

Deep Learning 기초

-Sung Kim 교수 동영상 자료 인용

2020. 11. 9(월)

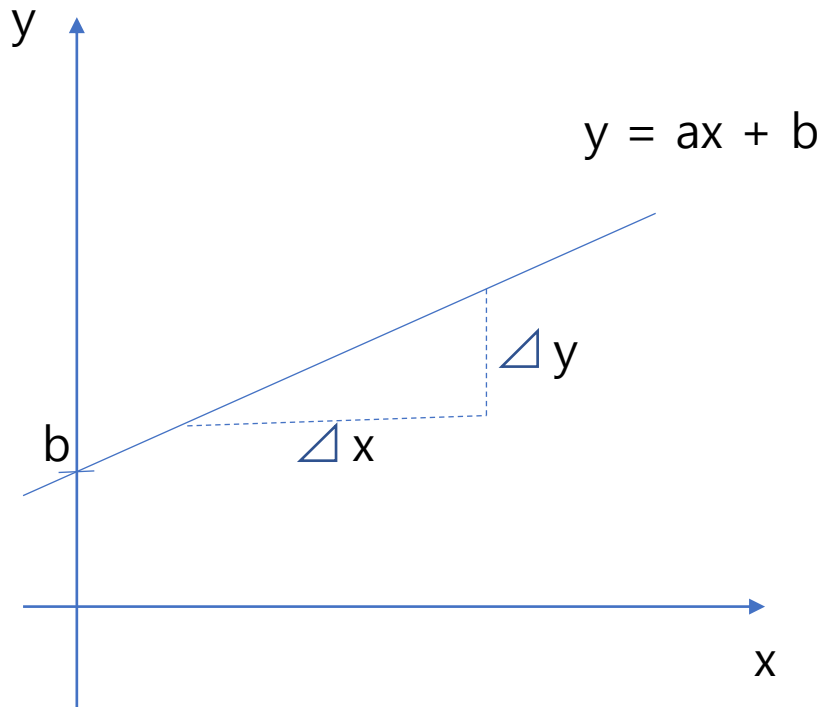
- 확률 이론
- 확률 분포(Probability Distribution)의 개념
 - 확률변수(Random variable)
 - . 이산(discrete) 확률 변수와 연속(continuous) 확률 변수
 - 확률함수와 확률밀도함수
(probability function and probability density function)
- 확률분포의 기대값과 분산
(Expected Value and Variance)
- 이산확률분포(Discrete Probability Distribution)
 - 이항분포
 - 다항분포
- 연속확률분포(Continuous Probability Distribution)
 - 균일분포(uniform distribution)
 - 정규분포(Normal distribution)
- 표본 및 표집분포

- 통계적 추정
- 단일모집단에 관한 가설검정
- 두 모집단에 관한 가설검정
- 분산분석
- 상관분석과 회귀분석의 기초
- 회귀분석의 통계적 추정
- 비모수 통계학

- 제1부 사전주제
 - 서문
 - 데이터 마이닝 프로세스 개요
- 제2부 데이터 탐색 및 차원축소
 - 데이터 시각화
 - 차원축소
- 제3부 성능평가
 - 분류와 예측의 성능평가
- 제4부 예측 및 분류 방법
 - 다중선형회귀분석
 - K-근접 이웃분석
 - 나이브 베이즈
 - 분류회귀나무
 - 로지스틱 회귀분석
 - 신경망
 - 판별분석
- 제5부 레코드들 간의 관계 마이닝
 - 연관규칙
 - 군집분석
- 제6부 시계열 예측
 - 시계열 데이터 분석
 - 회귀분석을 기반으로 한 예측
 - 평활법
- 제7부 사례

일차함수와 미분

- 일차함수 : 기울기 a , y 절편 b



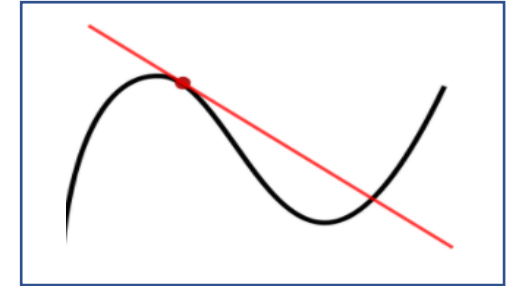
- 함수의 기울기

$$\bar{k} = \frac{\Delta y}{\Delta x} = \frac{f(x) - f(a)}{x - a}$$

- 미분(derivative) 또는 도함수

- 함수의 점에서의 미분은 그 점에서의 접선의 기울기와 같다.

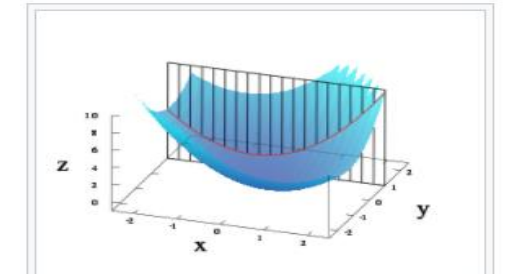
- 표기법 : $\frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f$



- 편미분(Partial derivative)

- 다변수 함수의 특정 변수를 제외한 나머지 변수들을 상수로 생각하여 미분하는 것이다.

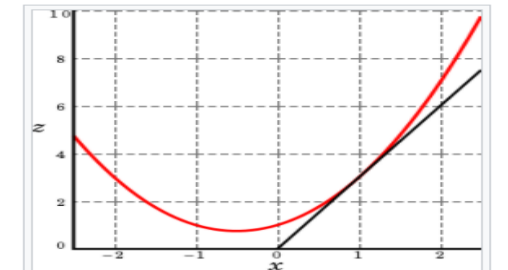
$$z = f(x, y) = x^2 + xy + y^2 \quad \frac{\partial z}{\partial x} = 2x + y$$



$z = x^2 + xy + y^2$ 의 그래프. $y = 1$ 로 잘라보면, xz -평면과 평행하는 빨간색 곡선을 얻으며, 점 $(1, 1)$ 에서 곡선의 접선은 역시 xz -평면과 평행한다.

- 체인 룰(chain rule)

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$



위 그래프의 평면 $y = 1$ 에 의한 절단면. 점 $(1, 1)$ 에서의 접선의 기울기는 3이다.

압정 던지기와 Maximum Likelihood Estimation(MLE) – (1/2)

인용자료 : KOOC 문일철 교수 동영상 강의 자료
인공지능 및 기계학습 개론 I

○ 압정던지기 실험

○ 압정을 던져서 Head와 Tail 이 나오는 실험에서(Bernoulli experiment, i.i.d)

- Head 가 나올 확률 : $P(H) = \theta$
- Tail 이 나올 확률: $P(T) = 1 - \theta$

○ i.i.d 조건을 만족하고 있는 상황에서, 압정을 5번 던진 경우,

- HHTHT 이 나온 경우,
- $P(HHTHT) = \theta\theta(1-\theta)\theta(1-\theta) = \theta^3(1-\theta)^2$

• Binomial distribution

- 연속적인 것이 아닌 discrete 한 이산적인 사건(true, false 또는 앞면, 뒷면 등)의 확률 분포
- . 압정을 던지면 Head 또는 Tail 이 나오는 이산적인 사건이니까, 확률 분포라는 것이 있을 수 있다.

○ Maximum Likelihood Estimation

○ $P(D/\theta) = \theta^{a_H}(1 - \theta)^{a_T}$

- θ 가 주어진 상황에서 Data가 관측될 확률을 정의한 것
. D(data)는 Head 와 Tail로 구성된 우리가 관측한 Data 이다.

- 우리의 **가설(Hypothesis)**
. 압정 던지기 결과는 θ 라는 **binomial 확률 분포를 따른다는 가설**.
. 어떻게 하면 우리의 가설이 강해 질 수 있나(가설이 '참' 인 방향으로)

→ ① 만약 binomial distribution 보다 더 좋은 분포가 있다면,
그것을 따르면 된다(더 많은 데이터와 실험이 필요할 것임)

→ ② **binomial distribution**을 따른다는 것을 인정하고,
 θ 를 최적화해서 상기 가설을 강하게 하다. 즉, **Best condition of θ 를 찾는 과정**임.
어떤 θ 를 선택했을 때, Data를 가장 잘 설명할 수 있을까, 그것을 찾는 것이 **확률의 요체**라 할 수 있음.

압정 던지기와 Maximum Likelihood Estimation(MLE) – (2/2)

○ Maximum Likelihood Estimation (계속)

- Data를 잘 설명할 수 있는 θ 를 선택할 수 있는, 한 가지 방법이 MLE(확률의 추론)이다.

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(D/\theta)$$

- . $P(D/\theta)$ 를 최대화 하는 argument θ 를 찾아내고, 그것을 $\hat{\theta}$ 라고 함.
" θ 가 주어졌을 때, Data를 관측할 확률을 알 수 있고, 이것을 최대화하는 θ 를 찾아내자는 것이다. "

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(D/\theta) = \operatorname{argmax}_{\theta} \theta^{a_H} (1 - \theta)^{a_T}$$

- . 계산을 진행하기가 어려워 log function을 이용한다.

- . P 가 최대화 되는 점은 마찬가지로 $\ln P$ 가 최대화되는 점과 동일하다는 특성을 이용한다.

$$\begin{aligned} - \hat{\theta} &= \operatorname{argmax}_{\theta} \ln(P(D/\theta)) = \operatorname{argmax}_{\theta} \ln(\theta^{a_H} (1 - \theta)^{a_T}) \\ &= \operatorname{argmax}_{\theta} (a_H \ln \theta + a_T \ln(1 - \theta)) \end{aligned}$$

- . 그러면 이제 최대화 문제가 된다.

여기에서 θ 를 optimize 하여, 이 수식을 최대화 시켜 주는 것이 된다.

최대값, 최소값을 구하는 방법을 이용, 즉 미분하여 zero로 놓고 푸는 방법 (극점을 이용하는 방법)

- 극점을 이용하는 방법 활용

$$\cdot \frac{d}{d\theta} (a_H \ln \theta + a_T \ln(1 - \theta)) = 0$$

$$\frac{a_H}{\theta} - \frac{a_T}{1 - \theta} = 0$$

$$\theta = \frac{a_H}{a_H + a_T} \rightarrow \frac{\text{헤드가 나온 횟수}}{\text{던져진 횟수}}$$

- 우리가 흔히 알고 있는 (관측된 횟수/전체 던져진 횟수),

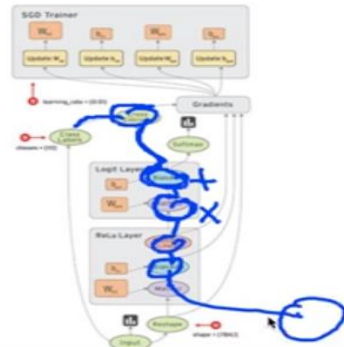
- . 이러한 간략한 확률이라는 것이, 사실은 binomial distribution, MLE(Maximum Likelihood Estimation), 최적화 과정을 거쳐서 나온 수식이 된다.
- . 이것이 MLE 관점에서 본, 최적화된 $\hat{\theta}$ (수정된, 최적화된 파라미터 값)이 된다.

TensorFlow

- TensorFlow™ is an open source software library for numerical computation using data flow graphs.
- Python!

What is a Data Flow Graph?

- Nodes in the graph represent mathematical operations
- Edges represent the multidimensional data arrays (tensors) communicated between them.



- 그래프는 노드(node), 노드와 노드를 연결하는 엣지(edge)로 구성된다.
- Data Flow Graph는 노드가 하나의 operation이다.
- Edge는 데이터(데이터 어레이)이며, 텐서(tensor) 라고도 하며, 이들이 노드로 들어와서 동작(연산)이 된다.
- 이러한 일련의 동작 결과로 내가 원하는 결과를 얻을 수 있게 된다.
- 노드를 따라서 tensor가 돌아다닌다(흘러 다닌다) 하여 tensorflow 라고 함.

Hello TensorFlow!

import tensorflow as tf

```
# Create a constant op  
# This op is added as a node to the default graph  
hello = tf.constant("Hello, TensorFlow!")
```

```
# start a TF session  
sess = tf.Session()
```

```
# run the op and get result  
print(sess.run(hello))
```

```
b'Hello, TensorFlow!'
```

Graph 에 하나의 노드가 있고,
그 노드에 'Hello, TensorFlow'
의 문자열이 들어 있는 것이다.

Computational Graph를 실행하기
위해서는 Session()을 만들고,

sess.run()으로 실행할 수 있는데,
위에서 만들어 놓은 hello 라는 노드
를 실행한다.

- Tensorflow에서 tf.constant() 라는 노드를 한 개 만들고, session() 을 만든 다음에, 그 노드를 실행 시킨 것이다.

Placeholder

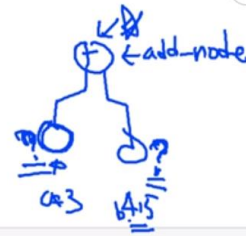
```

a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
add_node = a + b # + provides a shortcut for tf.add(a, b)

print(sess.run(add_node, feed_dict={a: 3, b: 4.5}))
print(sess.run(add_node, feed_dict={a: [1,3], b: [2, 4]}))

```

7.5
[3. 7.]



- 그래프는 미리 그려 놓고 실행시키는 단계에서 값들을 넣어 주고 싶을 때, placeholder 라는 특별한 노드를 만들어 준다.

- a, b 라는 두개의 placeholder 노드를 만들고,
- add 노드를 만들었음.
- Session()을 만들고, sess.run()으로 add를 실행시키는데, 이때 feed_dict를 이용하여 계산할 값을 넘겨준다.
- 한 개의 값 뿐만 아니라 array 값을 넣어줄 수도 있다.

[다시 정리하면 다음과 같다]

- (1) Build graph(tensors) using Tensorflow operations
 - 그래프를 정의할 때 placeholder 노드를 정의할 수 있다.
- (2) Feed data and run graph(operation)


```
sess.run(op, feed_dict = {x: x_data})
```

 - placeholder 노드를 만들면 그래프를 실행할 때, feed_dict로 값을 넘겨준다.
- (3) Update variables in the graph
(and return values)

Linear Regression : Hypothesis, Cost Function, Gradient Descent Algorithm – (1/4)

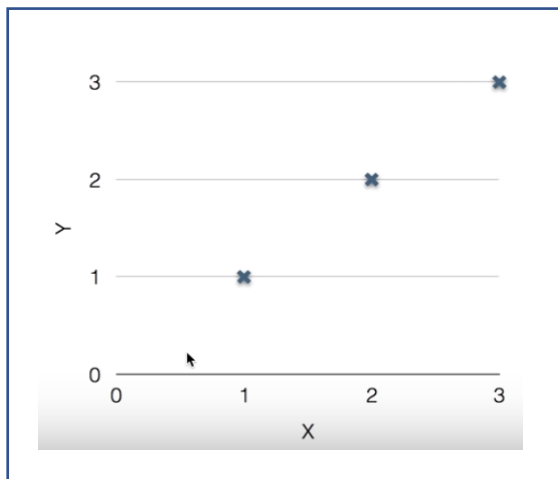
○ 아주 간단한 데이터 예제

- x : feature, y: label

Regression
(Data)

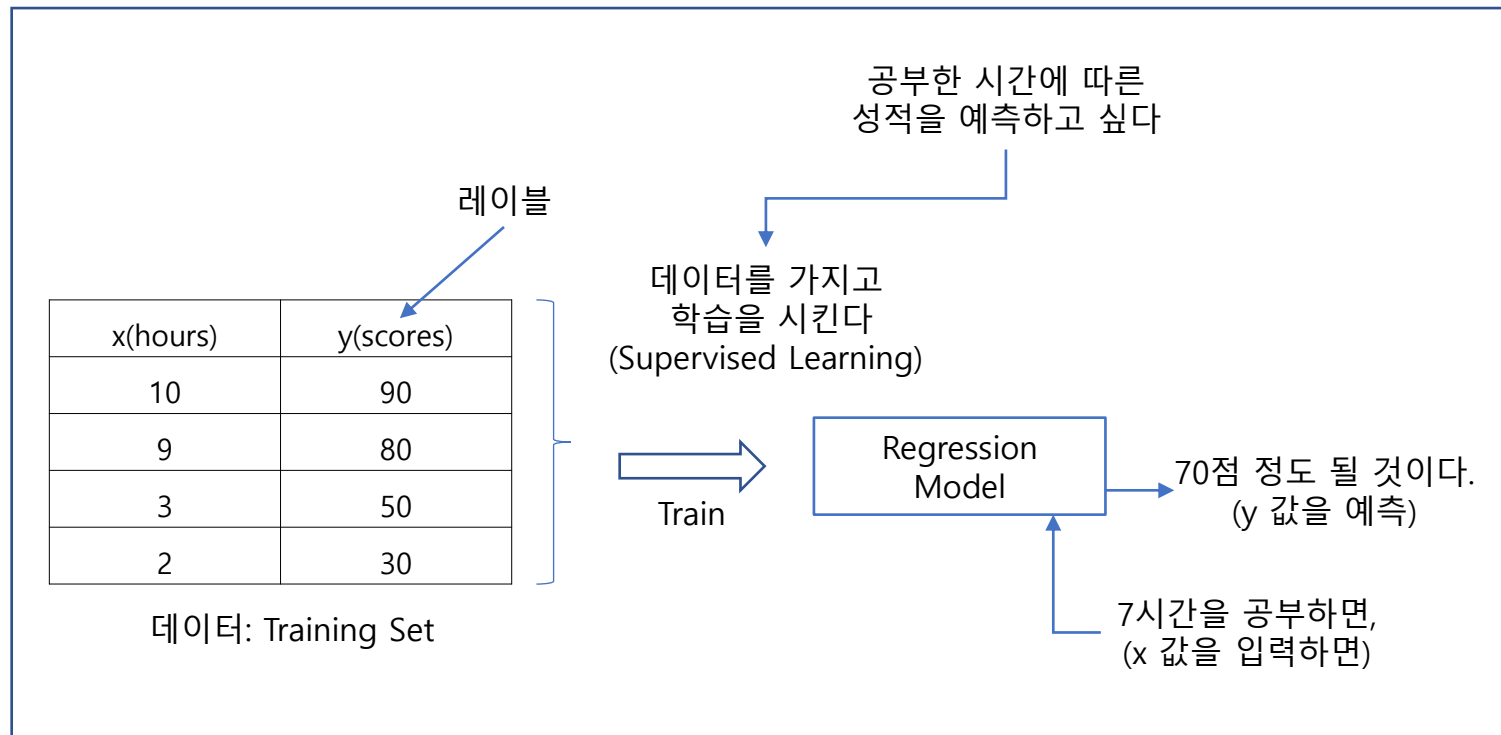
x	y
1	1
2	2
3	3

3개의 학습 데이터를
그래프로 표현



○ 회귀 모델(Regression Model)의 학습

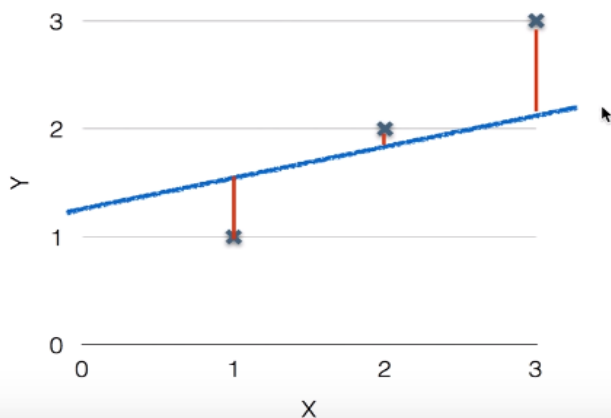
- (Liner) Hypothesis(가설)



Linear Regression : Hypothesis, Cost Function, Gradient Descent Algorithm – (2/4)

- 좋은 가설(Hypothesis) : 실제 데이터와 가설 Data 거리가 가까운 것
- Cost Function(또는 loss Function) : 우리가 세운 가설과 실제 Data의 차이 정도를 나타냄

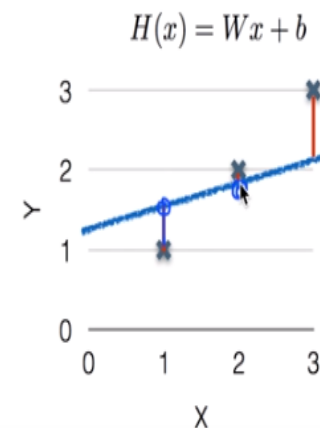
Which hypothesis is better?



- How fit the line to our (training) data

$$\frac{(H(x^{(1)}) - y^{(1)})^2 + (H(x^{(2)}) - y^{(2)})^2 + (H(x^{(3)}) - y^{(3)})^2}{3}$$

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



○ Cost Function은 W와 b의 함수

Cost function

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

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

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

Cost 함수가 최소가 되는 W, b 값을 우리가 가지고 있는 데이터를 통해서 구한다.

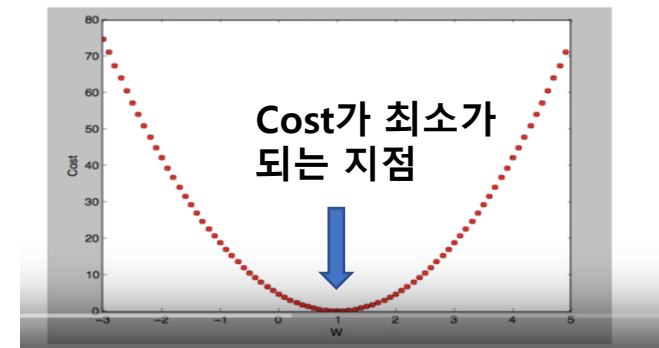
가설의 값 실제 데이터 값

○ What cost(W) looks like ?

- Simplified hypothesis
(설명을 쉽게 하기 위해서 $b = 0$ 가정)

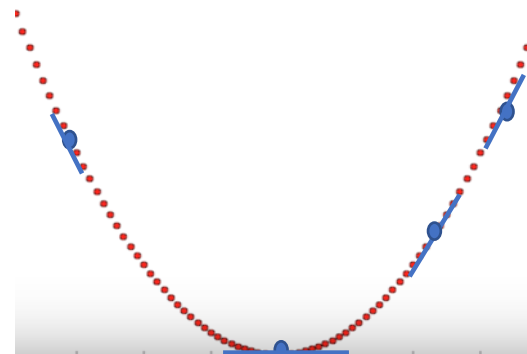
$$H(x) = Wx$$

$$cost(W) = \frac{1}{m} \sum_{k=1}^m (H(x^{(i)}) - y^{(i)})^2$$



How it works ?

- 최저점을 어떻게 찾아 갈 것인가



각 해당 지점에서 경사도를 계산하고, 경사도가 낮은 방향으로 조금씩 내려감.

최종적으로 경사도가 zero가 되는 지점이 최저점이 된다.

○ Gradient Descent Algorithm

- Formal Definition

$$cost(W) = \frac{1}{2m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})^2$$

$$W := W - \alpha \frac{\partial}{\partial W} \frac{1}{2m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})^2$$

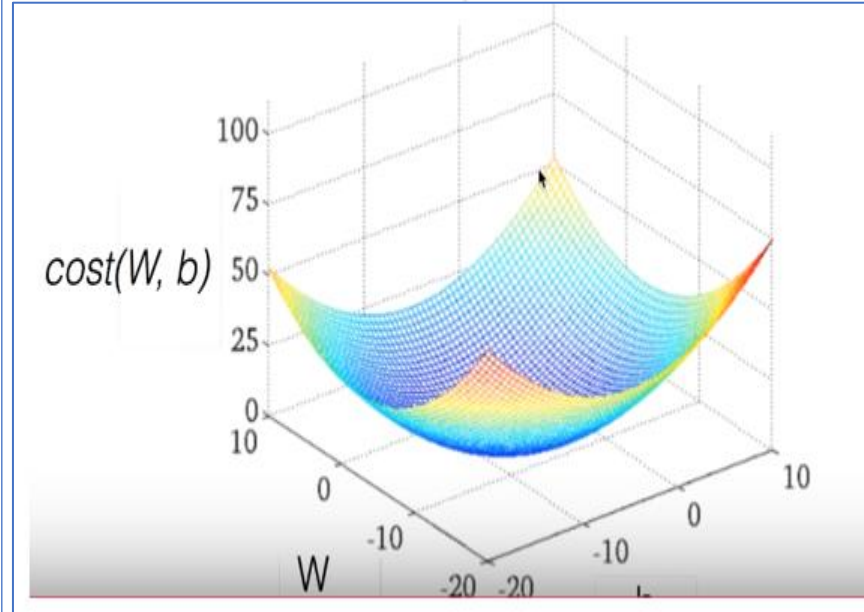
$$W := W - \alpha \frac{1}{2m} \sum_{i=1}^m 2(Wx^{(i)} - y^{(i)})x^{(i)}$$

$$W := W - \alpha \frac{1}{m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})x^{(i)}$$

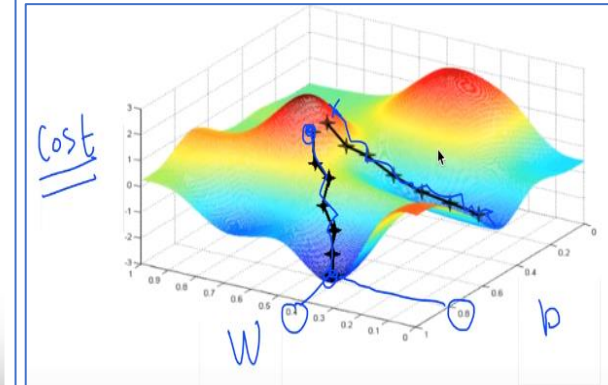
이 수식을 기계적으로 적용만 시키면, cost function을 최소화 하는 W를 구해내고, 그것이 바로 Linear Regression의 핵심인, 학습과정을 통해서 모델을 만든다고 할 수 있다.

○ Convex Function

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



- cost function 이 convex function 이 되는 것이 중요
- 시작점이 어디든 도착점은 항상 최저점
- Gradient Descent algorithm 을 이용



시작점에 따라 서로 다른 최저점으로 내려 온다면, 우리 알고리즘이 잘못된 것이다.

```

##-----
##   placeholder를 이용한 Linear Regression
##   - Gradient Descent의 구현
##-----

import tensorflow as tf

# tf Graph input
w = tf.Variable(tf.random_normal([1]), name = 'weight')

X = tf.placeholder(tf.float32, shape = [None]) # 1차원, element 수는 지정않음.
Y = tf.placeholder(tf.float32, shape = [None])

# Our hypothesis : Linear Model, XW
hypothesis = X * w
# cost/lost function
cost = tf.reduce_mean(tf.square(hypothesis-Y))

##-----
# Minimize : Gradient Descent Magic
#optimizer = tf.train.GradientDescentOptimizer(learning_rate = 0.01)
#train = optimizer.minimize(cost)

learning_rate = 0.01
gradient = tf.reduce_mean((w*X - Y)*X)
descent = w - learning_rate*gradient
update = w.assign(descent)

# Launch the graph in a session
sess = tf.Session( )

# Initialize global variables in the graph
sess.run(tf.global_variables_initializer())

for step in range(201):
    cost_val, update_val = sess.run([cost, update], feed_dict = {X: [1,2,3,4,5],
                                                                    Y: [1,2,3,4,5]})
    if step % 20 ==0 :
        print(step, cost_val, update_val)

```

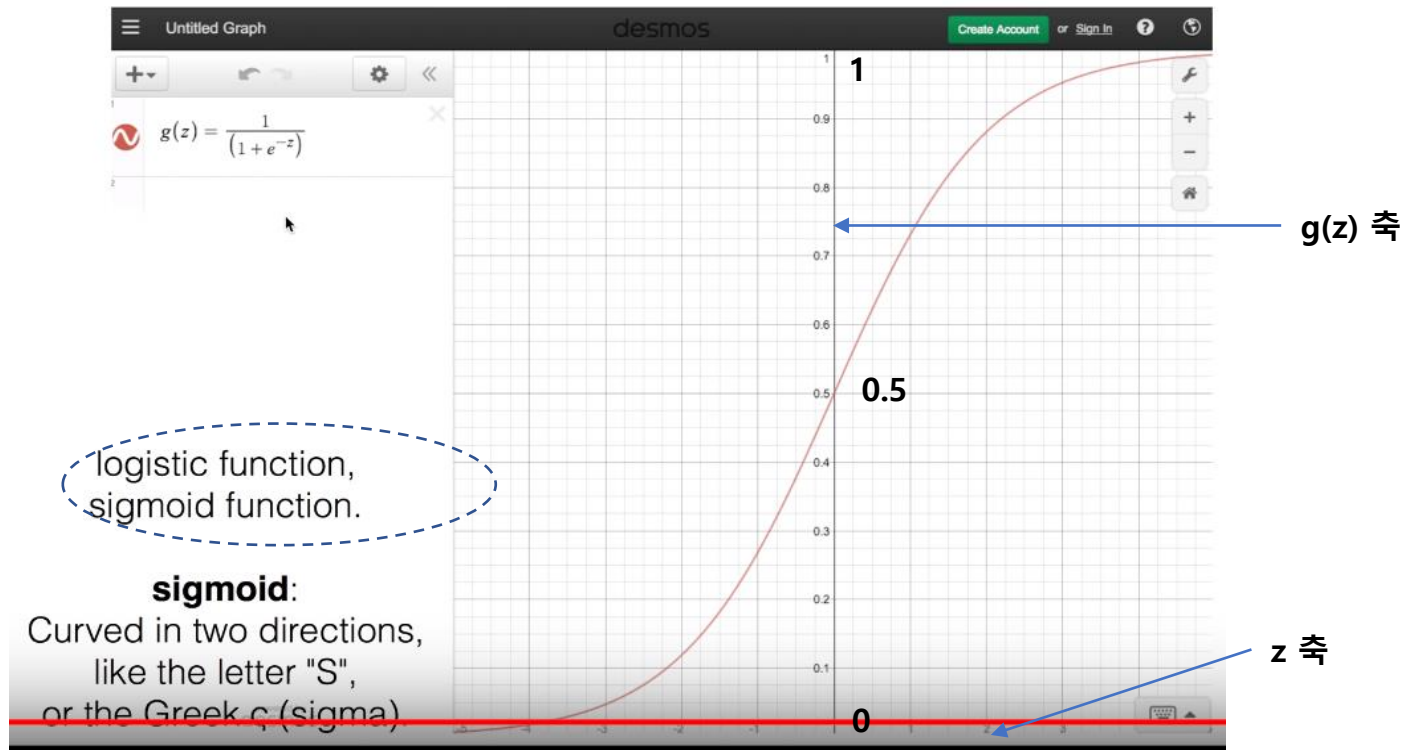
$$W := W - \alpha \frac{1}{m} \sum_{i=1}^m (Wx^{(i)} - y^{(i)})x^{(i)}$$

Logistic (Regression) Classification : Logistic Hypothesis – (1/3)

- Regression은 숫자를 예측하는 것이지만, **Binary Classification**은 두 개 중 한 개의 정해진 **category**를 정하는 것임
- 모든 입력 값에 대해서 0 ~ 1 사이의 값이 나오는 함수가 필요함.
(Binary Classification을 위해서)

$$g(z) = \frac{1}{(1 + e^{-z})} \quad \text{: Sigmoid Function}$$

Logistic Function



Logistic Hypothesis

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

Sigmoid function의 변수 z 자리에
Linear Regression Hypothesis Wx 를 넣은 것이다.

이것이 Logistic Classification의 가설(Hypothesis)의
함수가 된다.

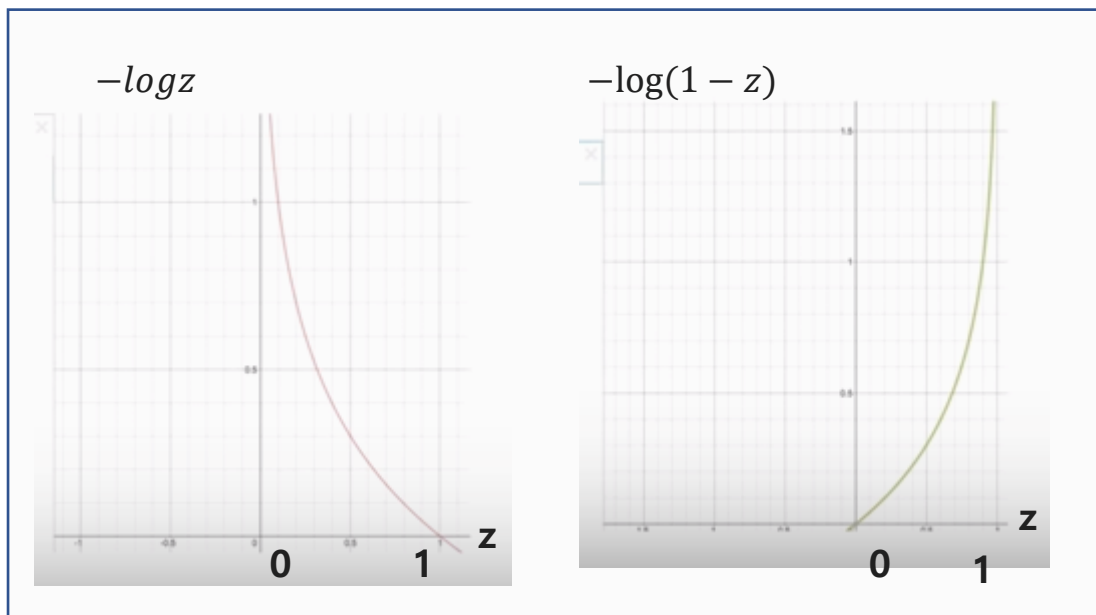
Logistic (Regression) Classification : Logistic Cost Function and Gradient Decent Algorithm – (2/3)

○ cost function for logistic

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

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

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



○ Minimize cost – Gradient decent algorithm

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

$$W := W - \alpha \frac{\partial}{\partial W} \text{cost}(W)$$

- gradient descent 를 적용하기 위해서 cost function의 미분을 하게 되는데, 수식이 복잡하므로 여기서는 생략함.
- 컴퓨터가 미분 계산을 하므로, 이런 알고리즘을 사용한다는 것을 이해하면 됨.
- 텐소플로에서 아래와 같은 library를 이용하여 구현함.


```
# cost function
cost = tf.reduce_mean(-tf.reduce_sum(Y*tf.log(hypothesis)
                                     + (1-Y)*tf.log(1-hypothesis)))

# Minimize
a = tf.Variable(0.1) # Learning rate, alpha
optimizer = tf.train.GradientDescentOptimizer(a)
train = optimizer.minimize(cost)
```

Logistic (Regression) classification

- Hypothesis

$$H_L(x) = Wx$$

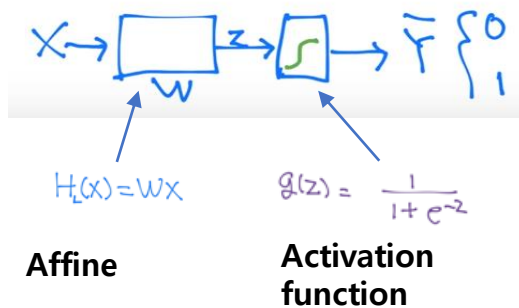
$$z = H_L(x), \quad g(z)$$


- Wx 라는 linear hypothesis로 출발하였지만, 출력 값이 임의의 실수 값이 되므로, logistic(binary) classification에 사용하기에는 부적합
- $H(x) = z$ 라 놓고, $g(z)$ 라는 함수가 있다면, $g(z)$ 이 $H(x)=Wx$ 의 큰 출력 값들을 압축해서 0과 1 사이의 값으로 변경

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

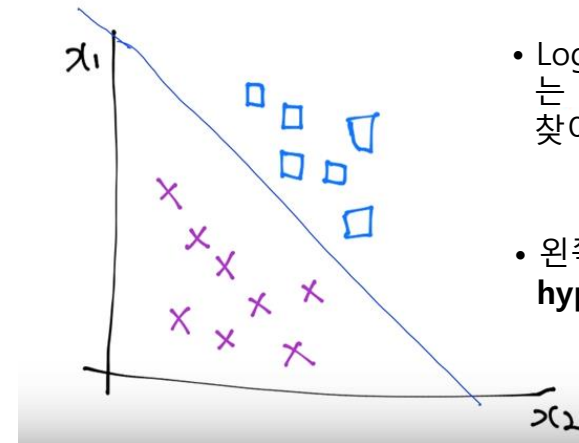
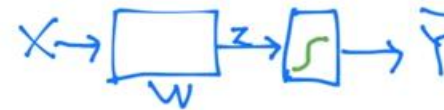
$$H_R(x) = g(H_L(x))$$

Logistic classification의 Hypothesis 함수로 표현



Logistic (Regression) classification

$$g(z) = \frac{1}{1 + e^{-z}} \quad H_R(x) = g(H_L(x))$$



- Logistic classification에서 학습을 한다는 의미는 네모와 엑스를 구분하는 선을 찾아내는 것임.
- 왼쪽 그림은 2차원 이지만, 고차원인 경우 **hyperplane** 이라 함.

Multinomial Classification(Softmax) : 데이터를 여러 개로 분류함 (1/2)

Multinomial Classification

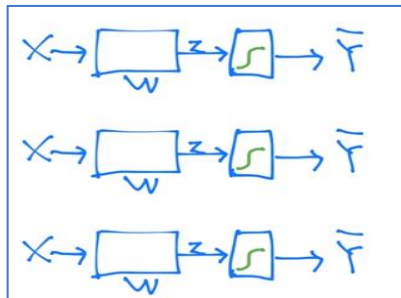
- 데이터를 여러 개로 분류함.

x1 (hours)	x2 (attendance)	y (grade)
10	5	A
9	5	A
3	2	B
2	4	B
11	1	C

3개로 분류

$$XW=Y$$

- 독립된 벡터들로 3번 계산하여 구할 수 있을 것임
- 독립적으로 n(3)번 계산 하는 것은 복잡한 측면이 있음.



독립적인 3개의 W 값을 matrix로 간단하게 표현

Sigmoid가 적용되기 전의 값이며, Sigmoid를 적용하면 각각 0~1 사이 값이 됨

$$\begin{bmatrix} w_{A1} & w_{A2} & w_{A3} \\ w_{B1} & w_{B2} & w_{B3} \\ w_{C1} & w_{C2} & w_{C3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} w_{A1}x_1 + w_{A2}x_2 + w_{A3}x_3 \\ w_{B1}x_1 + w_{B2}x_2 + w_{B3}x_3 \\ w_{C1}x_1 + w_{C2}x_2 + w_{C3}x_3 \end{bmatrix} = \begin{bmatrix} \bar{y}_A \\ \bar{y}_B \\ \bar{y}_C \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}$$

SOFTMAX

$$XW=Y \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$$

SOFTMAX

Softmax Activation function

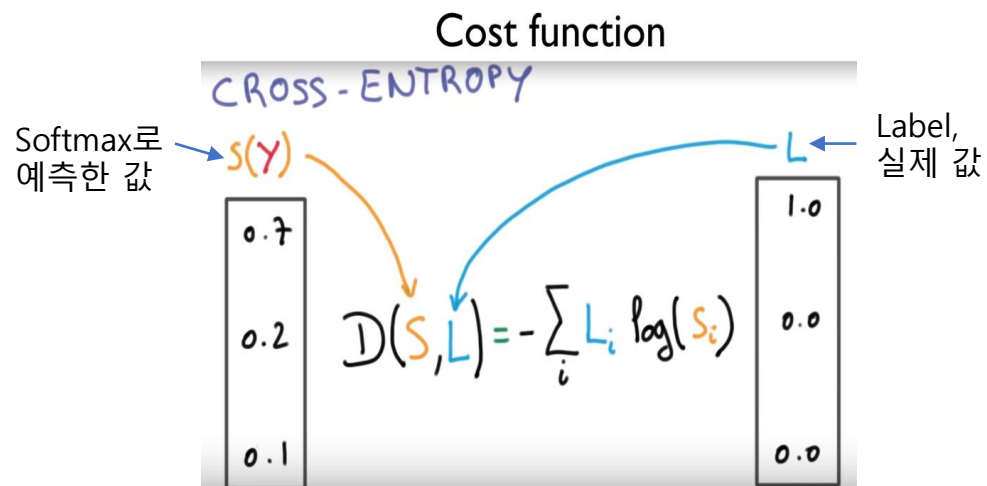
Softmax hypothesis

SCORES → PROBABILITIES

① 0~1
② z > 1

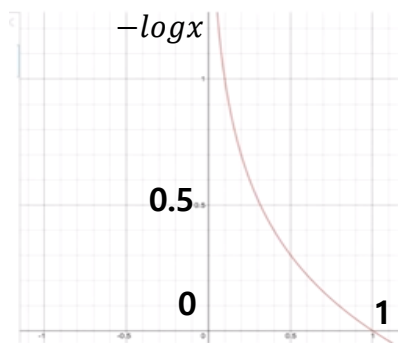
각각은 0 ~ 1 사이의 값이며, 모두 합해서 1 이 됨. 따라서, 각각의 확률로 볼 수 있다.

Multinomial Classification(Softmax) : 데이터를 여러 개로 분류함 (2/2)



- Cross Entropy를 Cost function으로 사용한다.

$$-\sum_i L_i \log(\hat{y}_i) = \sum_i L_i * -\log(\hat{y}_i)$$



\hat{y}_i 의 값은 softmax 결과 값
이므로 0~1 사이 값임

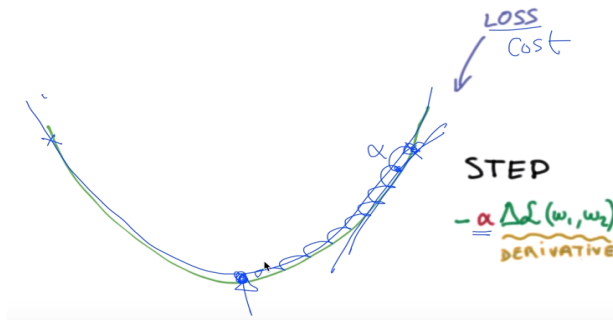
Cost function

LOSS

$$\mathcal{L} = \frac{1}{N} \sum_i \mathcal{D}(S(WX_i + b), L_i)$$

TRAINING SET

Gradient descent



α : learning rate

```
##-----
## Multinomial Classification : softmax , Sung Kim 교수 강의자료
##-----
```

```
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]]
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]]

X = tf.placeholder("float", [None, 4])
Y = tf.placeholder("float", [None, 3])
nb_classes = 3

W = tf.Variable(tf.random_normal([4, nb_classes]), name = 'weight')
b = tf.Variable(tf.random_normal([nb_classes]), name = 'bias')

# tf.nn.softmax computes softmax activations
# softmax = exp(Logits)/reduce_sum(exp(Logits), dim)
hypothesis = tf.nn.softmax(tf.matmul(X, W) + b)

# Cross entropy cost/Loss
cost = tf.reduce_mean(-tf.reduce_sum(Y * tf.log(hypothesis), axis = 1))
optimizer = tf.train.GradientDescentOptimizer(learning_rate = 0.1).minimize(cost)

# Launch graph
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for step in range(2001):
        sess.run(optimizer, feed_dict = {X: x_data, Y: y_data})
        if step % 200 == 0:
            print(step, sess.run(cost, feed_dict = {X: x_data, Y: y_data}))

# Test & one-hot encoding
a = sess.run(hypothesis, feed_dict = {X: [[1, 11, 7, 9]]})
print(a, sess.run(tf.argmax(a, 1)))

all = sess.run(hypothesis, feed_dict = {X: [[1, 11, 7, 9],
                                             [1, 3, 4, 3],
                                             [1, 1, 0, 1]]})

print(all, sess.run(tf.argmax(all, 1)))
```



One-hot encoding 표현.

```
tf.matmul(X, W) + b
hypothesis = tf.nn.softmax(tf.matmul(X, W) + b)
```

LOSS

$$\mathcal{L} = \frac{1}{N} \sum_i \mathcal{D}(s(w x_i + b), L_i)$$

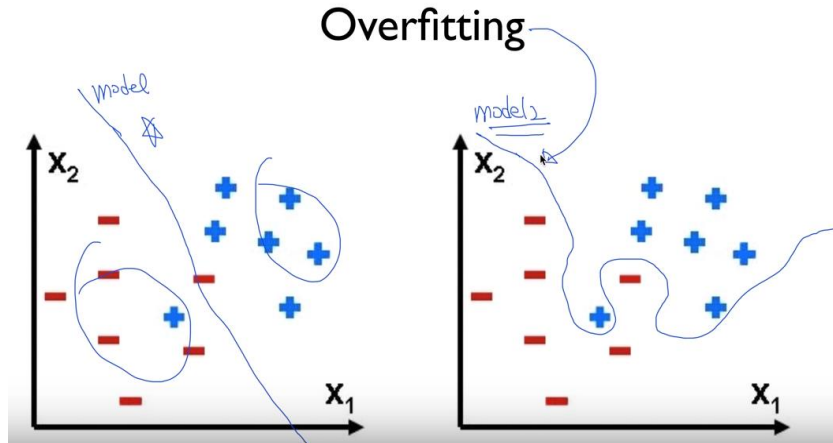
TRAINING SET

$$\mathcal{D}(s, L) = - \sum_i L_i \log(s_i)$$

STEP

$$-\alpha \Delta \mathcal{L}(w, w_2)$$

DERIVATIVE



Solutions for overfitting

- More training data!
- Reduce the number of features
- Regularization

Let's not have too big numbers in the weight
(Weight 가 너무 큰 값을 갖지 않도록 함)

Regularization

- Let's not have too big numbers in the weight

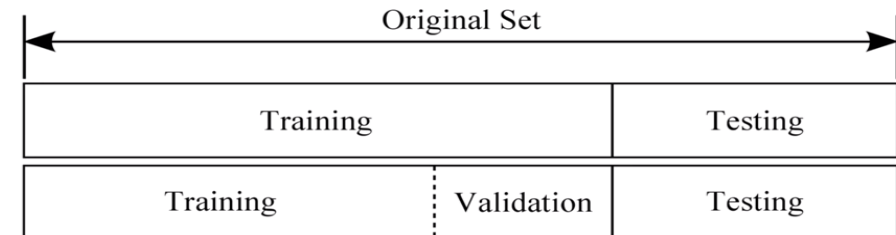
LOSS

$$\mathcal{L} = \frac{1}{N} \sum_i \mathcal{D}(s(w x_i + b), L_i) + \lambda \sum W^2$$

TRAINING SET

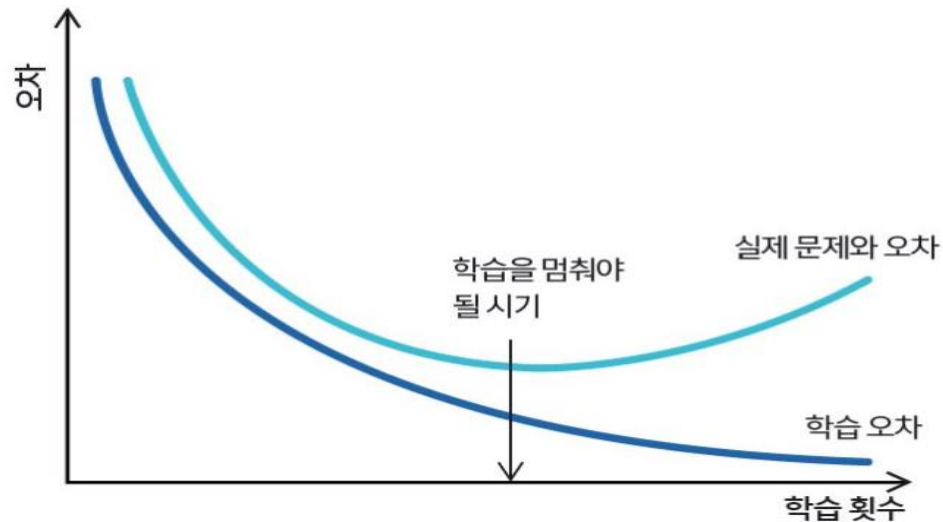
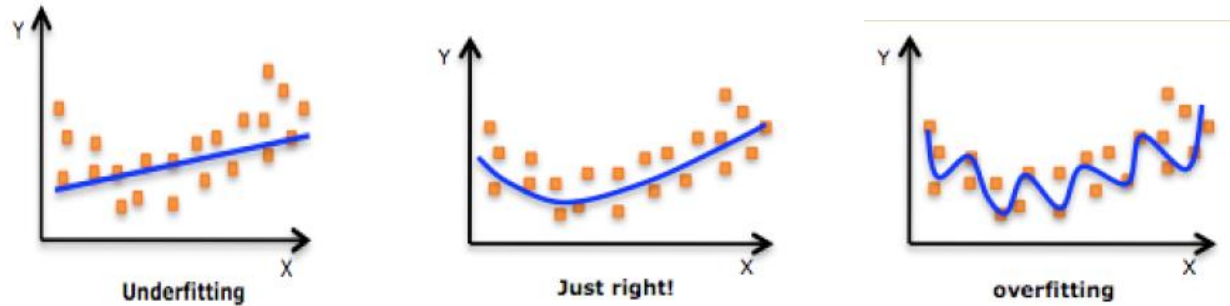
λ : regularization strength

Training, validation and test sets



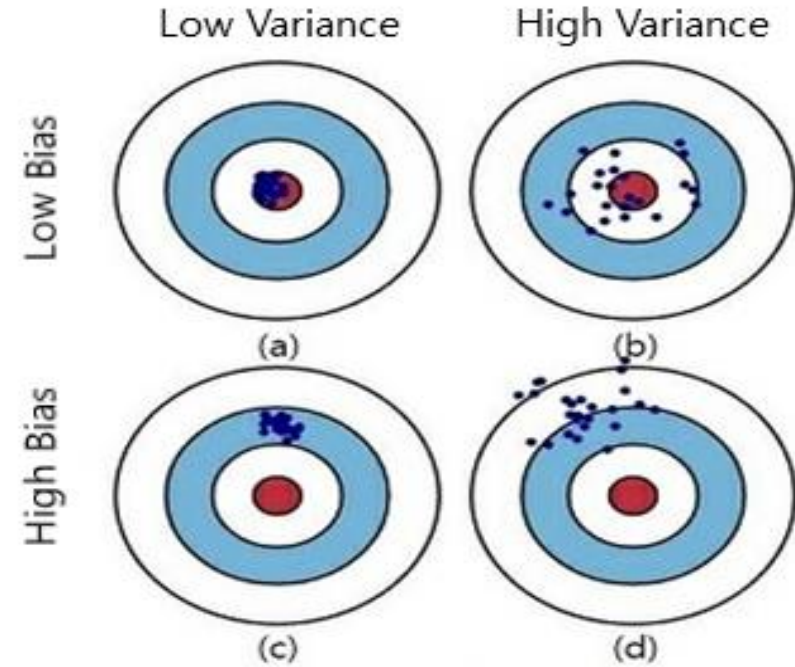
Overfitting, bias and variance – (2/4)

- 학습 데이터(training dataset)에는 잘 맞지만, 테스트 데이터(testing dataset) 나 실제 사용(예측)에 잘 맞지 않는 경우



○ Bias vs. Variance 의 의미

- Bias, Variance → 기계 학습 모델의 loss 또는 error 임
- approximation(추정)과 참값 사이에 생길 수 있는 error → bias
- 모델에 적용하게 될 향후 데이터들의 다양성에 의한 error → variance



Bias and variance tradeoff – Cross Validation – (3/4)

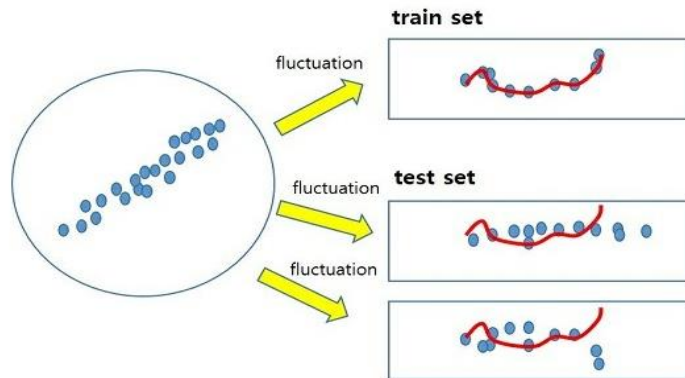
○ train set 과 test set의 관계가 bias and variance tradeoff 와 연관

- train data에 너무 잘 맞는 학습 → 높은 모델 복잡도 → test data의 높은 variance
- 반대로 단순한 모델 복잡도 → 덜된 학습 → bias 증가

➔ bias and variance tradeoff

○ 전체 dataset을 train data, test data 등의 subset으로 나눌 때, subset 간에 각기 다른 변동(fluctuation)을 포함

- train data에 너무 맞추면 test data에 대한 추정 값의 variance 가 증가



- training 할 때는 bias 나 variance의 어느 한 쪽을 보고 학습하는 것이 아니라, 이 둘을 합한 error 가 작아지도록 학습함

○ overfitting 해결 방법

- feature의 개수를 줄이는 방안, 정규화(Regularization)

○ 교차검증

- dataset의 크기가 작은 경우, 교차 검증은 모든 데이터가 최소 한번은 test dataset으로 쓰이도록 함

	A	B	C	D	E
Cross Validation Iteration 1	Test	Train	Train	Train	Train
Cross Validation Iteration 2	Train	Test	Train	Train	Train
Cross Validation Iteration 3	Train	Train	Test	Train	Train
Cross Validation Iteration 4	Train	Train	Train	Test	Train
Cross Validation Iteration 5	Train	Train	Train	Train	Test

Regularization

- Let's not have too big numbers in the weight

The diagram shows the cost function $\mathcal{L} = \frac{1}{N} \sum_i \mathcal{D}(s(w x_i + b), L_i) + \lambda \sum W^2$. Annotations include:

- A blue arrow labeled "LOSS" pointing to the first term.
- A blue arrow labeled "TRAINING SET" pointing to the index i in the summation.
- A blue arrow labeled "regularization strength" pointing to the coefficient λ .

Cost 함수에서 최소값을 찾는 것이 우리의 목적인데, 여기에 이 항목을 넣어준다.

λ 는 regularization strength 로서,
 0 이면 regularization 을 하지 않는 것이고,
 1 이면 regularization 을 아주 강하게 하는 것임.
 0.001 이면 regularization 하긴 하지만 중요하지는 않음.

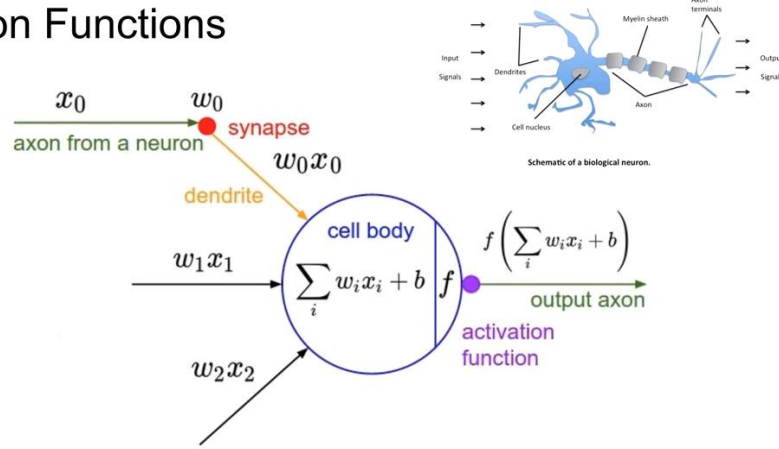
이 것을 텐서플로를 이용하여 구현하면, 다음과 같이 간단하게 구현할 수 있다.

```
l2reg = 0.001 * tf.reduce_sum(tf.square(W))
```

이렇게 계산된 *l2reg* 값을 cost 값과 더한 다음 최소값을 찾아 나가게 되는 것이다.

NN(Neural Network) Basic

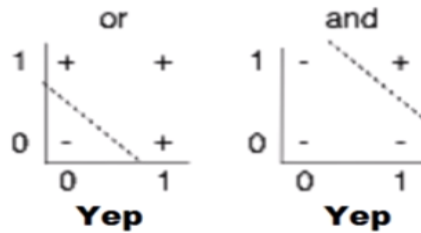
Activation Functions



(Simple) XOR problem: linearly separable?

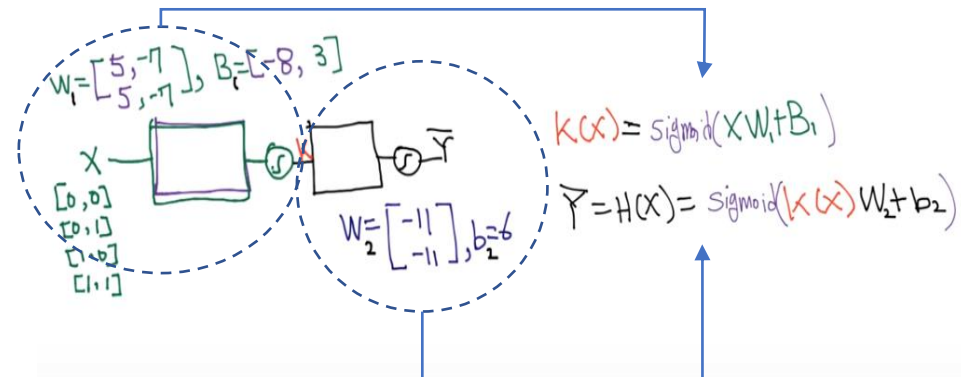


(Simple) AND/OR problem: linearly separable?

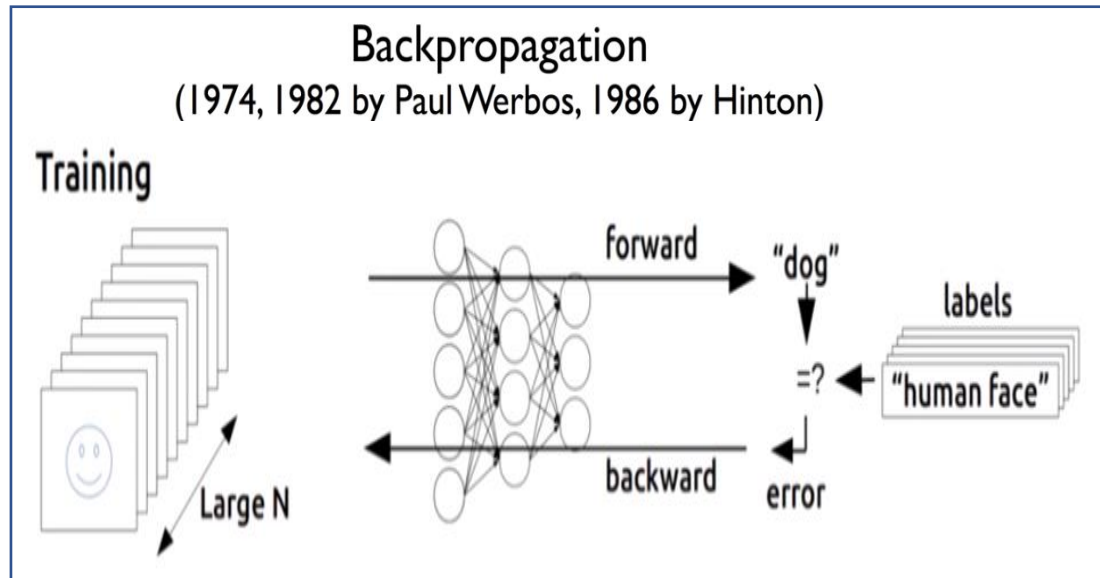


Weight 와 bias가 2차원 벡터로 되고,
두 개의 유닛을 연결한 Neural Network 으로
XOR 문제가 해결됨을 보이고 있음.

NN



Back4 Propagation - Basic(1/2)



① forward propagation

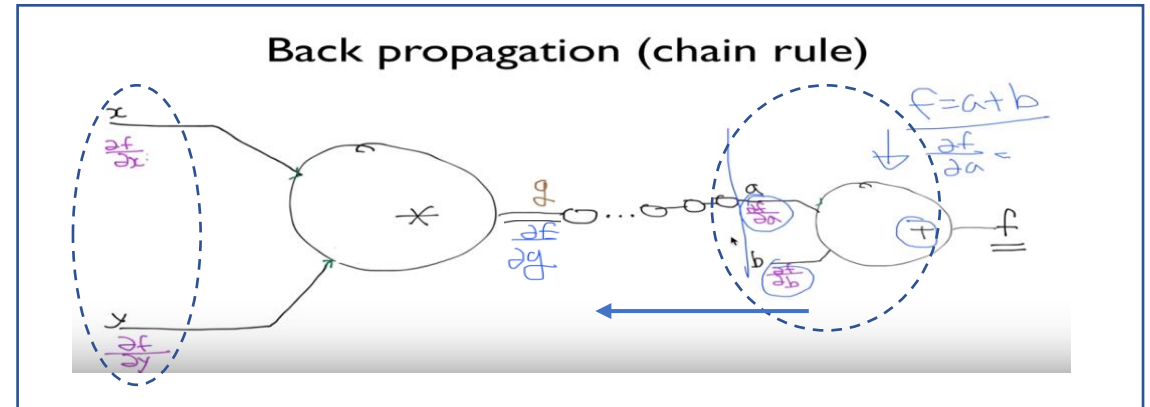
- 실제 학습 데이터에서 값을 가져온다. ($w = -2, x = 5, b = 3$)
- 그리고, 그래프에 값을 입력하고 계산한다.

② backward propagation

- 각 노드에서 편미분을 하고, chain rule을 이용하여 편미분한 미분 값을 역전파 한다.

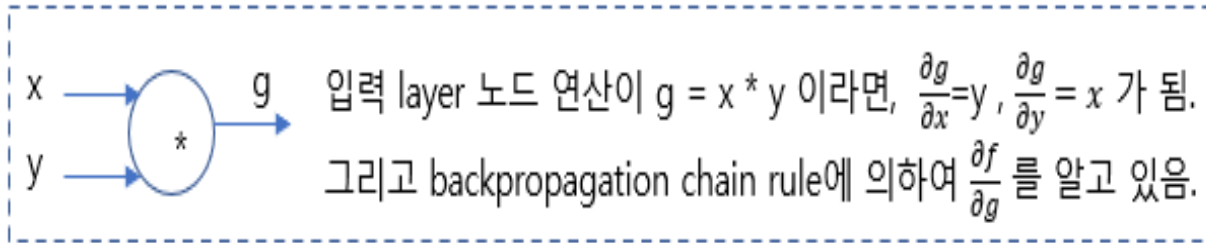
○ Chain rule of back propagation : 중간에 노드가 많은 경우

- 첫번째 노드 : $g = x * y$, 마지막 노드 : $f = a + b$ 라고 할 때,
- 궁극적으로 구하려는 것은 x, y 가 f 에 미치는 영향 $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$ 이다



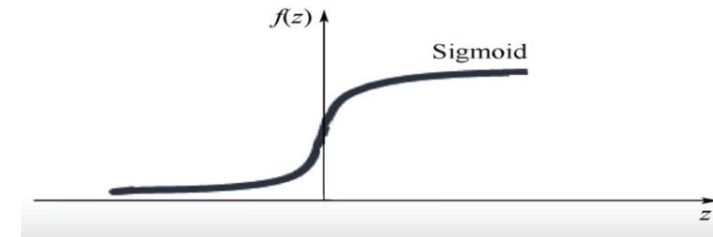
- 마지막 노드의 연산자가 덧셈이고, 두 입력이 a, b 라면, $f = a + b$ 의 수식이 되고, $\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}$ 를 구할 수 있게 된다.
- 화살표 방향으로 backpropagation 계산을 쭉 해 나가면, $\frac{\partial f}{\partial g}$ 값을 알 수 있다. (chain rule 계산 방법).
- 첫 번째 노드의 연산이 $*$ 라면, $g = x * y$ 이므로 $\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}$ 는 계산할 수 있다.
- 최종적으로 첫번째 노드에서 $\frac{\partial f}{\partial x}$ 를 구하기 위해서, $\frac{\partial f}{\partial g}$ 를 이용하여 $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \times \frac{\partial g}{\partial x} = \frac{\partial f}{\partial g} * y$ 가 된다.

Back Propagation-vanishing gradient and ReLU (2/2)

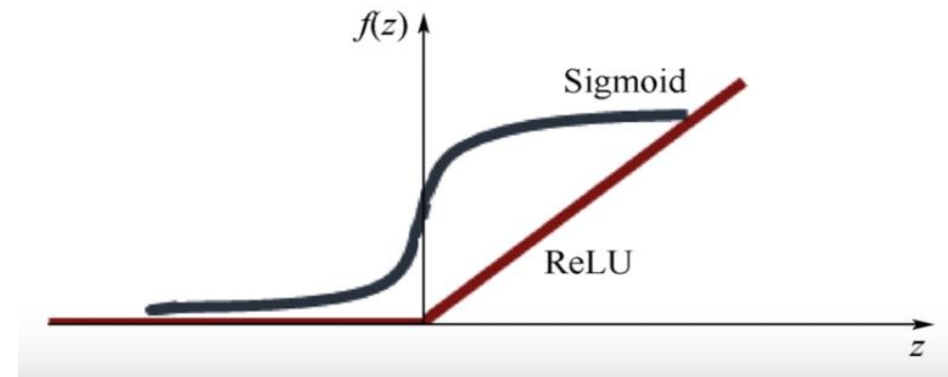


- y 값이 sigmoid 함수와 연산 노드를 몇 개 거쳐서 왔다고 생각하면, 값의 범위는 0 ~ 1 사이의 값이 된다. 만약 Sigmoid 함수 통과 결과 0.01 이라고 한다면, $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} * 0.01$ 이 된다.
- 이 노드에서 구한 미분 값 $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} * 0.01$ 은 앞쪽 노드들로 전달되면서 각 해당 노드의 미분 값과 곱해 질 것임.
- 곱해질 값들도 sigmoid function을 통과한 0 ~ 1 사이의 값, 0.01 등이 될 것이고, 결국에는, chain rule을 적용되어 0 ~ 1 사이의 값이 계속 곱해지니까, 점점 0에 가까워 질 것이다.

Sigmoid!



- 모든 입력 값에 대한 출력 값이 0 ~ 1 사이의 값이고, chain rule을 적용해서 이 값들이 곱해져 가서 점점 0에 가까워 진다는 것이 문제였다.
- 이것을 1보다 작아지지 않게 만들면 되지 않을까, 그래서 간단하게 만들어진 것이 ReLU 이다.

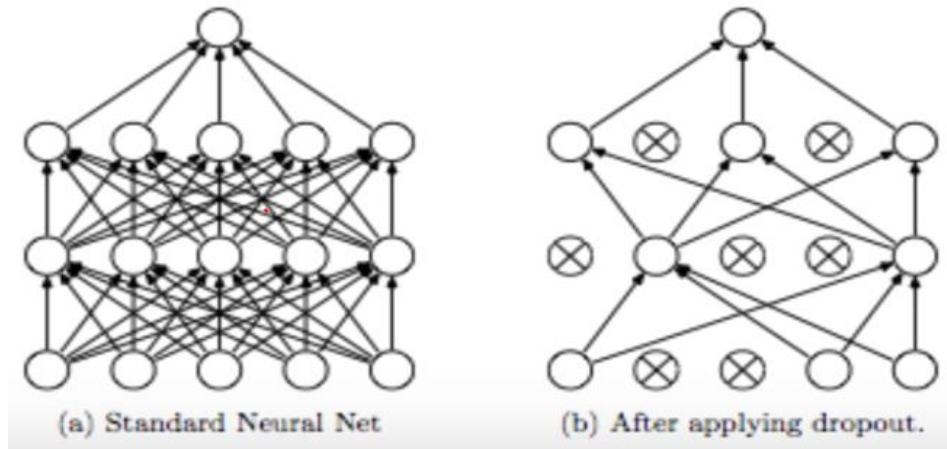


- ReLU는 z 가 0보다 작은 모든 값은 0 (off, - non active)이고, z 가 0 보다 큰 값이면 그 값에 비례해서 계속 큰 값을 출력하는 activation function 이다.

NN dropout and model ensemble

○ drop out

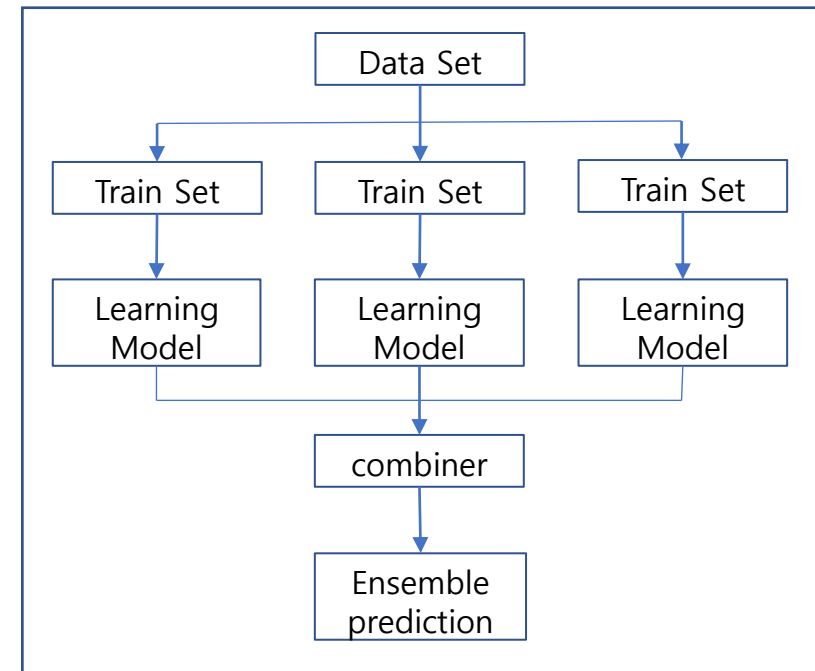
- 네트워크 일부를 생략하고 학습을 진행하며, 테스트 할 때는 모두 사용함.



- 모델 결합(model combination)을 하게 되면 학습의 성능을 높일 수 있는데, Drop out 은 이 개념을 적용한 것임.
- Drop out은 여러 개의 모델을 만드는 대신 모델 결합과 유사한 효과를 내기 위해 훈련이 진행되는 동안 랜덤하게 일부 뉴런을 정지
- 정지된 뉴런의 조합 만큼 지수함수적으로 다양한 모델을 학습시키는 것과 마찬가지로 여서 모델 결합의 효과를 있다.

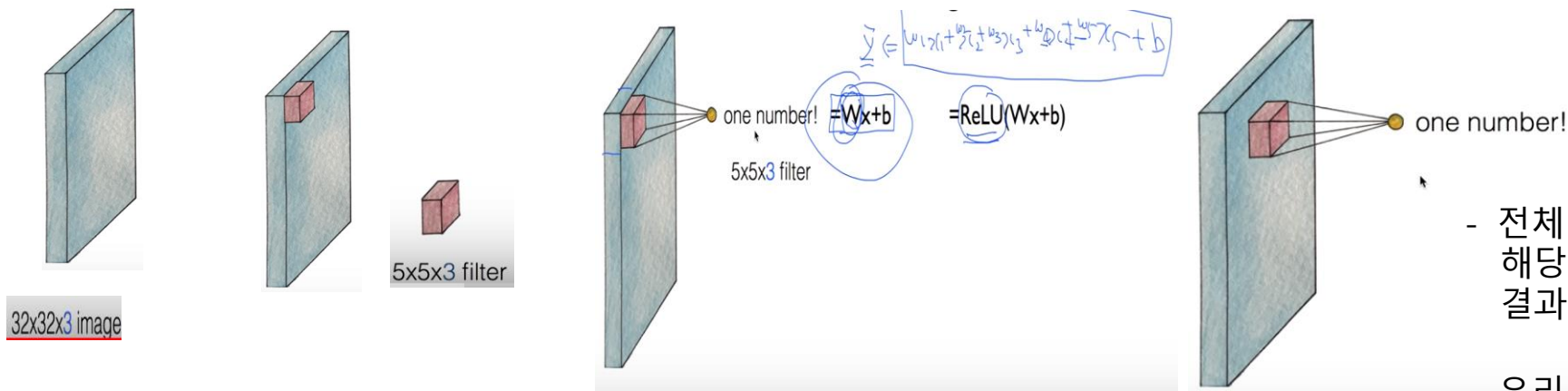
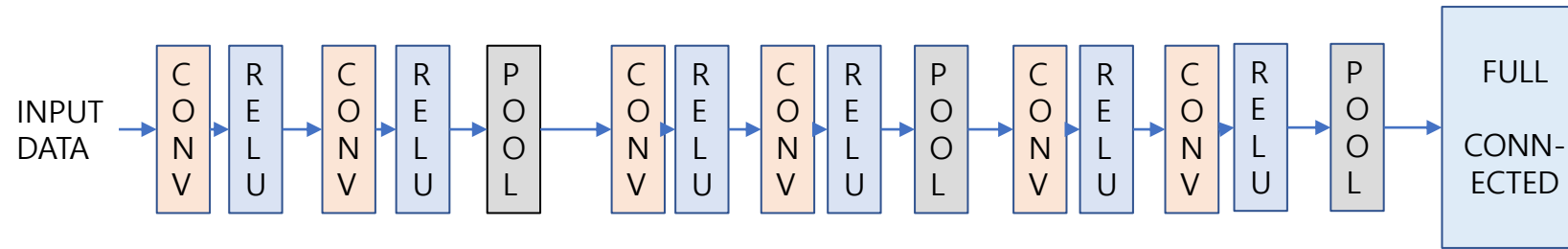
○ Ensemble

- 동일한 NN(Neural Network)을 여러 개 구성해 놓고 같은 training data로 각각의 NN에 트레이닝 시키고 나서 나중에 합치게 되면 성능개선이 됨



- W 초기 값이 random 이기 때문에 동일하게 구성된 NN 이라 하더라도 트레이닝 동작이 달라지게 되며,
- 이로 인해서 약간씩 다른 결과가 나오는데 이를 합치게 되면 더 많은 train data로 training 한 효과가 나오게 됨.

CNN : Convolutional Neural Network (1/7)

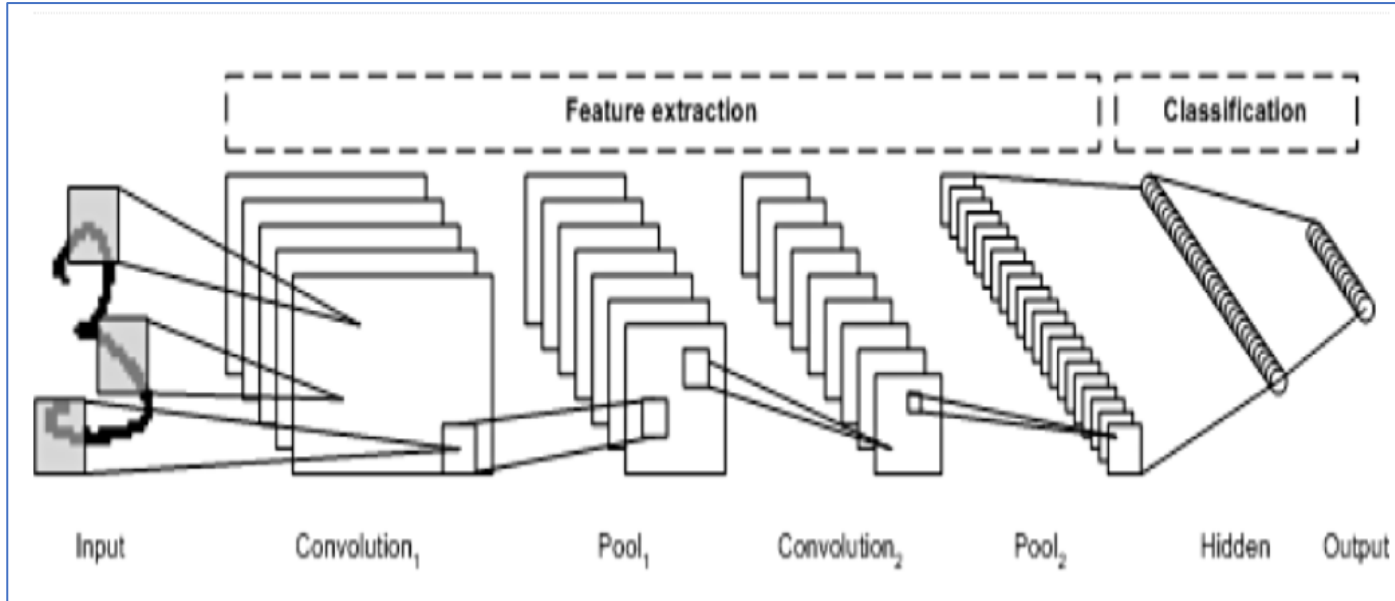


- 32x32x3 이미지에 대해서 5x5x3 filter는 해당위치에서 $Wx+b$ 의 계산방식으로 한 개의 계산 결과 값을 가진다.
- 계산 결과 값은 w 값의 형태에 따라서 한 개의 숫자로 계산된다.
- **Weight** 값은 계산 결과 값을 결정하는 **filter** 값이 된다.
- 그리고 그 계산 결과 값을 ReLU activation 함수에 넣으면 ReLU 결과 값이 계산되는 것이다.

- 전체 이미지에 대해서 filter가 움직이면서, 해당 위치에서 계산을 하여 one number 결과 값을 가져온다.

- 우리가 몇 개의 'one number' 값을 구할 것 인지가 중요한데, 실제로 이 값들을 알아야 **weight** 의 개수도 정하고, 그 다음으로 어떻게 넘길 것인가도 설계를 할 수 있다.

- CNN은 이미지의 특징을 추출하는 부분과 클래스를 분류하는 부분으로 구성

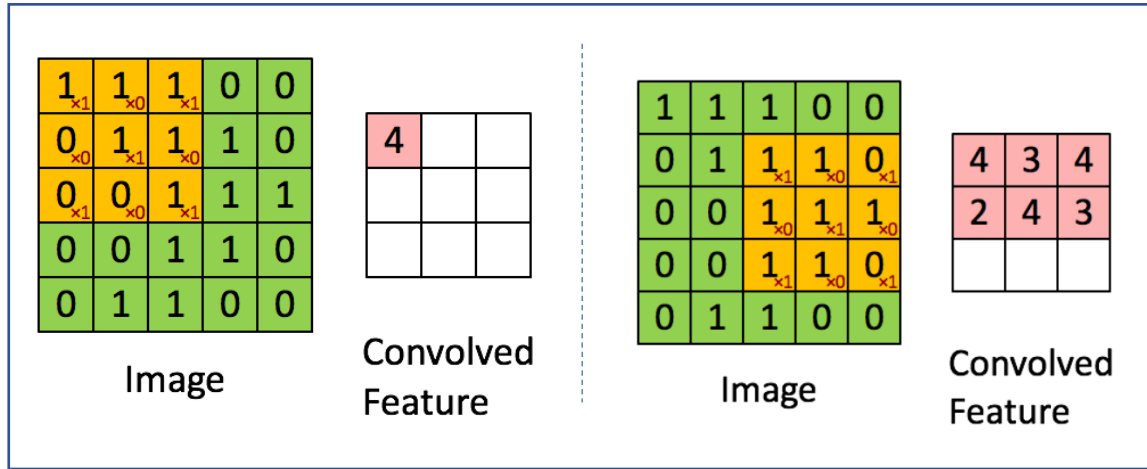


- FC(Fully Connected) Layer 만으로 구성된
인공신경망의 데이터는 1차원(배열)로 한정됨.
 - 컬러사진은 3차원(R,G,B) 데이터 배열이므로,
FC 신경망 으로 학습시켜야 할 경우 1차원으로
평면화 해야함.
 - 평면화 과정에서 공간 정보 유실 등이 있을 수
있는데, CNN은 feature extraction 과정을
통하여 공간 정보를 유지함.

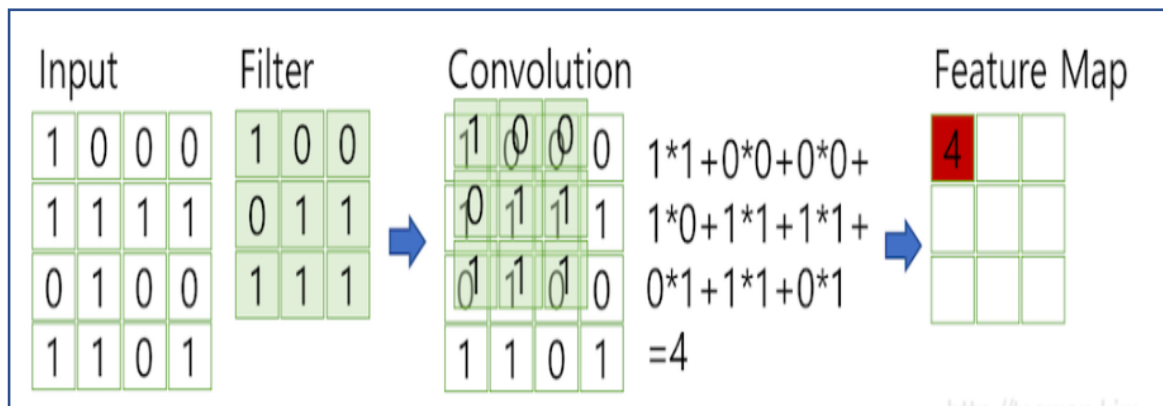
○ CNN의 주요 용어

- | | |
|---------------------|---------------------------|
| - Convolution(합성곱) | - 채널(Channel) |
| - 필터(filter) | - 커널(Kernel) |
| - 스트라이드(Stride) | - 패딩(padding) |
| - 피쳐 맵(Feature Map) | - 액티베이션 맵(Activation Map) |
| - 풀링(Pooling) 레이어 | |

① Convolution (합성곱)

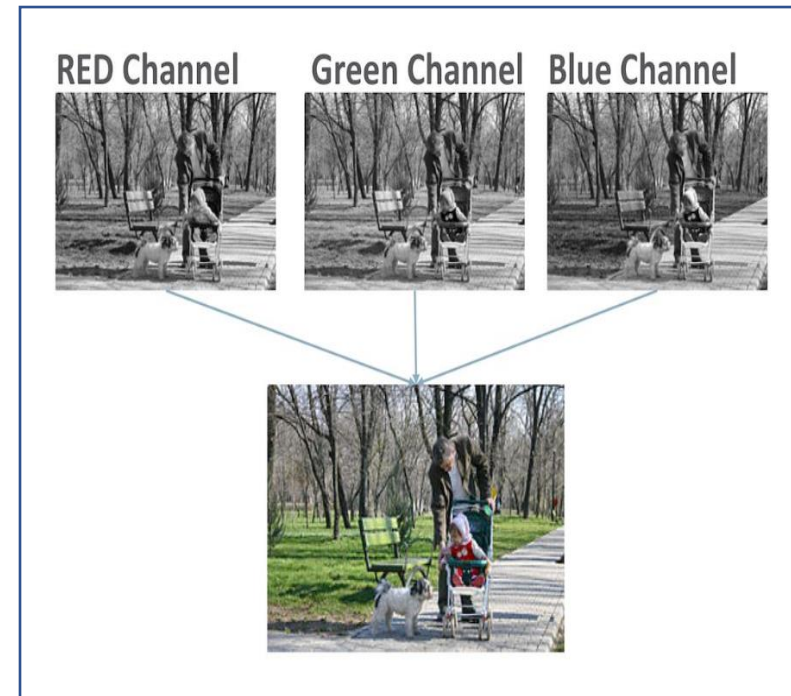


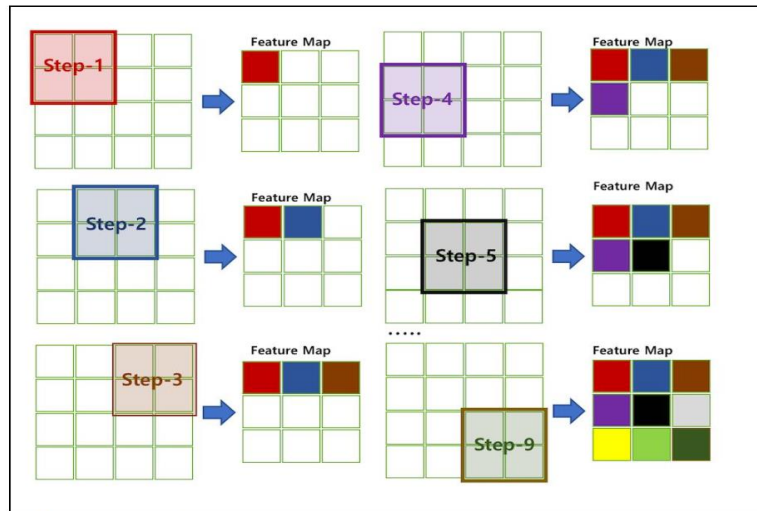
③ 필터(Filter) & 스트라이드(Stride)



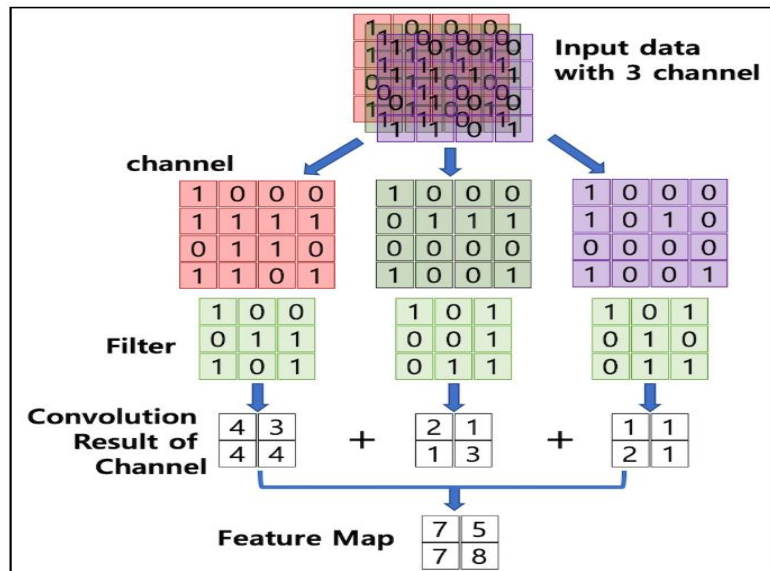
② 채널(Channel)

- 컬러 이미지는 3개의 채널로 구성
 . 높이 39 픽셀, 폭 31 픽셀인 컬러 데이터 shape 은 (39, 31, 3)
- Convolution Layer에 유입되는 입력 데이터에는 n개의 필터가 적용
 . 1개의 필터는 1개 Feature map의 채널이 됨



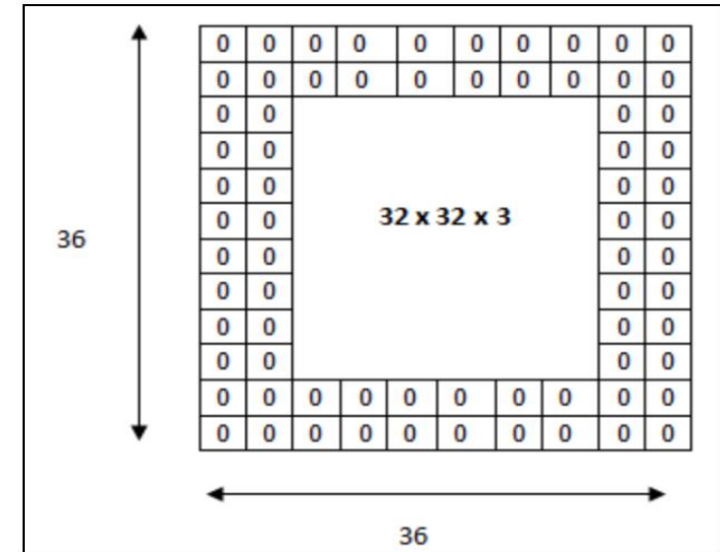


Feature Map
작성 과정
(stride = 1)



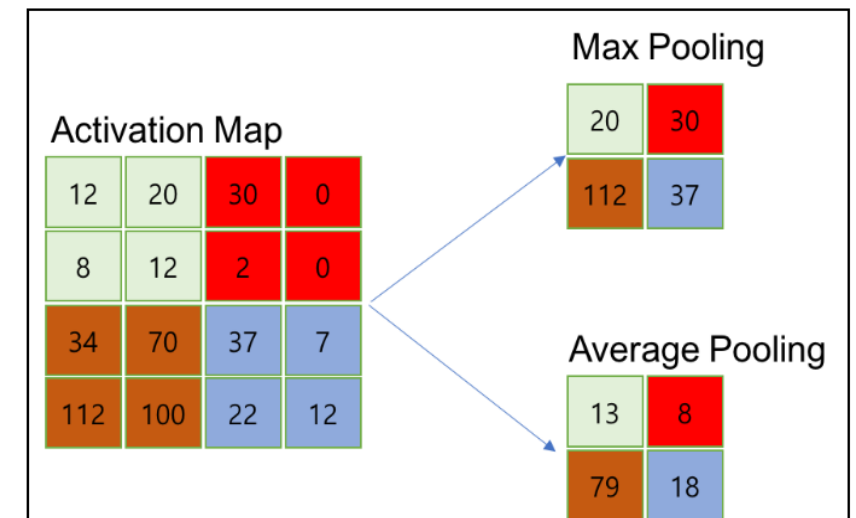
멀티 채널
입력 데이터에
필터를 적용한
합성곱
계산 절차

③ 패딩(Padding)

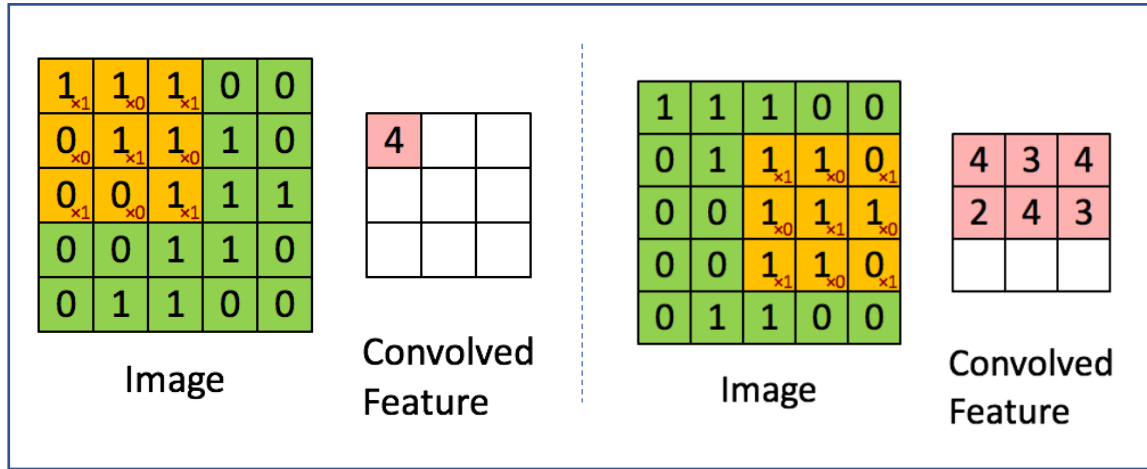


④ Pooling Layer

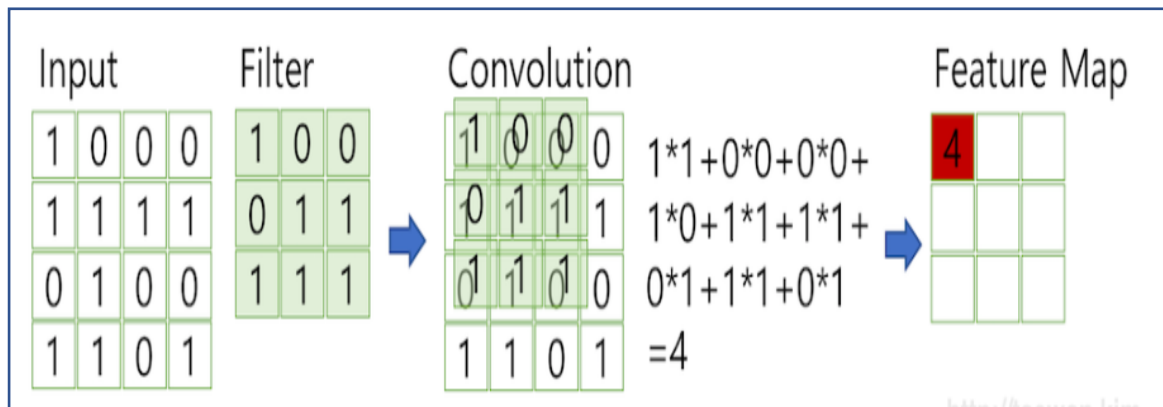
- Feature map 이 activation function 을 통과하면 activation map이 된다.



① Convolution (합성곱)

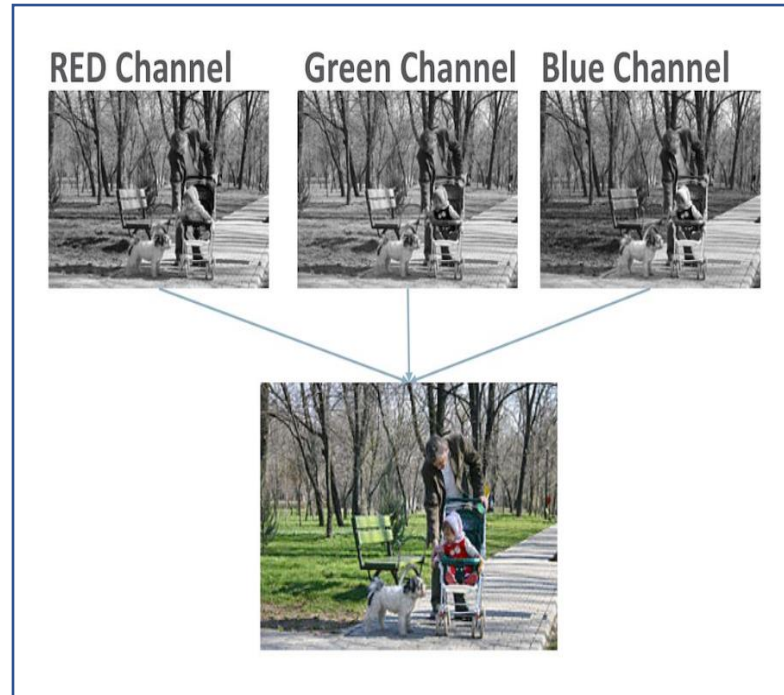


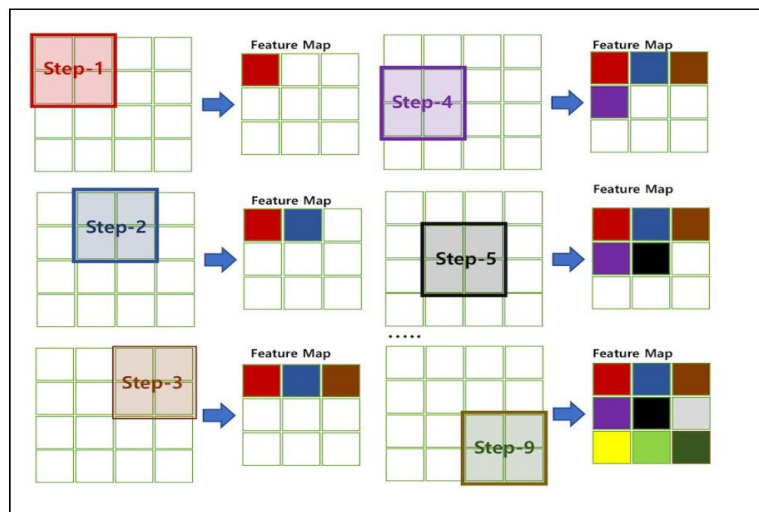
③ 필터(Filter) & 스트라이드(Stride)



② 채널(Channel)

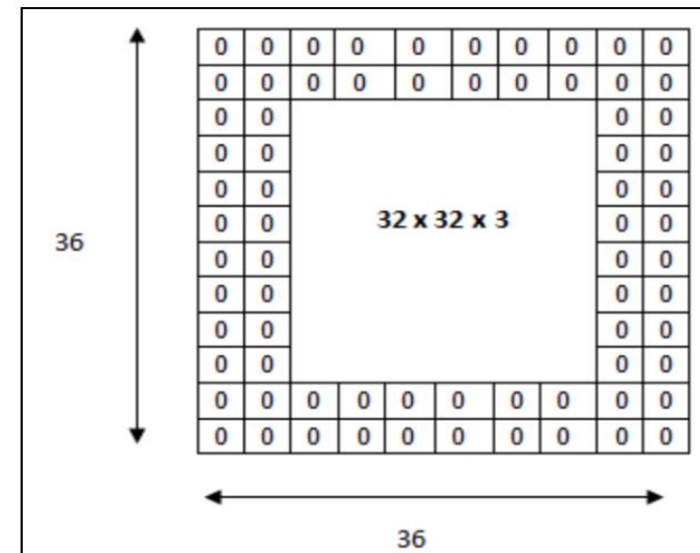
- 컬러 이미지는 3개의 채널로 구성
 . 높이 39 픽셀, 폭 31 픽셀인 컬러 데이터 shape 은 (39, 31, 3)
- Convolution Layer에 유입되는 입력 데이터에는 n개의 필터가 적용
 . 1개의 필터는 1개 Feature map의 채널이 됨





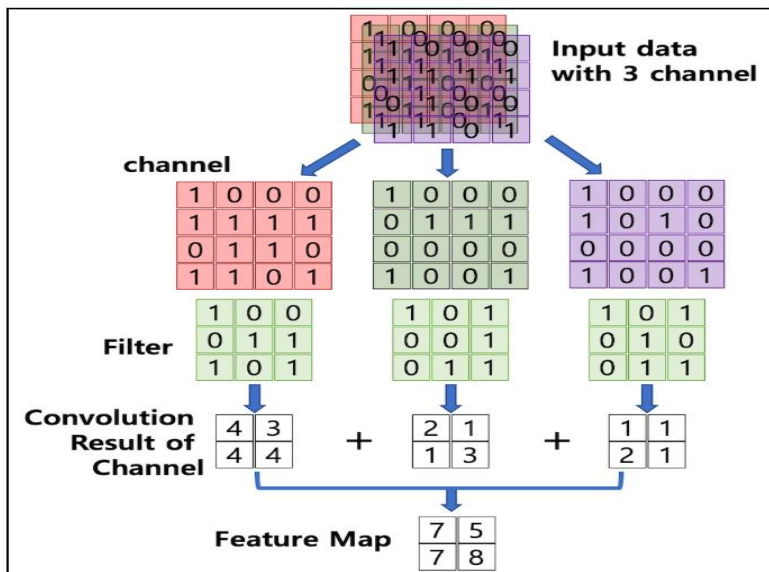
Feature Map
작성 과정
(stride = 1)

③ 패딩(Padding)

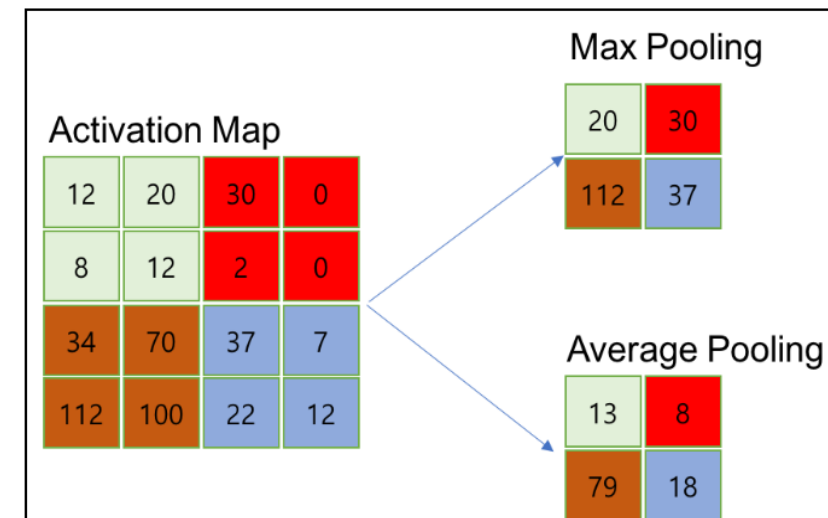


④ Pooling Layer

- Feature map 이 activation function 을 통과하면 activation map이 된다.

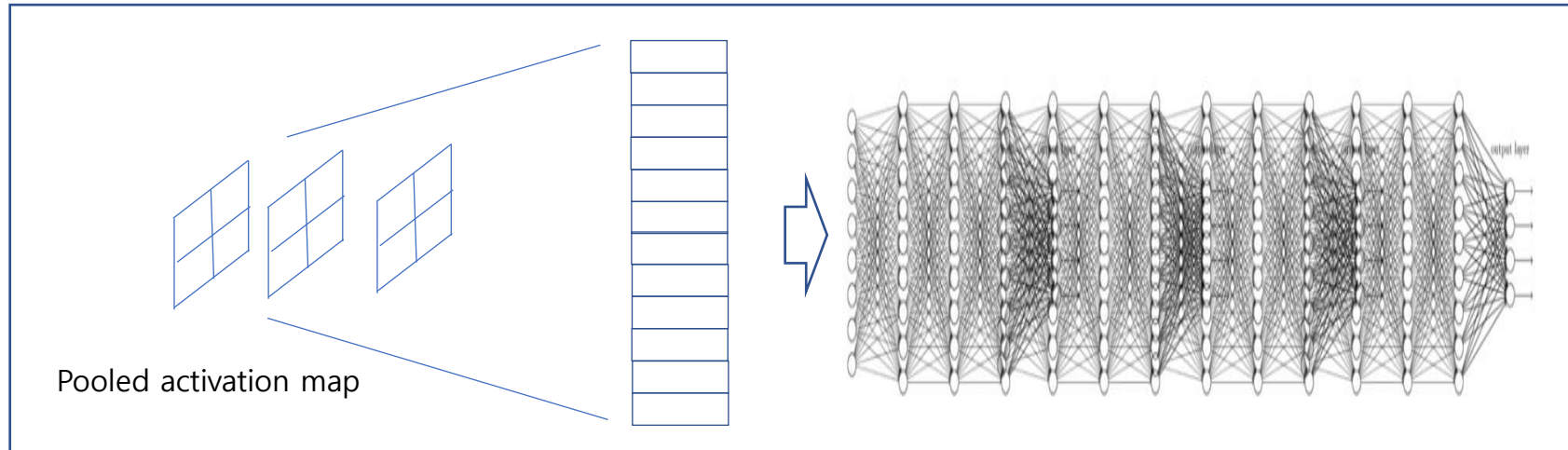


멀티 채널
입력 데이터에
필터를 적용한
합성곱
계산 절차



Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



앞에서 convolution에 대해서 다루었고, **Convolution**에서 나온 어떤 벡터를 ReLU 함수에 입력을 하면 된다.

그리고 **pooling**은 **sampling** 하는 방법으로 앞에서 설명되었음.
이러한 layer들을 쌓는 것은 연구자가 알아서 정할 수 있다.

마지막에, 보통 pooling을 하게 되는데, Pooling 결과 값을 $3 \times 3 \times 10$ 이었다고 할 때, 이것을 x data로 보고, 연구자가 원하는 만큼 깊이를 정한 fully neural network 에 넣어서, 마지막 layer가 softmax classifier가 되어, 해당 label 들을 분류할 수 있게 된다.

① Convolutional Layer 출력 데이터 크기 산정

- 입력 데이터 높이 : H
- 입력 데이터 폭 : W
- 필터 높이 : FH
- 필터 폭 : FW
- Stride 크기: S
- 패딩 사이즈 : P

$$\text{Output Height} = \text{OH} = \frac{H+2P-FH}{S} + 1$$

$$\text{Output Width} = \text{OW} = \frac{W+2P-FW}{S} + 1$$

- 위 식의 결과는 자연수 이어야 함.
- Convolution layer 에 이어서 pooling layer가 온다면 Feature map의 행과 열의 크기는 pooling size의 배수가 되어야 한다.

Output Depth = 해당 convolutional Layer에 적용되는 필터의 개수

② Pooling Layer 출력 데이터 크기 산정

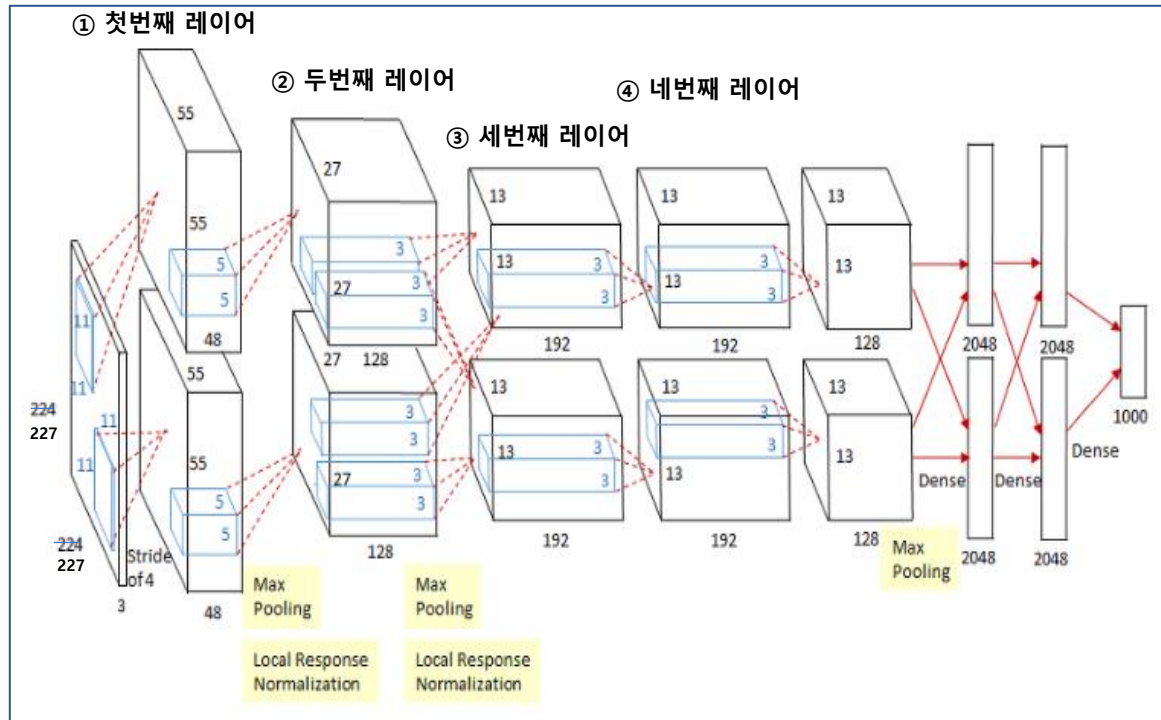
- Pooling Layer에서 일반적인 pooling size는 정사각형임
- Pooling size를 Stride 같은 크기로 만들어서, 모든 요소가 한번씩 pooling 되도록 한다.
- 입력 데이터의 행과 열의 크기는 Pooling 사이즈의 배수이어야 함

$$\text{Output Row Size} = \frac{\text{Input Row Size}}{\text{Pooling Size}}$$

$$\text{Output Column Size} = \frac{\text{Input Column Size}}{\text{Pooling Size}}$$

○ AlexNet 의 구조

- 8개 레이어 : 5개 convolution layer와 3개의 Full-connected layer로 구성
- 2개의 GPU로 병렬연산을 수행하기 위한 병렬적인 구조로 설계



- 입력 데이터는 227 x 227 x 3 의 컬러 이미지 데이터

- ① 첫번째 레이어 : convolution layer
 - 96개의 11 x 11 필터 적용, stride = 4
 - . Output 높이와 폭 : $(227-11)/4 + 1 = 55$
 - . Output Depth : 96

output shape : (55 x 55 x 96)
- ② 두번째 레이어 : pooling 및 Local Response Normalization
 - 3 x 3 필터 적용, stride2, Max pooling
 - . Output 높이와 폭: $(55-3)/2 + 1 = 27$
 - . Output depth: 96
 - . Local Response Normalization 적용 (256)
 - (요즘은 적용하지 않음)

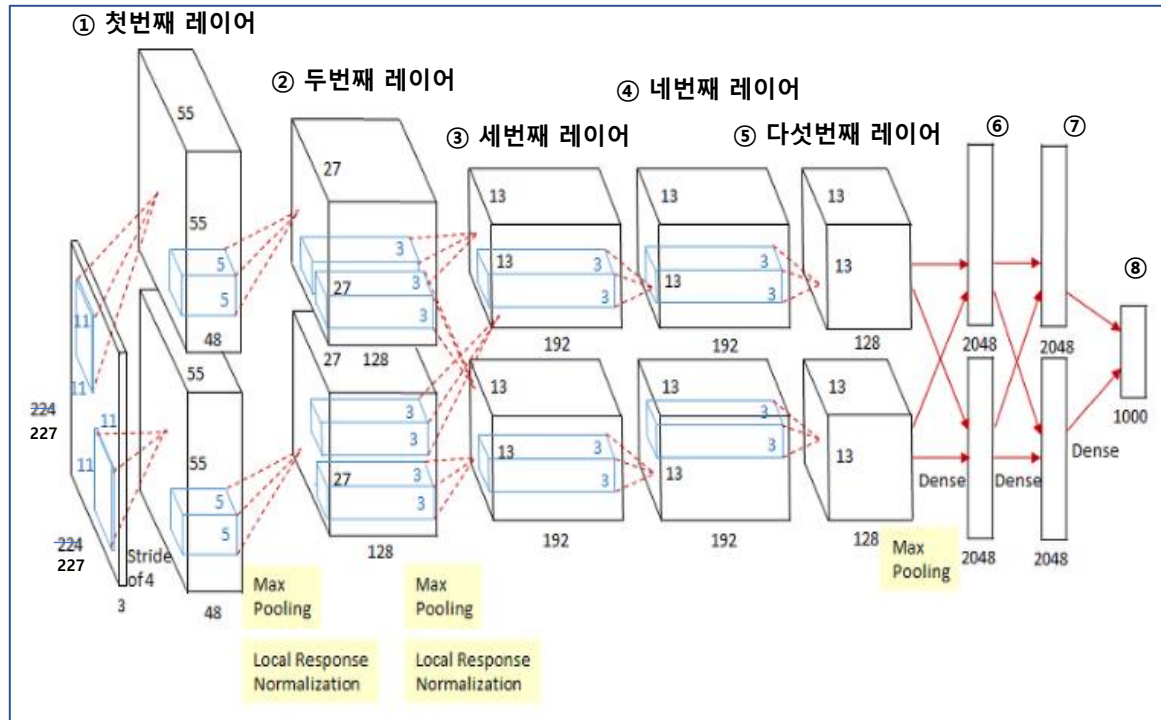
output shape : (27 x 27 x 256)
- ③ 세번째 레이어 : pooling 및 Local Response Normalization
 - 3 x 3 필터 적용, stride2, Max pooling
 - . Output 높이와 폭: $(27-3)/2 + 1 = 13$
 - . Output depth: 256
 - . Local Response Normalization 적용(384)
 - (요즘은 적용하지 않음)

output shape : (13 x 13 x 384)
- ④ 네번째 레이어 : pooling Layer
 - 3 x 3 필터 적용, stride1, pad 1, Max pooling
 - . Output 높이와 폭: $(13+2-3)/1 + 1 = 13$
 - . Output depth: 384
 - (pooling 은 depth 에 영향주지 않음)

output shape : (13 x 13 x 384)

○ AlexNet 의 구조

- 8개 레이어 : 5개 convolution layer와 3개의 Full-connected layer로 구성
- 2개의 GPU로 병렬연산을 수행하기 위한 병렬적인 구조로 설계



⑤ 다섯번째 레이어 : convolution 및 Max Pooling

- 256개의 3 x 3 필터, stride = 1, pad1

. Output 높이와 폭 : $(13+2-3)/1 + 1 = 13$

. Output Depth : 256

output shape : (13 x 13 x 256)

- Max pooling (3 x 3 필터, stride 2)

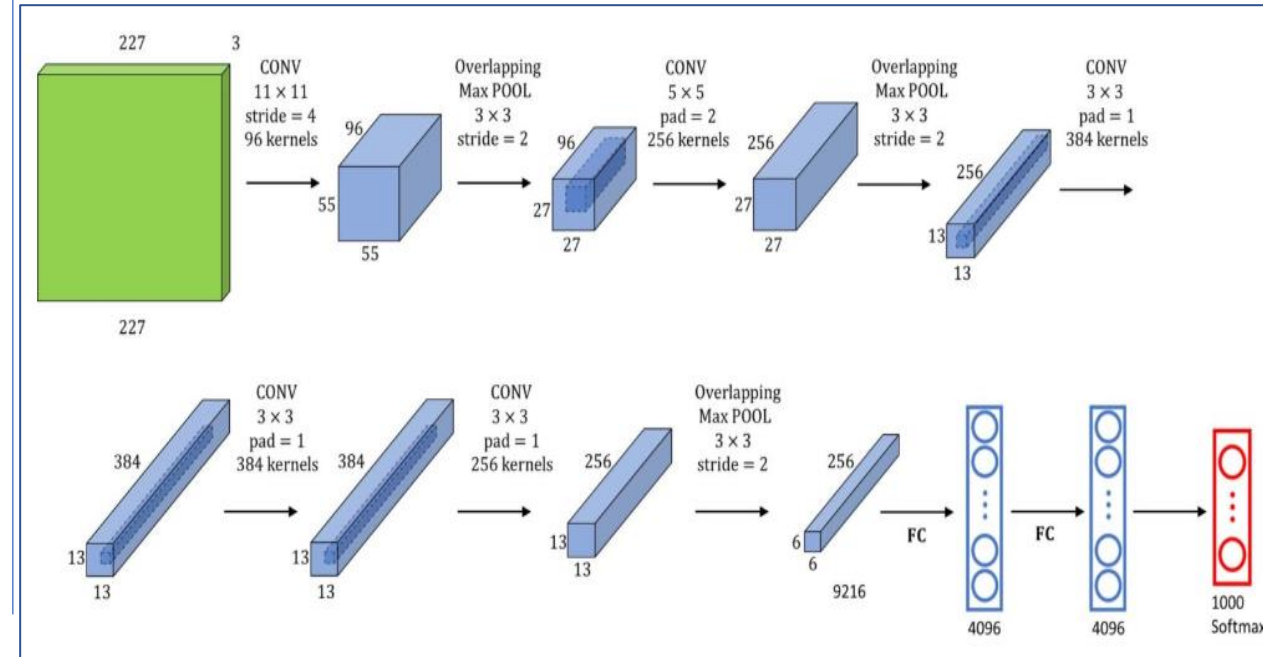
. Output 높이와 폭 : $(13-3)/2 + 1 = 6$

. Output Depth : 256

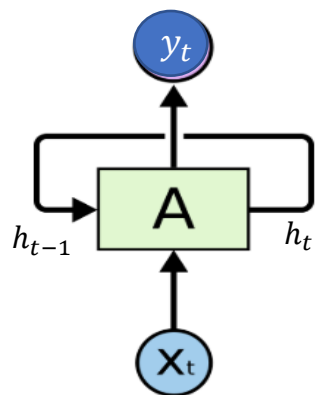
output shape : (6 x 6 x 256)

⑥, ⑦ layer : Full Connected Layer

⑧ Softmax

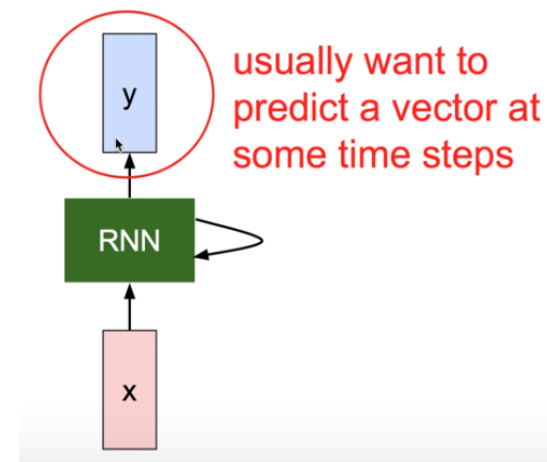


○ RNN은 스스로를 반복하면서 **이전 단계에서 얻은 정보가 지속되도록** 한다.

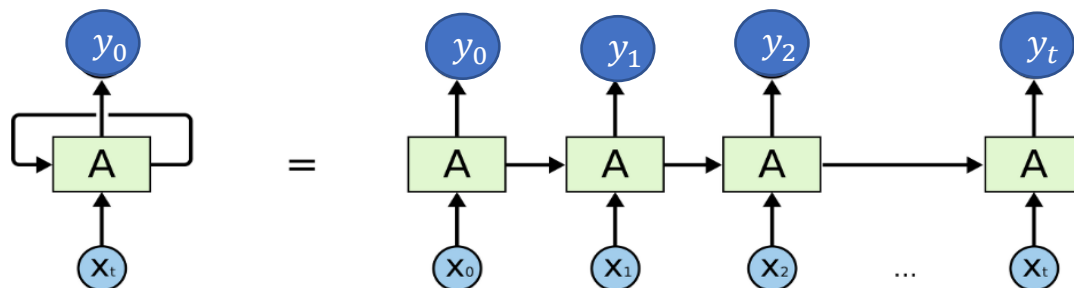


○ A 한 개로 표시

- A는 input x_t 를 받아서 y_t 로 내보낸다.
- A를 둘러싼 반복은 다음 단계에서의 Network 가 이전 단계의 정보(h_{t-1})를 받는다는 것을 보여준다.



○ Unfolding : sequence 개념



이전의 state 가 현재 state에 영향을 미치므로 series data 에 적합하다.

○ RNN에는 **State** 라는 개념이 있음.

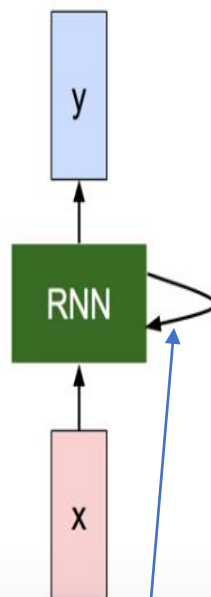
- State를 먼저 계산하고, 그 state를 이용하여 y 를 계산
- State 계산 시, one-time step 이전의 State가 입력으로 사용

We can process a sequence of vectors x by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

new state old state input vector at some time step

some function with parameters W

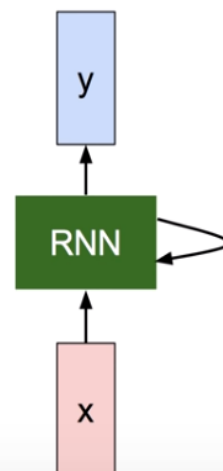


Notice: the same function and the same set of parameters are used at every time step.

- 기본적인 (Vanilla)RNN 연산 방법은 WX 형태이며, h_{t-1} 과 x_t 의 두 입력 값에 각각의 weight를 만들어 주는 것임.
- 그리고 \tanh (sigmoid와 비슷한 형태)를 이용함.
- y_t 의 계산은 계산된 h_t 에 또다른 weight를 곱함(WX 의 형태

(Vanilla) Recurrent Neural Network

The state consists of a single "hidden" vector h :



$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

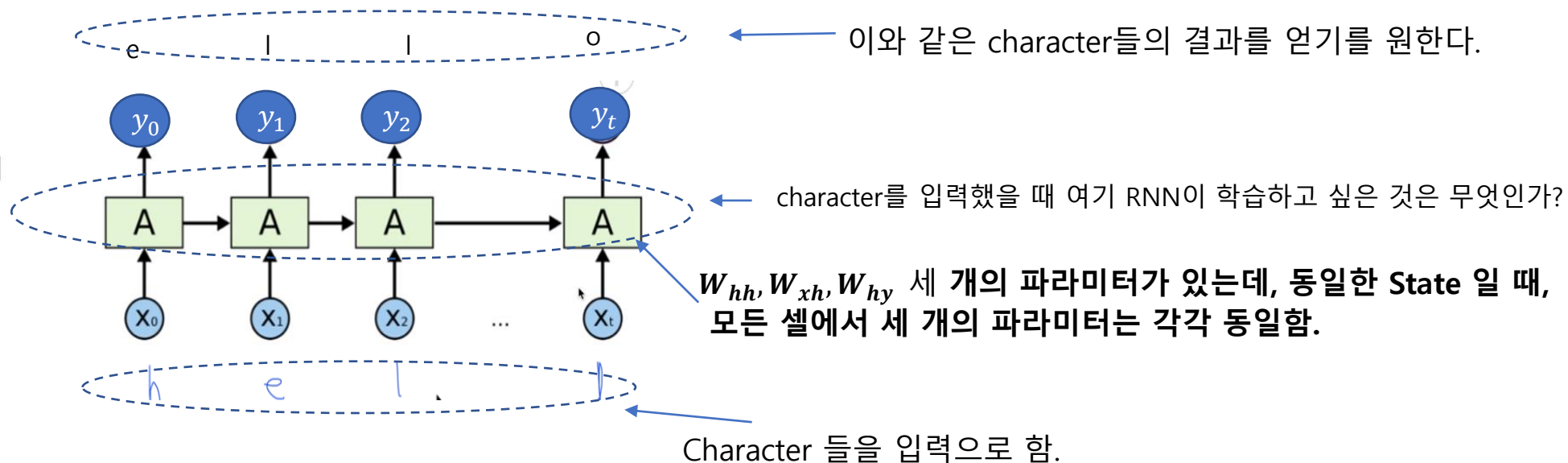
- y_t 가 몇 개의 vector로 나올 것인가 하는 것은 W_{hy} 의 형태에 달려있다. h_t vector 형태도 W_{hh} 형태에 달려있다

- 간단한 Language Model : 현재의 character가 있을 때 그 다음 Character가 무엇인지 예측
 - 연산 과정

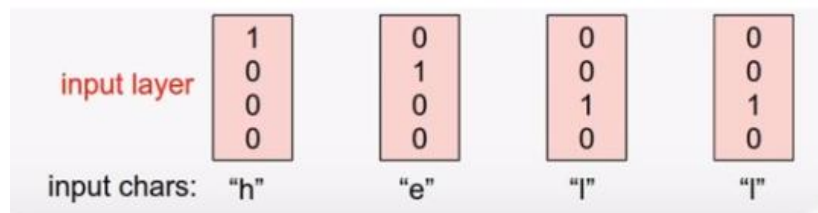
Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"

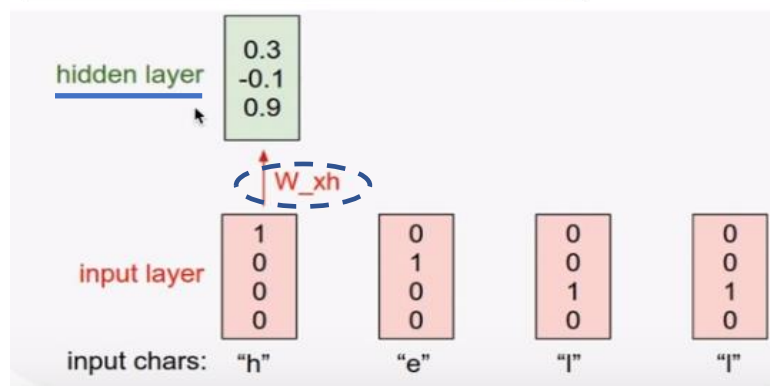


① 입력을 벡터로 표현 ; one-hot encoding



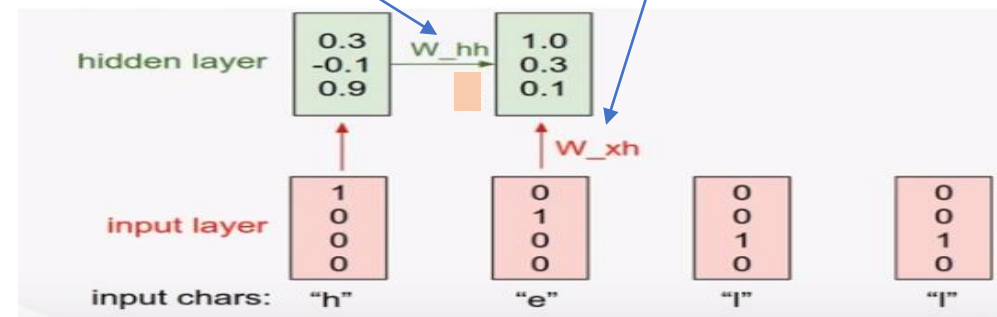
② Hidden Layer 첫번째 셀 계산

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

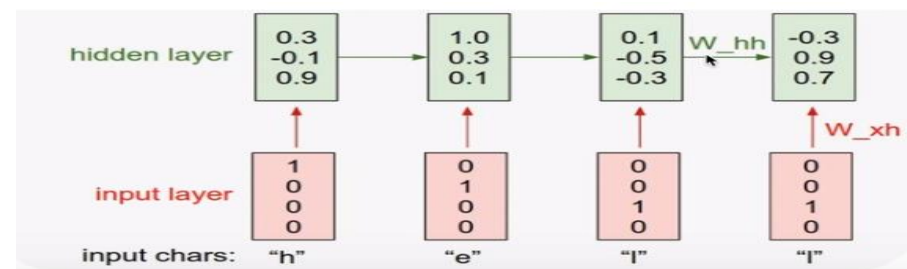
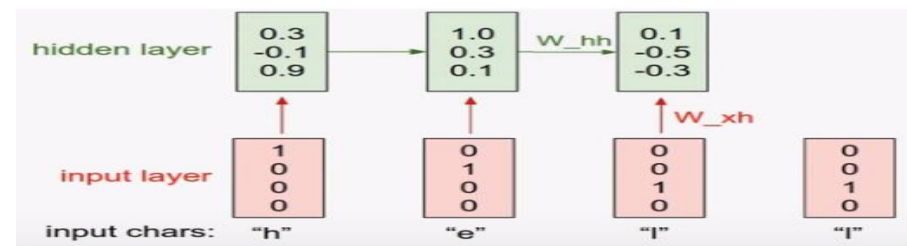


③ Hidden Layer 두번째 셀 계산

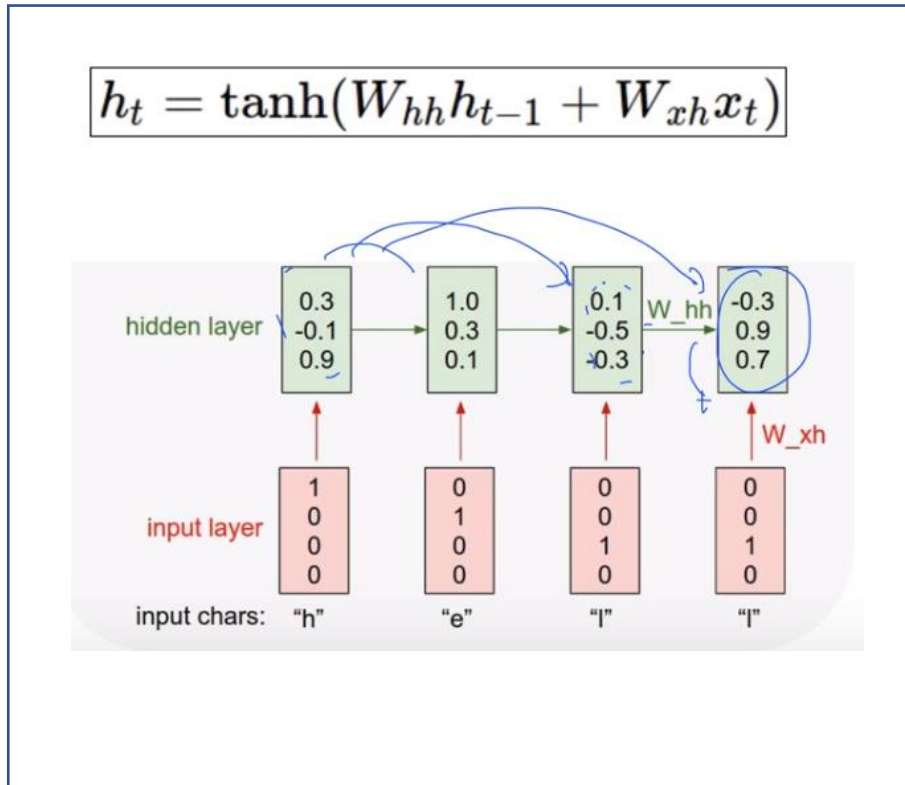
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$



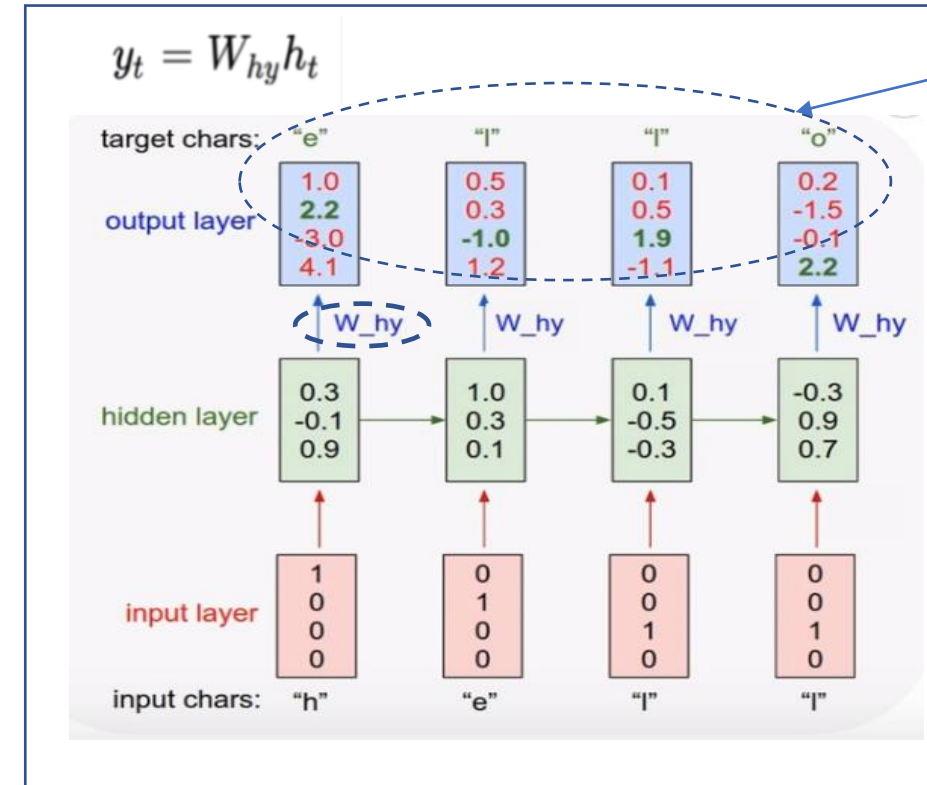
④ Hidden Layer 셀의 단계적 계산



- ⑤ hidden layer 값들은 그 이전의 값들이 영향을 미치고 있음을 알 수 있음

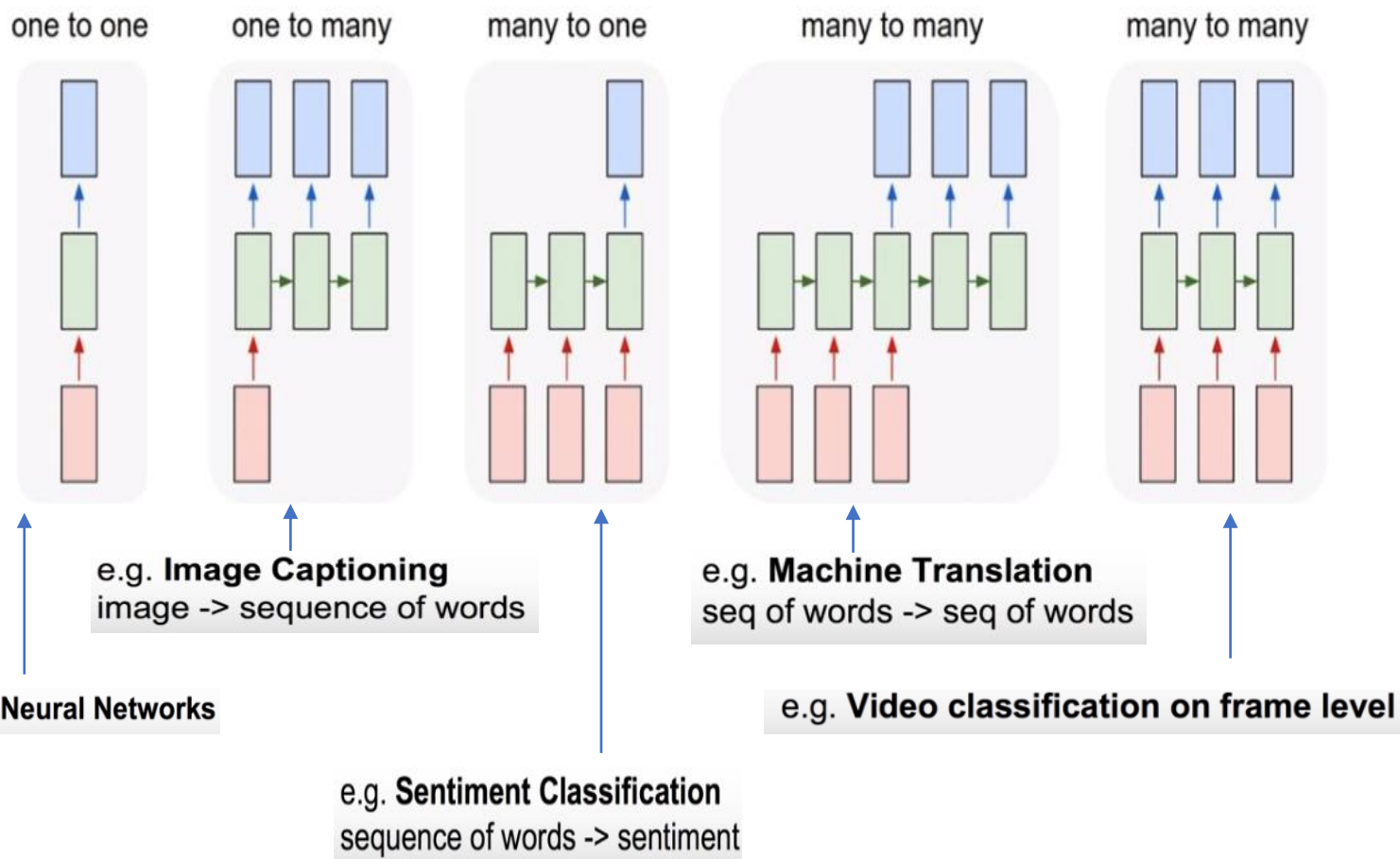


- ⑥ Output Layer 벡터 값들의 계산이며, 학습과정을 통하여 W 값들이 정해질 것임

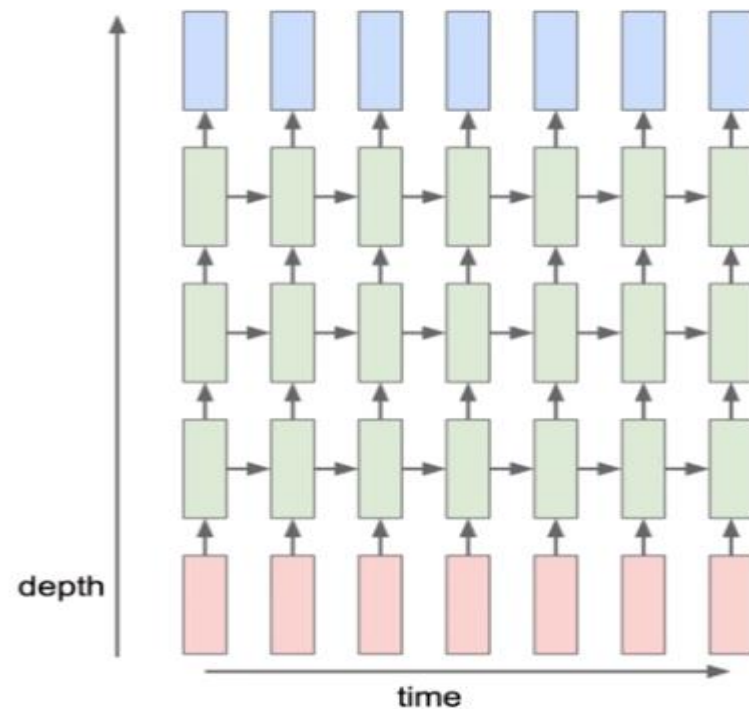


Softmax를 적용하고 학습을 할 수 있을 것임.

Recurrent Networks offer a lot of flexibility:



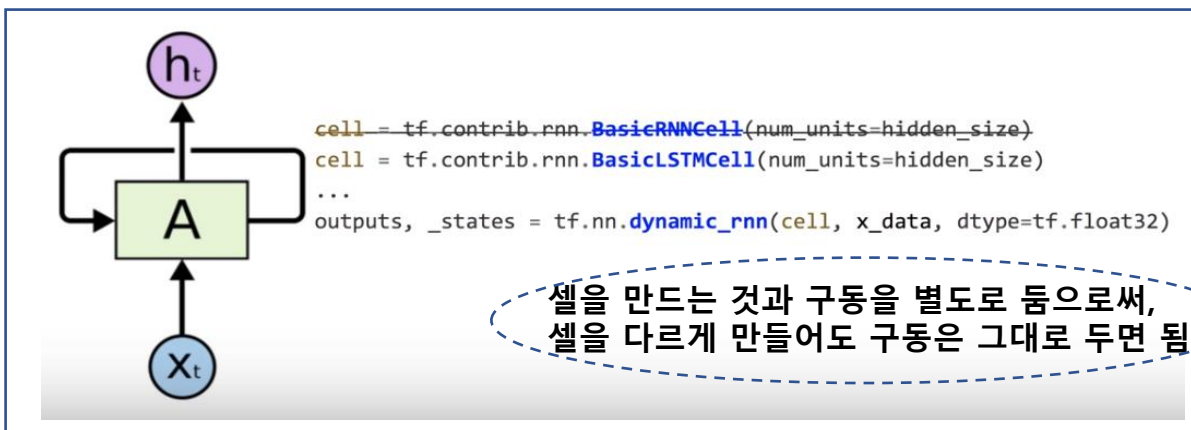
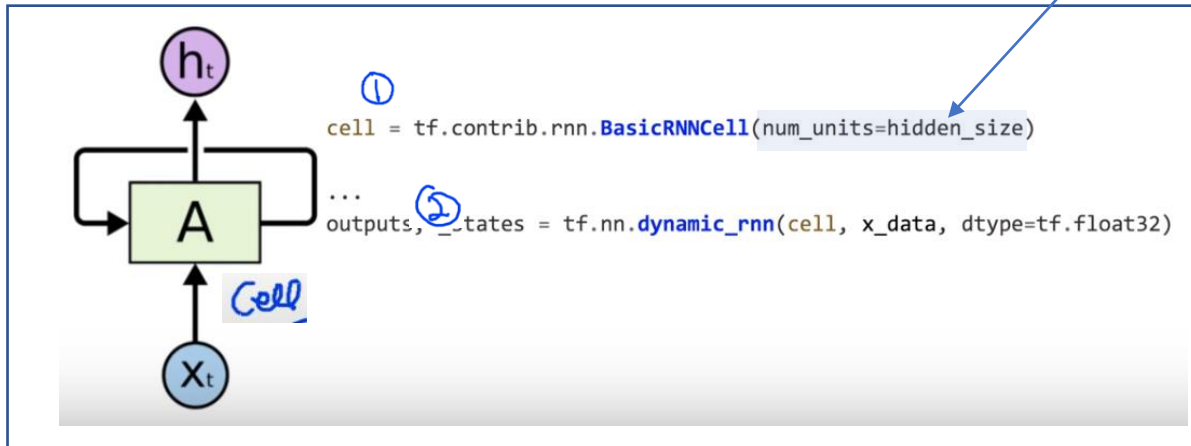
Multi-Layer RNN



RNN도 여러 개의 layer를 둘 수 있다.
이렇게 되면 더 복잡한 학습이 가능하다는 것이다.

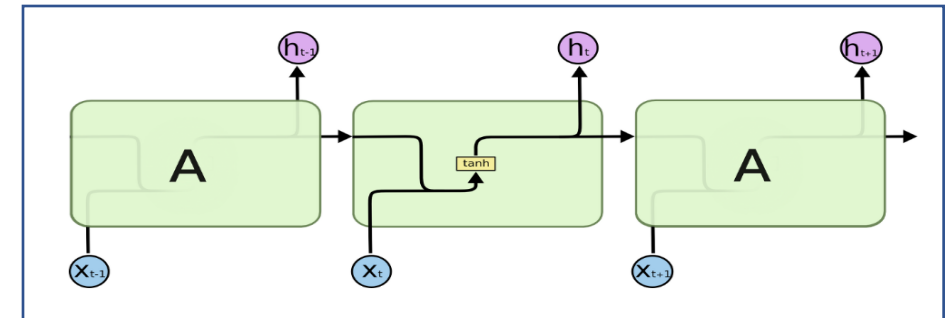
- ① Cell 을 만든다
 - 기본 RNN, LSTM, GRU 등의 셀이 있음
 - **hidden_size** 결정
- ② Cell 을 구동한다.
 - output 과 state 의 값을 내어준다

셀에서 나가는 출력[h(t)]의 크기를 나타내며, 임의의 값으로 정해줄 수 있다.

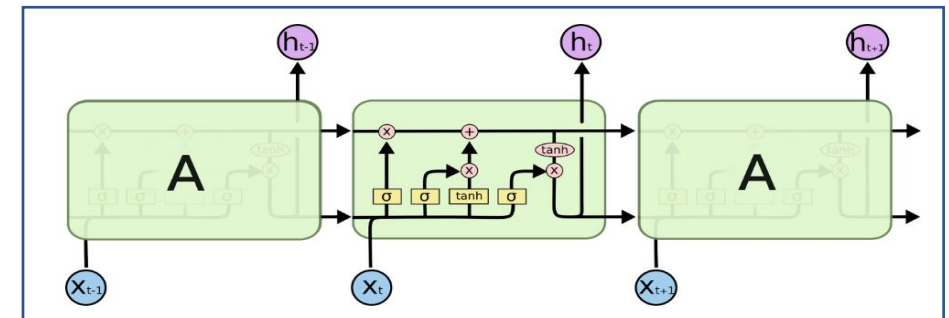


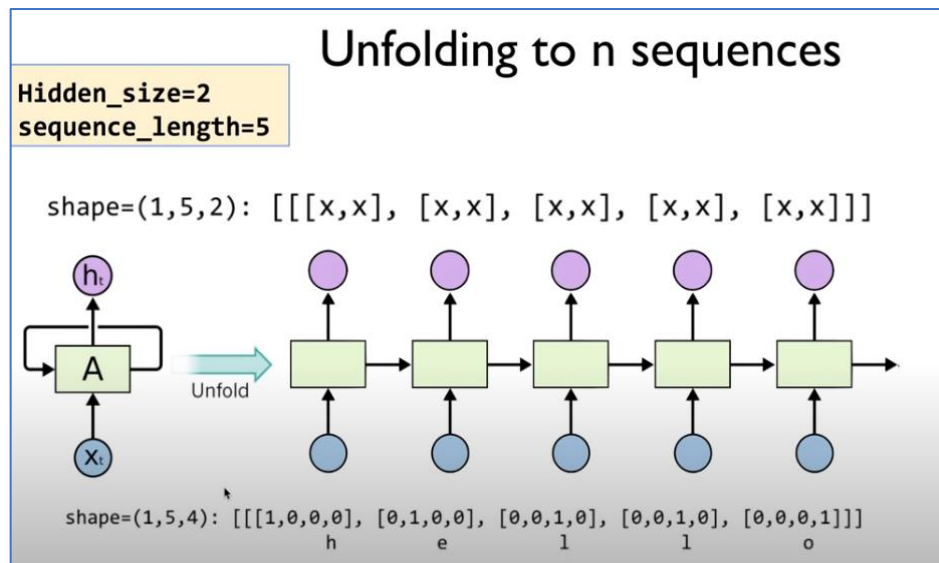
셀을 만드는 것과 구동을 별도로 둬으로써, 셀을 다르게 만들어도 구동은 그대로 두면 됨.

[RNN 셀]

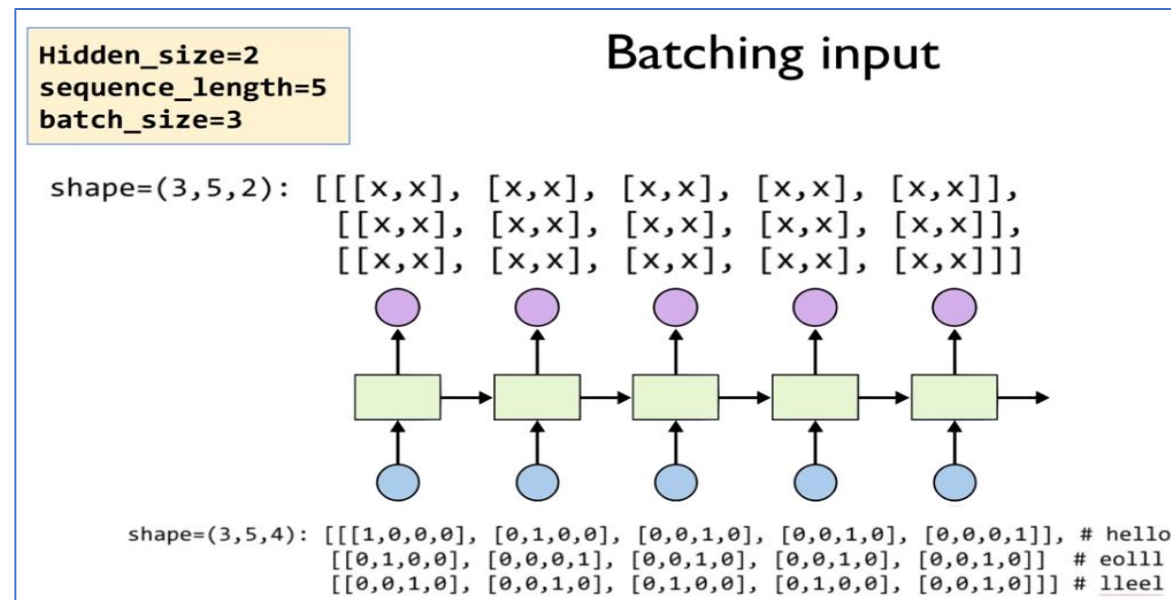


[LSTM 셀]

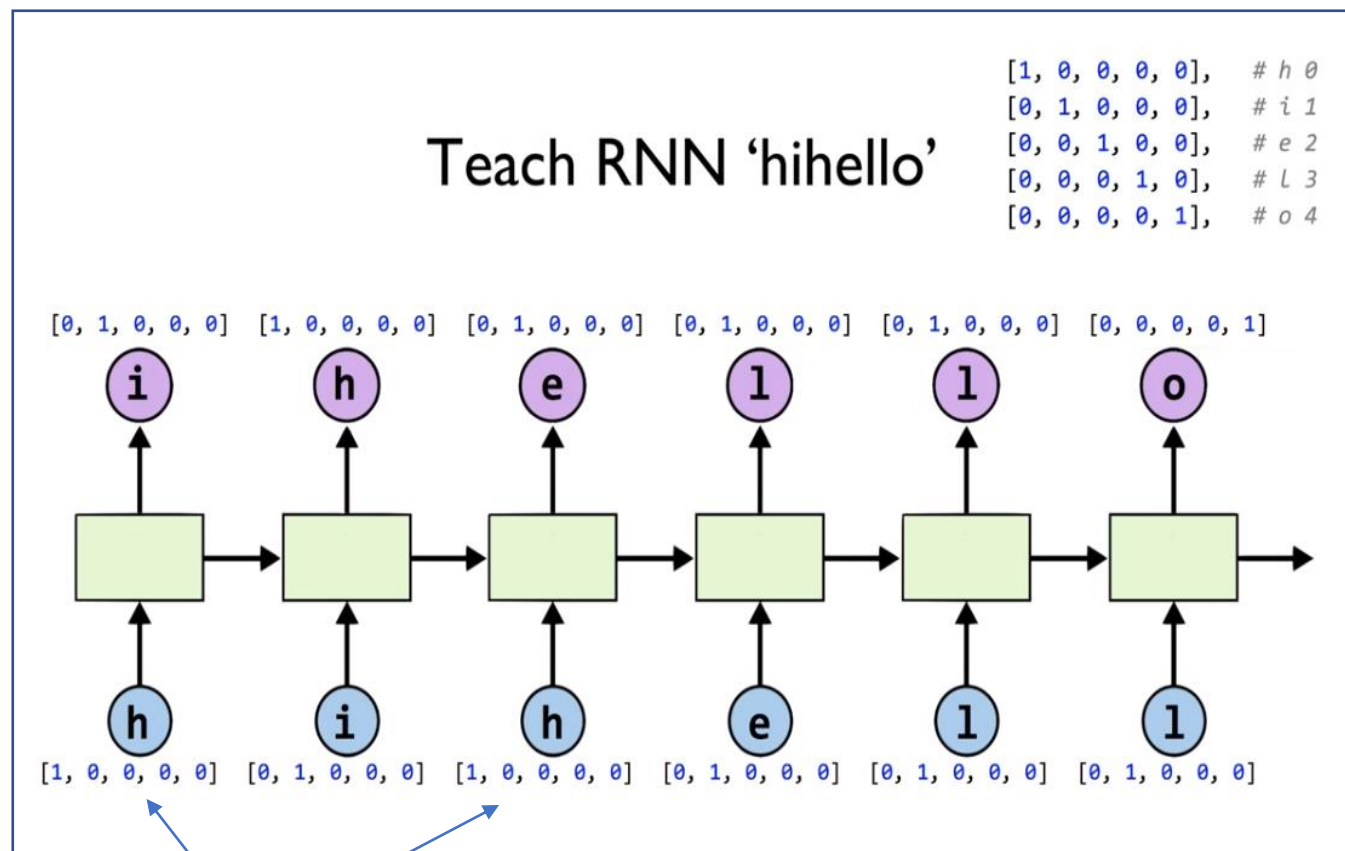




- **Input shape : (1, 5, 4)** (batch size, sequence_length, input dimension)
 - **Output shape : (1, 5, 2)** (batch size, sequence_length, hidden size)
- . Input dimension 은 입력 데이터 형태에 따라 결정됨.
. Hidden size 는 셀을 만들 때 값을 정해야 함.
. sequence length 는 unfold 했을 때, 몇 개의 series data 또는 sequence data를 입출력으로 할 것인가에 따라 결정됨.



- 문자열을 한 줄 씩 넣으면 비효율적이며, 여러 줄을 한꺼번에 입력. 효율적으로 만드는 것은 한 번에 여러 줄을 주는 것이다. 이것을 batch size 라고 함.
(상기 그림의 batch size = 3 임)



[one-hot encoding]

- text: 'hihello'
- unique chars (vocabulary, voc):
h, i, e, l, o
- voc index:
h:0, i:1, e:2, l:3, o:4

- Text의 unique 한 문자의 수가 one-hot encoding의 size 가 된다.
- 문자에 index 값을 할당(dictionary) 하고, index 값을 one-hot encoding 하였음.

○ RNN Parameters 를 결정함.

- **hidden_size = 5** (출력도 one-hot encode로 표현되어야 함)
- **input_dim = 5**
- **batch_size = 1**
- **Sequence_length = 6**

- 동일한 입력 h 에 대해서 출력이 각각 l, e 가 되어야 함
- 일반적인 forward net 으로는 쉽지 않으며, 이전의 문자가 무엇이었는지 알아야 정확한 출력 값을 낼 수 있다.

Data creation

```

idx2char = ['h', 'i', 'e', 'l', 'o'] # h=0, i=1, e=2, l=3, o=4
x_data = [[0, 1, 0, 2, 3, 3]] # hihell
x_one_hot = [[[1, 0, 0, 0, 0], # h 0
               [0, 1, 0, 0, 0], # i 1
               [1, 0, 0, 0, 0], # h 0
               [0, 0, 1, 0, 0], # e 2
               [0, 0, 0, 1, 0], # l 3
               [0, 0, 0, 1, 0]]] # l 3
y_data = [[1, 0, 2, 3, 3, 4]] # ihello
X = tf.placeholder(tf.float32,
                  [None, sequence_length, input_dim]) # X one-hot
Y = tf.placeholder(tf.int32, [None, sequence_length]) # Y Label

```

* x_data 를 그대로 사용하지 않고, one hot으로 바꾸어 사용한다.

* x_one_hot의 shape = (1,6,5) 임

batch_size → 1
sequence length → 6
Input_dim → 5

Feed to RNN

```

X = tf.placeholder(
    tf.float32, [None, sequence_length, hidden_size]) # X one-hot
Y = tf.placeholder(tf.int32, [None, sequence_length]) # Y Label

cell = tf.contrib.rnn.BasicLSTMCell(num_units=hidden_size,
    state_is_tuple=True)
initial_state = cell.zero_state(batch_size, tf.float32)
outputs, _states = tf.nn.dynamic_rnn(
    cell, X, initial_state=initial_state, dtype=tf.float32)

```

```

x_one_hot = [[[1, 0, 0, 0, 0], # h 0
               [0, 1, 0, 0, 0], # i 1
               [1, 0, 0, 0, 0], # h 0
               [0, 0, 1, 0, 0], # e 2
               [0, 0, 0, 1, 0], # l 3
               [0, 0, 0, 1, 0]]] # l 3
y_data = [[1, 0, 2, 3, 3, 4]] # ihello

```

input_dim

Cost: sequence_loss

```
outputs, _states = tf.nn.dynamic_rnn(
    cell, X, initial_state=initial_state, dtype=tf.float32)
weights = tf.ones([batch_size, sequence_length])

sequence_loss = tf.contrib.seq2seq.sequence_loss(
    logits=outputs, targets=Y, weights=weights)
loss = tf.reduce_mean(sequence_loss)
train = tf.train.AdamOptimizer(learning_rate=0.1).minimize(loss)
```

Training

```
prediction = tf.argmax(outputs, axis=2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(2000):
        l, _ = sess.run([loss, train], feed_dict={X: x_one_hot, Y: y_data})
        result = sess.run(prediction, feed_dict={X: x_one_hot})
        print(i, "loss:", l, "prediction: ", result, "true Y: ", y_data)

    # print char using dic
    result_str = [idx2char[c] for c in np.squeeze(result)]
    print("\tPrediction str: ", ''.join(result_str))
```


Cost: sequence_loss

```
outputs, _states = tf.nn.dynamic_rnn(
    cell, X, initial_state=initial_state, dtype=tf.float32)
weights = tf.ones([batch_size, sequence_length])

sequence_loss = tf.contrib.seq2seq.sequence_loss(
    logits=outputs, targets=Y, weights=weights)
loss = tf.reduce_mean(sequence_loss)
train = tf.train.AdamOptimizer(learning_rate=0.1).minimize(loss)
```

Training

```
prediction = tf.argmax(outputs, axis=2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(2000):
        l, _ = sess.run([loss, train], feed_dict={X: x_one_hot, Y: y_data})
        result = sess.run(prediction, feed_dict={X: x_one_hot})
        print(i, "loss:", l, "prediction: ", result, "true Y: ", y_data)

    # print char using dic
    result_str = [idx2char[c] for c in np.squeeze(result)]
    print("\tPrediction str: ", ''.join(result_str))
```

```
prediction = tf.argmax(outputs, axis=2)
```

Results

```
with tf.Session() as sess:
```

```
    sess.run(tf.global_variables_initializer())
```

```
    for i in range(2000):
```

```
        l, _ = sess.run([loss, train], feed_dict={X: x_one_hot, Y: y_data})
```

```
        result = sess.run(prediction, feed_dict={X: x_one_hot})
```

```
        print(i, "loss:", l, "prediction: ", result, "true Y: ", y_data)
```

```
        # print char using dic
```

```
        result_str = [idx2char[c] for c in np.squeeze(result)]
```

```
        print("\tPrediction str: ", ''.join(result_str))
```

```
0 loss: 1.55474 prediction: [[3 3 3 3 4 4]] true Y: [[1, 0, 2, 3, 3, 4]] Prediction str: lllloo
1 loss: 1.55081 prediction: [[3 3 3 3 4 4]] true Y: [[1, 0, 2, 3, 3, 4]] Prediction str: lllloo
2 loss: 1.54704 prediction: [[3 3 3 3 4 4]] true Y: [[1, 0, 2, 3, 3, 4]] Prediction str: lllloo
3 loss: 1.54342 prediction: [[3 3 3 3 4 4]] true Y: [[1, 0, 2, 3, 3, 4]] Prediction str: lllloo
...
1998 loss: 0.75305 prediction: [[1 0 2 3 3 4]] true Y: [[1, 0, 2, 3, 3, 4]] Prediction str: ihello
1999 loss: 0.752973 prediction: [[1 0 2 3 3 4]] true Y: [[1, 0, 2, 3, 3, 4]] Prediction str: ihello
```

처음에는 loss 값이 상당히 크다

하지만 학습이 진행되면서, loss 값은 떨어지고, Prediction은 우리가 원하는 값으로 접근해 가고, 문자열도 잘 예측하게 된다.

우리가 만든 RNN이 그 다음 문자를 예측하게 되었다.