Deep Learning and Generative Adversarial Network (GAN)

Basics

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

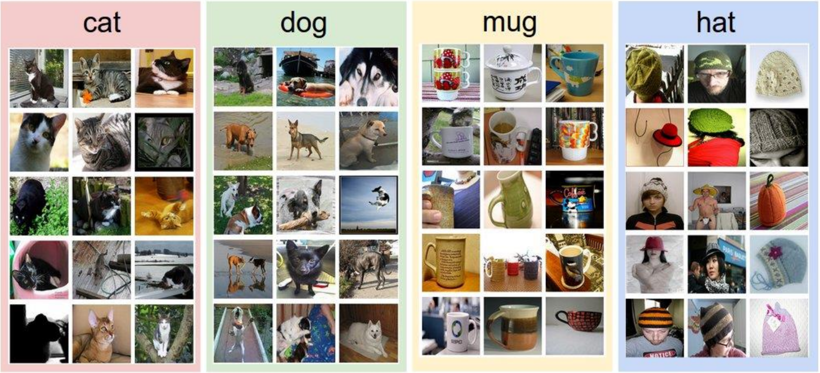
Deep Learning Basics and Software
Linear Regression

Regression (Examples)

- Exam Score Prediction (Linear Regression)

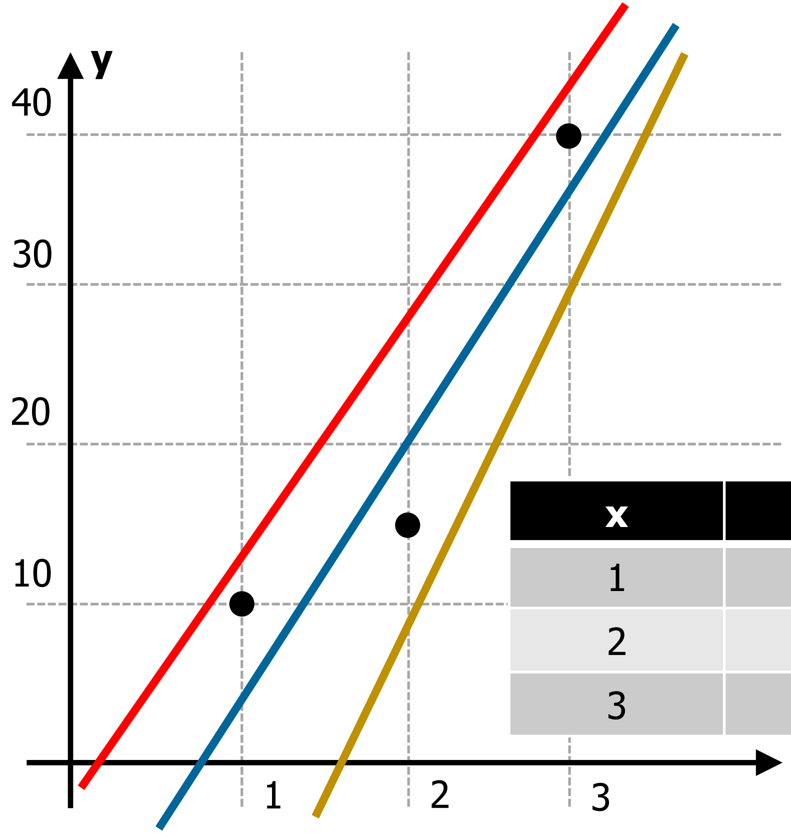
Classification (Examples)

- Pass/Fail (Binary Classification)
- Letter Grades (Multi-Level Classification)



Linear Regression

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



x	y
1	10
2	15
3	40

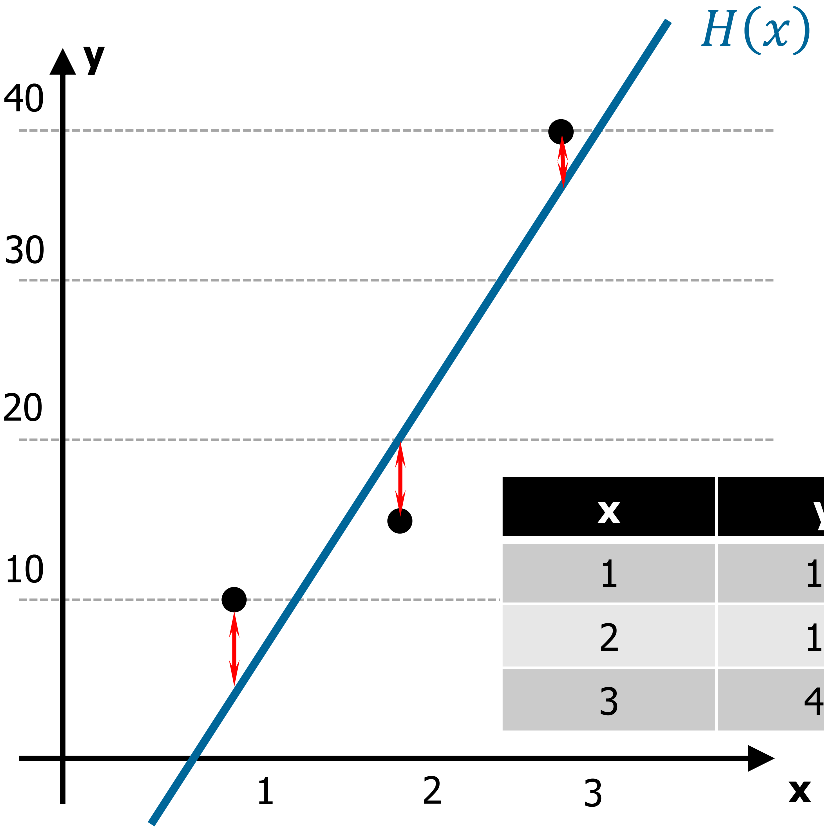
- Cost Function (or Loss Function)
  - How to fit the line to training data
  - The difference between model values and real measurements:

m: The number of training data

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

$H(x) = Wx + b$

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

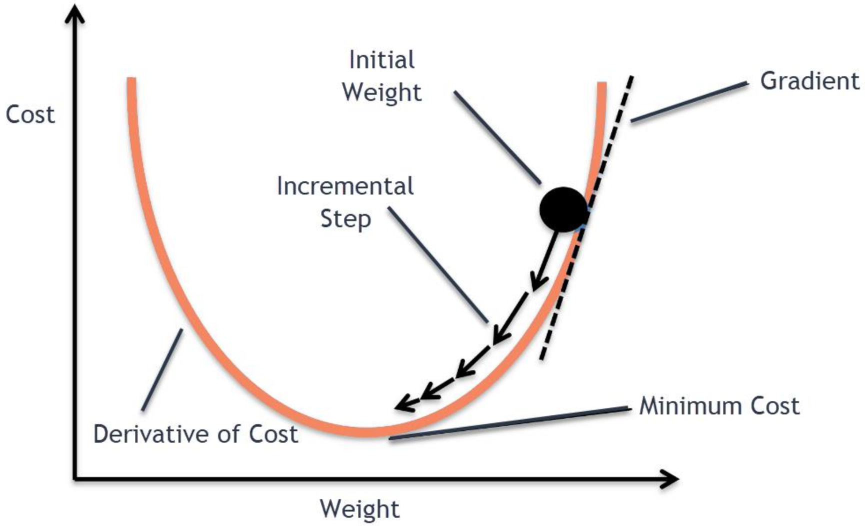


x	y
1	10
2	15
3	40

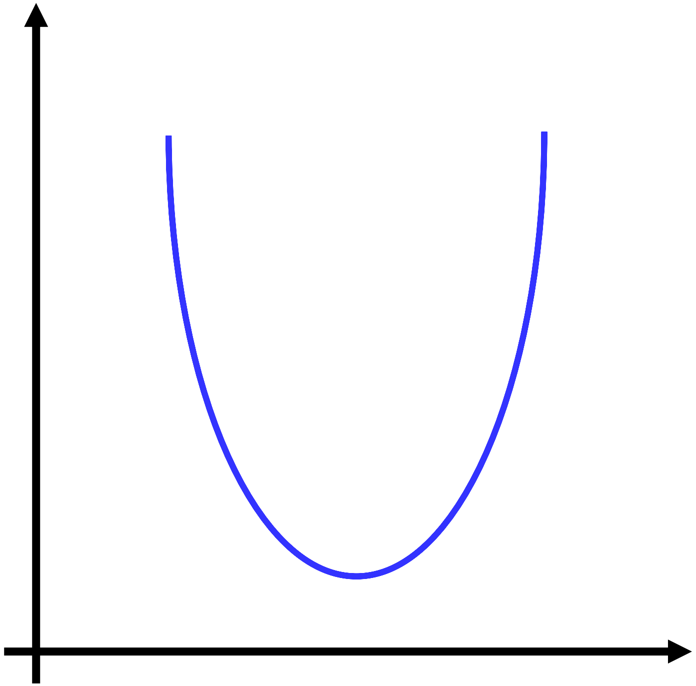
- Cost Function Minimization
  - Model: $H(x) = Wx + b$
  - Cost Function: $Cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}\left(H\left(x^i\right) - y^i\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(Wx^i + b - y^i\right)^2$
  - How to Minimize this Function? → **Gradient Descent Method**
    - Angle → Differentiation

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

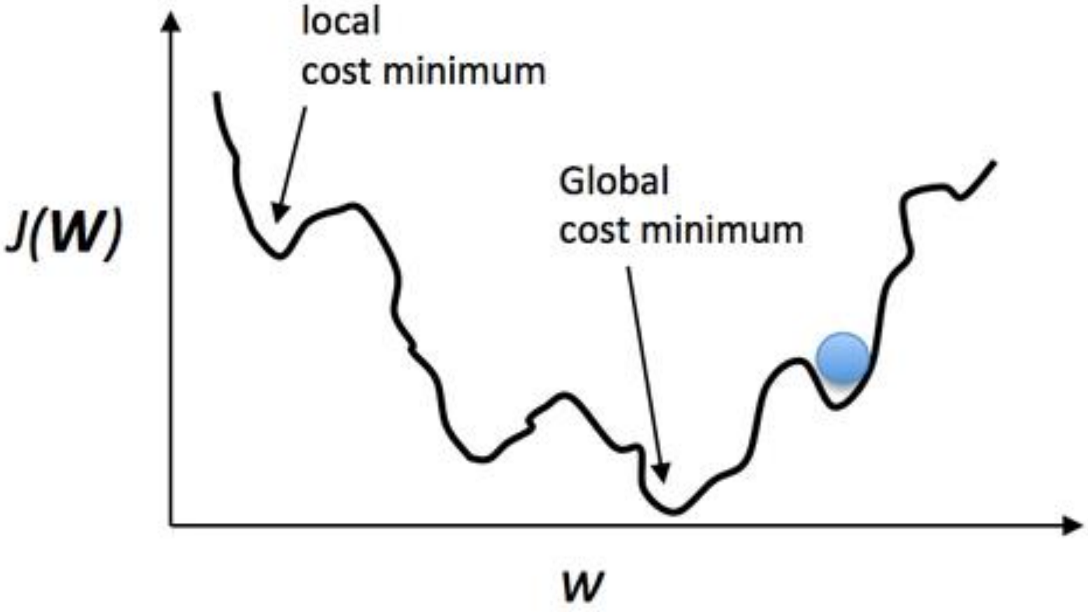
α: Learning rate



- Learning Rates
  - Too large: Overshooting
  - Too small: takes too long, stops in the middle
- How can we determine the learning rates?
  - Try several learning rates
    - Observe the cost function
    - Check it goes down in a reasonable rate



- Cost Function Minimization
  - Gradient Descent Method is only good for convex functions.



- 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:

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

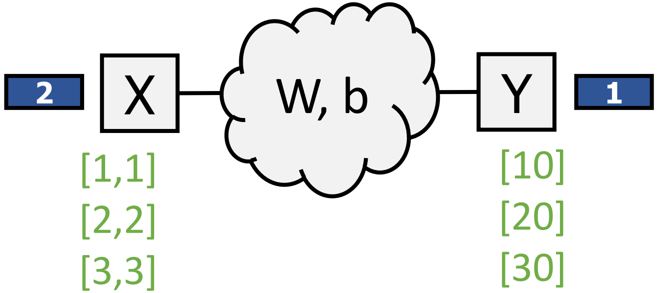
$$\underbrace{(x_1 \quad x_2 \dots \quad x_n)}_{X} \cdot \underbrace{\begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix}}_{W} = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

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

when $W = (w_1 \quad w_2 \dots \quad w_n)$

- TensorFlow
  - **<u>Linear Regression (1.5)</u>**
  - Linear Regression (2.5)
- PyTorch
  - Linear Regression
- Keras
  - Linear Regression

Linear Regression Implementation (TensorFlow 1.5)




```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 1.5)


```
 [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]]
```

- TensorFlow
  - Linear Regression (1.5)
  - **Linear Regression (2.5)**
- PyTorch
  - Linear Regression
- Keras
  - Linear Regression

Linear Regression Implementation (TensorFlow 2.5)


```python
import tensorflow as tf
import numpy as np
x_data = [[1,1],[2,2],[3,3]]
y_data = [[10],[20],[30]]
W = tf.Variable(np.random.randn())
b = tf.Variable(np.random.randn())

def linear_regression(x):
  return W*x+b
def mean_square(y_p,y_t):
  return tf.reduce_mean(tf.square(y_p-y_t))
def run_optimization():
  with tf.GradientTape() as g:
    model = linear_regression(x_data)
    cost = mean_square(model,y_data)
  gradients = g.gradient(cost,[W,b])
  tf.optimizers.SGD(0.01).apply_gradients(zip(gradients,[W,b]))

for step in range(1,2001):
  run_optimization()
  if step % 100 == 0:
    model = linear_regression(x_data)
    cost = mean_square(model, y_data)
    print("step : %i, cost : %f, W : %f, b: %f" %(step,cost,W.numpy(),b.numpy()))
```

```
step : 100, cost : 1.279252, W : 8.686304, b: 2.986167
step : 200, cost : 0.790499, W : 8.967365, b: 2.347421
step : 300, cost : 0.488480, W : 9.188255, b: 1.845286
step : 400, cost : 0.301851, W : 9.361896, b: 1.450563
step : 500, cost : 0.186525, W : 9.498391, b: 1.140274
step : 600, cost : 0.115261, W : 9.605690, b: 0.896360
step : 700, cost : 0.071225, W : 9.690037, b: 0.704621
step : 800, cost : 0.044012, W : 9.756341, b: 0.553896
step : 900, cost : 0.027197, W : 9.808461, b: 0.435412
step : 1000, cost : 0.016806, W : 9.849432, b: 0.342275
step : 1100, cost : 0.010385, W : 9.881641, b: 0.269059
step : 1200, cost : 0.006417, W : 9.906960, b: 0.211504
step : 1300, cost : 0.003966, W : 9.926862, b: 0.166261
step : 1400, cost : 0.002450, W : 9.942507, b: 0.130696
step : 1500, cost : 0.001514, W : 9.954804, b: 0.102739
step : 1600, cost : 0.000936, W : 9.964473, b: 0.080762
step : 1700, cost : 0.000578, W : 9.972073, b: 0.063487
step : 1800, cost : 0.000357, W : 9.978045, b: 0.049907
step : 1900, cost : 0.000221, W : 9.982741, b: 0.039234
step : 2000, cost : 0.000136, W : 9.986431, b: 0.030843
```

Linear Regression Implementation

- TensorFlow
  - Linear Regression (1.5)
  - Linear Regression (2.5)
- PyTorch
  - **Linear Regression**
- Keras
  - Linear Regression

Linear Regression Implementation (PyTorch)

Artificial Intelligence and Mobility Lab

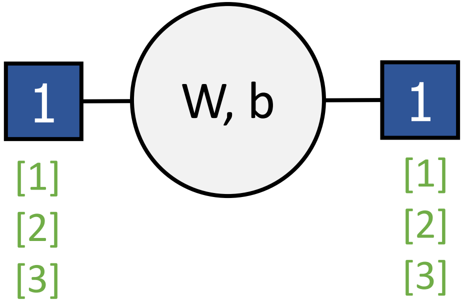
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
x_data = torch.FloatTensor([[1], [2], [3]])
y_data = torch.FloatTensor([[2], [4], [6]])
W = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
optimizer = optim.SGD([W, b], lr=0.01)
for epoch in range(2001):
    model = x_data * W + b
    cost = torch.mean((model - y_data) ** 2)
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print('Epoch {:4d} W: {:.3f}, b: {:.3f} Cost: {:.6f}'.format(epoch, W.item(), b.item(), cost.item()))
```

```
Epoch    0 W: 0.187, b: 0.080 Cost: 18.666666
Epoch  100 W: 1.746, b: 0.578 Cost: 0.048171
Epoch  200 W: 1.800, b: 0.454 Cost: 0.029767
Epoch  300 W: 1.843, b: 0.357 Cost: 0.018394
Epoch  400 W: 1.876, b: 0.281 Cost: 0.011366
Epoch  500 W: 1.903, b: 0.221 Cost: 0.007024
Epoch  600 W: 1.924, b: 0.174 Cost: 0.004340
Epoch  700 W: 1.940, b: 0.136 Cost: 0.002682
Epoch  800 W: 1.953, b: 0.107 Cost: 0.001657
Epoch  900 W: 1.963, b: 0.084 Cost: 0.001024
Epoch 1000 W: 1.971, b: 0.066 Cost: 0.000633
Epoch 1100 W: 1.977, b: 0.052 Cost: 0.000391
Epoch 1200 W: 1.982, b: 0.041 Cost: 0.000242
Epoch 1300 W: 1.986, b: 0.032 Cost: 0.000149
Epoch 1400 W: 1.989, b: 0.025 Cost: 0.000092
Epoch 1500 W: 1.991, b: 0.020 Cost: 0.000057
Epoch 1600 W: 1.993, b: 0.016 Cost: 0.000035
Epoch 1700 W: 1.995, b: 0.012 Cost: 0.000022
Epoch 1800 W: 1.996, b: 0.010 Cost: 0.000013
Epoch 1900 W: 1.997, b: 0.008 Cost: 0.000008
Epoch 2000 W: 1.997, b: 0.006 Cost: 0.000005
```

- TensorFlow
  - Linear Regression (1.5)
  - Linear Regression (2.5)
- PyTorch
  - Linear Regression
- Keras
  - **Linear Regression**

Linear Regression Implementation (Keras)


```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense

# Data
x_data = np.array([[1], [2], [3]])
y_data = np.array([[1], [2], [3]])
# Model, Cost, Train
model = Sequential()
model.add(Dense(1, input_dim=1))
model.compile(loss='mse', optimizer='adam')
model.fit(x_data, y_data, epochs=1000, verbose=0)
model.summary()
# Inference
print(model.get_weights())
print(model.predict(np.array([4])))
# Plot
plt.scatter(x_data, y_data)
plt.plot(x_data, y_data)
plt.grid(True)
plt.show()
```

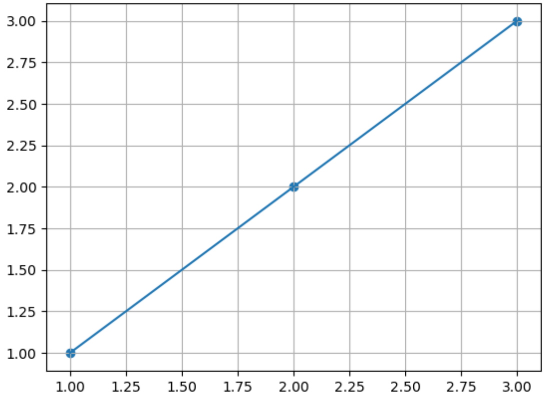


Model, Cost, Train

Linear Regression Implementation (Keras)

Artificial Intelligence and Mobility Lab

```
joongheon@joongheon-AB350M-Gaming-3: ~/Dropbox/codes_keras
joongheon@joongheon-AB350M-Gaming-3:~/Dropbox/codes_keras$ python keras_linearregression.py
/home/joongheon/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Con
version of the second argument of issubdtype from `float` to `np.floating` is deprecated. In
future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
2019-06-29 16:39:04.566966: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU sup
ports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
[array([[0.999888]], dtype=float32), array([0.00024829], dtype=float32)]
[[3.9998002]]
joongheon@joongheon-AB350M-Gaming-3:~/Dropbox/codes_keras$
```



Deep Learning Basics and Software

Binary Classification (Logistic Regression)

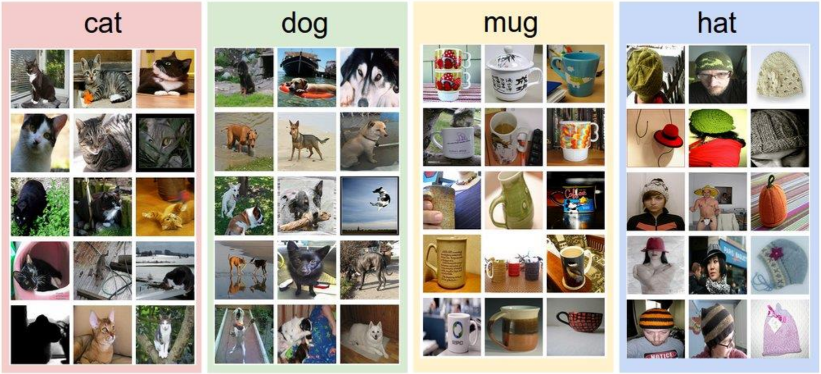
Regression and Classification

Regression (Examples)

- Exam Score Prediction (Linear Regression)

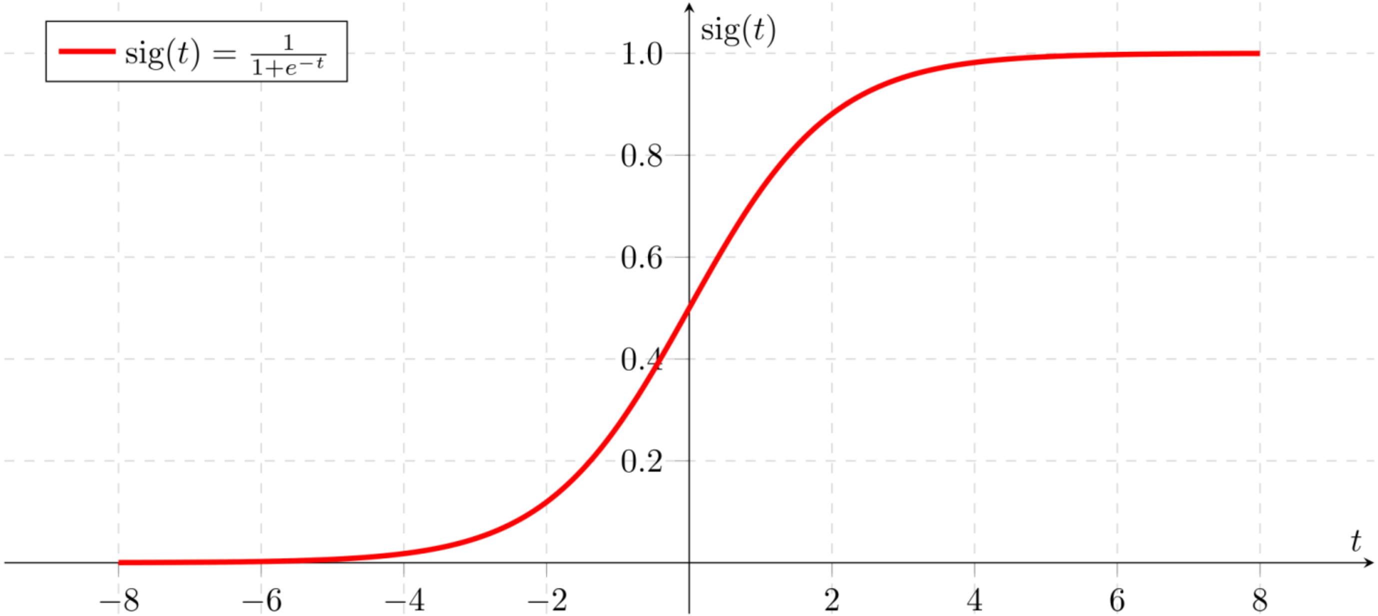
Classification (Examples)

- Pass/Fail (Binary Classification)
- Letter Grades (Multi-Level Classification)



- 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.



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

$H(x) = Wx + b$ or $H(X) = W^T X$

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

Binary Classification Model

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

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

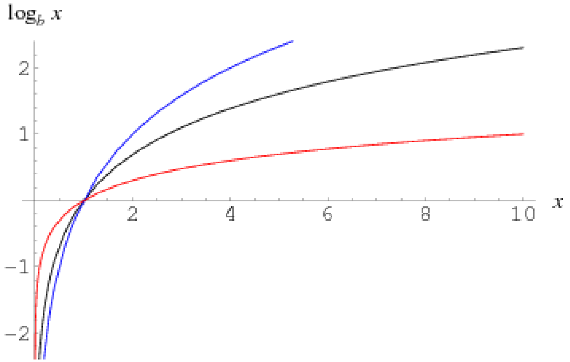
$$\text{Cost}(W) = \frac{1}{m}\sum c(H(x), y)$$

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

Understanding this Cost Function

Cost	**0**	**∞**	**∞**	**0**
$H(x)$	0	0	1	1
y	0	1	0	1

Log Function



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

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

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

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

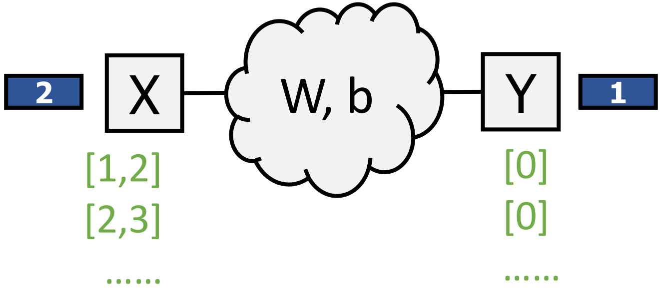
Gradient Descent Method

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

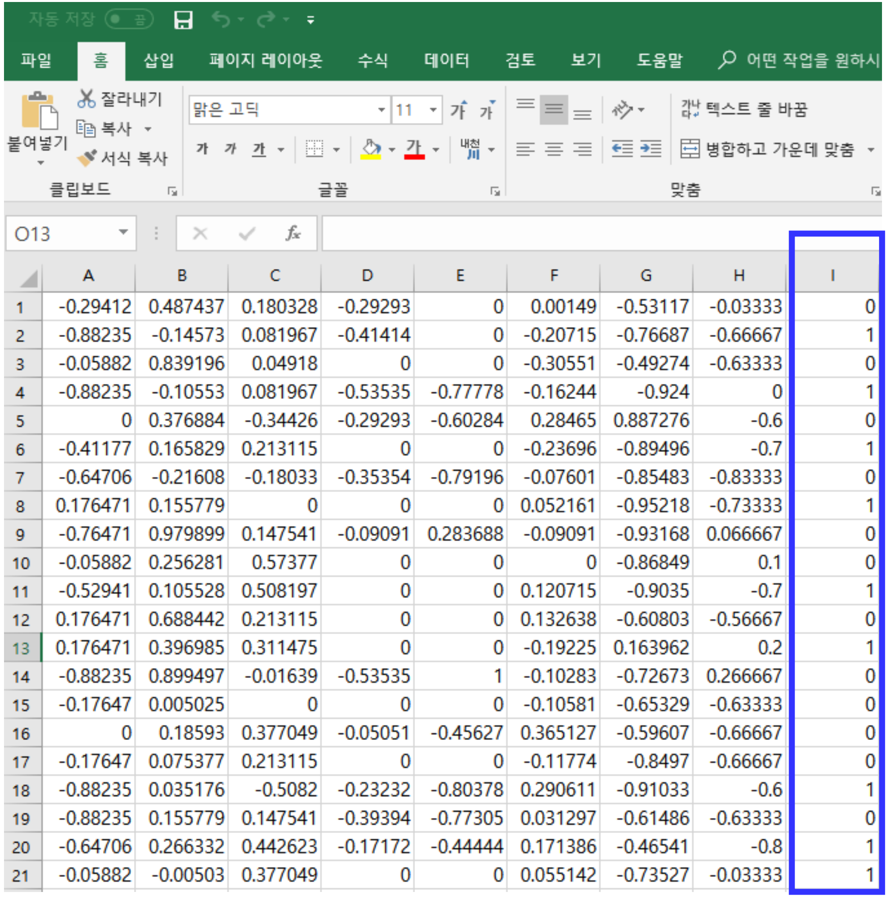
Binary Classification Implementation (TensorFlow)


```
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



CSV file



	A	B	C	D	E	F	G	H	I
1	-0.29412	0.487437	0.180328	-0.29293	0	0.00149	-0.53117	-0.03333	0
2	-0.88235	-0.14573	0.081967	-0.41414	0	-0.20715	-0.76687	-0.66667	1
3	-0.05882	0.839196	0.04918	0	0	-0.30551	-0.49274	-0.63333	0
4	-0.88235	-0.10553	0.081967	-0.53535	-0.77778	-0.16244	-0.924	0	1
5	0	0.376884	-0.34426	-0.29293	-0.60284	0.28465	0.887276	-0.6	0
6	-0.41177	0.165829	0.213115	0	0	-0.23696	-0.89496	-0.7	1
7	-0.64706	-0.21608	-0.18033	-0.35354	-0.79196	-0.07601	-0.85483	-0.83333	0
8	0.176471	0.155779	0	0	0	0.052161	-0.95218	-0.73333	1
9	-0.76471	0.979899	0.147541	-0.09091	0.283688	-0.09091	-0.93168	0.066667	0
10	-0.05882	0.256281	0.57377	0	0	0	-0.86849	0.1	0
11	-0.52941	0.105528	0.508197	0	0	0.120715	-0.9035	-0.7	1
12	0.176471	0.688442	0.213115	0	0	0.132638	-0.60803	-0.56667	0
13	0.176471	0.396985	0.311475	0	0	-0.19225	0.163962	0.2	1
14	-0.88235	0.899497	-0.01639	-0.53535	1	-0.10283	-0.72673	0.266667	0
15	-0.17647	0.005025	0	0	0	-0.10581	-0.65329	-0.63333	0
16	0	0.18593	0.377049	-0.05051	-0.45627	0.365127	-0.59607	-0.66667	0
17	-0.17647	0.075377	0.213115	0	0	-0.11774	-0.8497	-0.66667	0
18	-0.88235	0.035176	-0.5082	-0.23232	-0.80378	0.290611	-0.91033	-0.6	1
19	-0.88235	0.155779	0.147541	-0.39394	-0.77305	0.031297	-0.61486	-0.63333	0
20	-0.64706	0.266332	0.442623	-0.17172	-0.44444	0.171386	-0.46541	-0.8	1
21	-0.05882	-0.00503	0.377049	0	0	0.055142	-0.73527	-0.03333	1

Binary Classification Implementation (TensorFlow)

CSV data loading

x_data.shape[1] == 8

Model, Cost, Train

```python
import tensorflow as tf
import numpy as np

xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
x_data = xy[:, 0:-1]
y_data = xy[:, [-1]]

X = tf.placeholder(tf.float32, shape=[None, x_data.shape[1]])
Y = tf.placeholder(tf.float32, shape=[None, 1])
W = tf.Variable(tf.random_normal([x_data.shape[1], 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(0.01).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(100001):
        c, _ = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
        print(step, c)
    # Testing
    h, c, a = sess.run([model, prediction, accuracy], feed_dict={X: x_data, Y: y_data})
    print("\nHypothesis: ", h, "\nCorrect (Y): ", c, "\nAccuracy: ", a)
```

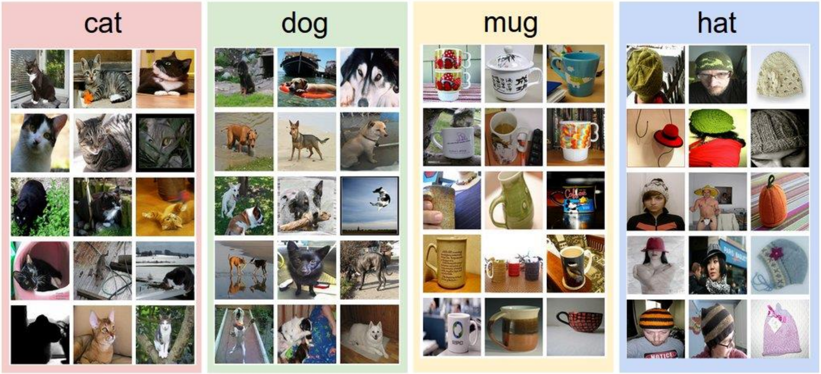
```
 [0.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
Accuracy:  0.76943344
```

Deep Learning Basics and Software

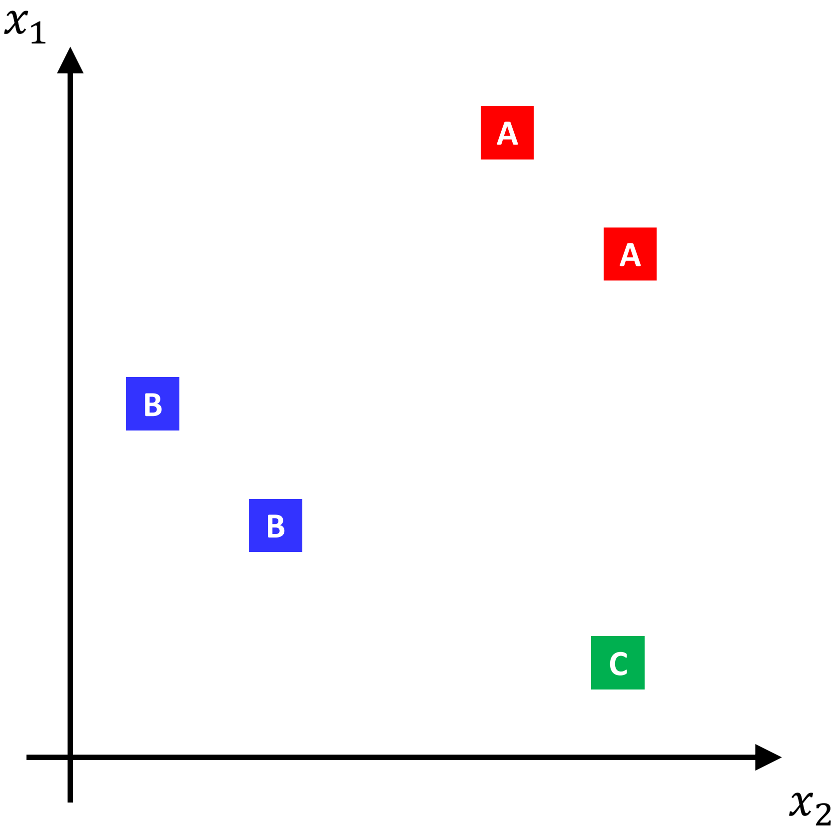
Softmax Classification

Regression (Examples)
• Exam Score Prediction (Linear Regression)

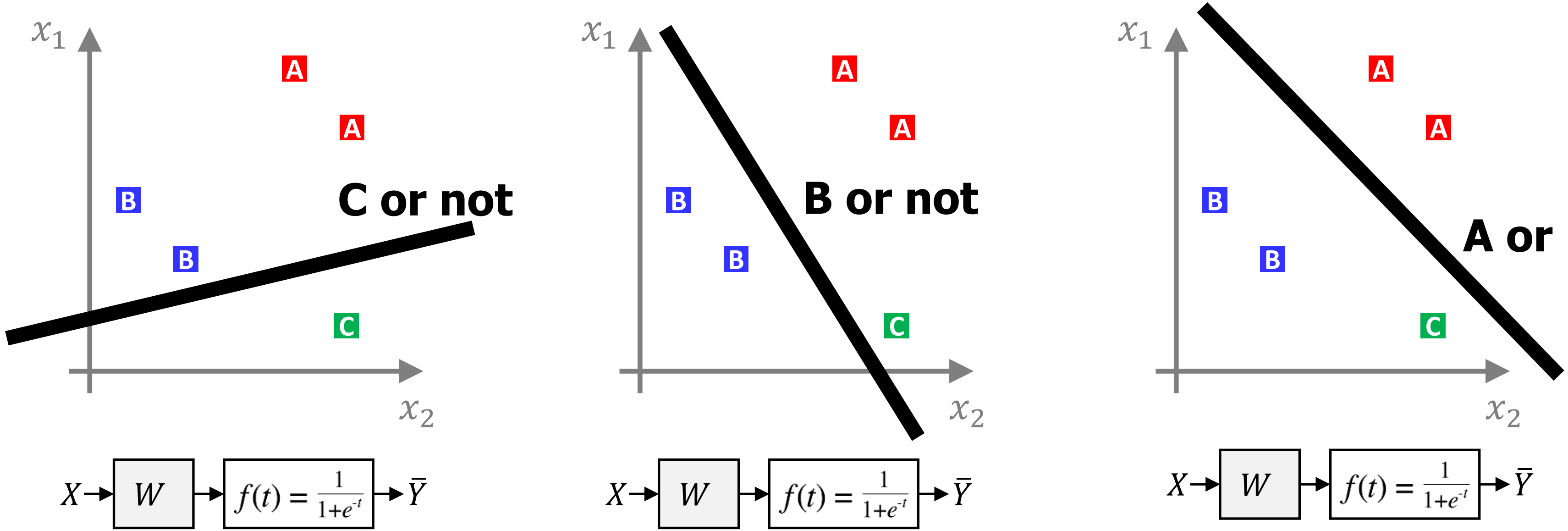
Classification (Examples)
• Pass/Fail (Binary Classification)
• Letter Grades (Multi-Level Classification)



Multinomial Classification (Softmax Classification)

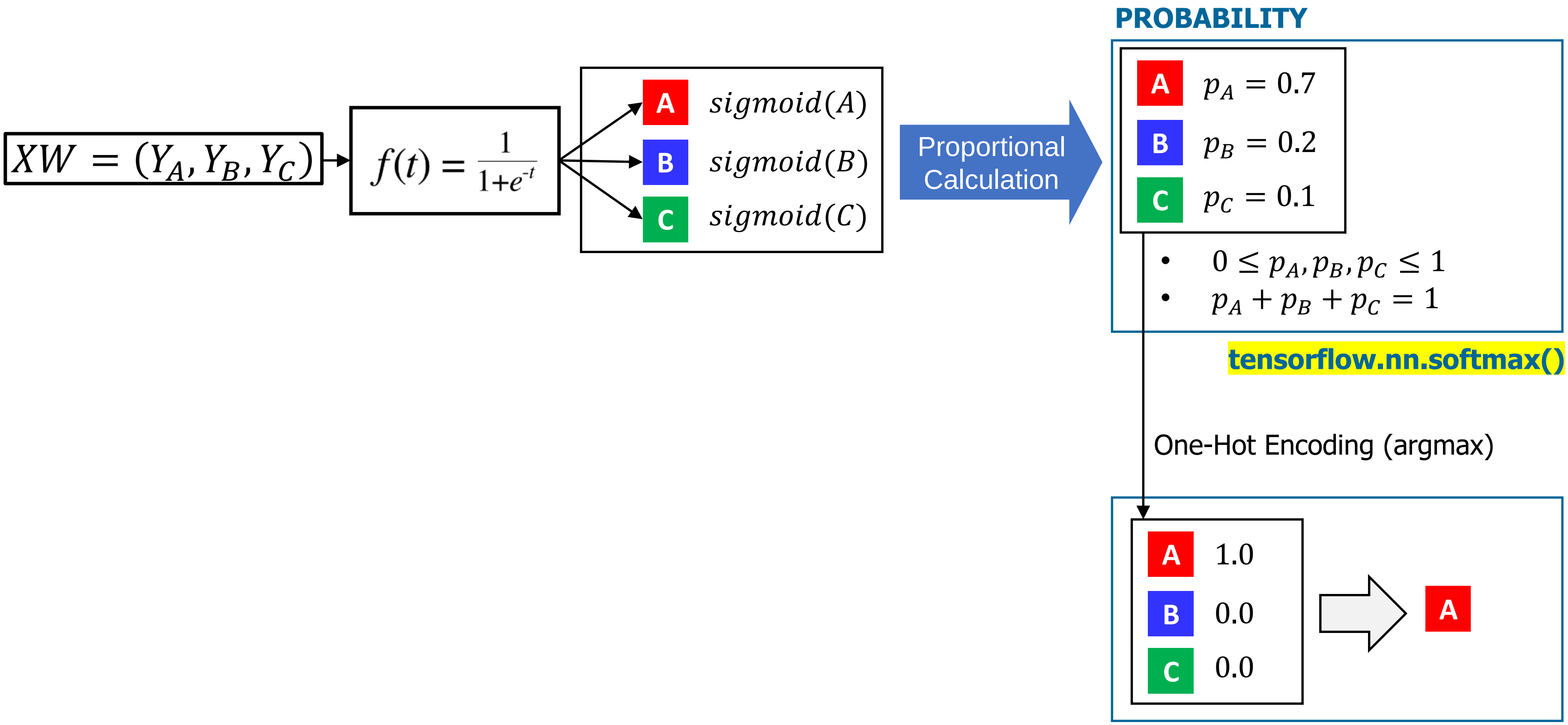


Multinomial Classification (Softmax Classification)



$$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \cdot \begin{array}{c} \begin{array}{ccc} \text{A} & \text{B} & \text{C} \end{array} \\ \begin{pmatrix} \boxed{\begin{matrix} W_{A1} \\ W_{A2} \end{matrix}} & \boxed{\begin{matrix} W_{B1} \\ W_{B2} \end{matrix}} & \boxed{\begin{matrix} W_{C1} \\ W_{C2} \end{matrix}} \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{ccc} \text{A} & \text{B} & \text{C} \end{array} \\ \begin{pmatrix} \boxed{x_1 \cdot W_{A1} + x_2 \cdot W_{A2}} & \boxed{x_1 \cdot W_{B1} + x_2 \cdot W_{B2}} & \boxed{x_1 \cdot W_{C1} + x_2 \cdot W_{C2}} \end{pmatrix} \end{array}$$

Multinomial Classification (Softmax Classification)



- Cost Function: **Cross-Entropy**

$S(Y) = \bar{Y}$ (estimated)
0.7
0.2
0.1

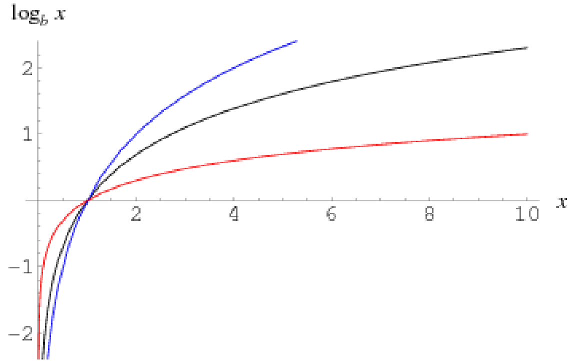
$$C(S, L) = -\sum_{\forall i} L_i \log(S_i)$$

$L = Y$ (real)
1.0
0.0
0.0

Understanding this Cost Function

L	S	Cost
[1,0,0]	[1,0,0]	$-1 \cdot \log 1 - 0 \cdot \log 0 - 0 \cdot \log 0 = 0$
	[0,1,0]	$-1 \cdot \log 0 - 0 \cdot \log 1 - 0 \cdot \log 0 = \infty$
	[0,0,1]	$-1 \cdot \log 0 - 0 \cdot \log 0 - 0 \cdot \log 1 = \infty$

Log Function



Softmax Classification Implementation (TensorFlow)


```python
import tensorflow as tf

x_data = [[1,2,1,1], [2,1,3,2], [3,1,3,4], [4,1,5,5], [1,7,5,5], [1,2,5,6], [1,6,6,6], [1,7,7,7]] # vectors
y_data = [[0,0,1], [0,0,1], [0,0,1], [0,1,0], [0,1,0], [0,1,0], [1,0,0], [1,0,0]] # one hot encoding
X = tf.placeholder(tf.float32, shape=[None, 4])
Y = tf.placeholder(tf.float32, shape=[None, 3])
W = tf.Variable(tf.random_normal([4, 3]))
b = tf.Variable(tf.random_normal([3]))

model_LC = tf.add(tf.matmul(X,W),b)
model = tf.nn.softmax(model_LC)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=model_LC, labels=Y))
train = tf.train.GradientDescentOptimizer(0.1).minimize(cost)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # Training
    for step in range(2001):
        c, _ = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
        print(step, c)
    # Testing
    test1 = sess.run(model, feed_dict={X: [[1,11,7,9]]})
    print(test1, sess.run(tf.argmax(test1, 1)))
```

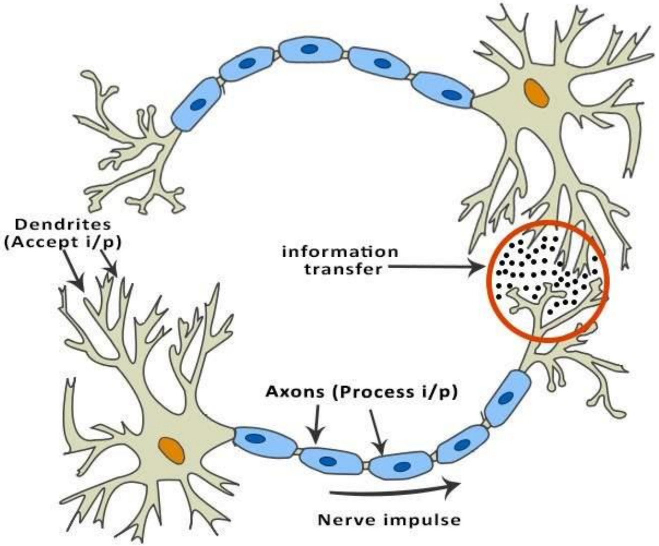
Model, Cost, Train

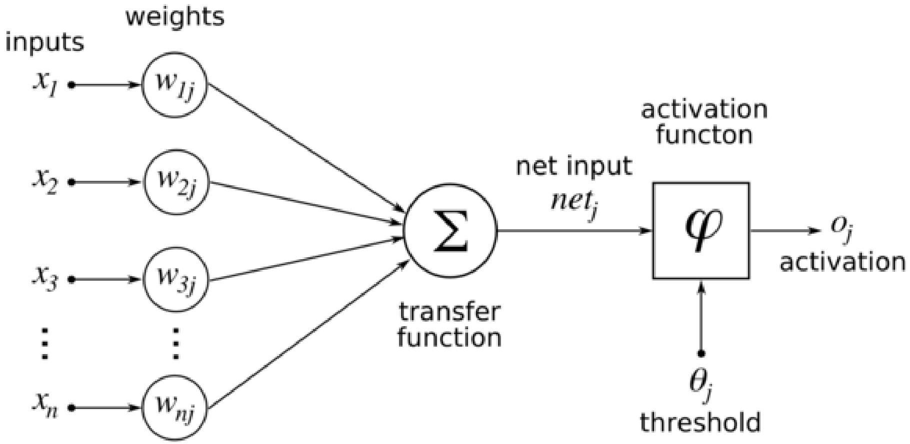
```
1988 0.16142774
1989 0.16136909
1990 0.16131032
1991 0.16125184
1992 0.16119315
1993 0.16113463
1994 0.16107623
1995 0.16101775
1996 0.16095944
1997 0.1609011
1998 0.16084275
1999 0.16078432
2000 0.16072604
[[7.2217123e-03 9.9276876e-01 9.6337890e-06]] [1]
```

Deep Learning Basics and Software

Neural Network (Nonlinear Functions)

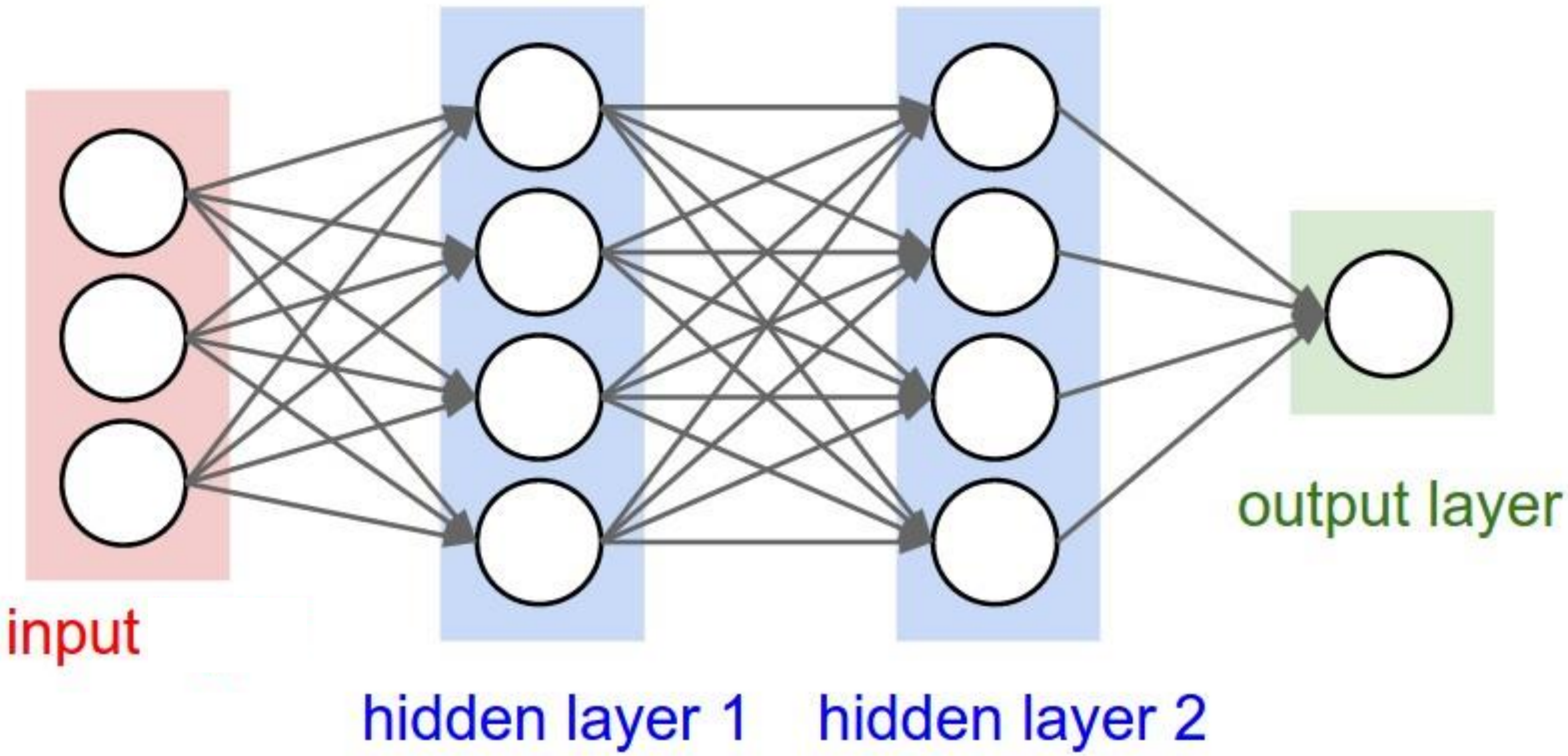
- Human Brain (Neuron)





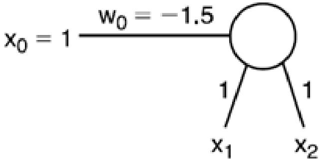
Binary Classification

Artificial Neural Networks (ANN): Multilayer Perceptron

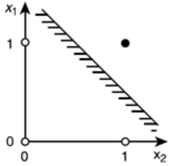


- Application to Logic Gate Design

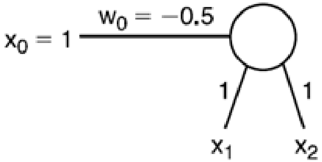
AND gate



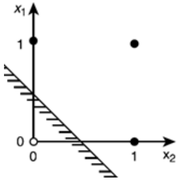
x_1	x_2	W · X	y
0	0	−1.5	0
0	1	−0.5	0
1	0	−0.5	0
1	1	0.5	1



OR gate



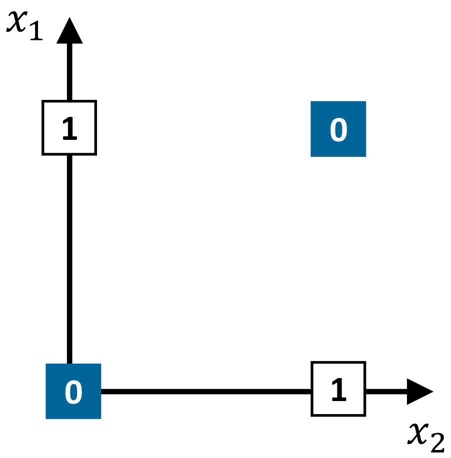
x_1	x_2	W · X	y
0	0	−0.5	0
0	1	0.5	1
1	0	0.5	1
1	1	1.5	1



- **What about XOR?**

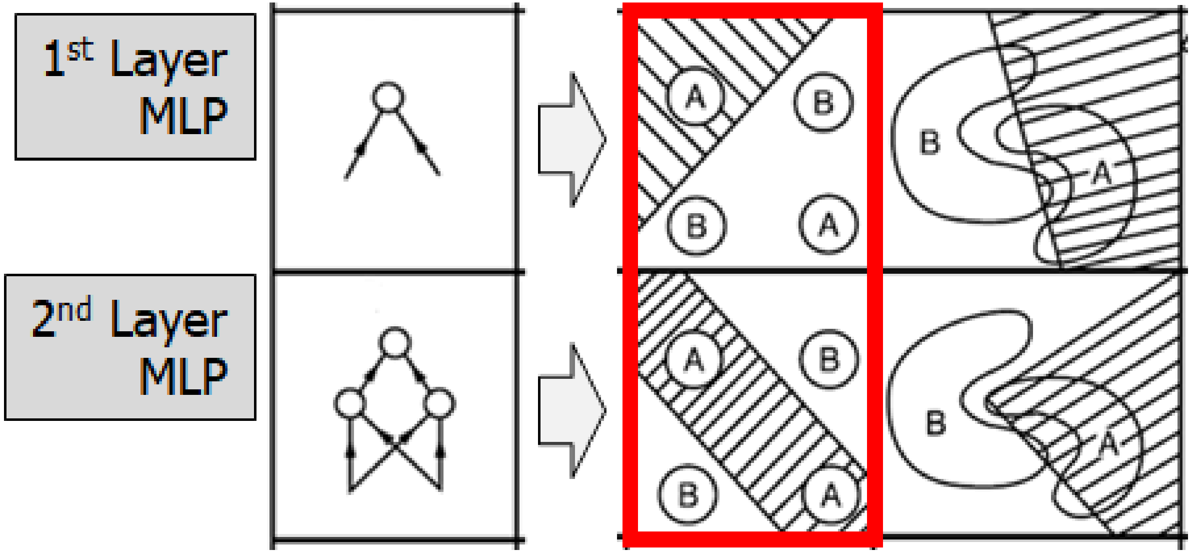
Artificial Neural Networks (ANN): Multilayer Perceptron

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

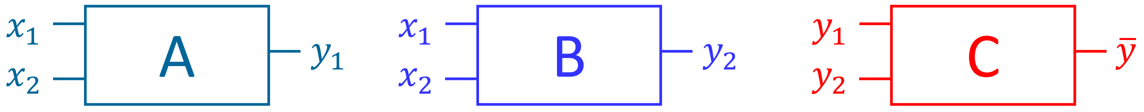


Mathematically proven by
Prof. Marvin Minsky at MIT (1969)

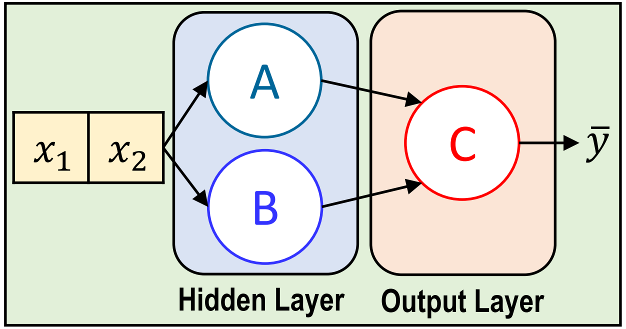
- Multilayer Perceptron (MLP)
  - Proposed by Prof. Marvin Minsky at MIT (1969)
  - Can solve XOR Problem



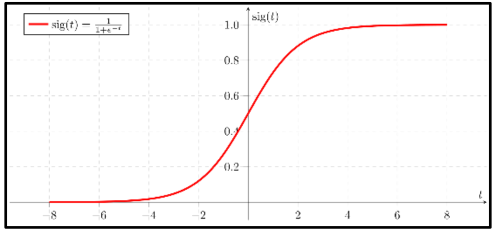
ANN: Solving XOR with MLP



$W = \begin{pmatrix}5\\5\end{pmatrix}, b = -8$ $W = \begin{pmatrix}-7\\-7\end{pmatrix}, b = 3$ $W = \begin{pmatrix}-11\\-11\end{pmatrix}, b = 6$

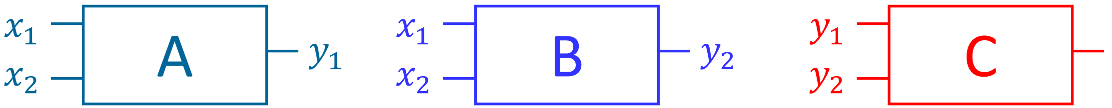


- $(x_1\ x_2) = (0\ 0)$
  - $(0\ 0)\begin{pmatrix}5\\5\end{pmatrix} + (-8) = -8$, i.e., $y_1 = Sigmoid(-8) \cong 0$
  - $(0\ 0)\begin{pmatrix}-7\\-7\end{pmatrix} + (3) = 3$, i.e., $y_2 = Sigmoid(3) \cong 1$
  - $(y_1\ y_2)\begin{pmatrix}-11\\-11\end{pmatrix} + (6) = -11 + 6 = -5$, i.e., $\bar{y} = Sigmoid(-5)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

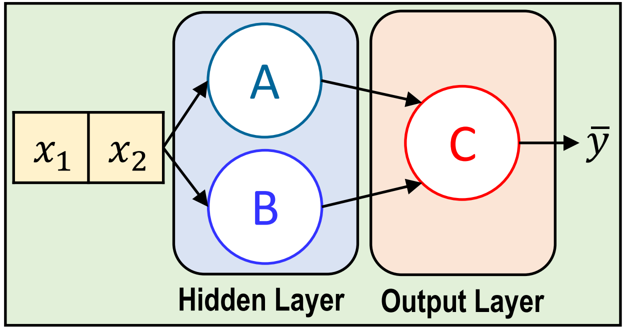


x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	**0**	**1**	**0**	**0**
0	1				**1**
1	0				**1**
1	1				**0**

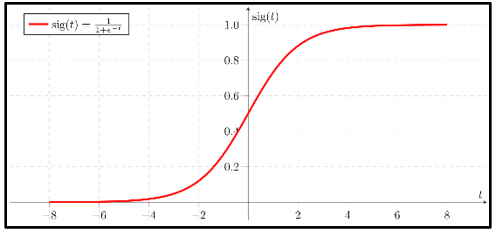
ANN: Solving XOR with MLP



$W = \begin{pmatrix} 5 \\ 5 \end{pmatrix}, b = -8$ $\quad$ $W = \begin{pmatrix} -7 \\ -7 \end{pmatrix}, b = 3$ $\quad$ $W = \begin{pmatrix} -11 \\ -11 \end{pmatrix}, b = 6$

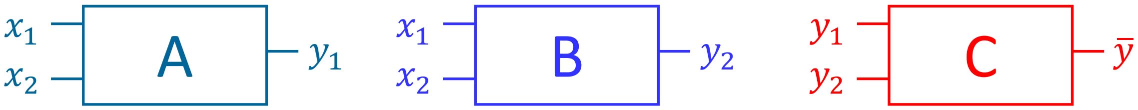


- $(x_1\ x_2) = (0\ 1)$
  - $(0\ 1)\begin{pmatrix} 5 \\ 5 \end{pmatrix} + (-8) = -3$, i.e., $y_1 = Sigmoid(-3) \cong 0$
  - $(0\ 1)\begin{pmatrix} -7 \\ -7 \end{pmatrix} + (3) = -4$, i.e., $y_2 = Sigmoid(-4) \cong 0$
  - $(y_1\ y_2)\begin{pmatrix} -11 \\ -11 \end{pmatrix} + (6) = 6$, i.e., $\bar{y} =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

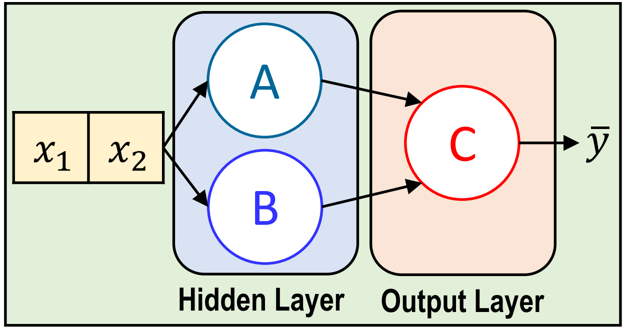


x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	**0**	**1**	**0**	**0**
0	1	**0**	**0**	**1**	**1**
1	0				**1**
1	1				**0**

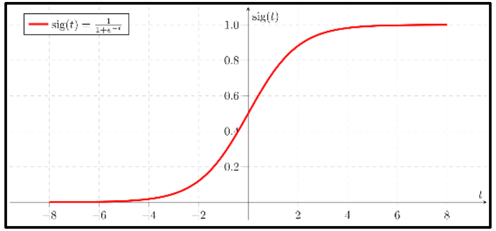
ANN: Solving XOR with MLP



$W = \begin{pmatrix} 5 \\ 5 \end{pmatrix}, b = -8$ $\quad$ $W = \begin{pmatrix} -7 \\ -7 \end{pmatrix}, b = 3$ $\quad$ $W = \begin{pmatrix} -11 \\ -11 \end{pmatrix}, b = 6$

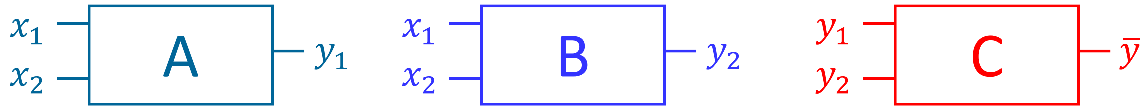


- $(x_1\ x_2) = (1\ 0)$
  - $(1\ 0)\begin{pmatrix} 5 \\ 5 \end{pmatrix} + (-8) = -3$, i.e., $y_1 = Sigmoid(-3) \cong 0$
  - $(1\ 0)\begin{pmatrix} -7 \\ -7 \end{pmatrix} + (3) = -4$, i.e., $y_2 = Sigmoid(-4) \cong 0$
  - $(y_1\ y_2)\begin{pmatrix} -11 \\ -11 \end{pmatrix} + (6) = 6$, i.e., $\bar{y} =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

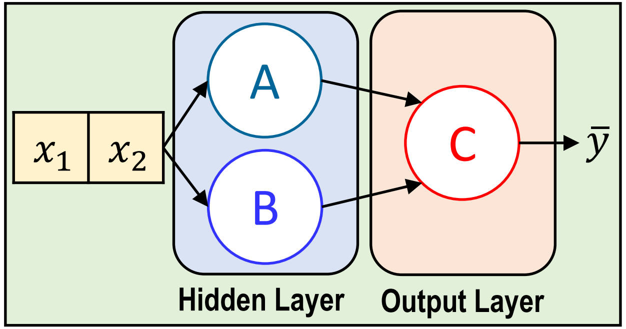


x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	**0**	**1**	**0**	**0**
0	1	**0**	**0**	**1**	**1**
1	0	**0**	**0**	**1**	**1**
1	1				**0**

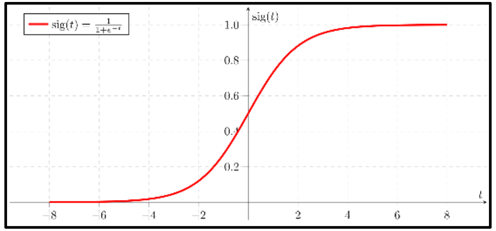
ANN: Solving XOR with MLP



$W = \begin{pmatrix} 5 \\ 5 \end{pmatrix}, b = -8$ $W = \begin{pmatrix} -7 \\ -7 \end{pmatrix}, b = 3$ $W = \begin{pmatrix} -11 \\ -11 \end{pmatrix}, b = 6$

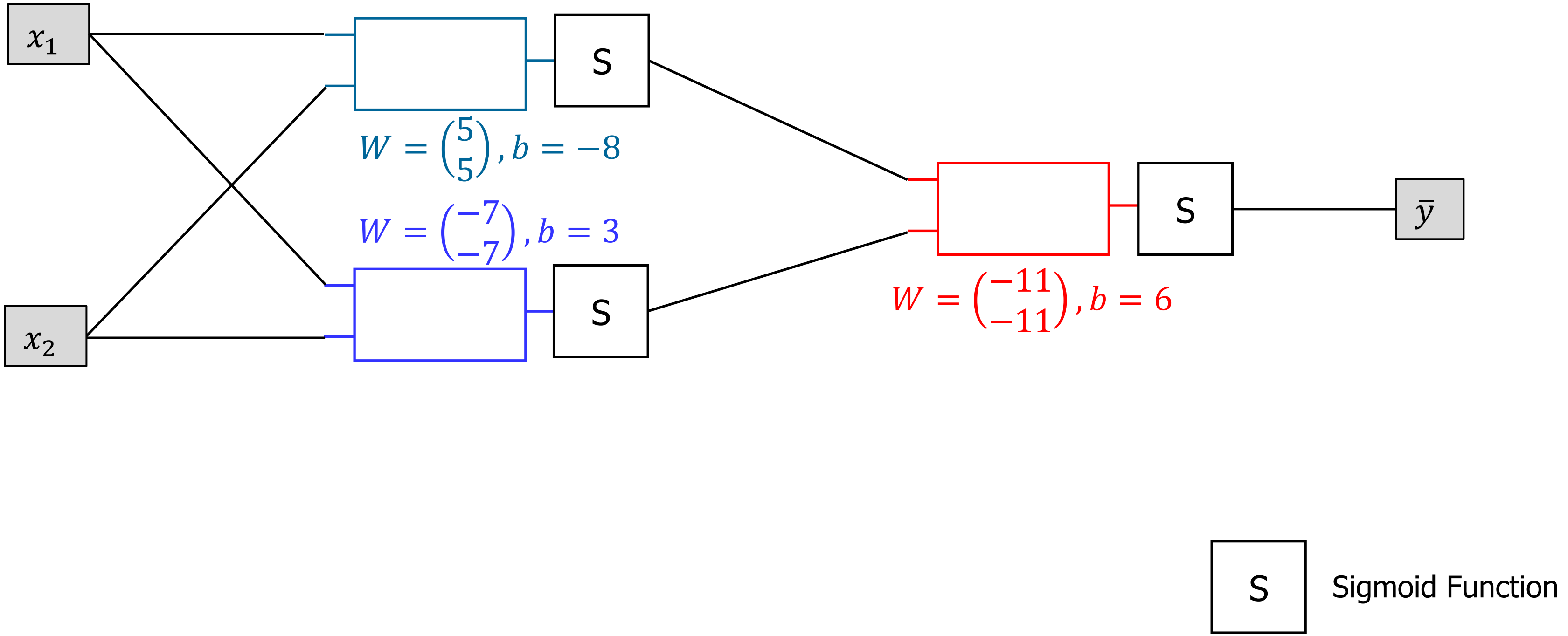


- $(x_1\ x_2) = (1\ 1)$
  - $(1\ 1)\begin{pmatrix} 5 \\ 5 \end{pmatrix} + (-8) = 2$, i.e., $y_1 = Sigmoid(2) \cong 1$
  - $(1\ 1)\begin{pmatrix} -7 \\ -7 \end{pmatrix} + (3) = -11$, i.e., $y_2 = Sigmoid(-11) \cong 0$
  - $(y_1\ y_2)\begin{pmatrix} -11 \\ -11 \end{pmatrix} + (6) = -5$, i.e., $\bar{y}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

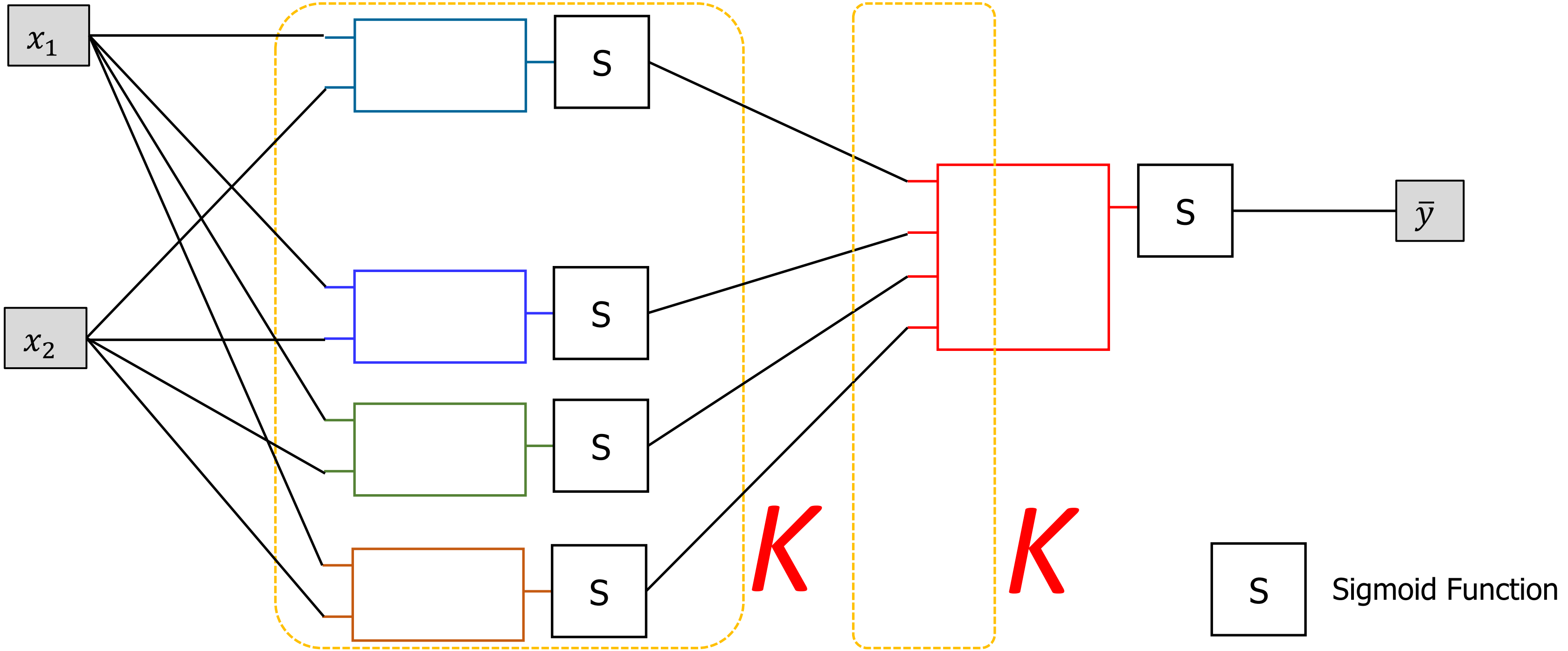


x_1	x_2	y_1	y_2	$\bar{y}$	XOR
0	0	0	1	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	0	0	0

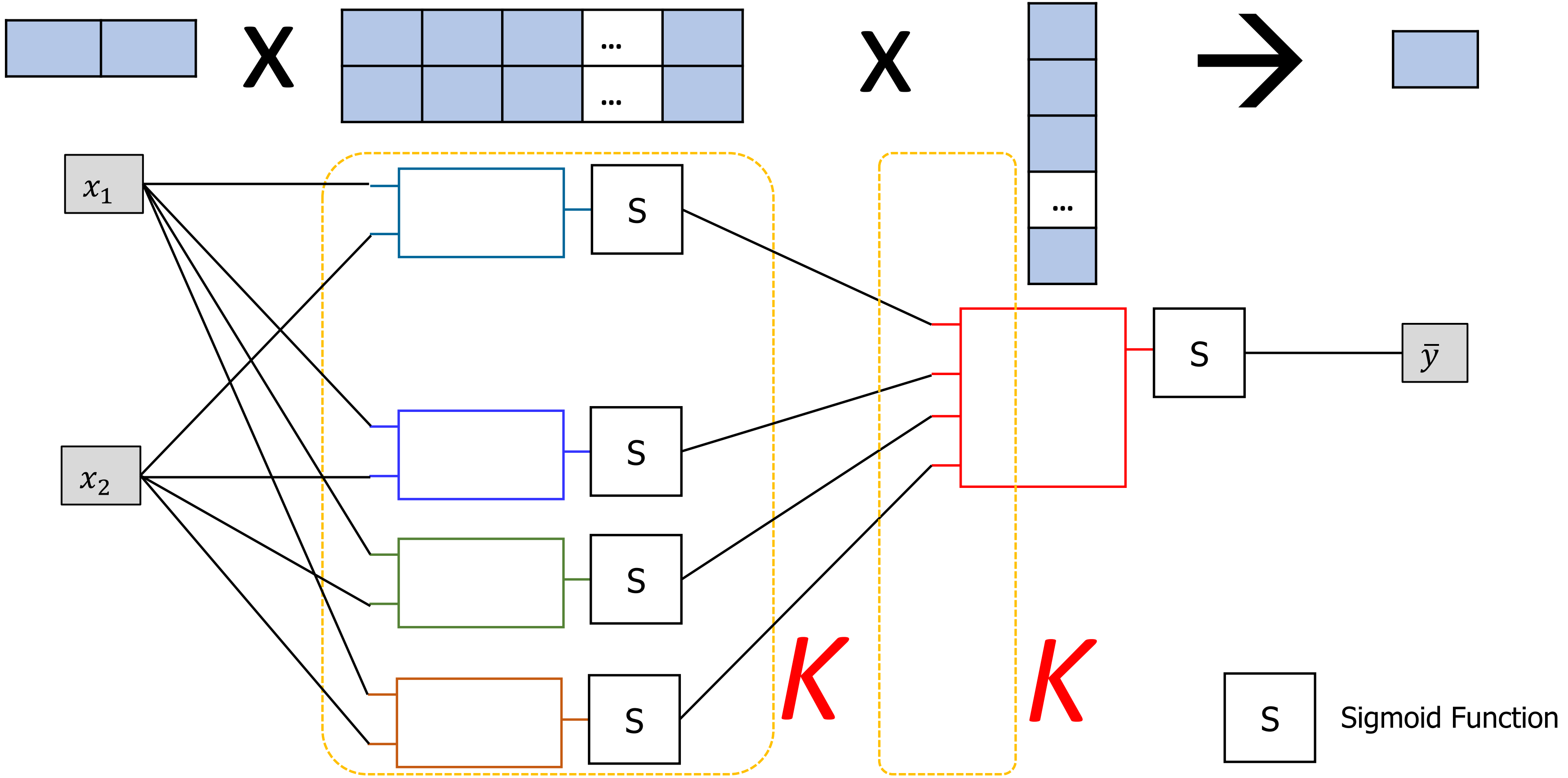
ANN: Solving XOR with MLP (Forward Propagation)



ANN: Solving XOR with MLP (Forward Propagation)

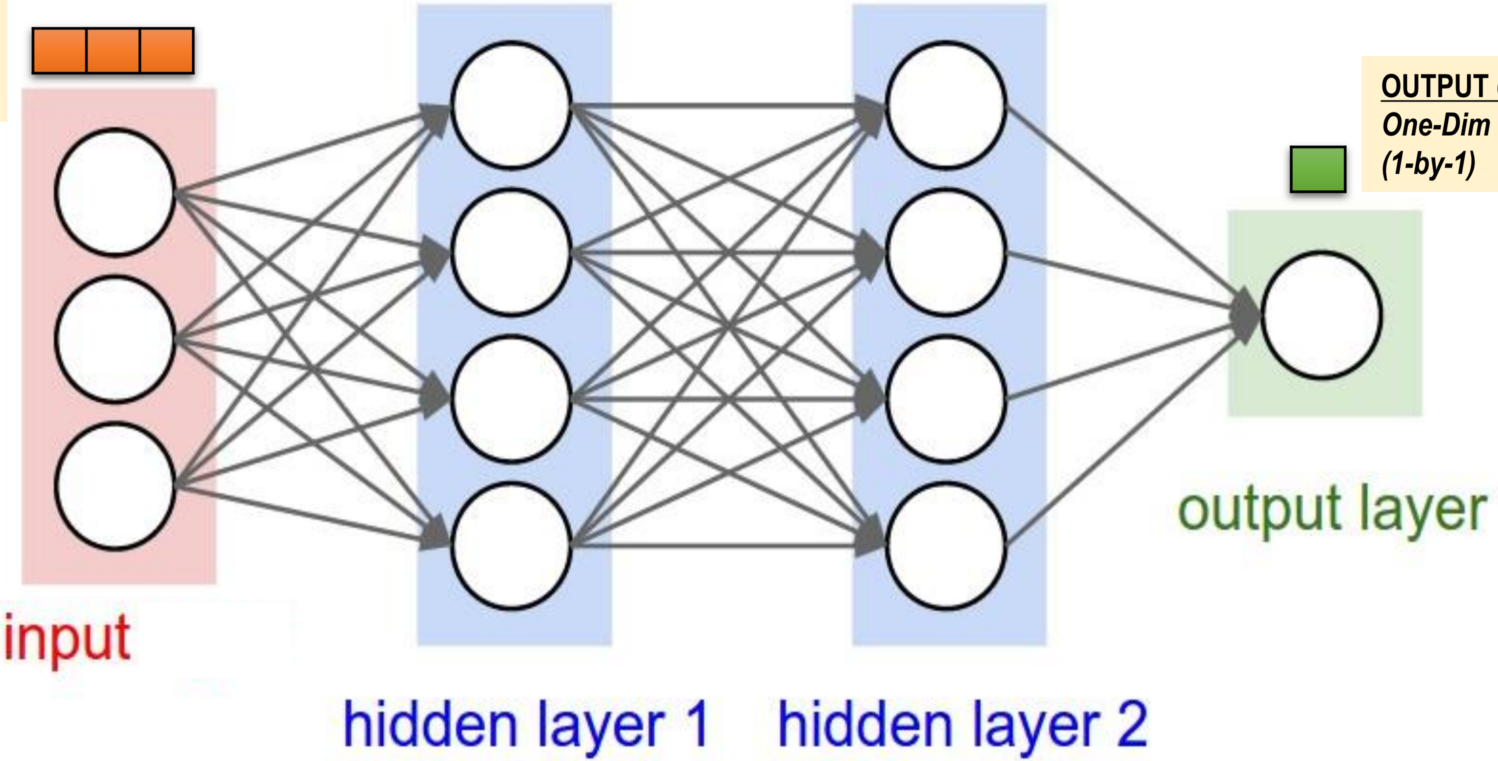


ANN: Solving XOR with MLP (Forward Propagation)



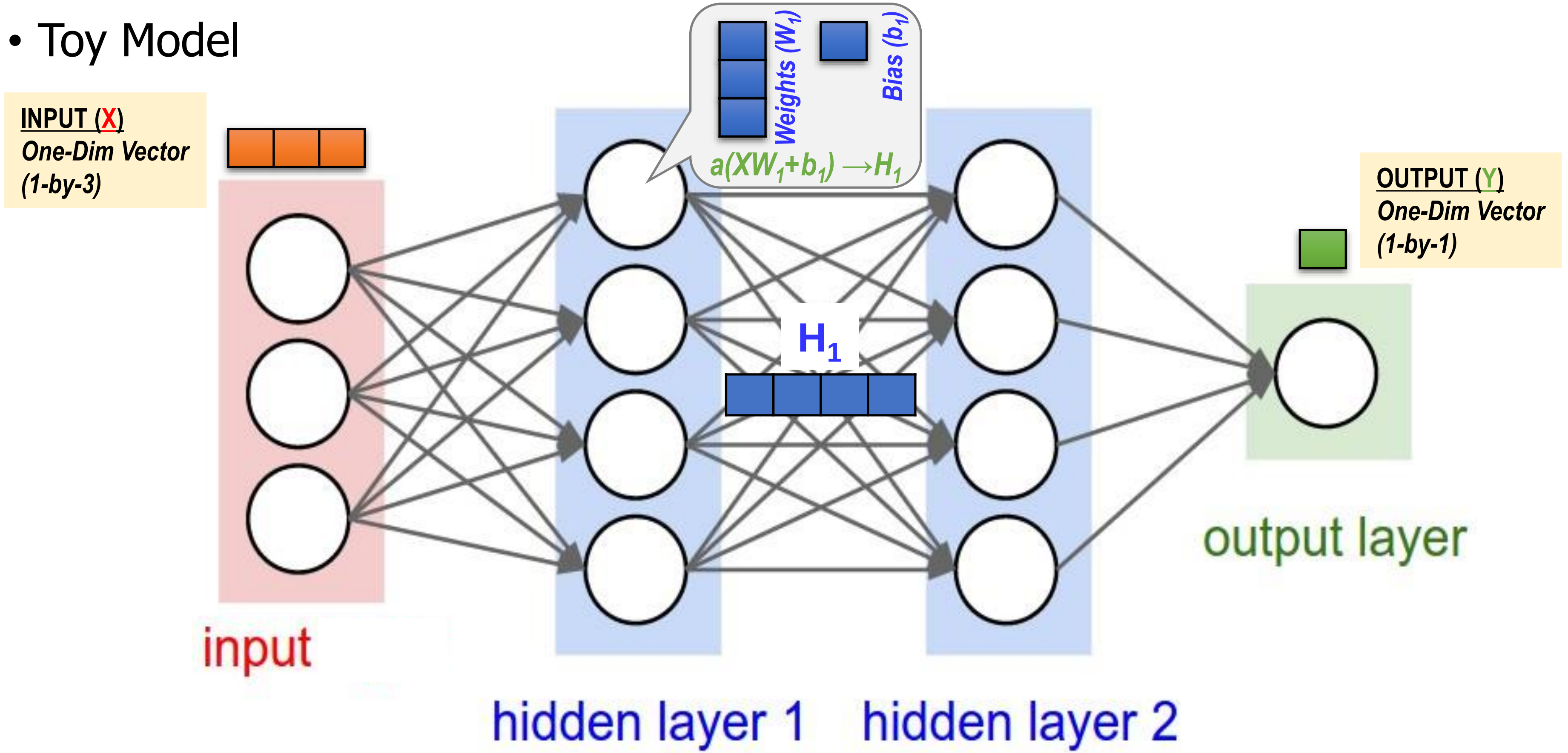
- Toy Model

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

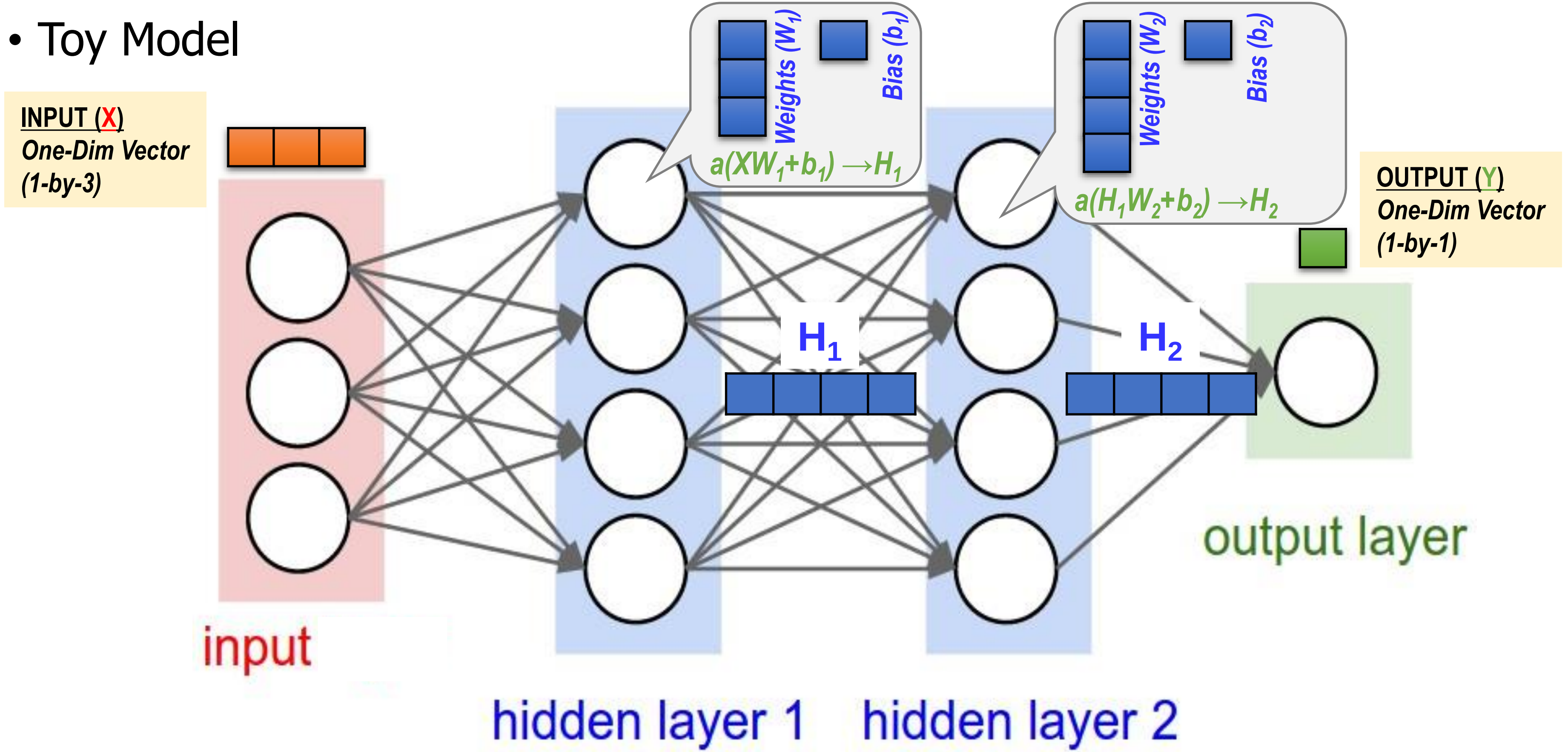


OUTPUT (Y)
One-Dim Vector
(1-by-1)

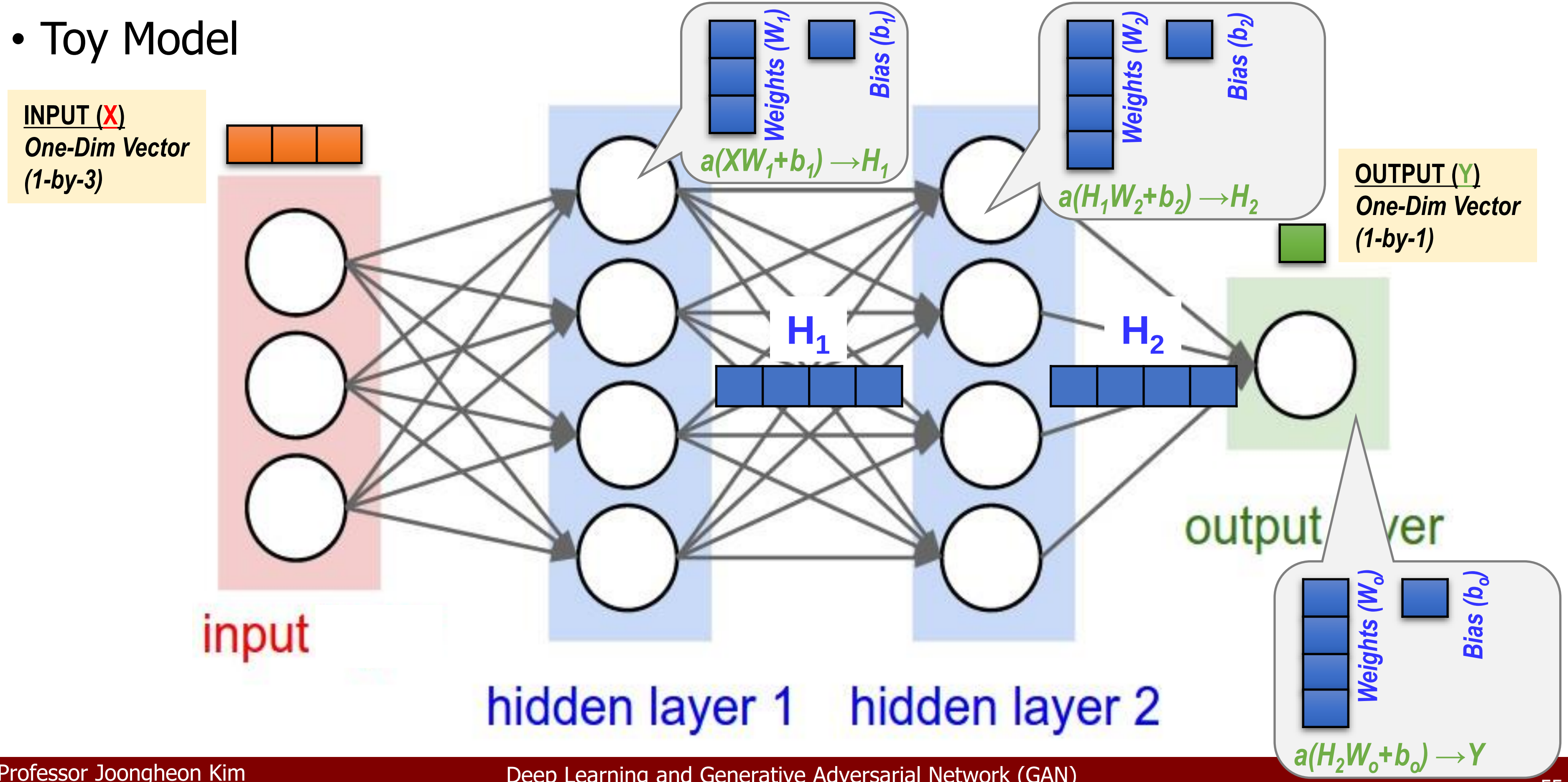
- Toy Model


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

- 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

- Toy Model

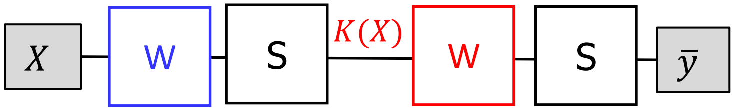

INPUT (X)
One-Dim Vector
(1-by-3)
Weights (W_1)
Bias (b_1)
$a(XW_1+b_1) \rightarrow H_1$
Weights (W_2)
Bias (b_2)
$a(H_1W_2+b_2) \rightarrow H_2$
OUTPUT (Y)
One-Dim Vector
(1-by-1)
H_1
H_2
input
hidden layer 1
hidden layer 2
output
ver
Weights (W_o)
Bias (b_o)
$a(H_2W_o+b_o) \rightarrow Y$

TensorFlow for ANN (XOR with Binary Classification)

```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)
```

```
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
```

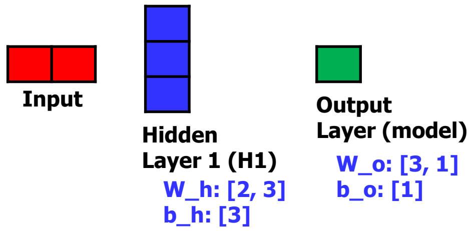
TensorFlow for ANN (XOR with ANN)



$$K(X) = Sigmoid(XW_1 + B_1)$$

$$\bar{Y} = H(X) = Sigmoid(K(X)W + b)$$

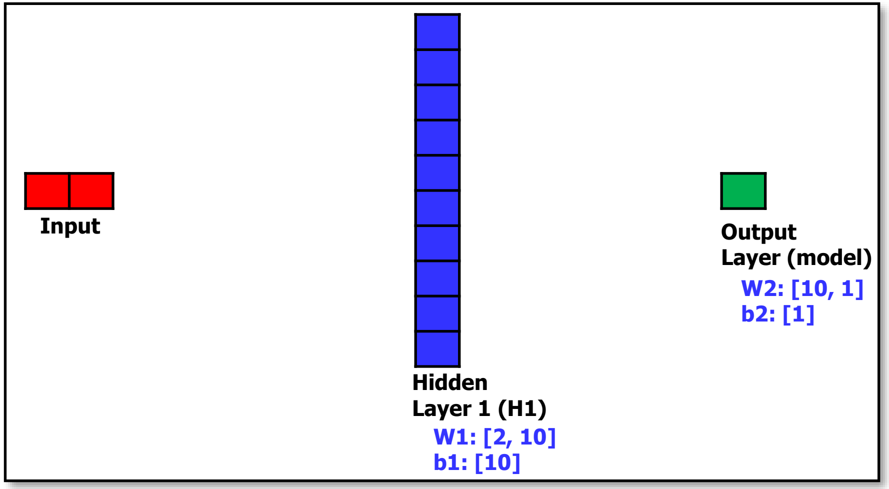
```python
import tensorflow as tf

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
```

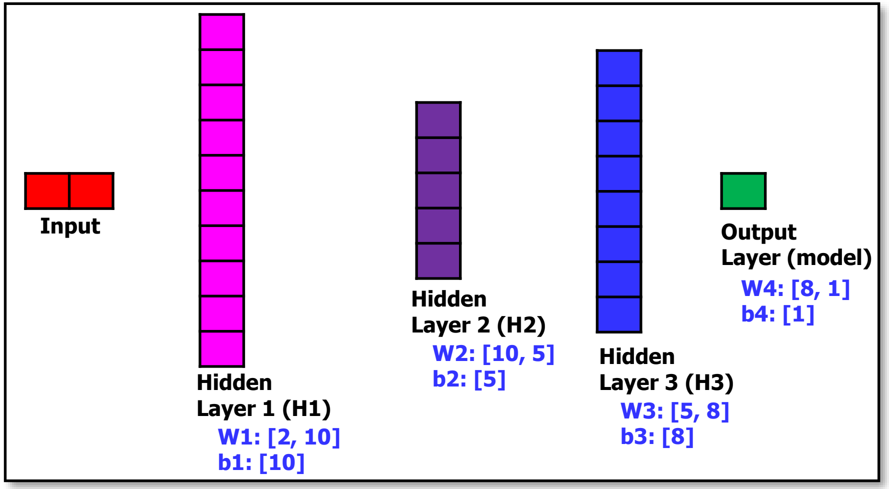
- **Wide** ANN for XOR

```
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

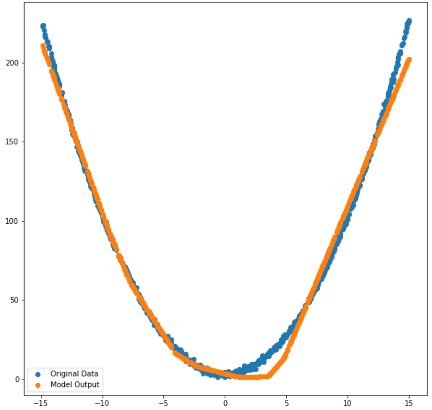
```
W1 = tf.Variable(tf.random_normal([2, 10]))
b1 = tf.Variable(tf.random_normal([10]))
H1 = tf.sigmoid(tf.matmul(X, W1) + b1)

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)
```

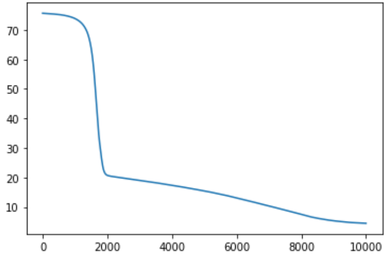


PyTorch for ANN

Artificial Intelligence and Mobility Lab

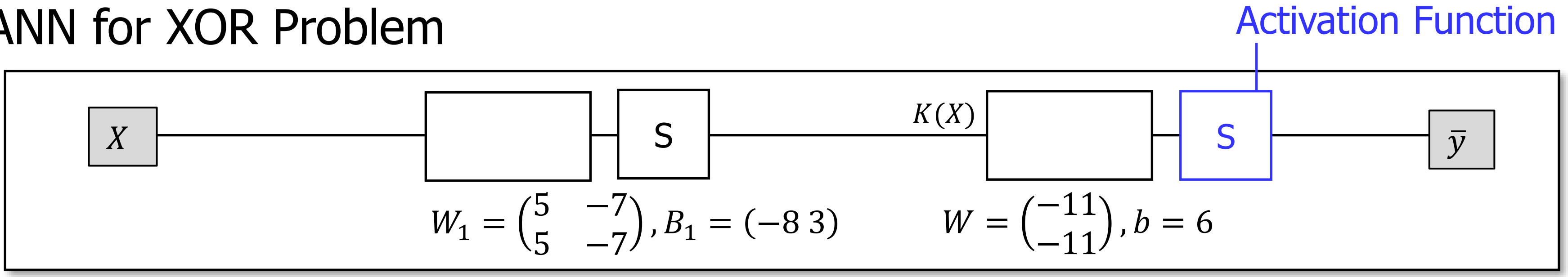
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import matplotlib.pyplot as plt
num_data = 1000
num_epoch = 10000
noise = init.normal_(torch.FloatTensor(num_data,1),std=1)
x = init.uniform_(torch.Tensor(num_data,1),-15,15)
y = (x**2) + 3
y_noise = y + noise
# Model Build
model = nn.Sequential(
        nn.Linear(1,6),
        nn.ReLU(),
        nn.Linear(6,10),
        nn.ReLU(),
        nn.Linear(10,6),
        nn.ReLU(),
        nn.Linear(6,1),
    )
loss_func = nn.L1Loss()
optimizer = optim.SGD(model.parameters(),lr=0.0002)
# Model Training
loss_array = []
for i in range(num_epoch):
    optimizer.zero_grad()
    output = model(x)
    loss = loss_func(output,y_noise)
    loss.backward()
    optimizer.step()
    loss_array.append(loss)
# Loss Graph
plt.plot(loss_array)
plt.show()
# Result Visualization
plt.figure(figsize=(10,10))
plt.scatter(x.detach().numpy(),y_noise,label="Original Data")
plt.scatter(x.detach().numpy(),output.detach().numpy(),label="Model Output")
plt.legend()
plt.show()
```



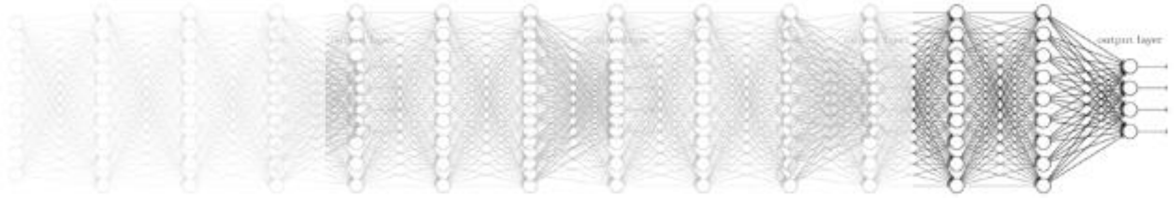


ANN: ReLU (Rectified Linear Unit)

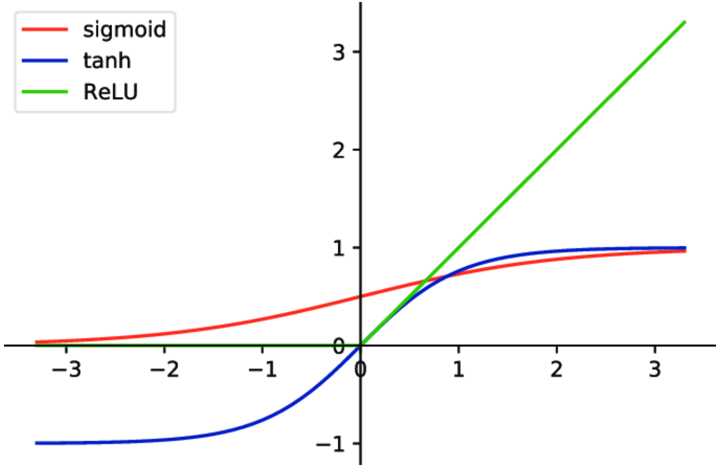
- ANN for XOR Problem

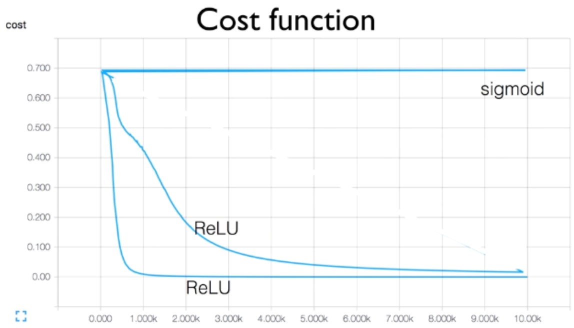


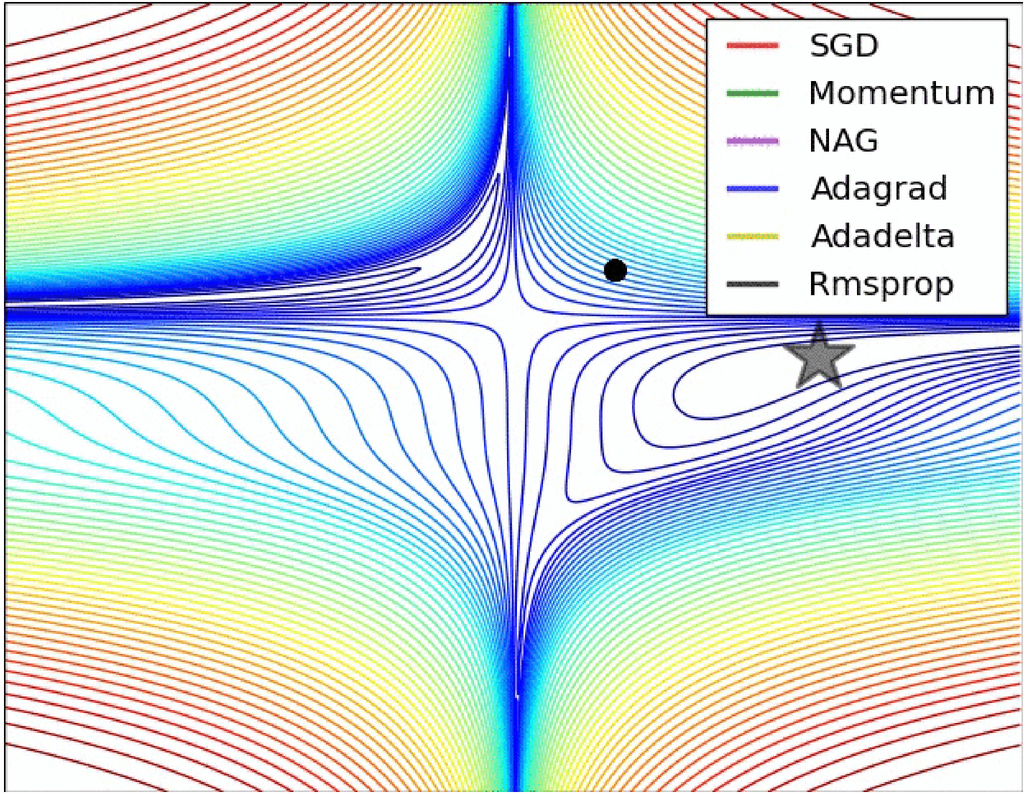
- Observation)
  - There exists cases when the accuracy is low even if the # layers is high. Why?
  - Answer)
    - The result of one ANN is the result of sigmoid function (**between 0 and 1**).
    - The numerous multiplication of this result converges to near zero.
      → **Gradient Vanishing Problem**



ANN: ReLU (Rectified Linear Unit)







- Deep Learning Revolution is Real

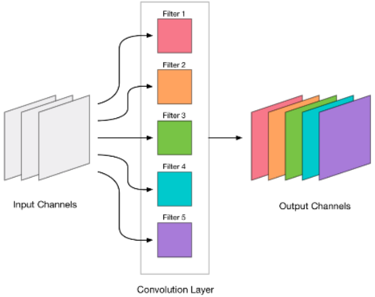
Our **labeled datasets** were **too small**.

Big-Data

Our **computers** were millions of times **too slow**.

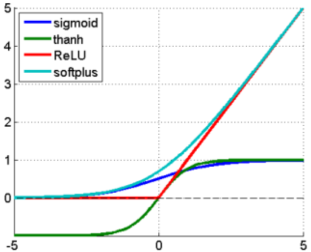
GPU

We only consider **one-dimensional vector** as an input.



Convolution Layers for Multi-Dimensional Inputs

We used the **wrong type of non-linearity (activation function).**



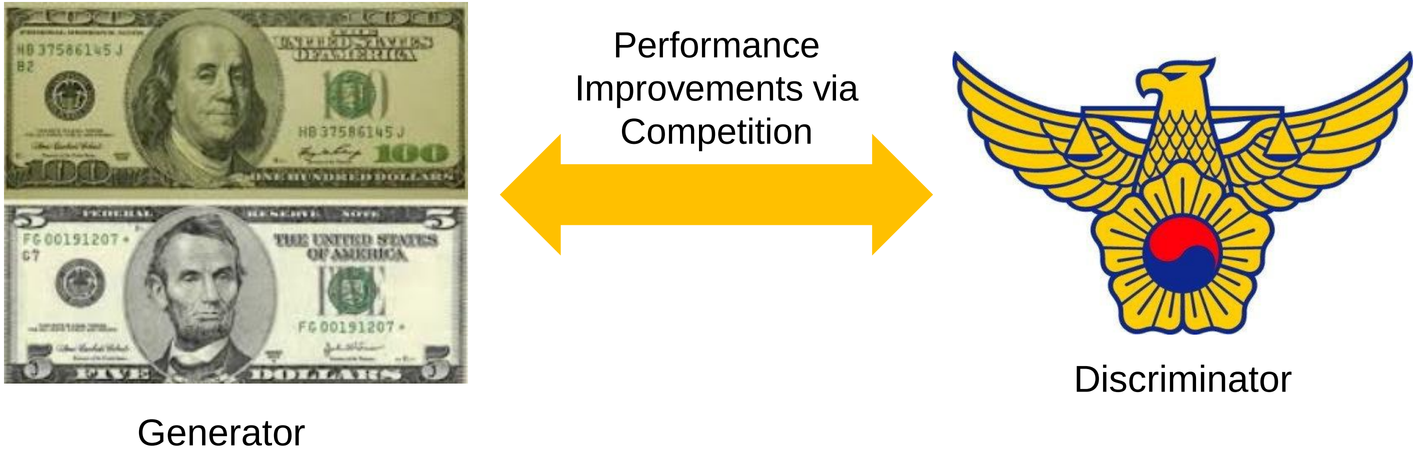
ReLU for solving Gradient Vanishing Problem

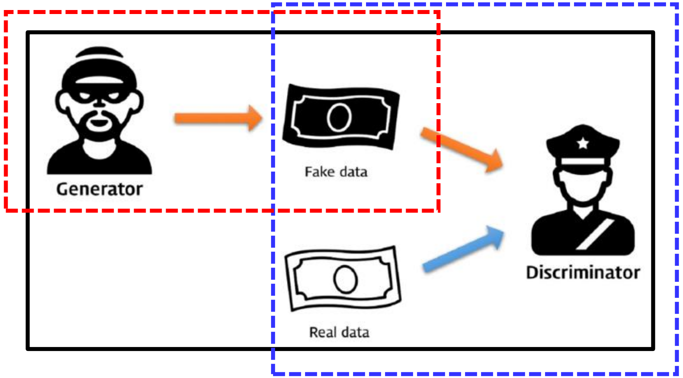
Deep Learning Basics and Software

Generative Adversarial Network (GAN)

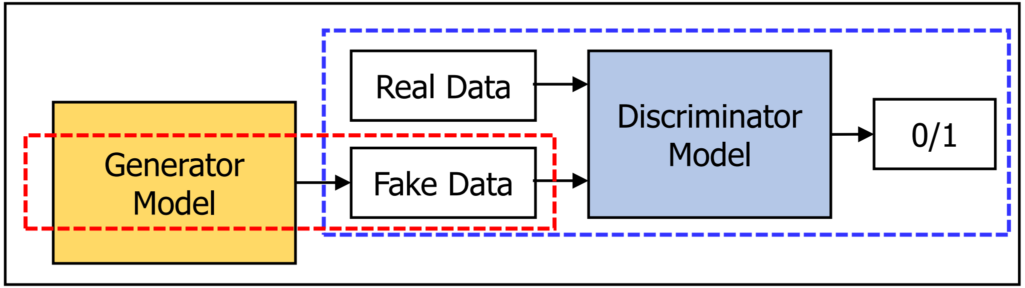
GAN Introduction

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





GAN architecture



Discriminator Model

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

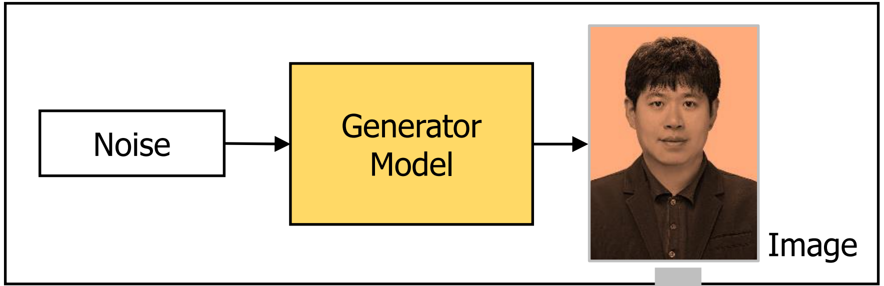
Supervised Learning

Generator Model

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

Unsupervised Learning

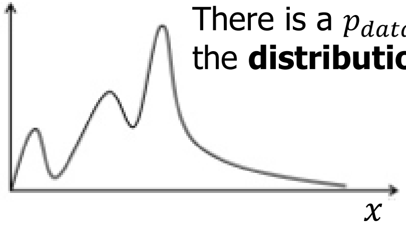
• Generative Model



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

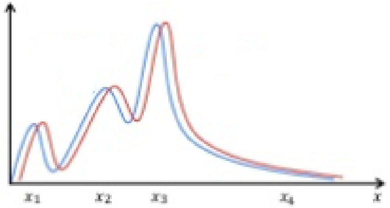
$p_{data}(x)$

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

Generator Model → Fake Data → Discriminator Model

Real Data → Discriminator Model → 0/1

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

x → 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

Random Noise (Latent Code)

NN

z → G → Fake Image

Fake image generated by G.

Fake Image

NN

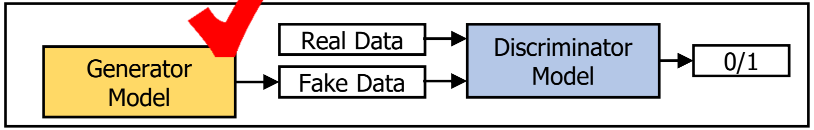
Fake Image → D → D(G(z))

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

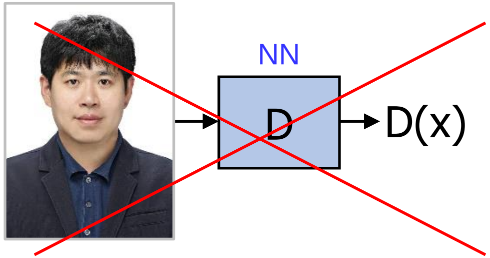
Sample latent code x from Gaussian distribution

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

G should minimize V(D,G)

Minimum when D(G(z))=1

Training with REAL



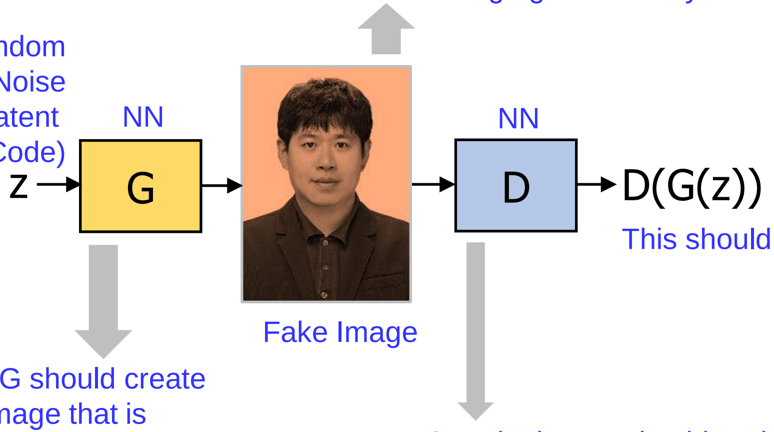
x

G has no inputs.

Training with FAKE

Fake image generated by G.

Random Noise (Latent Code)



This should be 1.

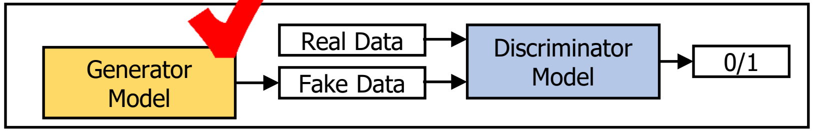
Fake Image

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

Our aim is to make this D be foolish.

GAN Introduction

GAN architecture



Objective of G

G is independent to this part

Sample latent code x from Gaussian distribution

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

G should minimize V(D,G)

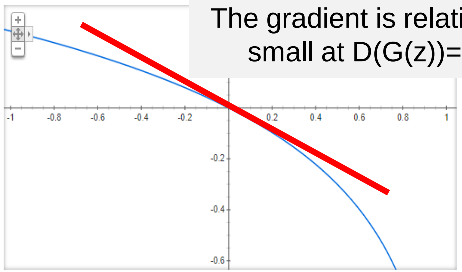
Minimum when D(G(z))=1

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

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

- 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.

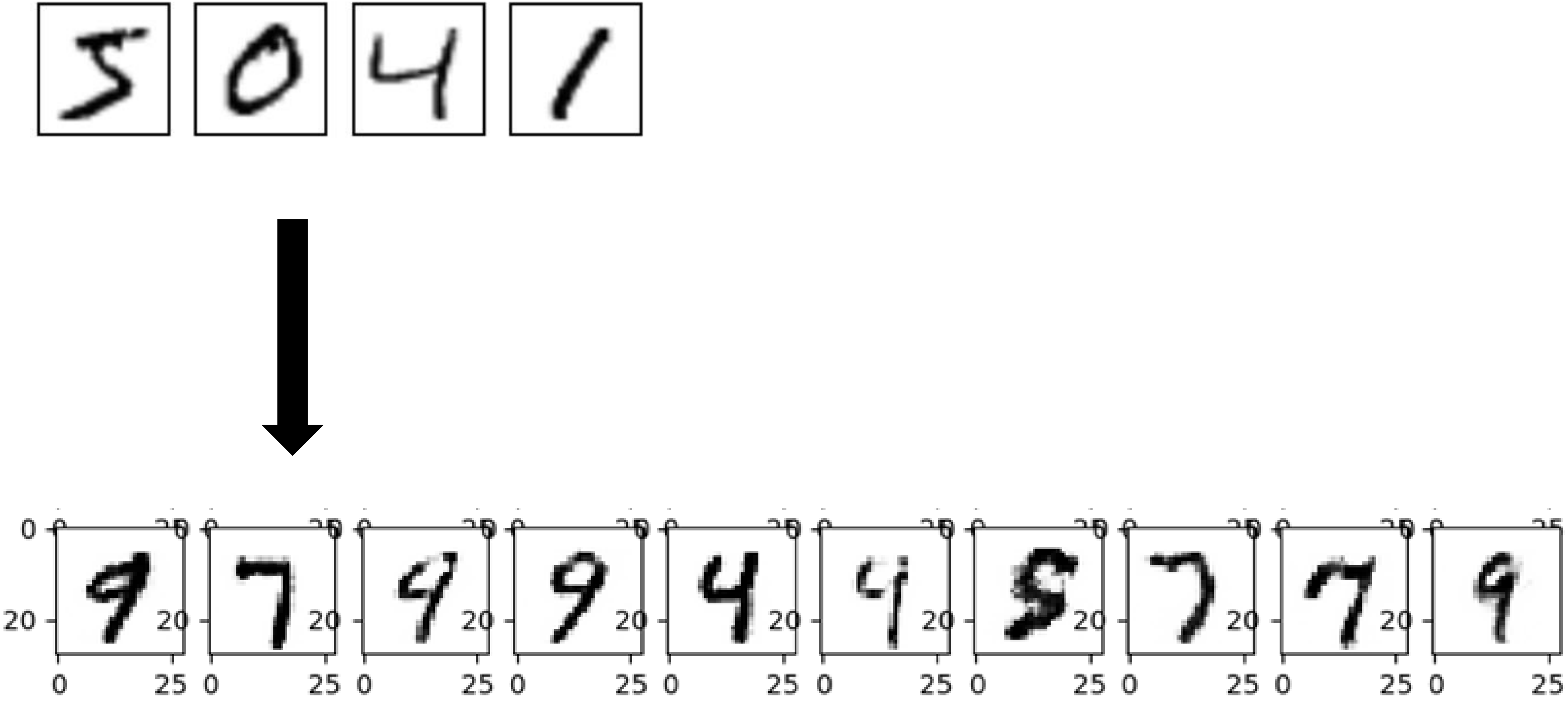
Deep Learning Theory and Software

Generative Adversarial Networks (GAN)

GAN Implementation

- GAN Theory
- **GAN Implementation**

GAN Introduction (TensorFlow for GAN)



Artificial Intelligence and Mobility Lab

```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.))
```

```python
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])
```

GAN Introduction (TensorFlow for GAN)

```
# Build Generator Neural Network
sample_G = nn_G(IN_G)

# Build Discriminator Neural Network (one from noise input, one from generated samples)
D_real = nn_D(IN_D)
D_fake = nn_D(sample_G)
vars_G = [W['HL_G'], W['OL_G'], b['HL_G'], b['OL_G']]
vars_D = [W['HL_D'], W['

# Cost, Train
cost_G = -tf.reduce_mean(tf.log(D_fake))
cost_D = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
train_G = tf.train.AdamOptimizer(0.0002).minimize(cost_G, var_list=vars_G)
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)

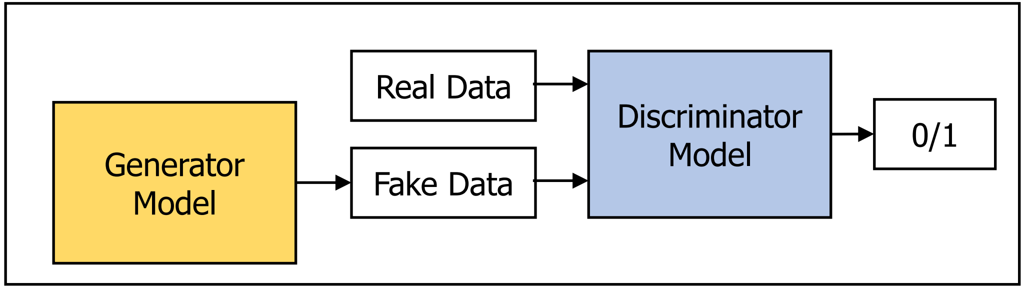
Maximize when D(x)=1

Maximize when D(G(z))=0

GAN Introduction (TensorFlow for GAN)

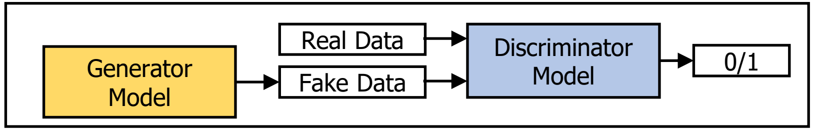

```
# Build Generator Neural Network
sample_G = nn_G(IN_G)

# Build Discriminator Neural Network (one from noise input, one from generated samples)
D_real = nn_D(IN_D)
D_fake = nn_D(sample_G)
vars_G = [W['HL_G'], W['OL_G'], b['HL_G'], b['OL_G']]
vars_D = [W['HL_D'], W['OL_D'], b['HL_D'], b['OL_D']]

# Cost, Train
cost_G = -tf.reduce_mean(tf.log(D_fake))
cost_D = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
train_G = tf.train.AdamOptimizer(0.0002).minimize(cost_G, var_list=vars_G)
train_D = tf.train.AdamOptimizer(0.0002).minimize(cost_D, var_list=vars_D)
```

GAN architecture



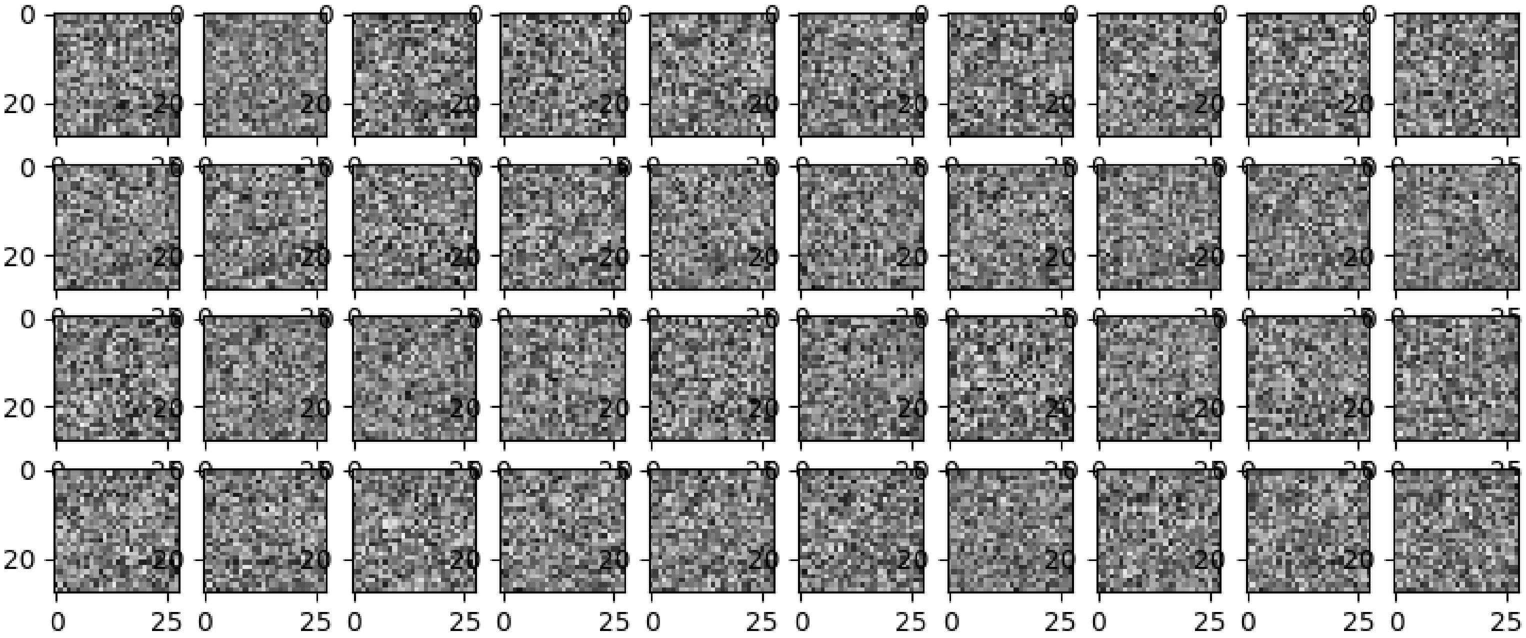
GAN Introduction (TensorFlow for GAN)

GAN architecture

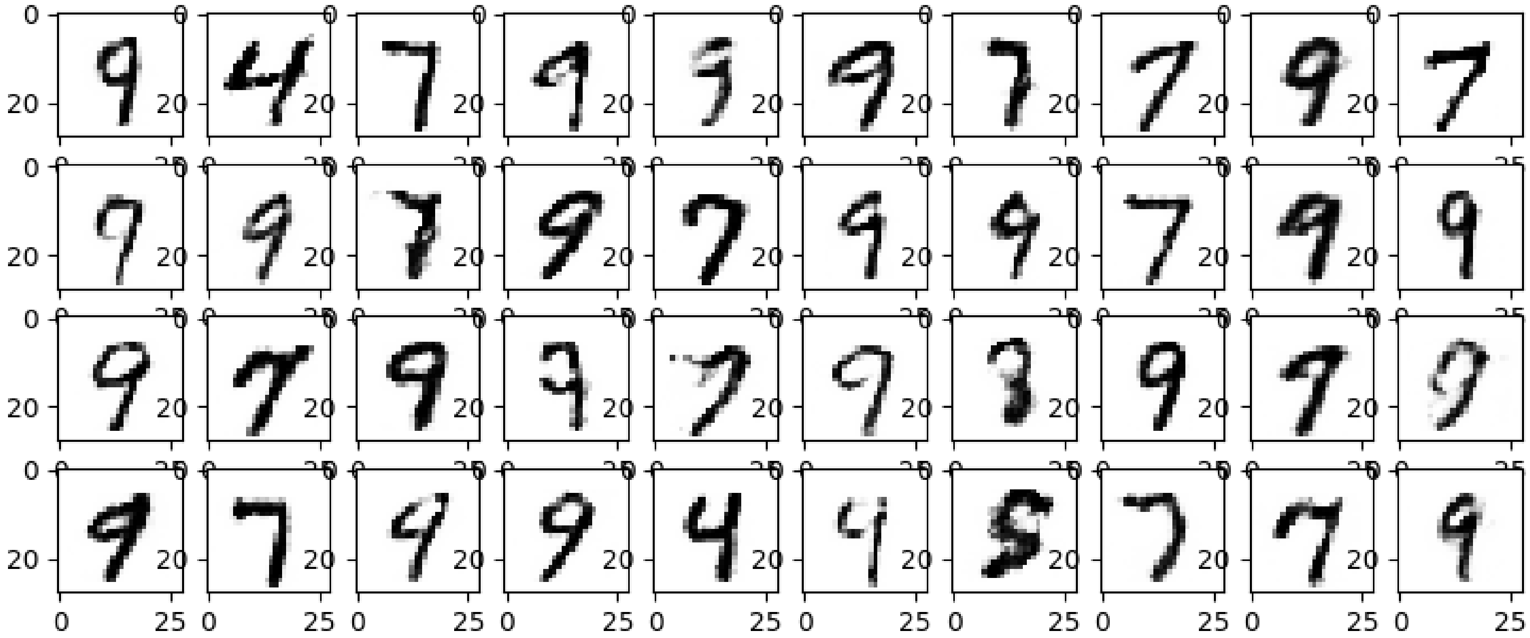



```python
# Session
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(1, num_steps+1):
        # Get the next batch of MNIST data
        batch_images, _ = mnist.train.next_batch(batch_size)
        # Generate noise to feed to the generator G
        z = np.random.uniform(-1., 1., size=[batch_size, dim_noise])
        # Train
        sess.run([train_G, train_D], feed_dict = {IN_D: batch_images, IN_G: z})
        f, a = plt.subplots(4, 10, figsize=(10, 4))
        for i in range(10):
            z = np.random.uniform(-1., 1., size=[4, dim_noise])
            g = sess.run([sample_G], feed_dict={IN_G: z})
            g = np.reshape(g, newshape=(4, 28, 28, 1))
            # Reverse colors for better display
            g = -1 * (g - 1)
            for j in range(4):
                # Generate image from noise. Extend to 3 channels for matplot figure.
                img = np.reshape(np.repeat(g[j][:, :, np.newaxis], 3, axis=2), newshape=(28, 28, 3))
                a[j][i].imshow(img)
    f.show()
    plt.draw()
    plt.waitforbuttonpress()
```

num_steps: 1



num_steps: 100000



Thank you for your attention!

- More questions?
  - joongheon@korea.ac.kr
- More details?
  - https://joongheon.github.io/