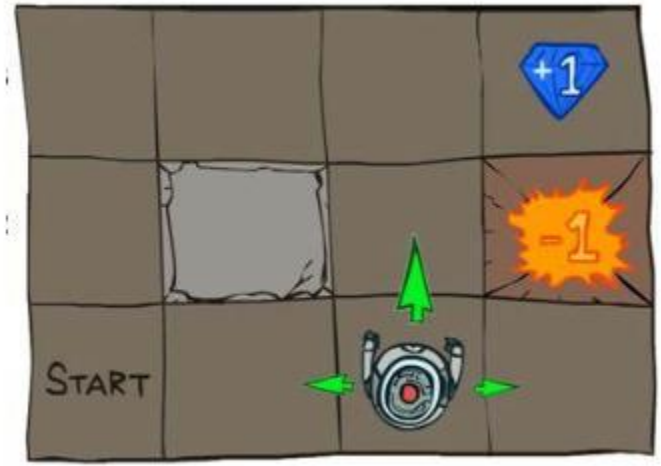
How can we use MDP to model agent in a maze?

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - **S: Set of states**
  - A: Set of actions
  - R: Reward function
  - T: Transition function
  - γ: Discount factor



S: location (x, y) if the maze is a 2D grid
- s_0: starting state
- s: current state
- s': next state
- s_t: state at time t

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - **A: Set of actions**
  - R: Reward function
  - T: Transition function
  - γ: Discount factor

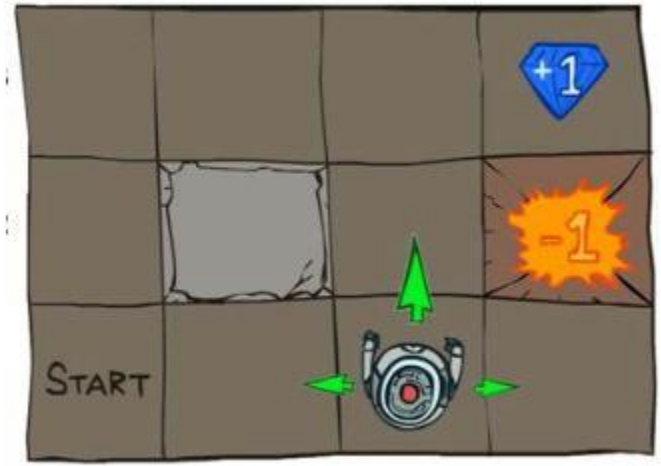


S: location (x, y) if the maze is a 2D grid

A: move up, down, left, or right

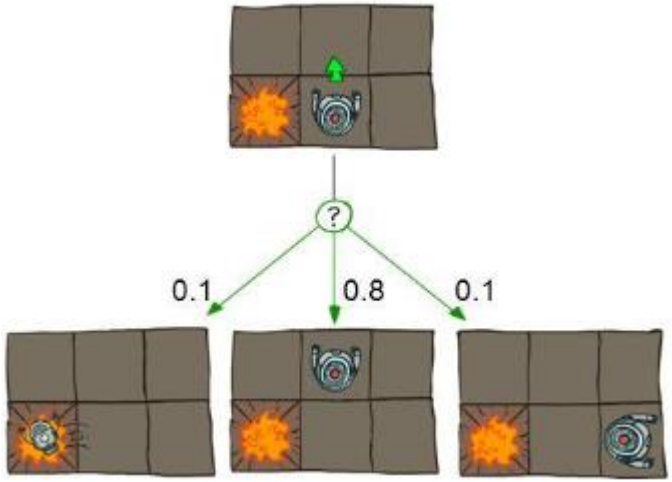
- $s \rightarrow s'$

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - A: Set of actions
  - **R: Reward function**
  - T: Transition function
  - γ: Discount factor



S: location (x, y) if the maze is a 2D grid
A: move up, down, left, or right
R: how good was the chosen action?
- $r = R(s, a, s')$
- -1 for moving (battery used)
- +1 for jewel? +100 for exit?

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - A: Set of actions
  - R: Reward function
  - **T: Transition function**
  - γ: Discount factor



Stochastic Transition

S: location (x, y) if the maze is a 2D grid
A: move up, down, left, or right
R: how good was the chosen action?
T: where is the robot's new location?
- $T = P(s'|s, a)$

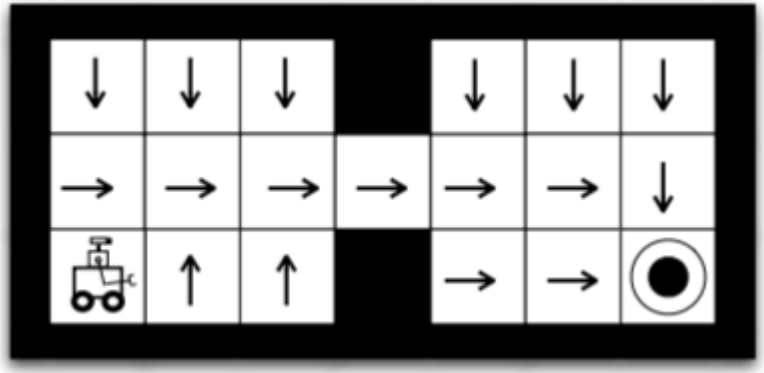
- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - A: Set of actions
  - R: Reward function
  - T: Transition function
  - **γ: Discount factor**

S: location (x, y) if the maze is a 2D grid
A: move up, down, left, or right
R: how good was the chosen action?
T: where is the robot's new location?
γ: how much does future reward worth?

- $0 \leq \gamma \leq 1$, [$\gamma \approx 0$: future reward is near 0 (immediate action is preferred)]

- Policy
  - $\pi: S \rightarrow A$
  - Maps states to actions
  - Gives an action for every state
- Return
  -

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$



Our goal:
Find π that maximizes expected return!

- State Value Function (V)

$$V^{\pi}(s) = E_{\pi}(R_t | s_t = s) = E_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s\right)$$

  - Expected return of starting at state s and following policy π
  - How much return do I expect starting from state s?

- Action Value Function (Q)

$$Q^{\pi}(s,a) = E_{\pi}(R_t | s_t = s, a_t = a) = E_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a\right)$$

  - Expected return of starting at state s, taking action a, and then following policy π
  - How much return do I expect starting from state s and taking action a?

- Our goal is to find the optimal policy

$$\pi^*(s) = \max_{\pi} R^{\pi}(s)$$

  - If $T(s'|s,a)$ and $R(s,a,s')$ are known, this is a **planning** problem.
  - We can use **dynamic programming** to find the optimal policy.

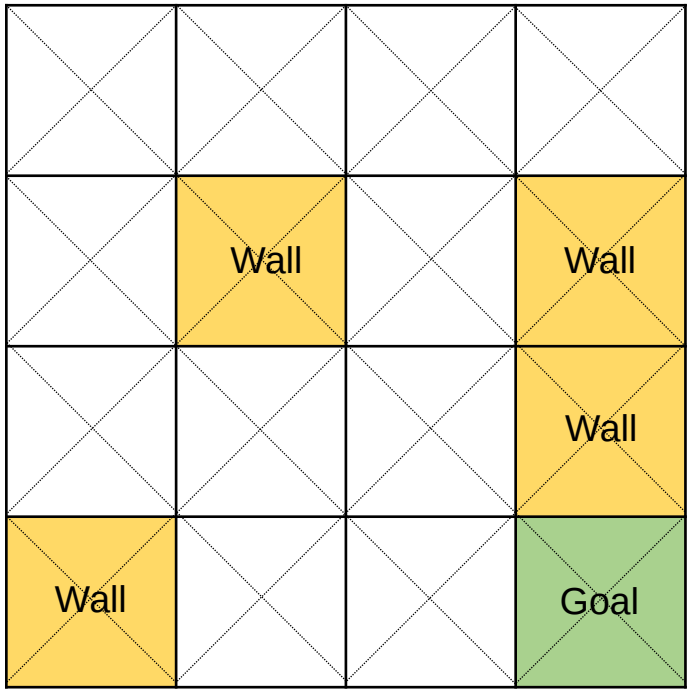
- Notes
  - Bellman Equation (Value Iteration)

$$\forall s \in S\colon V^*(s) = \max_a \sum_{s'} \{R(s,a,s') \cdot T(s,a,s') + \gamma V^*(s')\}$$

GAN and Reinforcement Learning
Reinforcement Learning (RL)
RL Implementation

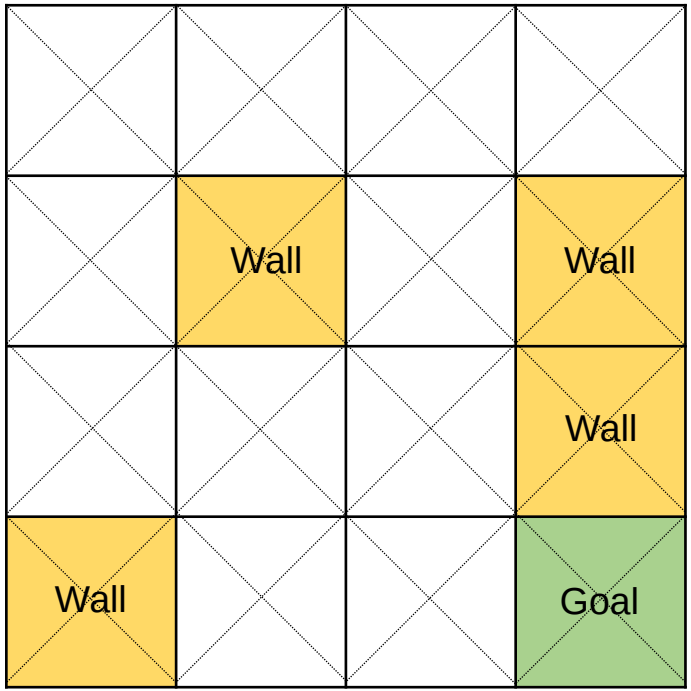
- RL Theory
- **RL Implementation**

Q-Learning (Basics)




```python
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym.envs.registration import register
import random

'''
Q-Table
        action |  L  |  D  |  R  |  U |
-----------------------------------------
state: 0       |     |     |     |    |
-----------------------------------------
state: 1       |     |     |     |    |
-----------------------------------------
state: 2       |     |     |     |    |
-----------------------------------------
state: ...     |     |     |     |    |
-----------------------------------------
'''

register(
    id='FrozenLake-v3',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={
        'map_name': '4x4',
        'is_slippery': False
    }
)

env = gym.make("FrozenLake-v3")
```

Q-Learning (Basics)




```python
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym.envs.registration import register
import random

'''
Q-Table
        action |  L  |  D  |  R  |  U |
-----------------------------------------
state: 0       |     |     |     |    |
-----------------------------------------
state: 1       |     |     |     |    |
-----------------------------------------
state: 2       |     |     |     |    |
-----------------------------------------
state: ...     |     |     |     |    |
-----------------------------------------
'''

register(
    id='FrozenLake-v3',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={
        'map_name': '4x4',
        'is_slippery': False
    }
)

env = gym.make("FrozenLake-v3")
```

- Environment setting

Q-Learning (Basics)

```
32  # Initialization with 0 in Q-table
33  Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
34  num_episodes = 1000 # Number of iterations
35
36  rList = []
37  successRate = []
38
39  def rargmax(vector):
40      m = np.amax(vector) # Return the maximum of an array or maximum along an axis (0 or 1)
41      indices = np.nonzero(vector == m)[0] # np.nonzero(True/False vector) => find the maximum
42      return random.choice(indices) # Random selection
43
44  for i in range(num_episodes): # Updates with num_episodes iterations
45      state = env.reset() # Reset
46      total_reward = 0 # Reward graph (1: success, 0: failure)
47      done = None
48
49      while not done: # The agent is not in the goal yet
50          action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
51          new_state, reward, done, _ = env.step(action) # Result of the chosen action
52
53          Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
54          total_reward += reward
55          state = new_state
56
57      rList.append(total_reward) # Reward appending
58      successRate.append(sum(rList)/(i+1)) # Success rate appending
```

```python
# Initialization with 0 in Q-table
Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
num_episodes = 1000 # Number of iterations

rList = []
successRate = []

def rargmax(vector):
    m = np.amax(vector) # Return the maximum of an array or maximum along an axis (0 or 1)
    indices = np.nonzero(vector == m)[0] # np.nonzero(True/False vector) => find the maximum
    return random.choice(indices) # Random selection

for i in range(num_episodes): # Updates with num_episodes iterations
    state = env.reset() # Reset
    total_reward = 0 # Reward graph (1: success, 0: failure)
    done = None

    while not done: # The agent is not in the goal yet
        action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
        new_state, reward, done, _ = env.step(action) # Result of the chosen action

        Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
        total_reward += reward
        state = new_state

    rList.append(total_reward) # Reward appending
    successRate.append(sum(rList)/(i+1)) # Success rate appending
```

- Randomly pick one when multiple argmax values exist

Q-Learning (Basics)

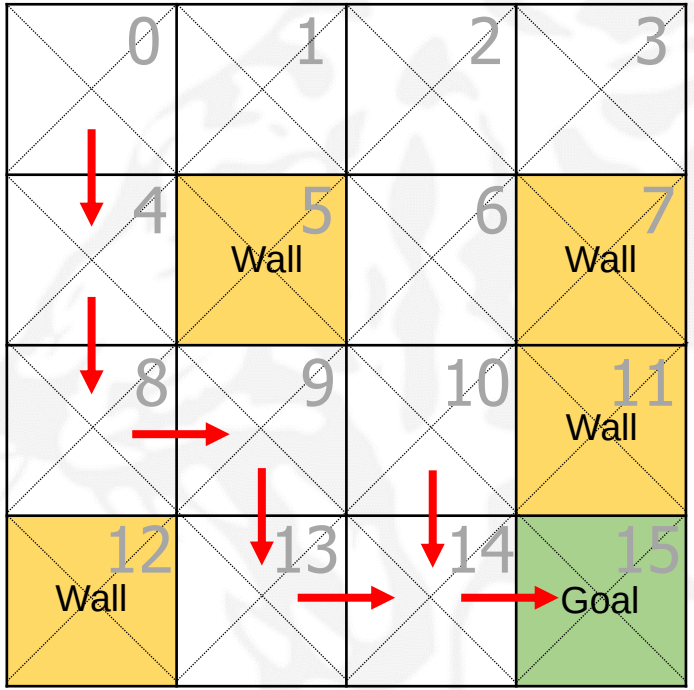
```
32  # Initialization with 0 in Q-table
33  Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
34  num_episodes = 1000 # Number of iterations
35
36  rList = []
37  successRate = []
38
39  def rargmax(vector):
40      m = np.amax(vector) # Return the max
41      indices = np.nonzero(vector == m)[0]
42      return random.choice(indices) # Rand
43
44  for i in range(num_episodes): # Updates
45      state = env.reset() # Reset
46      total_reward = 0 # Reward graph (1:
47      done = None
48
49      while not done: # The agent is not in the goal yet
50          action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
51          new_state, reward, done, _ = env.step(action) # Result of the chosen action
52
53          Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
54          total_reward += reward
55          state = new_state
56
57      rList.append(total_reward) # Reward appending
58      successRate.append(sum(rList)/(i+1)) # Success rate appending
```

- Iteration until the agent arrives at the goal or it cannot move anymore.
- (line 50) find the action which returns max Q value.
- (line 51) take the action which is the result of (line 50).
  - done: if the agent is at goal or cannot move anymore, done → True
- (line 53) Q-update
- (line 54) reward value accumulation
- (line 55) state value update for next iteration

```
'''
Final Q-Table
[[0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 0.]]
Success Rate :  0.903
'''
```

[L, D, R, U]

0 (D)
1
2
3
4 (D)
5
6
7
8 (R)
9 (D)
10 (D)
11
12
13 (R)
14 (R)
15



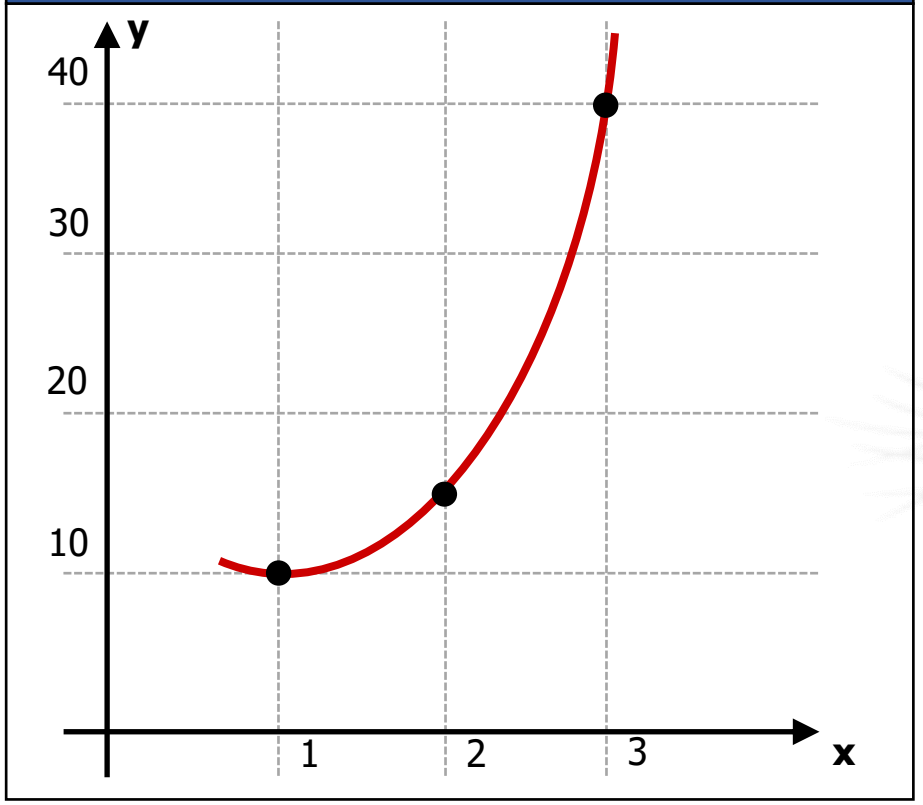
- Introduction
  - Theory
  - TensorFlow
- Generative Adversarial Network
- Reinforcement Learning
- **Deep Reinforcement Learning**

GAN and Reinforcement Learning
Deep Reinforcement Learning (DRL)
DRL Theory

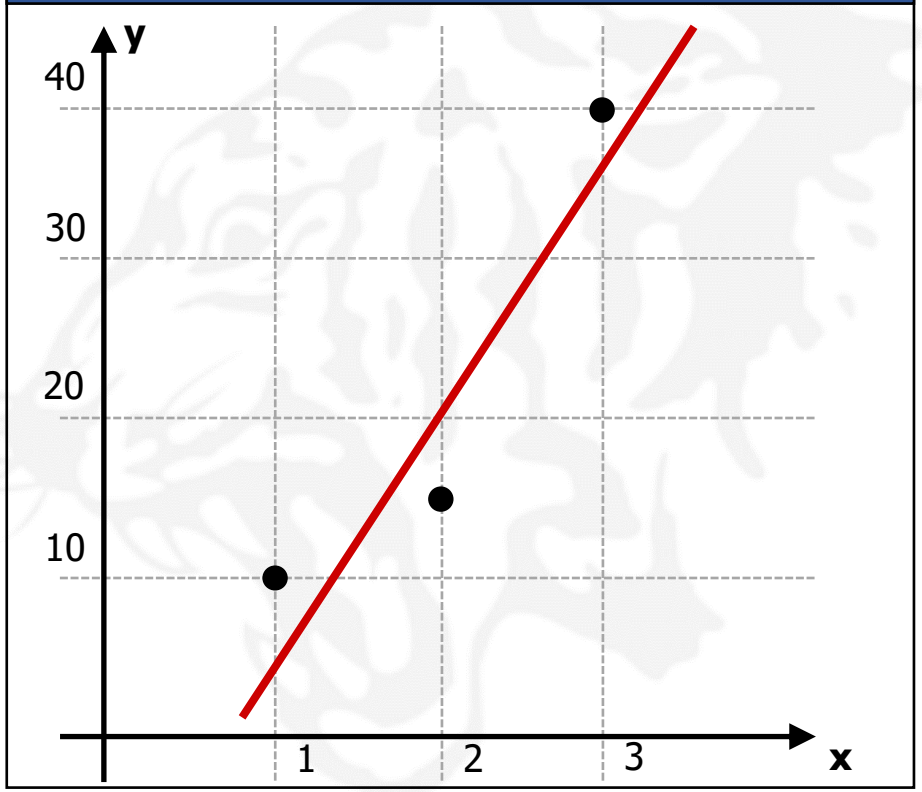
- **DRL Theory**

Interpolation vs. Linear Regression

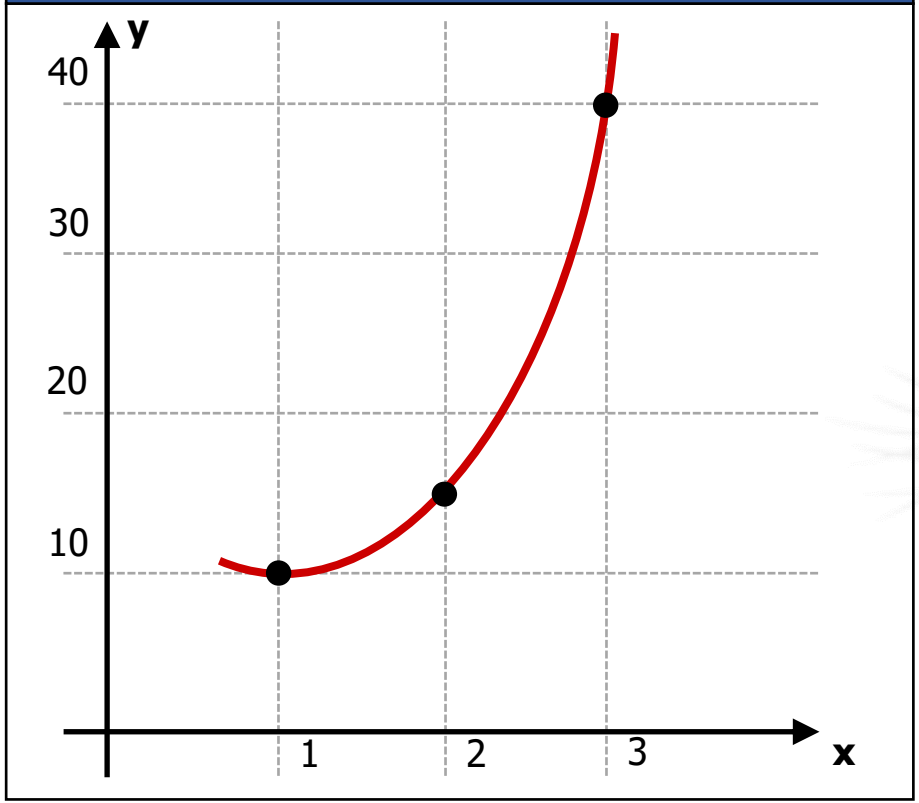
Interpolation



Linear Regression



Interpolation

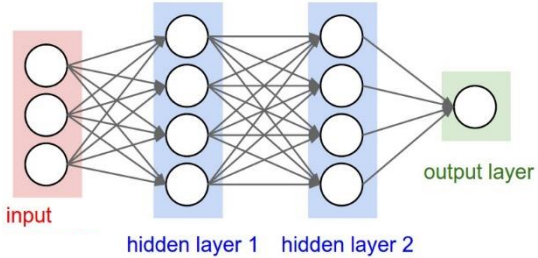


Interpolation with Polynomials

$$y = a_2 x^2 + a_1 x^1 + a_0$$

where three points are given.
→ Unique coefficients (a_0, a_1, a_2) can be calculated.

Is this related to
Neural Network Training?



$$Y = a(a(a(X \cdot W_1 + b_1) \cdot W_2 + b_2) \cdot W_o + b_o)$$

where training data/labels (X: data, Y: labels) are given.

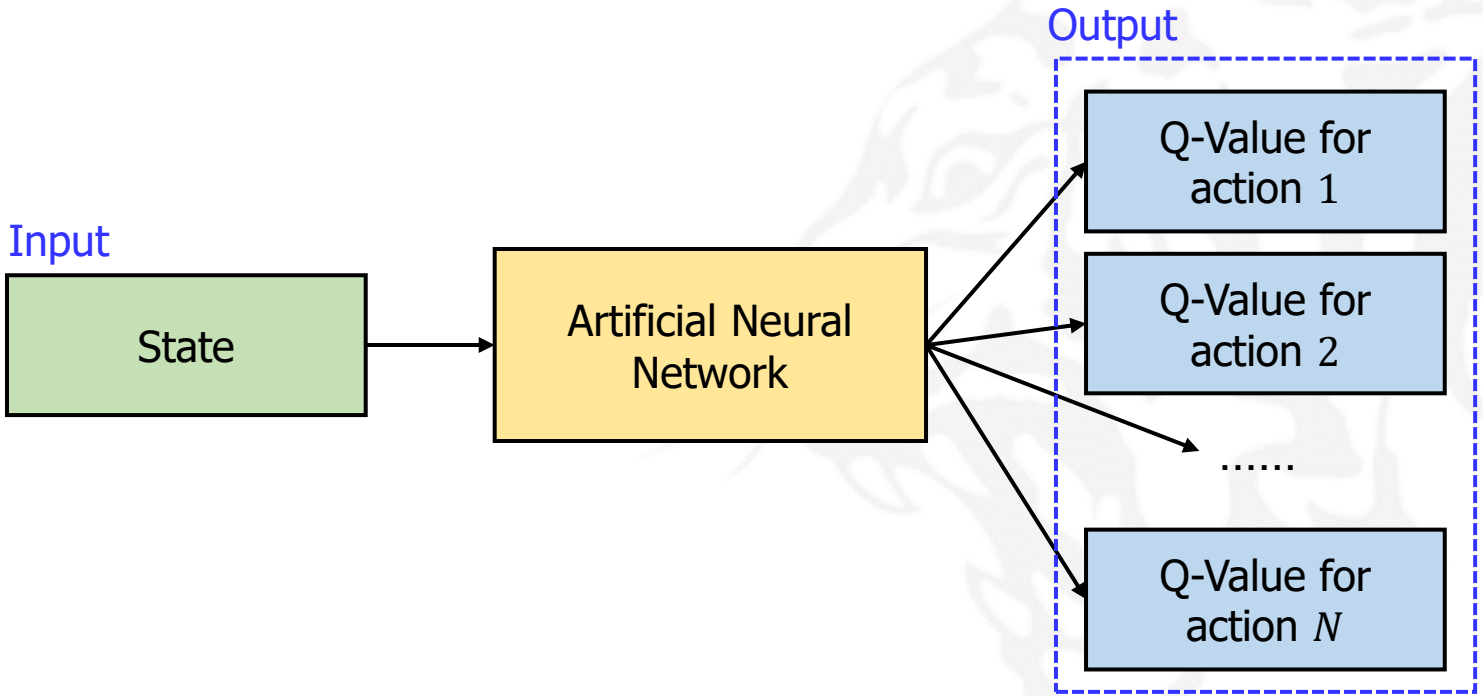
→ Find $W_1, b_1, W_2, b_2, W_o, b_o$

→ This is the mathematical meaning of neural network training.

→ **Function Approximation**

→ The most well-known function approximation with neural network: **Deep Reinforcement Learning**

- It is inefficient to make the Q-table for each state-action pair.
  → ANN is used to **approximate the Q-function**.



- ICML 2018 Tutorial
  - https://sites.google.com/view/icml2018-imitation-learning/

Imitation Learning Tutorial ICML 2018

- ICML 2019 Tutorial
  - https://slideslive.com/38917941/imitation-prediction-and-modelbased-reinforcement-learning-for-autonomous-driving

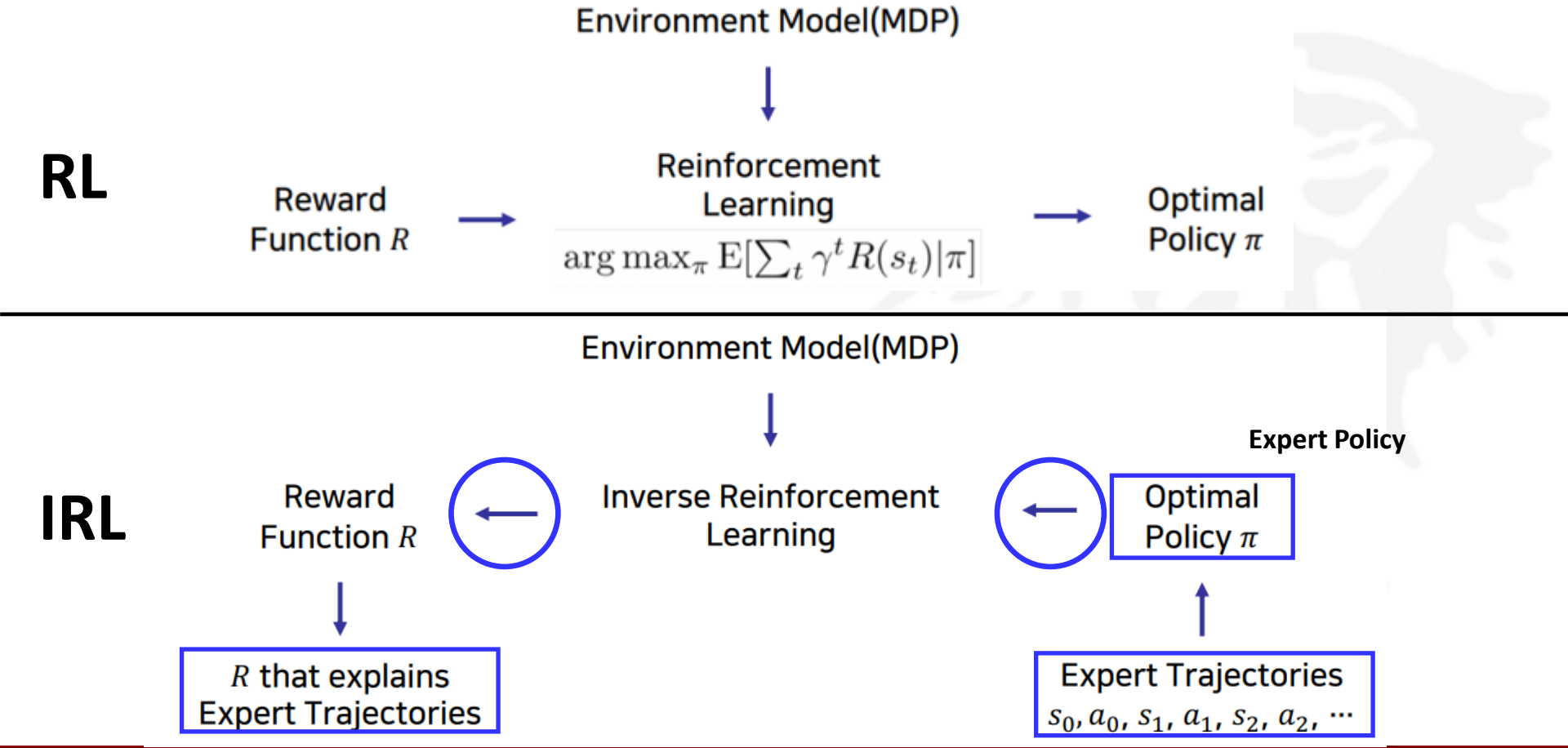
- Gameplay

Pro-Gamer

Trained Agent

The goal of Imitation Learning is to train a policy to mimic **the expert's demonstrations**

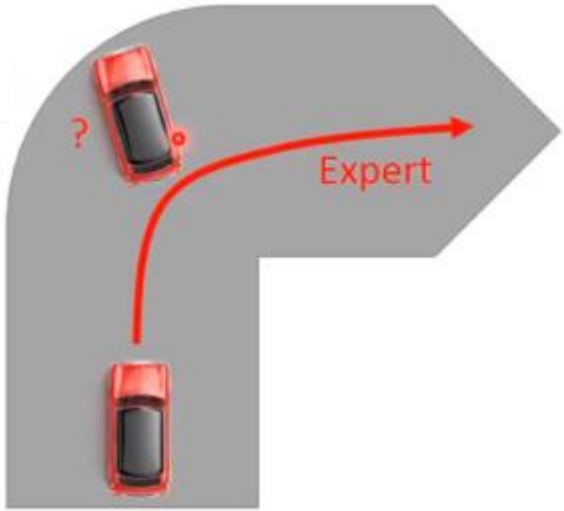
Inverse Reinforcement Learning (IRL)



- **Behavior Cloning**
  - Define $P^* = P(s|\pi^*)$ (distribution of states visited by **expert**)
  - **Learning objective**

$$argmin_{\boldsymbol{\theta}} E_{(s,a_E)\sim P^*} L(a_E, \pi_{\boldsymbol{\theta}}(s))$$
$$L\left(a_E, \pi_{\boldsymbol{\theta}}(s)\right) = \left(a_E - \pi_{\boldsymbol{\theta}}(s)\right)^2$$

  - **Discussion**
    - Works well when P^* close to the distribution of states visited by $\pi_{\boldsymbol{\theta}}$
    - **Minimize 1-step deviation error** along the expert trajectories



- Starcraft2

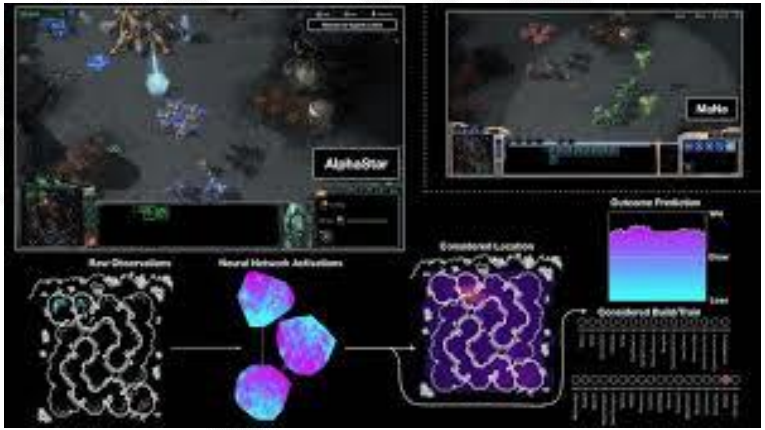
States: s = **minimap**, **screen**
Action: a **= select**, **drag**
Training set: $D = \{\tau := (s, a)\}$ from expert
Goal: learn $\pi_\theta(s) \rightarrow a$

States: s
Action: a
Policy: π_θ
- Policy maps states to actions : $\pi_\theta(s) \rightarrow a$
- Distributions over actions : $\pi_\theta(s) \rightarrow P(a)$

State Dynamics: $P(s'|s,a)$
- Typically not known to policy
- Essentially the simulator/environment

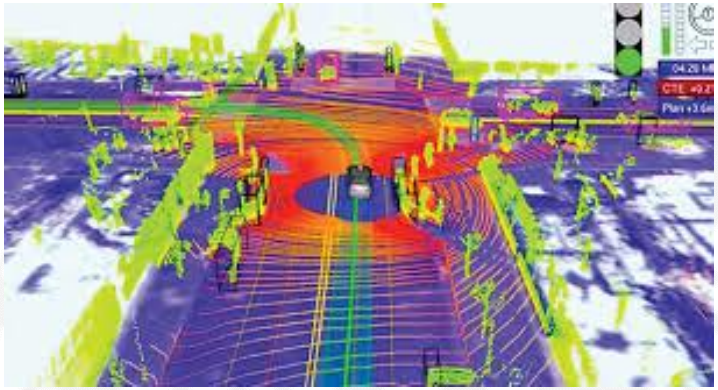
Rollout: sequentially execute $\pi_\theta(s_0)$ on initial state
- Produce trajectories $\boldsymbol{\tau}$

$\mathbf{P}(\boldsymbol{\tau}|\boldsymbol{\pi})$**:** distribution of trajectories induced by a policy
$\mathbf{P}(\mathbf{s}|\boldsymbol{\pi})$**:** distribution of states induced by a policy



- Autonomous Driving Control

States: s = **sensors**
Action: a **=** **steering wheel**, **brake**, …
Training set: $D = \{\tau := (s, a)\}$ from expert
Goal: learn $\pi_\theta(s) \rightarrow a$



- Smartphone Security

States: s = **apps**, …
Action: a **= use patterns**, …
Training set: $D = \{\tau := (s, a)\}$ from expert
Goal: learn $\pi_\theta(\mathrm{s}) \rightarrow$ a

- PPF/RFTN Injection Control in Medicine

States: s = **BIS**, **BP**, …
Action: a **= PPF**, **RFTN**, …
Training set: $D = \{\tau := (s, a)\}$ from expert
Goal: learn $\pi_\theta(\mathrm{s}) \rightarrow$ a

Thank you for your attention!

- More questions?
  - joongheon@korea.ac.kr