# ML_7

Team BMS

# INDEX

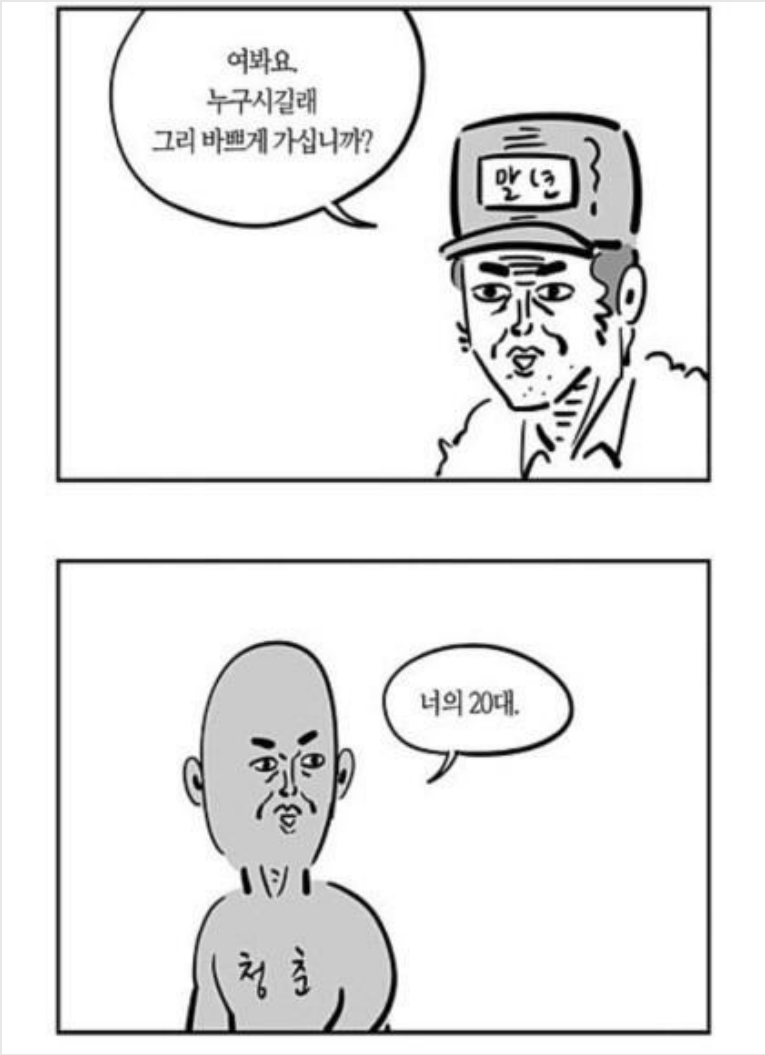
ML Study
7 Week

# Ice Breaking

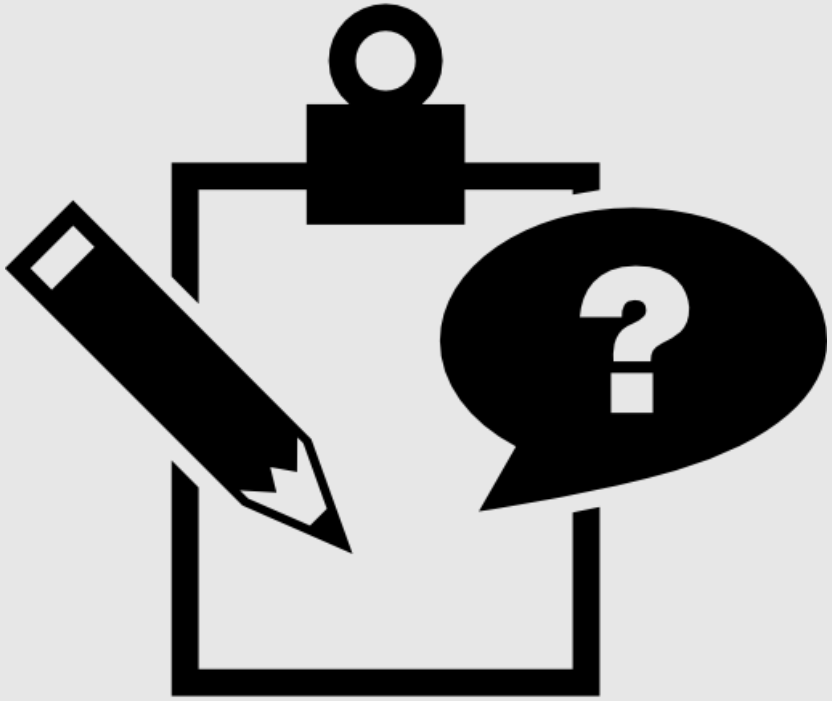
# Ice Breaking


〈http://www.asiae.co.kr/news/view.htm?sec=1120&idxno=2018042308505127147〉

# Ice Breaking




〈https://www.instiz.net/pt/4904516〉



# Review

# Test Answer

맞추라고 낸 문제는 파란색
함정 문제는 빨간색

● 주관식

1. 지도 학습 머신러닝 알고리즘의 대표적인 예시 2개? [앙상블 제외]

   SVM, Naïve Bayes, Knn, Linear Regression, Decision Tree, Logistic Regression, ANN, 등-

2. 비지도 학습으로 할 수 있는 것 예시 2개?
   [지도 학습으로 할 수 있는 것은 분류, 회귀, 등 이 있다.]

   차원 축소, 이상치 탐지, 노이즈 제거, 군집화, 연관규칙 생성, 등-

3. 지도 학습 성능 평가 지표 예시 4개? (분류, 회귀 상관 없음)

   회귀: MAE, RMSE, RAE, RSE, R2

   분류: ACC, Precision, Recall, F-Score, AUC, Log-loss

평가 지표는 알아두면 좋다.

4. Decision Tree에서 노드를 나눌 때 쓰이는 기법 중 예시 2개?

   Variance Reduction, Entropy, Gini-index

Random Forest때문에라도 알아두면 좋다.

# Test Answer

맞추라고 낸 문제는 파란색
함정 문제는 빨간색

● OX

1. Radial Basis Function(RBF)는 은닉층이 1개이다. ( ) O

2. Bias-Variance Trade off의 관점에서 봤을 때 일반적인 모형은 Bias를 줄이는 것이 목적이고, 앙상블 모형은 Variance를 줄이는 것이 목적이다. ( ) O

3. Bagging의 대표적인 알고리즘은 Random Forest이다. ( ) O

4. Support Vector Machine(SVM)은 OvA(One-Versus-All)의 대표적인 모형 중 하나이다. ( ) X

5. 정규방정식을 사용할 경우 Scaling이 필수적이다. ( ) X

6. RAM사용량 증가와 과적합을 막기 위해 사용되는 방법이 Regularization이다. ( ) O

7. L2 Norm은 'Manhattan Norm'이라고 불린다. ( ) X

8. Stochastic Gradient Descent(SGD)은 배치 크기가 1인 경사하강법 알고리즘이다. ( ) O

9. XGBoost는 Extreme Gradient Boosting의 약자로, Regularization항이 있다는 것과 병렬 계산이 가능하다는 것이 특징이다. ( ) O

10. Isomap은 각 포인트들의 이웃을 찾아서 그 이웃을 기반으로 한 가중치를 계산한 후 그 값들로 MDS를 진행하여 나오는 값이다. ( ) O

함정문제지만, 앙상블을 이해하고 있다면 충분히 풀 수 있다.

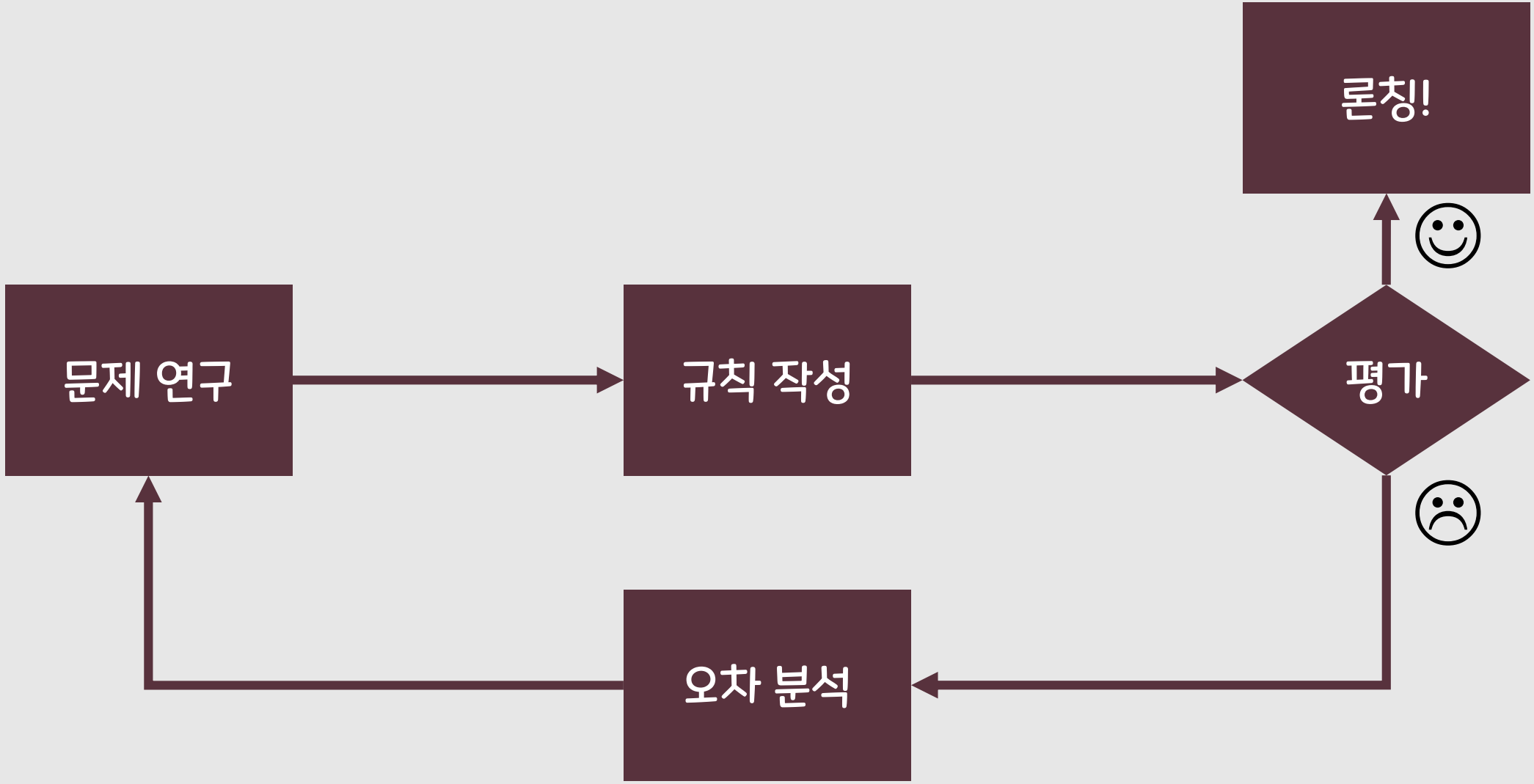
SVM이 연산이 오래 걸리는 이유이다.

함정문제이면서 회귀수업 문제이다.

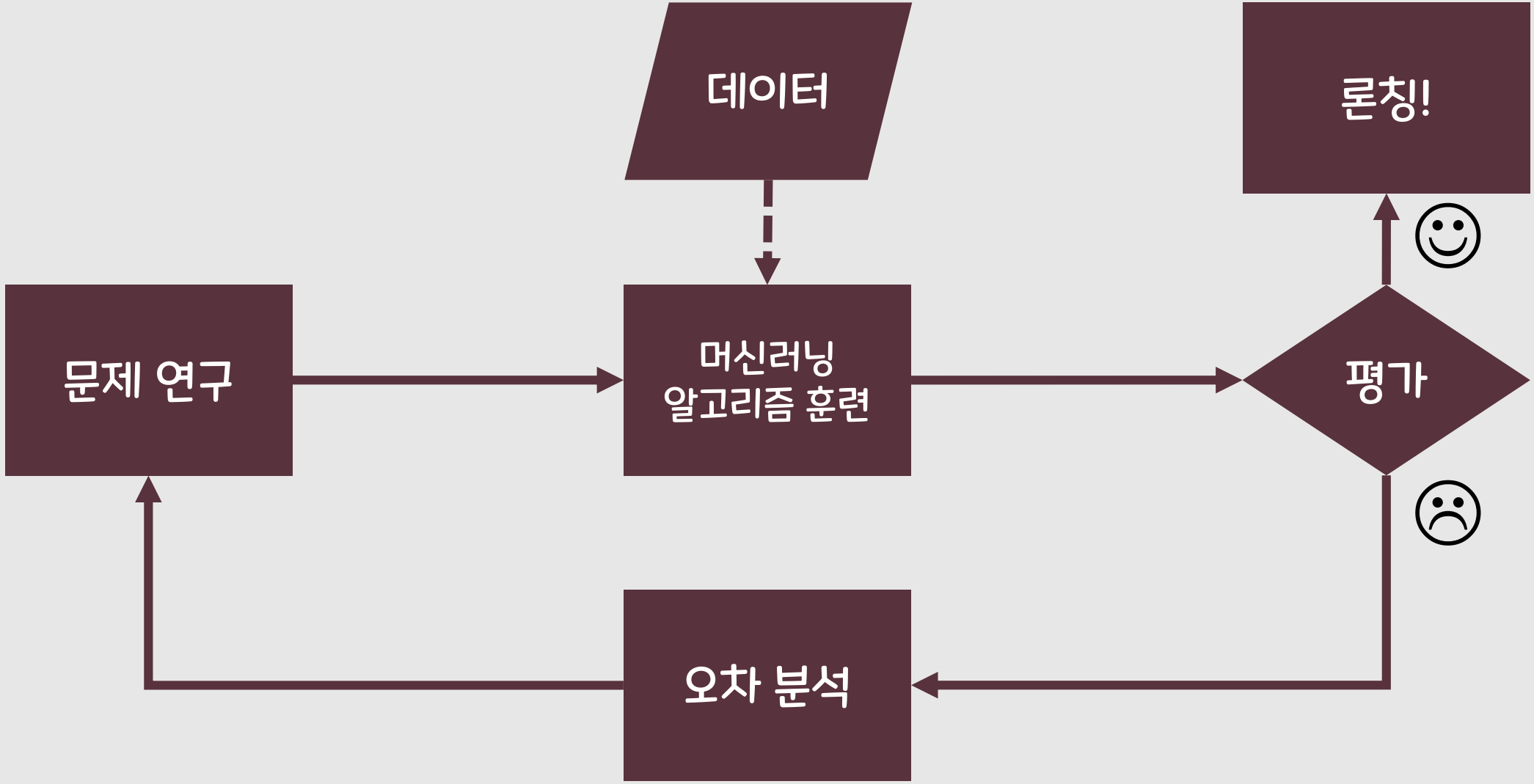
알아두어야 할 내용들이다.

알아두어야 할 내용들이다.

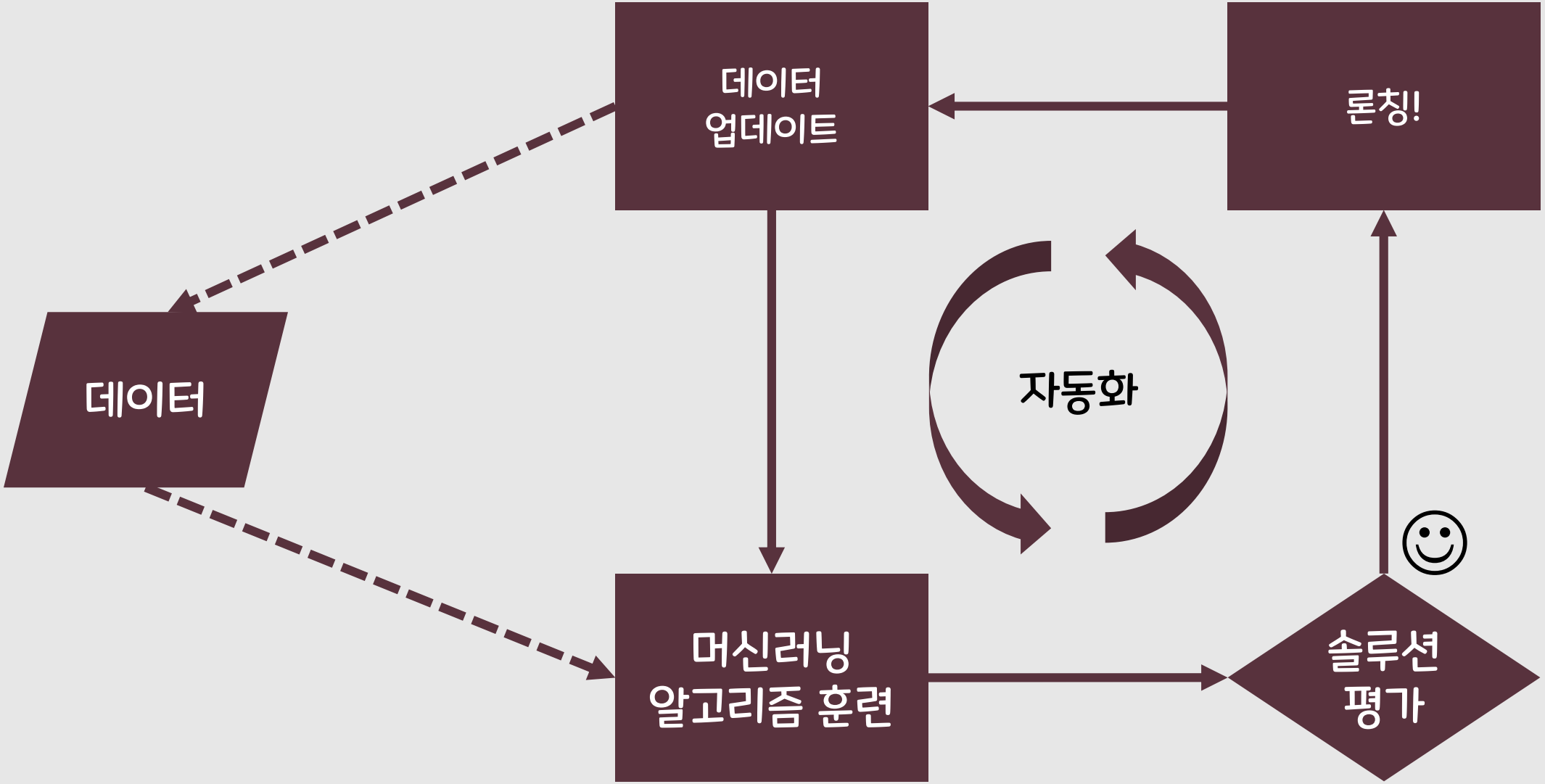
# Chapter 1: The Machine Learning Landscape



# Chapter 1: The Machine Learning Landscape



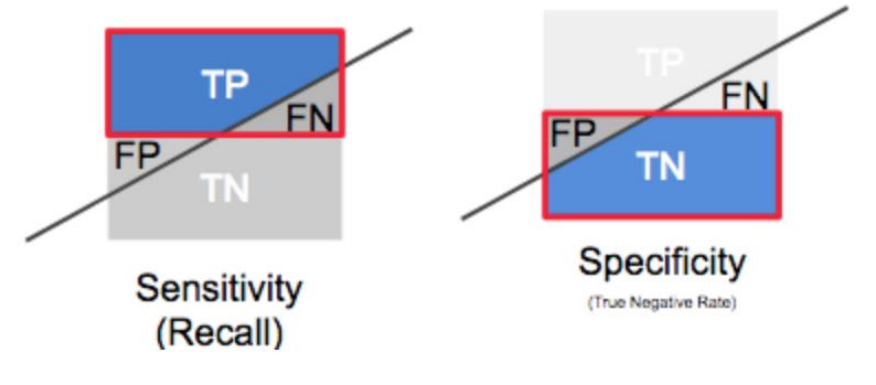
# Chapter 1: The Machine Learning Landscape

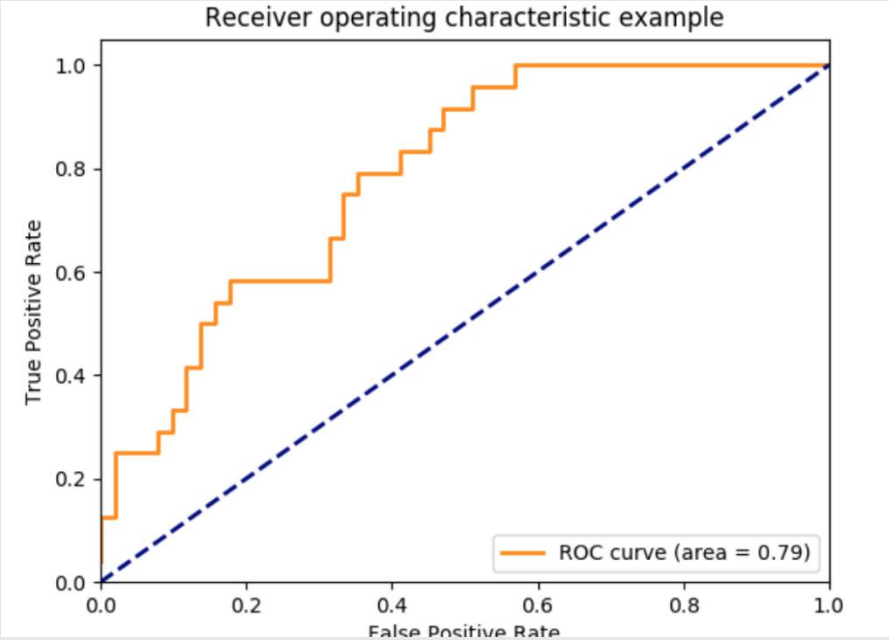


# Chapter 2: The Machine Learning Landscape

1. 큰 그림을 봅니다.
(Look at the Big Picture)
2. 데이터를 구합니다.
(Get the Data)
3. 데이터로부터 통찰을 얻기위해 탐색하고 시각화합니다.
(Discover and Visualize the Data to Gain Insights)
4. 머신러닝 알고리즘을 위해 데이터를 준비합니다.
(Prepare the Data for Machine Learning Algorithms)
5. 모델을 선택하고 훈련시킵니다.
(Select and Train a Model)
6. 모델을 상세하게 조정합니다.
(Fine-Tune Your Model)
7. 솔루션을 제시합니다.
(Present a Solution)
8. 시스템을 론칭하고 모니터링하고 유지보수합니다.
(Launch, Monitor, and Maintain Your System)

# Chapter 3: Classification



# Chapter 3: Classification

## F-Score

모델의 성능을 하나의 수로 표현할때, ROC나 PR 그래프의 AUC를 사용하면 되지만, AUC를 계산하려면 여러 Throughput에 대해서 Precision, Recall, Specificity 값을 측정해야 한다.
그렇다면 Throughput을 이미 알고 있거나 또는 다양한 Throughput에 대해서 어떤 Throughput이 좋은지를 하나의 수로 모델의 성능을 평가하려면 어떻게 해야할까? 이를 위해서 사용하는 것이 F-Score 라는 값이 있다.

When measuring how well you're doing, it's often useful to have a single number to describe your performance
When measuring how well you're doing, it's often useful to have a single number to describe your performance. We could define that number to be, for instance, the mean of your precision and your recall. This is exactly what the F1-score is.
https://www.quora.com/What-is-an-intuitive-explanation-of-F-score
F Score에 대한 계산은 다음 공식을 이용한다. 큰 의미상으로 보자면 Precision과 Recall에 대한 평균인데, 그냥 평균을 내면, 값의 외곡 현상이 생기기 때문에, 가중치를 주는 평균이라고 이해하면 된다.

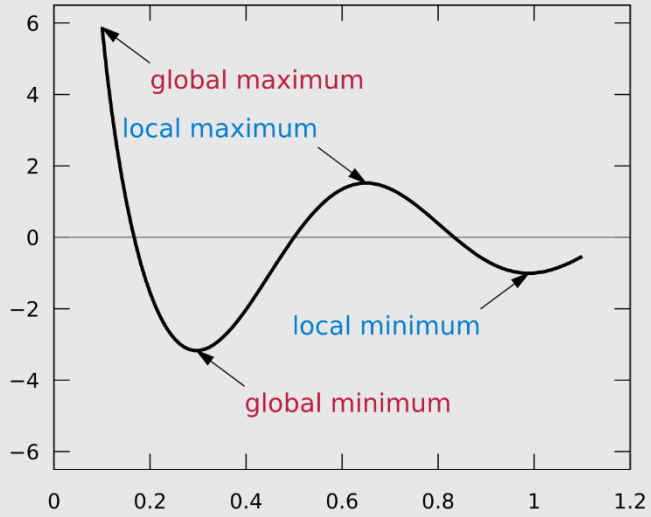
$$F_\beta = \frac{(1+\beta^2)(\mathrm{PREC}\cdot\mathrm{REC})}{(\beta^2\cdot\mathrm{PREC}+\mathrm{REC})}$$

특히 β가 1인 경우 (즉 F1)를 F1 Score라고 하고, 모델의 성능 평가 지표로 많이 사용한다.

〈http://bcho.tistory.com/tag/F1%20score〉

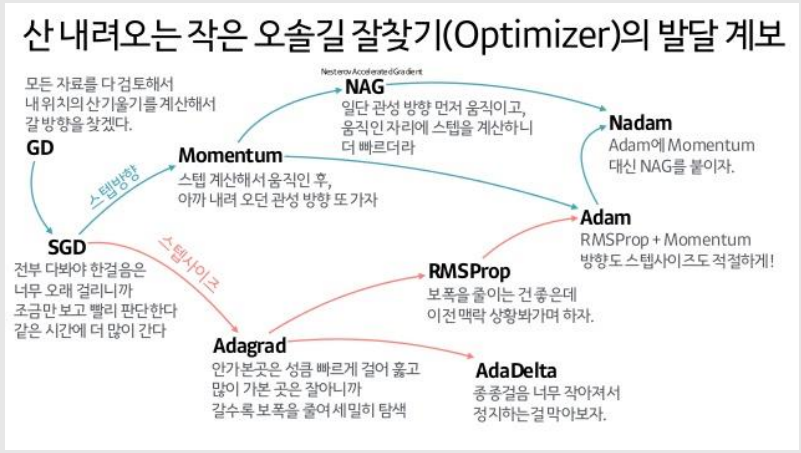
# Chapter 4: Training Models

## Gradient Descent

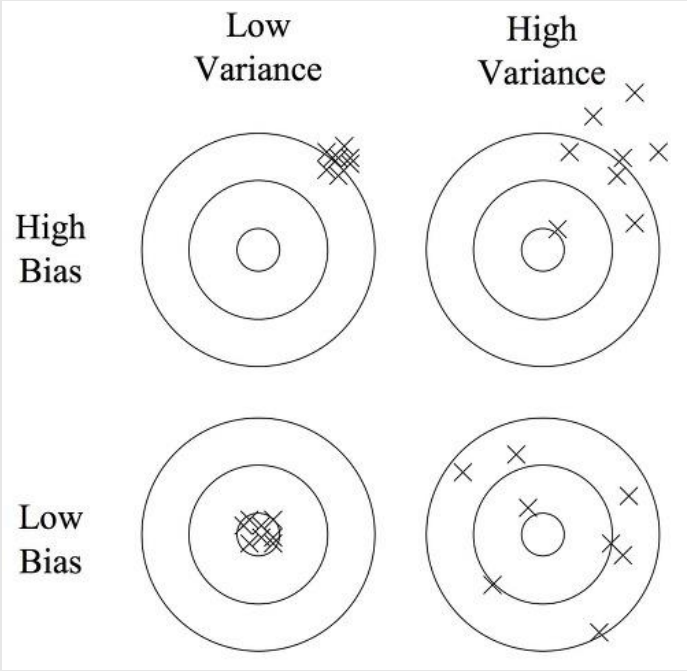



〈https://en.wikipedia.org/wiki/Maxima_and_minima〉

## Optimizer

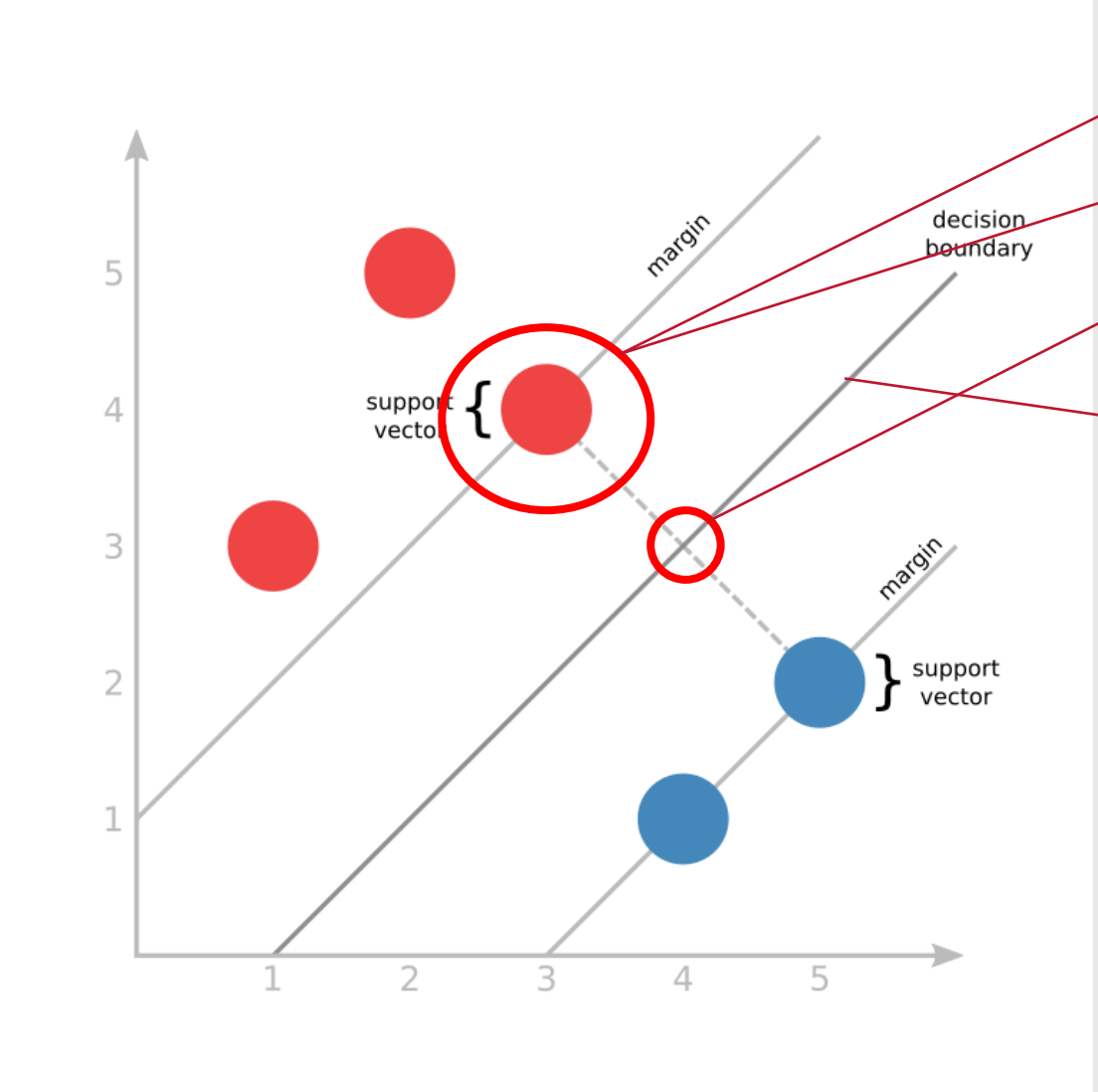



## Bias-Variance Trade-off




〈https://en.wikipedia.org/wiki/Maxima_and_minima〉
〈https://www.slideshare.net/yongho/ss-79607172〉
〈https://www.quora.com/What-is-the-best-way-to-explain-the-bias-variance-trade-off-in-layman%E2%80%99s-terms〉

# Chapter 5: Support Vector Machine




직각이라고 할 때

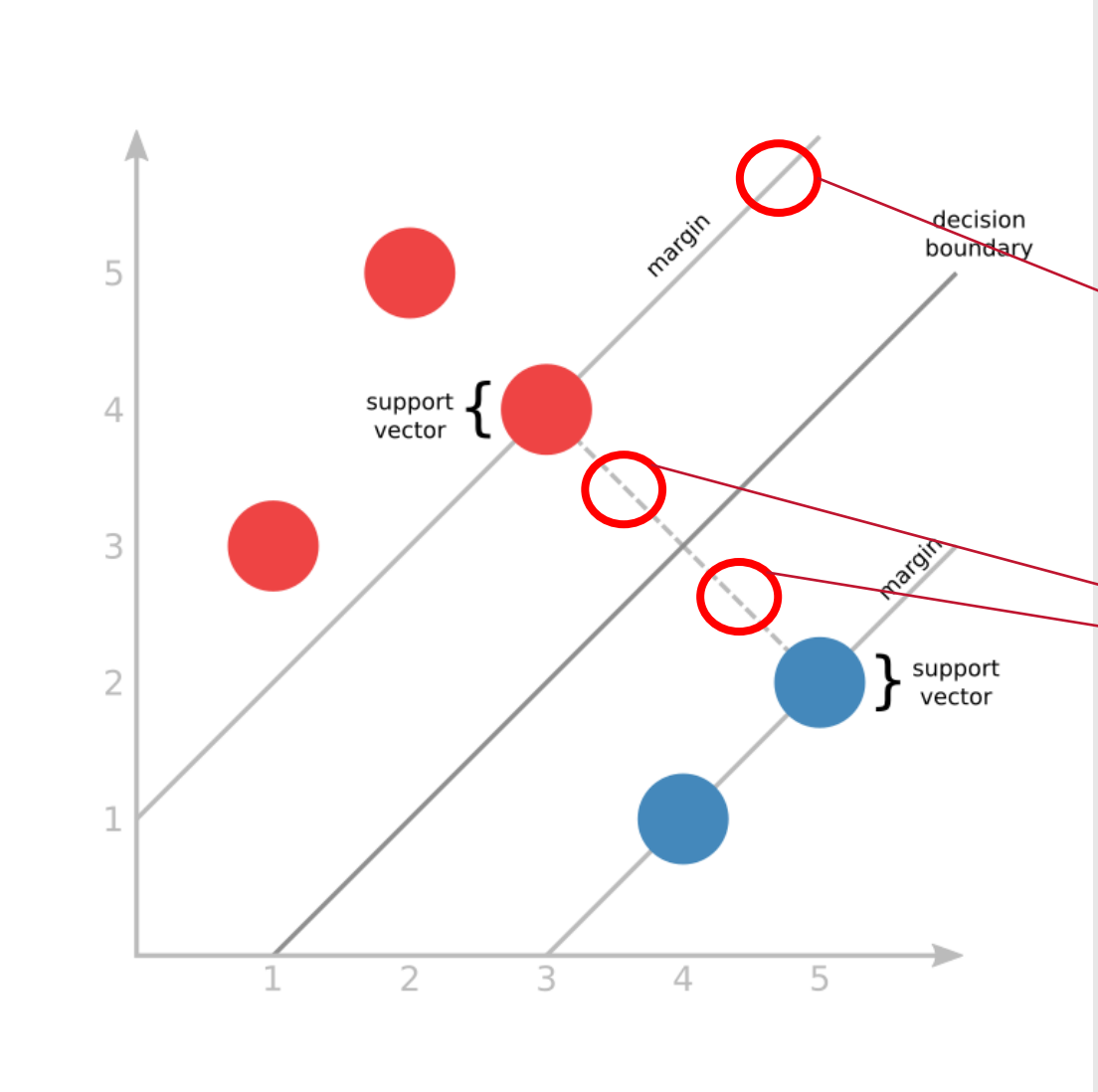
$$x = x_p + r\frac{w}{||w||}, f(x_p) = 0$$

$$f(x) = w \cdot x + b = w\left(x_p + r\frac{w}{||w||}\right) + b$$

$$= wx_p + b + r\frac{w \cdot w}{||w||} = r||w||$$

$$\Rightarrow f(x) = r||w|| \Rightarrow r = \frac{f(x)}{||w||} = \frac{a}{||w||}$$

# Chapter 5: Support Vector Machine




최적화 = 최대 마진 결정 경계

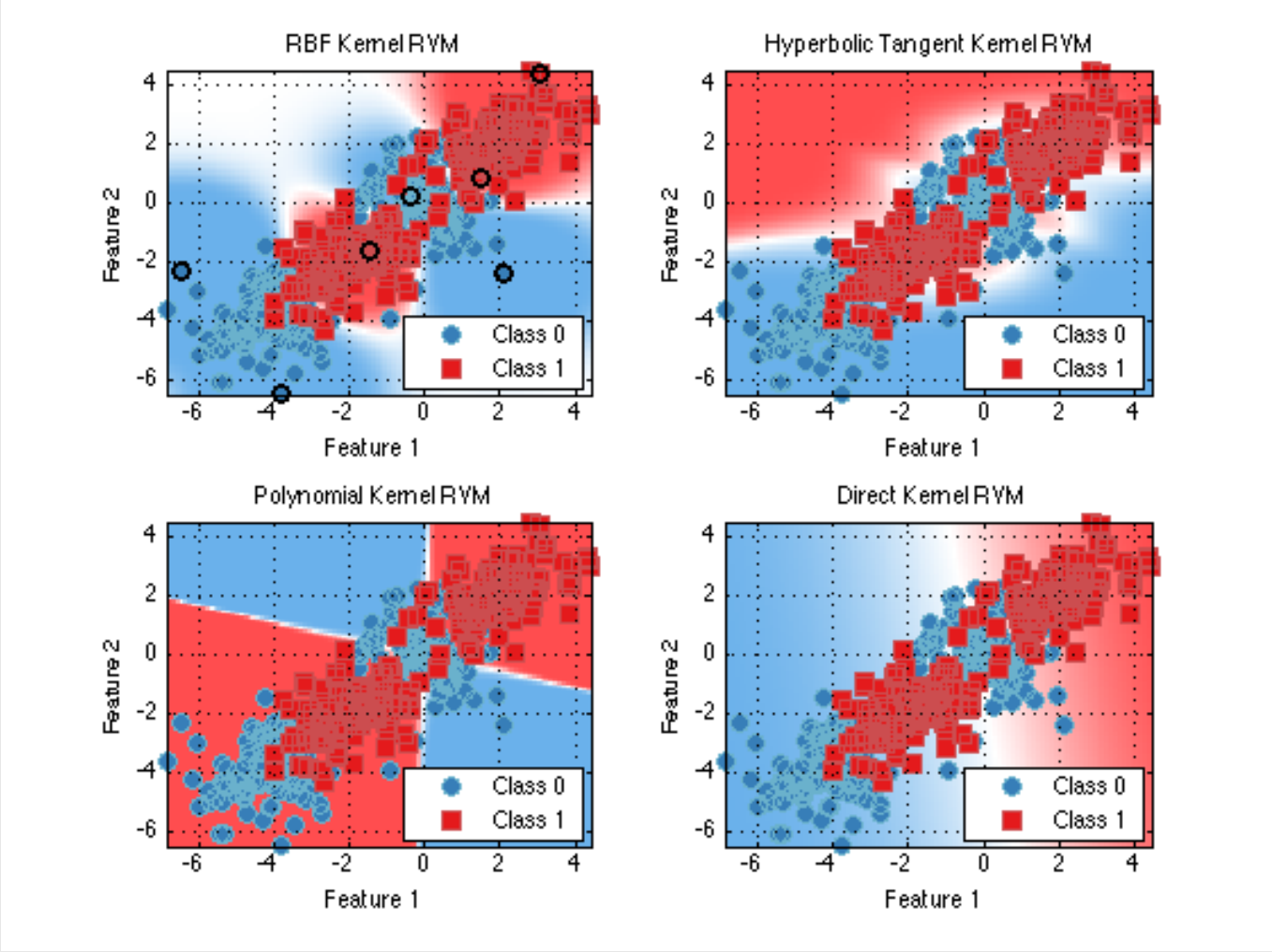
$$r = \frac{a}{||w||}$$

$$f(x) = w \cdot x + b = -a$$

$$\max_{w,b} 2r = \frac{2a}{||w||}$$

$$\Rightarrow \min_{w,b} ||w||$$

# Chapter 5: Support Vector Machine



# Chapter 6: Decision Tree

## Information Gain (Entropy)

$$I_E(f) = -\sum_{i=1}^{m} f_i \log_2 f_i$$

## Gini Index

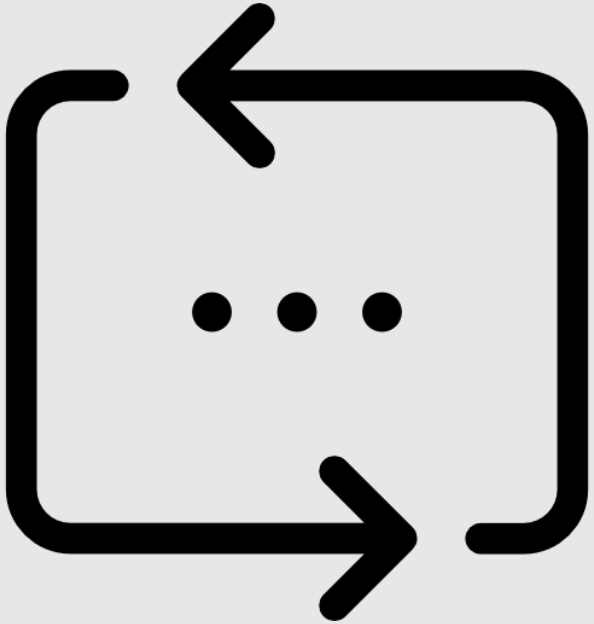
$$I_G(f) = \sum_{i=1}^{m} f_i(1 - f_i) = 1 - \sum_{i=1}^{m} f_i^2$$

## Variance Reduction

$$I_V(N) = \frac{1}{|S|}\sum_{i \in S}\sum_{j \in S}\frac{1}{2}(x_i - x_j)^2 - \left(\frac{1}{|S_t|}\sum_{i \in S_t}\sum_{j \in S_t}\frac{1}{2}(x_i - x_j)^2\right.$$

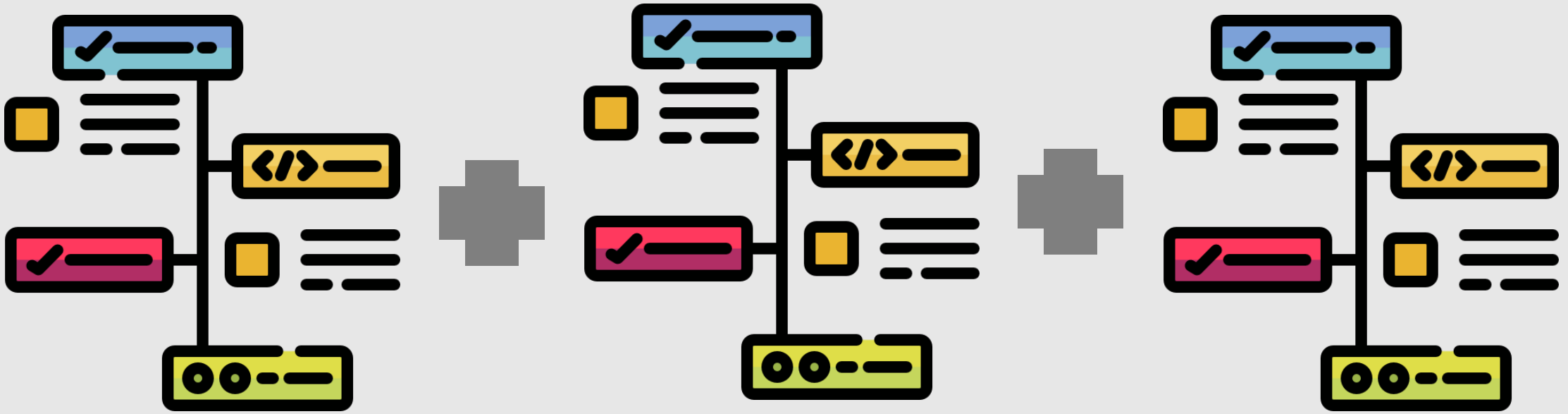
# Chapter 6: Decision Tree

재귀적 분기
(Recursive Partitioning)



가지치기
(Pruning)

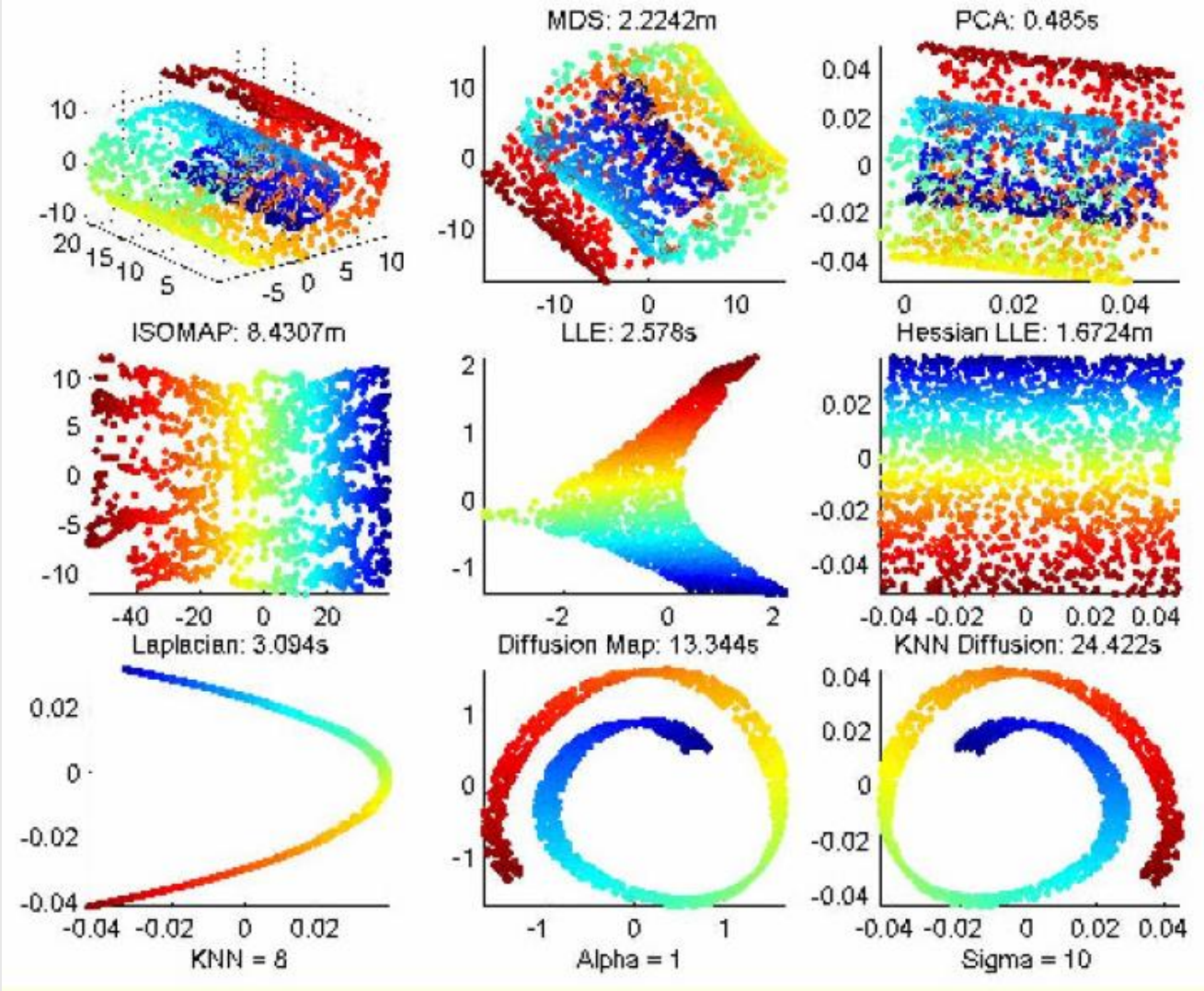
# Chapter 7: Ensemble Learning and Random Forests



여러 가지 동일한 종류의 혹은 서로 상이한 모형들의 예측/분류 결과를 종합하여
최종적인 의사결정에 활용하는 방법론

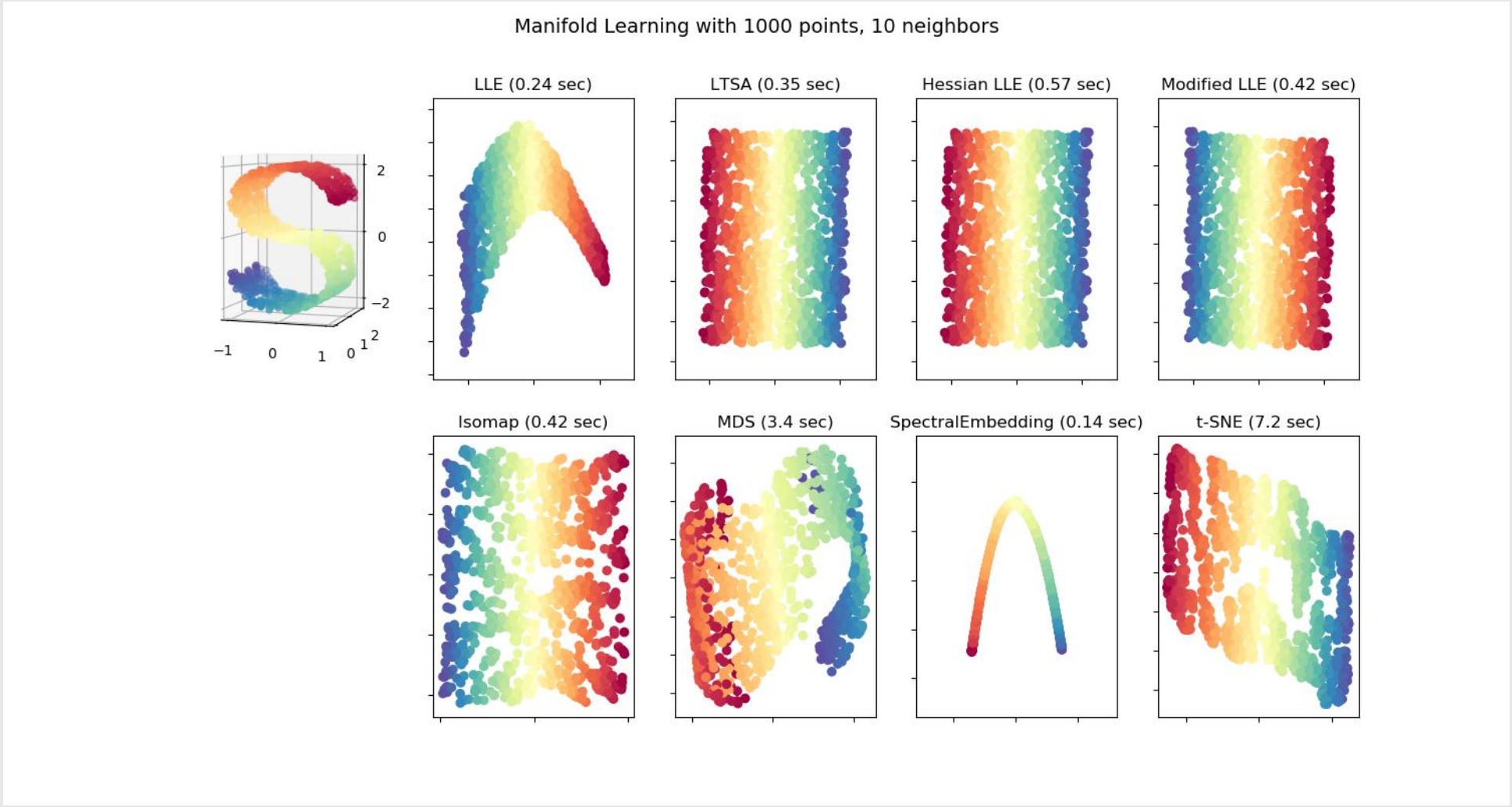
# Chapter 7: Ensemble Learning and Random Forests

# Chapter 8: Dimensionality Reduction



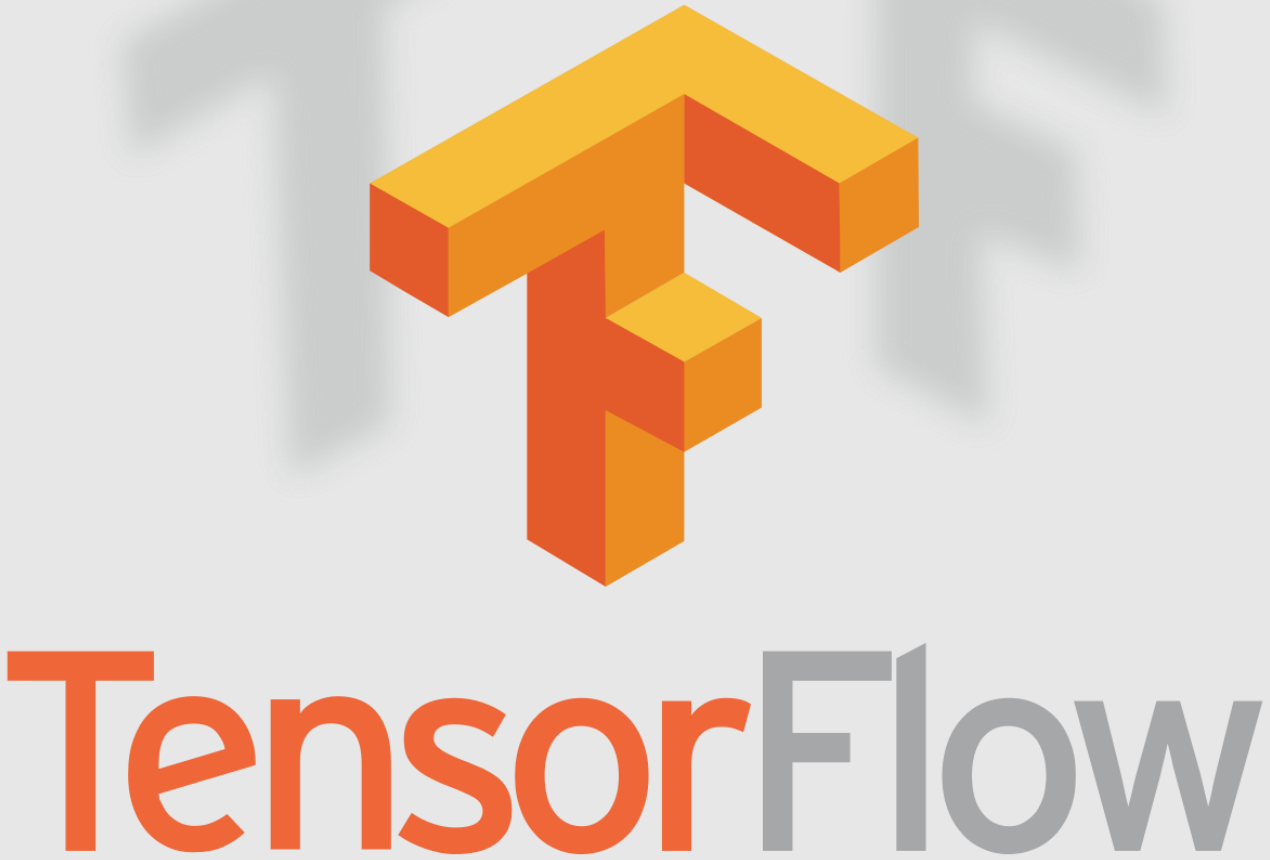

〈http://www.cs.haifa.ac.il/~rita/uml_course/add_mat/mani_presentation.pdf〉

# Chapter 8: Dimensionality Reduction



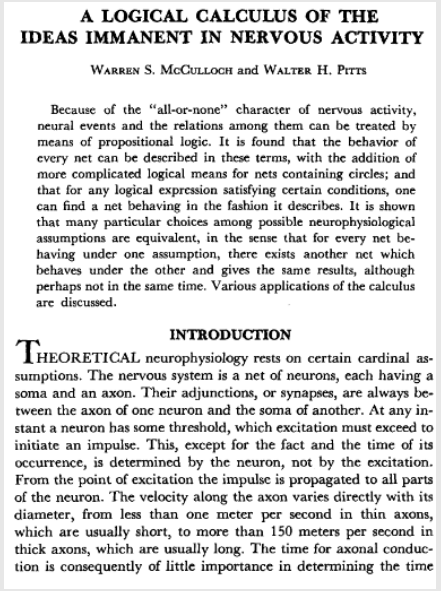

〈http://scikit-learn.org/stable/modules/manifold.html〉

# Chapter 9: Up and Running with TensorFlow



# Chapter 10: Introduction to Artificial Neural Networks

## 1943



A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY

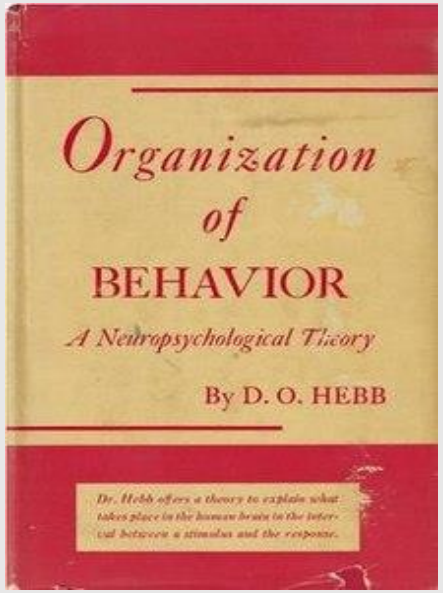
WARREN S. MCCULLOCH and WALTER H. PITTS

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

INTRODUCTION

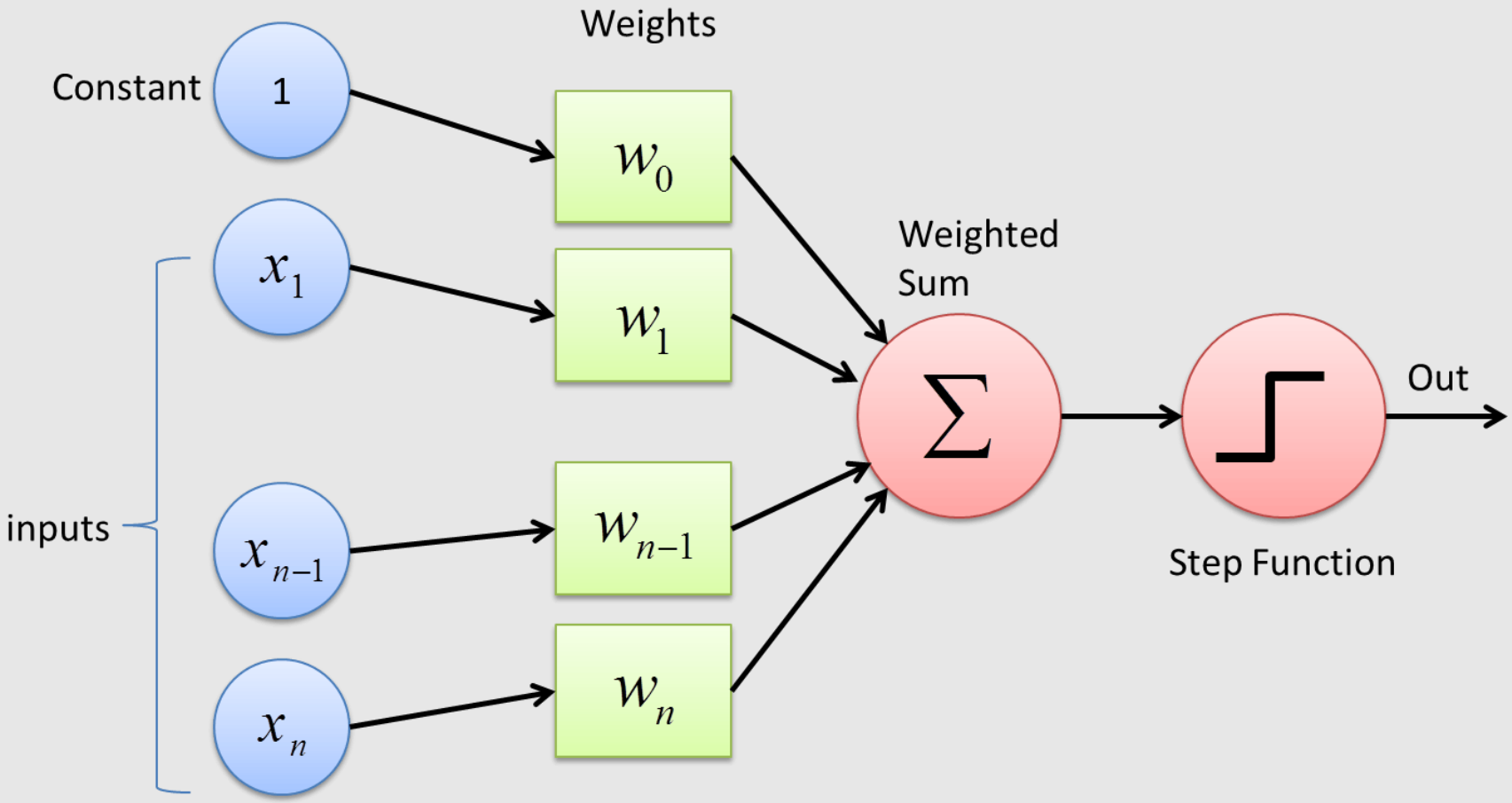
THEORETICAL neurophysiology rests on certain cardinal assumptions. The nervous system is a net of neurons, each having a soma and an axon. Their adjunctions, or synapses, are always between the axon of one neuron and the soma of another. At any instant a neuron has some threshold, which excitation must exceed to initiate an impulse. This, except for the fact and the time of its occurrence, is determined by the neuron, not by the excitation. From the point of excitation the impulse is propagated to all parts of the neuron. The velocity along the axon varies directly with its diameter, from less than one meter per second in thin axons, which are usually short, to more than 150 meters per second in thick axons, which are usually long. The time for axonal conduction is consequently of little importance in determining the time

## 1949




## 1957

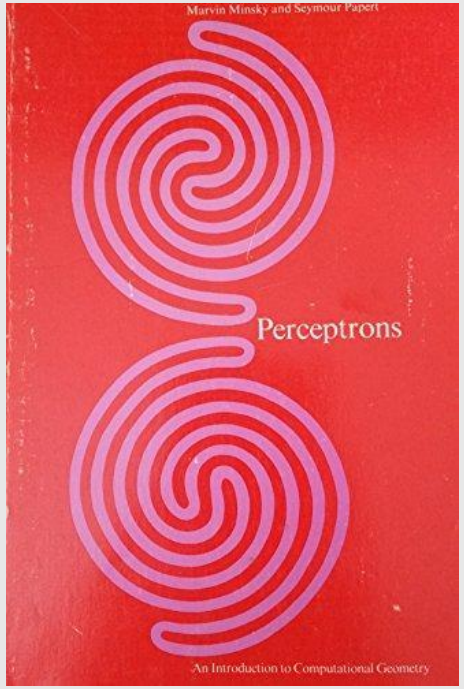
# Chapter 10: Introduction to Artificial Neural Networks



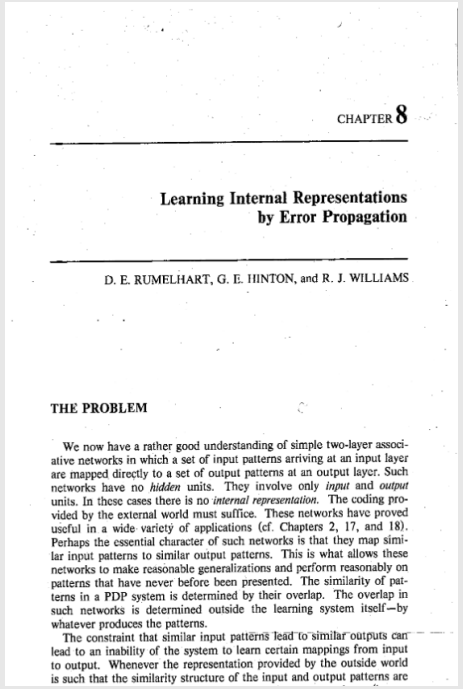

〈https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53〉

# Chapter 10: Introduction to Artificial Neural Networks

## 1969




## 1986



CHAPTER 8

**Learning Internal Representations by Error Propagation**

D. E. RUMELHART, G. E. HINTON, and R. J. WILLIAMS

**THE PROBLEM**

We now have a rather good understanding of simple two-layer associative networks in which a set of input patterns arriving at an input layer are mapped directly to a set of output patterns at an output layer. Such networks have no *hidden* units. They involve only *input* and *output* units. In these cases there is no *internal representation*. The coding provided by the external world must suffice. These networks have proved useful in a wide variety of applications (cf. Chapters 2, 17, and 18). Perhaps the essential character of such networks is that they map similar input patterns to similar output patterns. This is what allows these networks to make reasonable generalizations and perform reasonably on patterns that have never before been presented. The similarity of patterns in a PDP system is determined by their overlap. The overlap in such networks is determined outside the learning system itself—by whatever produces the patterns.

The constraint that similar input patterns lead to similar outputs can lead to an inability of the system to learn certain mappings from input to output. Whenever the representation provided by the outside world is such that the similarity structure of the input and output patterns are

## 2011

**Deep Sparse Rectifier Neural Networks**

**Xavier Glorot**
DIRO, Université de Montréal
Montréal, QC, Canada
glorotxa@iro.umontreal.ca

**Antoine Bordes**
Heudiasyc, UMR CNRS 6599
UTC, Compiègne, France
and
DIRO, Université de Montréal
Montréal, QC, Canada
antoine.bordes@hds.utc.fr

**Yoshua Bengio**
DIRO, Université de Montréal
Montréal, QC, Canada
bengioy@iro.umontreal.ca

**Abstract**

While logistic sigmoid neurons are more biologically plausible than hyperbolic tangent neurons, the latter work better for training multi-layer neural networks. This paper shows that rectifying neurons are an even better model of biological neurons and yield equal or better performance than hyperbolic tangent networks in spite of the hard non-linearity and non-differentiability at zero, creating sparse representations with true zeros, which seem remarkably suitable for naturally sparse data. Even though they can take advantage of semi-supervised setups with extra-unlabeled data, deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labeled datasets. Hence, these results can be seen as a new milestone in the attempts at understanding the difficulty in training deep but purely supervised neural networks, and closing the performance gap between neural networks learnt with and without unsupervised pre-training.

**1 Introduction**

Many differences exist between the neural network models used by machine learning researchers and those used by computational neuroscientists. This is in part because the objective of the former is to obtain computationally efficient learners, that generalize well to new examples, whereas the objective of the latter is to abstract out neuroscientific data while obtaining explanations of the principles involved, providing predictions and guidance for future biological experiments. Areas where both objectives coincide are therefore particularly worthy of investigation, pointing towards computationally motivated principles of operation in the brain that can also enhance research in artificial intelligence. In this paper we show that two common gaps between computational neuroscience models and machine learning neural network models can be bridged by using the following linear by part activation : $\max(0, x)$, called the rectifier (or hinge) activation function. Experimental results will show engaging training behavior of this activation function, especially for *deep architectures* (see Bengio (2009) for a review), i.e., where the number of hidden layers in the neural network is 3 or more.

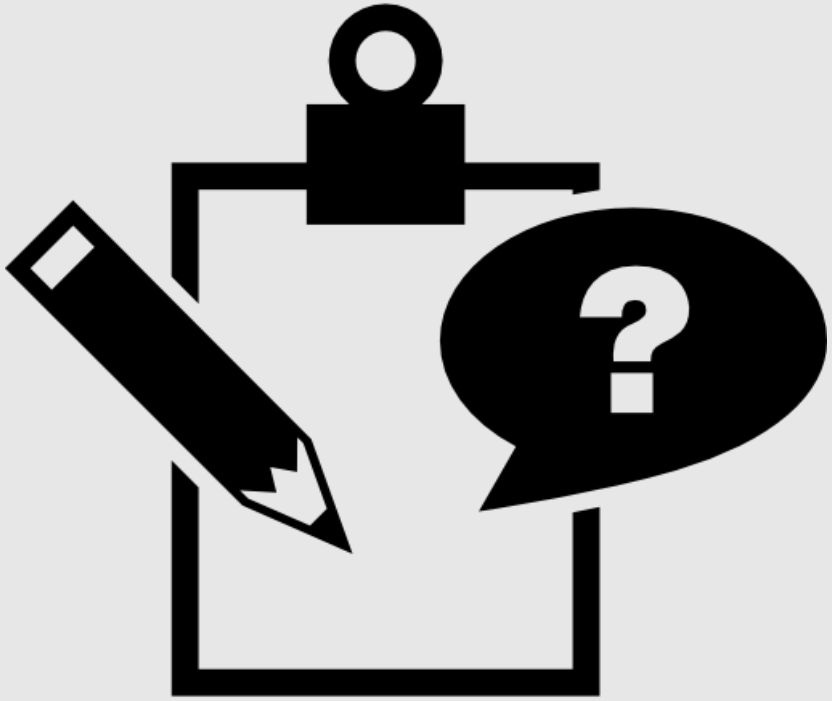
Recent theoretical and empirical work in statistical machine learning has demonstrated the importance of learning algorithms for deep architectures. This is in part inspired by observations of the mammalian visual cortex, which consists of a chain of processing elements, each of which is associated with a different representation of the raw visual input. This is particularly clear in the primate visual system (Serre *et al.*, 2007), with its sequence of processing stages: detection of edges, primitive shapes, and moving up to gradually more complex visual shapes. Interestingly, it was found that the features learned in deep architectures resemble those observed in the first two of these stages (in areas V1 and V2 of visual cortex) (Lee *et al.*, 2008), and that they become increasingly invariant to factors of variation (such as camera movement) in higher layers (Goodfellow *et al.*, 2009).

Appearing in Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15. 

**1995**: Support-Vector Machine
**1999**: Convolution Neural Network
**2006**: Deep Belief Network

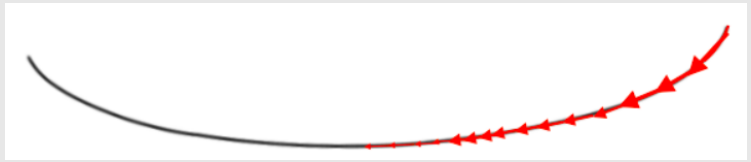
# Chapter 10: Introduction to Artificial Neural Networks

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

import tensorflow as tf

x = tf.placeholder("float", [None, 784])
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))

matm=tf.matmul(x,W)
y = tf.nn.softmax(tf.matmul(x,W) + b)
y_ = tf.placeholder("float", [None,10])

cross_entropy = -tf.reduce_sum(y_*tf.log(y))
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}
```

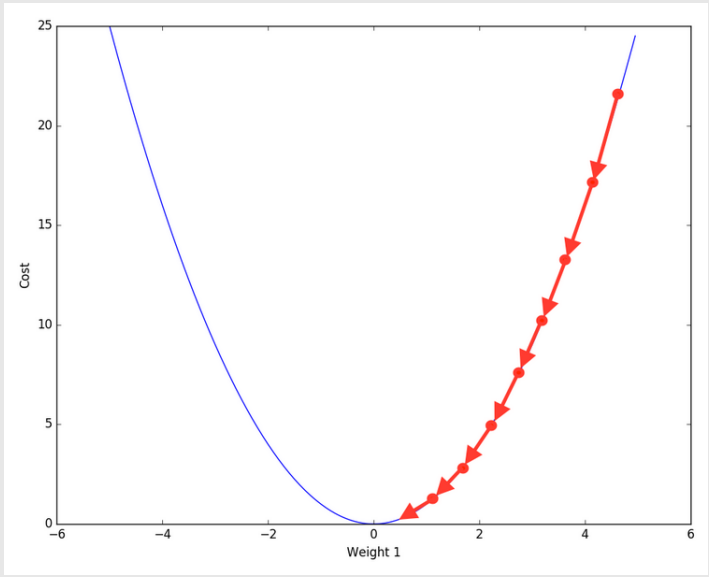


# Training Deep Neural Nets

# Gradient Vanishing



VS

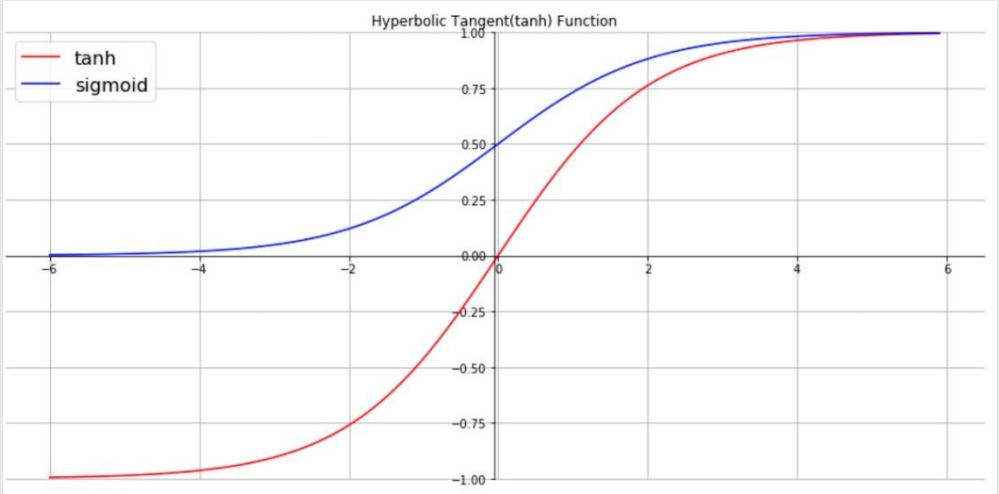



〈https://brunch.co.kr/@chris-song/39〉

- Vanishing gradient problem은 activation function을 선택하는 문제에 의존적으로 발생
- 각 단계의 기울기가 너무 작으면, 반복이 발생할 때 마다 Weight가 충분히 변하지 않기 때문에 더 큰 반복이 수렴 될 때까지 필요
- 설정된 반복 횟수에서 Weight가 최소에 근접하지 않을 수 있고 정말 작은 Gradient로는 훈련 자체가 불가능

# Gradient Vanishing

- Sigmoid, Tanh, 등 요즘 많이 사용하는 activation function은 매우 비선형적인 방식으로 그들의 input을 매우 작은 output range로 squash해서 넣음
  ➔ 이렇게 되어 버린 input space에서는 큰 변화가 있더라도, output에서 작은 변화를 보인다. (Gradient가 작기 때문)
- 이러한 현상은 여러 레이어로 쌓을 때 더욱 악화된다.




〈http://taewan.kim/post/tanh_diff/〉

# Initial Point Setting

## Xiavier(Glorot), 2010

- ReLu 등장 후에 Glorot이 2010년에 제안한 방법으로 vanshing gradient 문제를 해결하기 위해 만들어짐(9)
- **특별한 분야가 아닐 경우 대부분 Glorot 을 사용함**
- **Input 과 output neuron의 수에 기반해서 초기화의 스케일을 정함**
- Initialization weight 가 0일 경우 각 레이어를 통과하면서 signal이 쪼그러 들어 0이 되고 반대로 weight가 너무 크다면 signal은 너무도 커져버린다는 점에 주목한 방법
- Uniform distribution 에서
  - x = sqrt( σ / (in + out))
- Normal distribution 에서
  - sqrt(2. / (in + out))

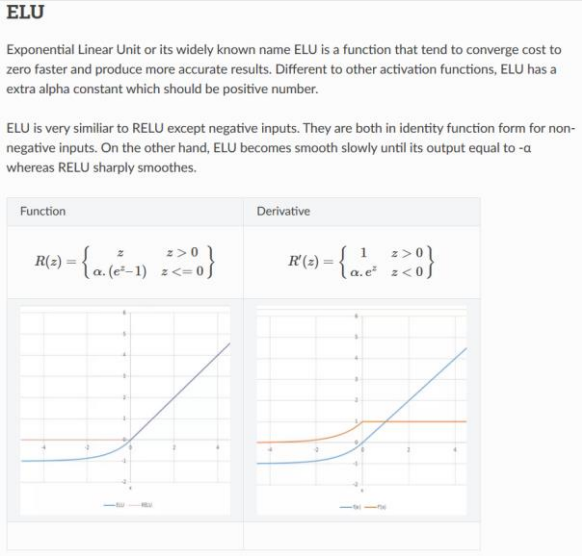
## He, 2015

- Glorot과 유사하지만 **Neuron의 out size**를 고려하진 않음
- **ReLu가 0 이하의 신호를 제거하기 때문에 분산을 두배 주어 분산을 유지한다는 의도**
- Uniform distribution 에서
  - [-limit, limit] 안에서의 uniform distribution
  - Limit = limit is sqrt(σ / fan_in)
- Normal distribution 에서
  - center가 0 이고 stddev = sqrt(2 / fan_in) 인 정규분포

- 논문만 따져보면 사실 이론적으로 둘 다 유사
  (둘 다 초기 파라미터 분포에 대한 좋은 Variance를 찾는 것이 목적)
- 논문에서 Xavier는 Uniform Distribution에,
  He는 Gaussian Distribution에 사용된다 했지만 어디에 사용하여도 무방

〈https://flonelin.wordpress.com/2018/01/28/weight-initalizer-%EC%A2%85%EB%A5%98/〉

# Various ReLU



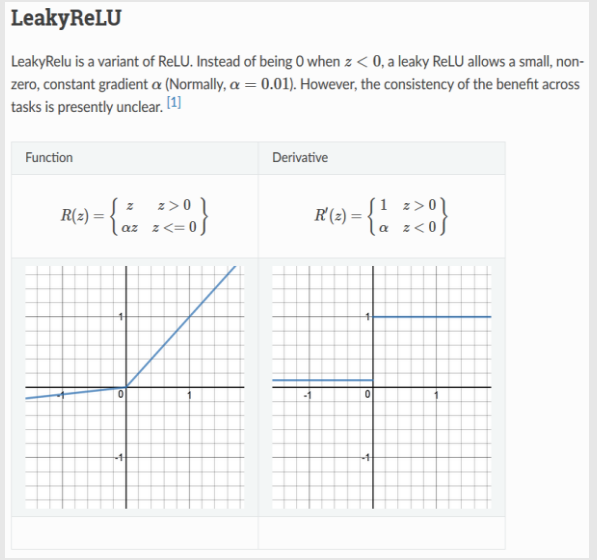
**ELU**

Exponential Linear Unit or its widely known name ELU is a function that tend to converge cost to zero faster and produce more accurate results. Different to other activation functions, ELU has a extra alpha constant which should be positive number.

ELU is very similiar to RELU except negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.

| Function | Derivative |
|---|---|
| $R(z) = \begin{cases} z & z > 0 \\ \alpha.(e^z - 1) & z <= 0 \end{cases}$ | $R'(z) = \begin{cases} 1 & z > 0 \\ \alpha.e^z & z < 0 \end{cases}$ |

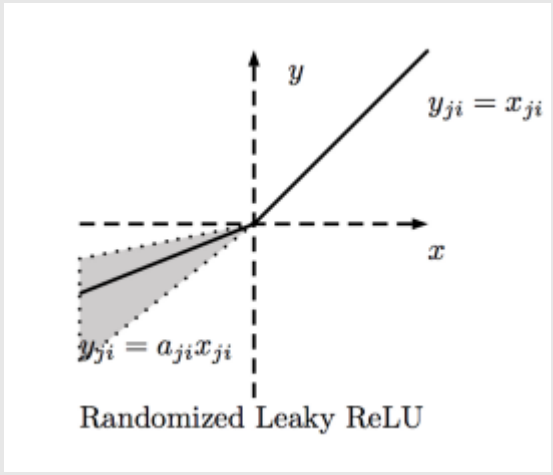
〈https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html〉



**LeakyReLU**

LeakyRelu is a variant of ReLU. Instead of being 0 when $z < 0$, a leaky ReLU allows a small, non-zero, constant gradient $\alpha$ (Normally, $\alpha = 0.01$). However, the consistency of the benefit across tasks is presently unclear. [1]

| Function | Derivative |
|---|---|
| $R(z) = \begin{cases} z & z > 0 \\ \alpha z & z <= 0 \end{cases}$ | $R'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases}$ |




〈https://www.gabormelli.com/RKB/Randomized_Leaky_Rectified_Linear_Activation_Function〉

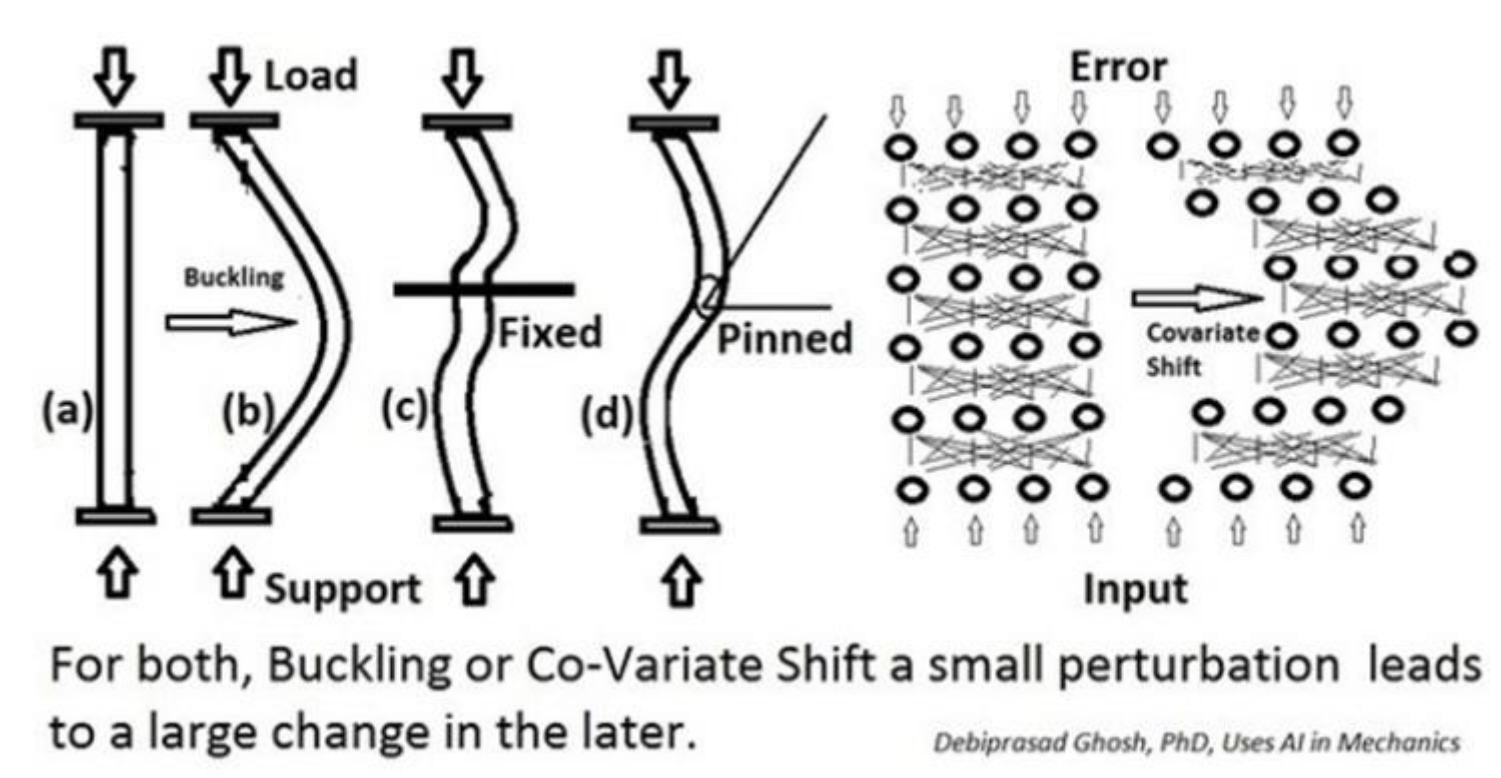
- Strong alternative to ReLU
- ELU can produce negative outputs
- Very Slow

- Attempt to fix the ‘dying ReLU’
- As it possess linearity, it can’t be used for the complex computation

- Same as ‘Leaky ReLU’

# Internal Covariate Shift

Network의 각 층이나 Activation 마다 input의 distribution이 달라지는 현상을 의미




〈http://blog.naver.com/PostView.nhn?blogId=laonple&logNo=220808903260〉

# Internal Covariate Shift


〈http://www.asiae.co.kr/news/view.htm?idxno=2016051413525091379〉

# Whitening

$$\hat{X} = Cov(X)^{-1/2}X, Cov(X) = E[(X - E[X])(X - E[X])^{\top}].$$

그러나 이런 naive한 approach에서는 크게 두 가지 문제점들이 발생하게 된다.

1. multi variate normal distribution으로 normalize를 하려면 inverse의 square root를 계산해야 하기 때문에 필요한 계산량이 많다.
2. mean과 variance 세팅은 어떻게 할 것인가? 전체 데이터를 기준으로 mean/variance를 training마다 계산하면 계산량이 많이 필요하다.

〈http://sanghyukchun.github.io/88/〉

## 수리적인 내용은 아래의 논문 참조

Efficient BackProp

Yann LeCun[1], Leon Bottou[1], Genevieve B. Orr[2], and Klaus-Robert Müller[3]

[1] Image Processing Research Department AT& T Labs - Research, 100 Schulz Drive, Red Bank, NJ 07701-7033, USA
[2] Willamette University, 900 State Street, Salem, OR 97301, USA
[3] GMD FIRST, Rudower Chaussee 5, 12489 Berlin, Germany
{yann,leonb}@research.att.com, gorr@willamette.edu, klaus@first.gmd.de
originally published in
Orr, G. and Müller, K. "Neural Networks: tricks of the trade",
Springer, 1998.

**Abstract.** The convergence of back-propagation learning is analyzed so as to explain common phenomenon observed by practitioners. Many undesirable behaviors of backprop can be avoided with tricks that are rarely exposed in serious technical publications. This paper gives some of those tricks, and offers explanations of why they work.
Many authors have suggested that second-order optimization methods are advantageous for neural net training. It is shown that most "classical" second-order methods are impractical for large neural networks. A few methods are proposed that do not have these limitations.

**1 Introduction**

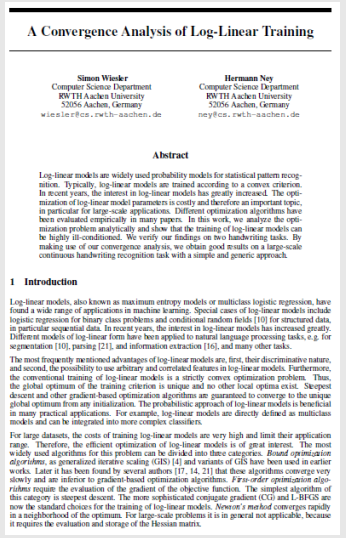
Backpropagation is a very popular neural network learning algorithm because it is conceptually simple, computationally efficient, and because it often works. However, getting it to work well, and sometimes to work at all, can seem more of an art than a science. Designing and training a network using backprop requires making many seemingly arbitrary choices such as the number and types of nodes, layers, learning rates, training and test sets, and so forth. These choices can be critical, yet there is no foolproof recipe for deciding them because they are largely problem and data dependent. However, there are heuristics and some underlying theory that can help guide a practitioner to make better choices.

In the first section below we introduce standard backpropagation and discuss a number of simple heuristics or tricks for improving its performance. We next discuss issues of convergence. We then describe a few "classical" second-order non-linear optimization techniques and show that their application to neural network training is very limited, despite many claims to the contrary in the literature. Finally, we present a few second-order methods that do accelerate learning in certain cases.

**2 Learning and Generalization**

There are several approaches to automatic machine learning, but much of the successful approaches can be categorized as *gradient based learning methods*. The

〈LeCun et al, 1998〉



A Convergence Analysis of Log-Linear Training

**Simon Wiesler**
Computer Science Department
RWTH Aachen University
52056 Aachen, Germany
wiesler@cs.rwth-aachen.de

**Hermann Ney**
Computer Science Department
RWTH Aachen University
52056 Aachen, Germany
ney@cs.rwth-aachen.de

**Abstract**

Log-linear models are widely used probability models for statistical pattern recognition. Typically, log-linear models are trained according to a convex criterion. In recent years, the interest in log-linear models has greatly increased. The optimization of log-linear model parameters is costly and therefore an important topic, in particular for large-scale applications. Different optimization algorithms have been evaluated empirically in many papers. In this work, we analyze the optimization problem analytically and show that the training of log-linear models can be highly ill-conditioned. We verify our findings on two handwriting tasks. By making use of our convergence analysis, we obtain good results on a large-scale continuous handwriting recognition task with a simple and generic approach.

**1 Introduction**

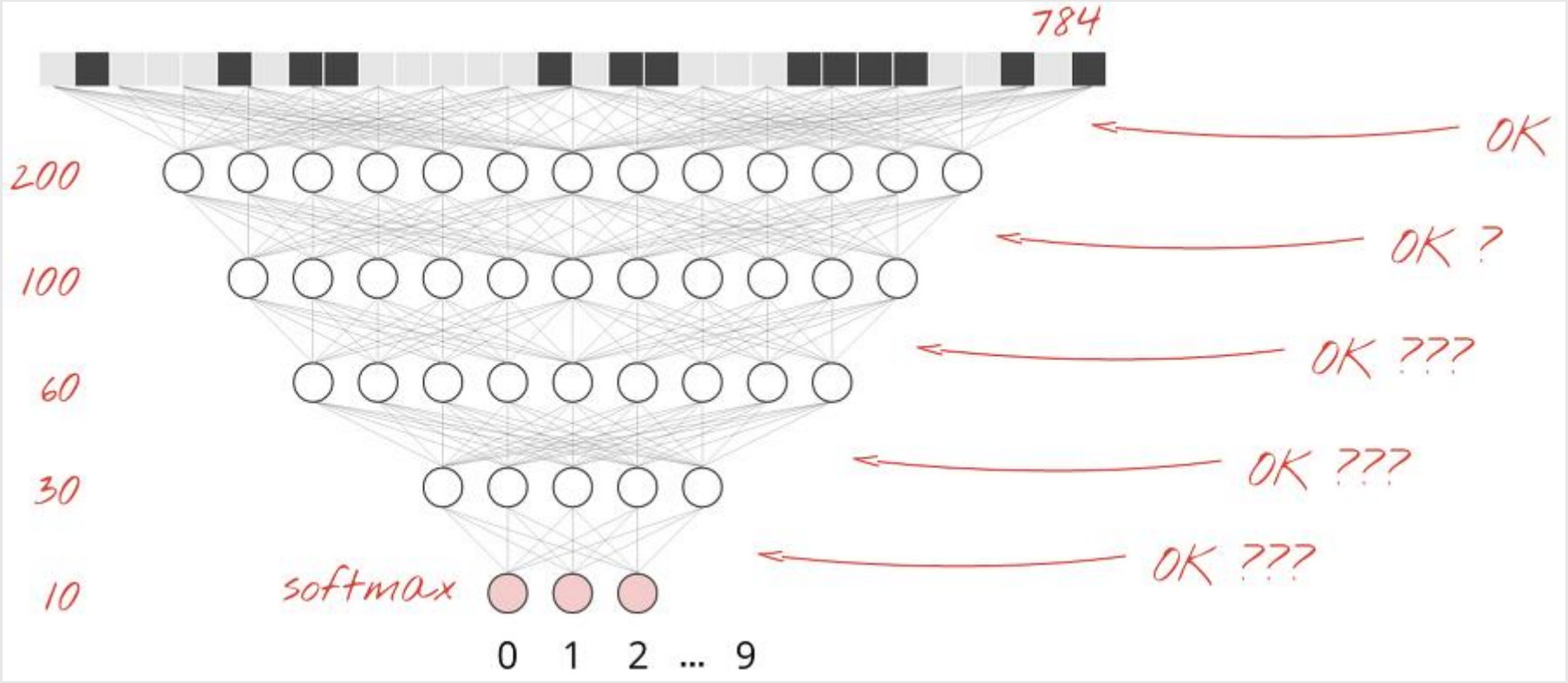
Log-linear models, also known as maximum entropy models or multiclass logistic regression, have found a wide range of applications in machine learning. Special cases of log-linear models include logistic regression for binary class problems and conditional random fields [10] for structured data, in particular sequential data. In recent years, the interest in log-linear models has increased greatly. Different models of log-linear form have been applied to natural language processing tasks, e.g. for segmentation [10], parsing [21], and information extraction [16], and many other tasks.

The most frequently mentioned advantages of log-linear models are, first, their discriminative nature, and second, the possibility to use arbitrary and correlated features in log-linear models. Furthermore, the conventional training of log-linear models is a strictly convex optimization problem. Thus, the global optimum of the training criterion is unique and no other local optima exist. Steepest descent and other gradient-based optimization algorithms are guaranteed to converge to the unique global optimum from any initialization. The probabilistic approach of log-linear models is beneficial in many practical applications. For example, log-linear models are directly defined as multiclass models and can be integrated into more complex classifiers.

For large datasets, the costs of training log-linear models are very high and limit their application range. Therefore, the efficient optimization of log-linear models is of great interest. The most widely used algorithms for this problem can be divided into three categories. *Bound optimization algorithms*, as generalized iterative scaling (GIS) [4] and variants of GIS have been used in earlier works. Later it has been found by several authors [17, 14, 21] that these algorithms converge very slowly and are inferior to gradient-based optimization algorithms. *First-order optimization algorithms* require the evaluation of the gradient of the objective function. The simplest algorithm of this category is steepest descent. The more sophisticated conjugate gradient (CG) and L-BFGS are now the standard choices for the training of log-linear models. *Newton's method* converges rapidly in a neighborhood of the optimum. For large-scale problems it is in general not applicable, because it requires the evaluation and storage of the Hessian matrix.

〈Wiesler & Ney et al, 2011〉

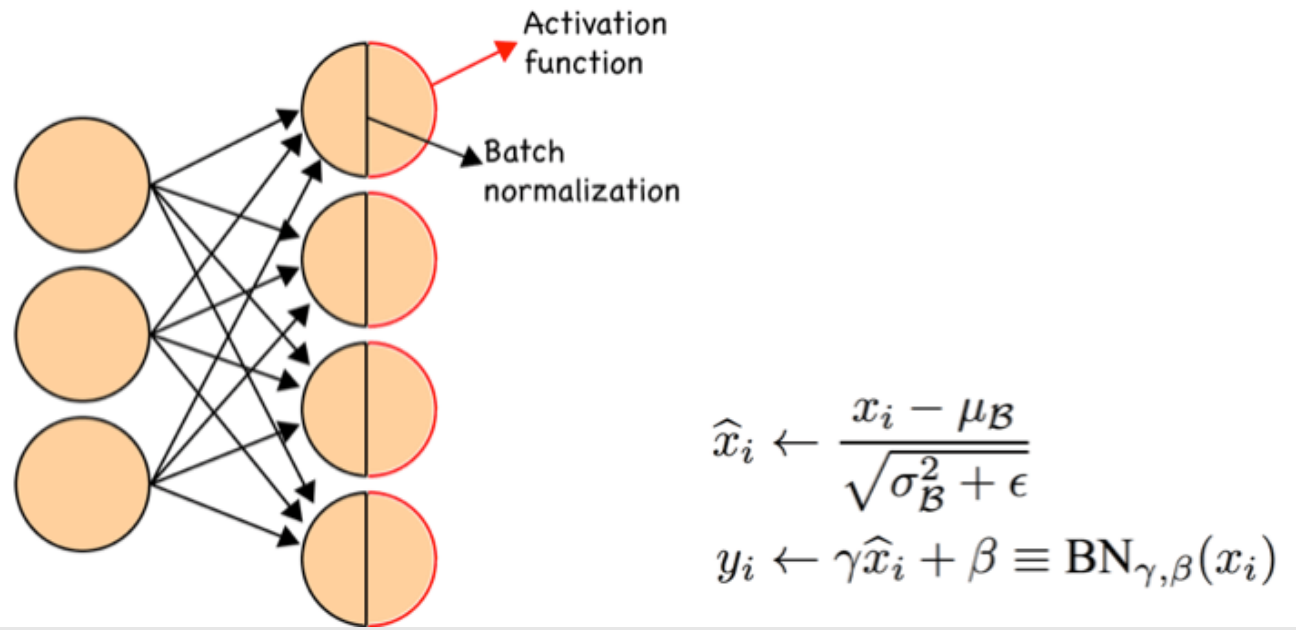
# Whitening




〈http://eyeofneedle.tistory.com/25〉

# Batch Normalization

1. 각 차원들이 서로 independent하다고 가정하고 각 차원 별로 따로 estimate를 하고 그 대신 표현형을 더 풍성하게 해 줄 linear transform도 함께 learning한다
2. 전체 데이터에 대해 mean/variance를 계산하는 대신 지금 계산하고 있는 batch에 대해서만 mean/variance를 구한 다음 inference를 할 때에만 Real mean/variance를 계산한다




$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

<http://dongyukang.com/%EB%B0%B0%EC%B9%98-%EC%A0%95%EA%B7%9C%ED%99%94-%EB%85%BC%EB%AC%B8%EC%9D%84-%EC%9D%BD%EA%B3%A0/>

# Batch Normalization

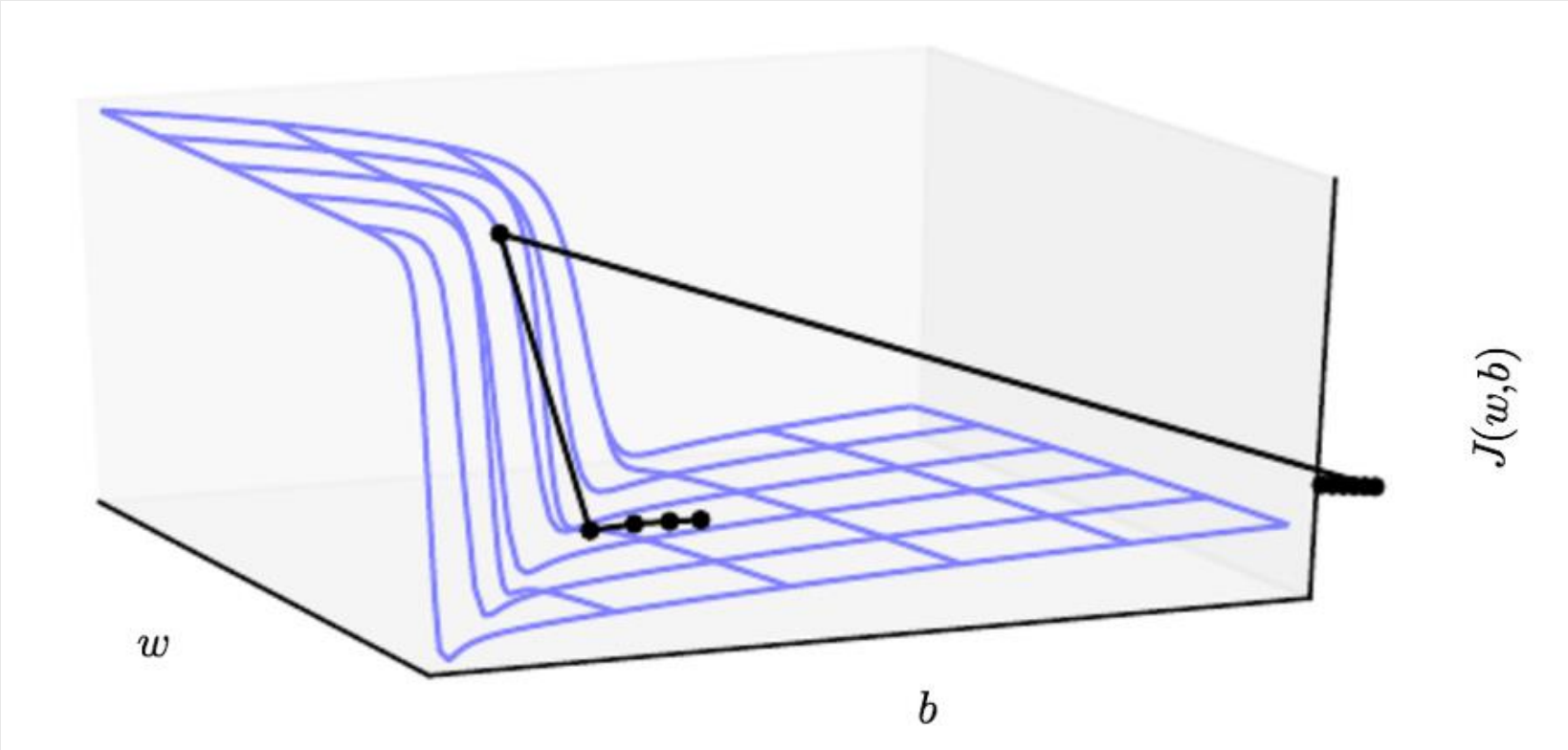

<https://en.wikipedia.org/wiki/Moving_average#/media/File:Moving_Average_Types_comparison_-_Simple_and_Exponential.png>

# Batch Normalization

매우 빠른 연산속도

자체적인 Regularization

# Gradient Clipping




<http://dhhwang89.tistory.com/90>

# Gradient Clipping

**Algorithm 1** Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
**if** $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
**end if**

이러한 문제에 대처하는 한가지 방법은 learning rate를 매우 작게 설정하는 것입니다. 이 솔루션은 학습 속도를 매우 느리게 만들고, 잘못하면 local minima에 빠지게 만듭니다. 위의 솔루션보다 더 좋은 솔루션은 gradient clipping :: norm-clipping을 사용하는 것입니다. gradient clipping의 개념은 gradient의 최대 갯수를 제한하고, gradient가 최대치를 넘게되면 gradient의 크기를 재조정해서 gradient의 크기를 조정하는 것입니다. 이러한 gradient clipping은 최적화 알고리즘이 가야하는 방향은 그대로 유지하면서 업데이트되야하는 step의 크기(learning rate)를 자동으로 조정합니다.

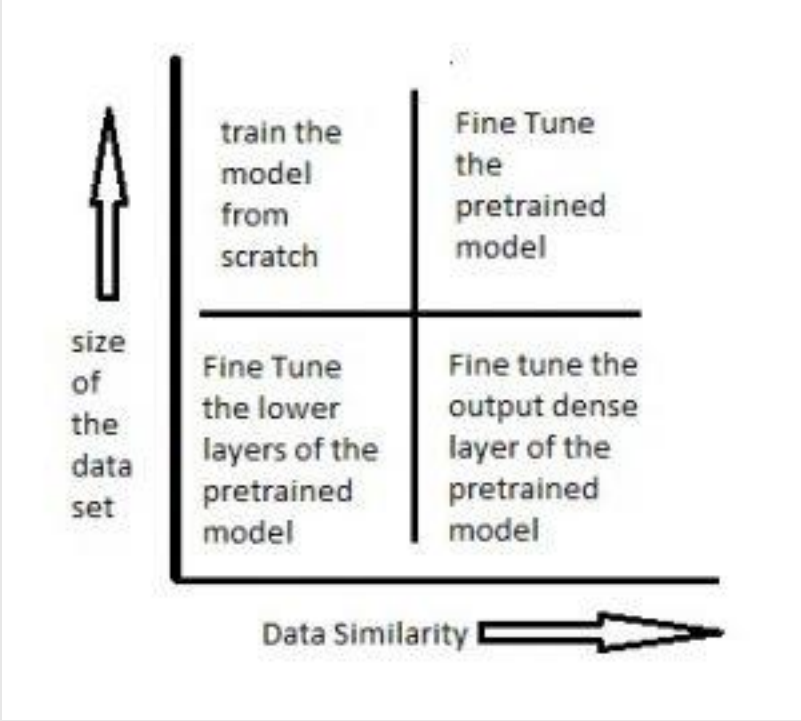
<http://dhhwang89.tistory.com/90>

# Transfer Learning

**기존의 만들어진 모델을 사용하여 새로운 모델을 만들 시 학습을 빠르게 하며, 예측을 더 높이는 방법**

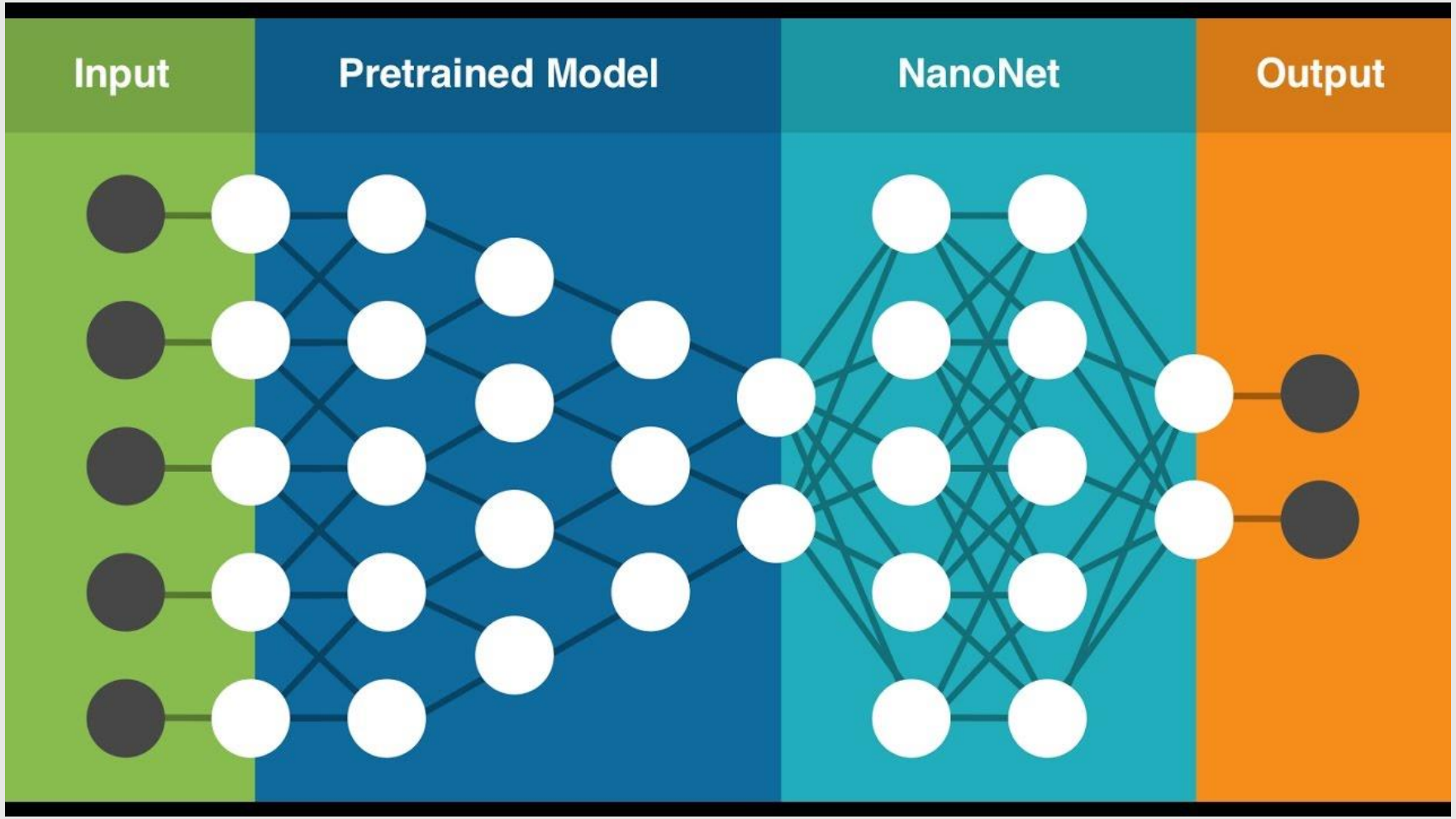
1. 실질적으로 처음부터 학습시키는 일은 많지 않다.
   대부분의 문제는 이미 학습된 모델을 사용해서 문제를 해결할 수 있다.
2. 복잡한 모델일수록 학습시키기 어렵다.
   어떤 모델은 시간이 오래 걸릴 수 있으며, 비싼 GPU 여러 대를 사용하기도 한다.
3. Layers의 개수, activation, hyper parameters등등 고려해야 할 사항들이 많으며, 실질적으로 처음부터 학습시키려면 많은 시도가 필요합니다.
4. 이미 구조가 훌륭하다고 입증된 구조를 사용하면 되는데 굳이 여러 구조를 시험해볼 필요가 없다.

# Transfer Learning



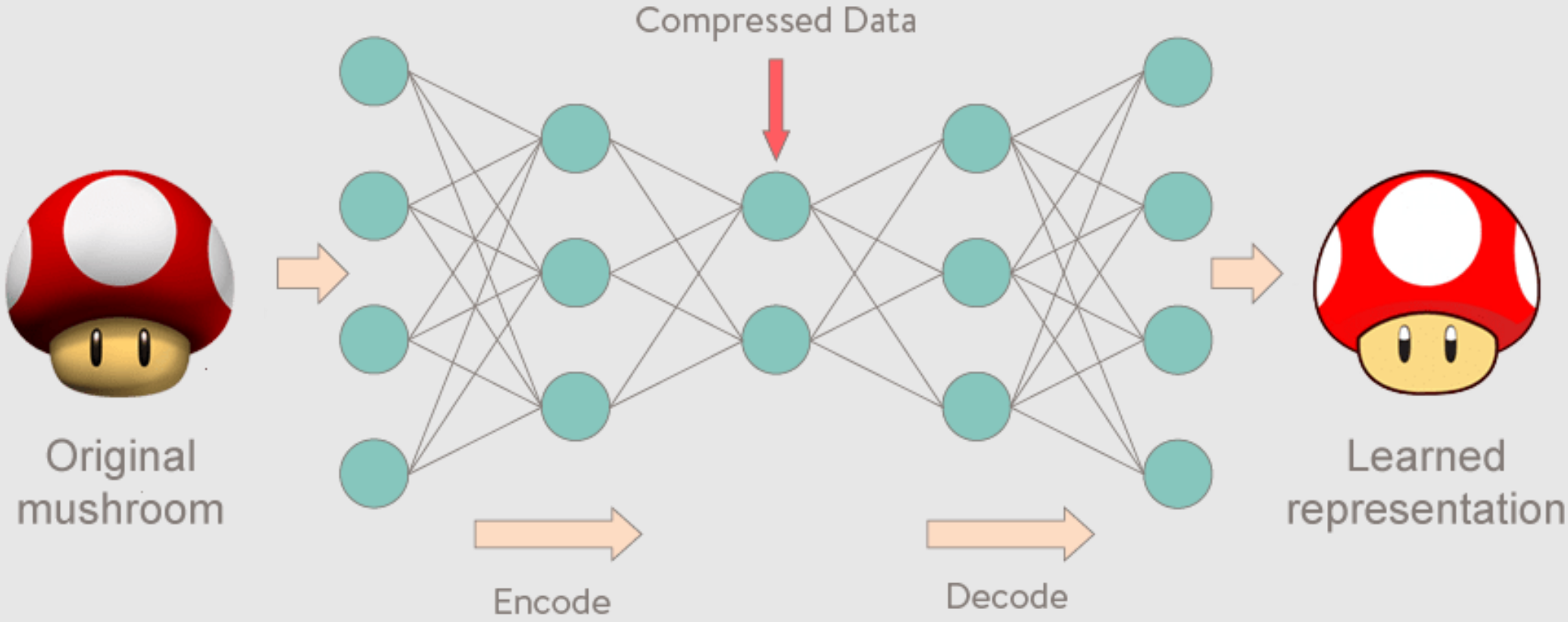

〈https://www.analyticsvidhya.com/blog/2017/06/transfer-learning-the-art-of-fine-tuning-a-pre-trained-model/〉

# Transfer Learning




〈https://www.youtube.com/watch?v=8ZWMQcd7KSo〉

# Autoencoder



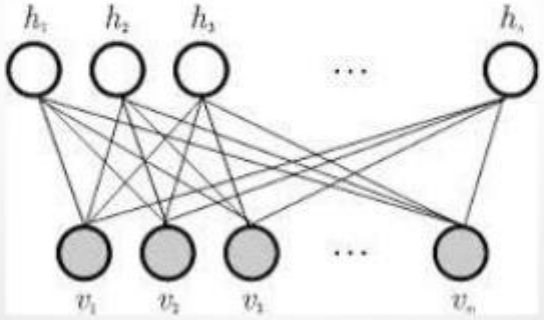

〈http://hugrypiggykim.com/2018/01/16/fds-fraud-detection-system-with-autoencoder/〉

# Restricted Boltzmann Machines

## Restricted Boltzmann Machines

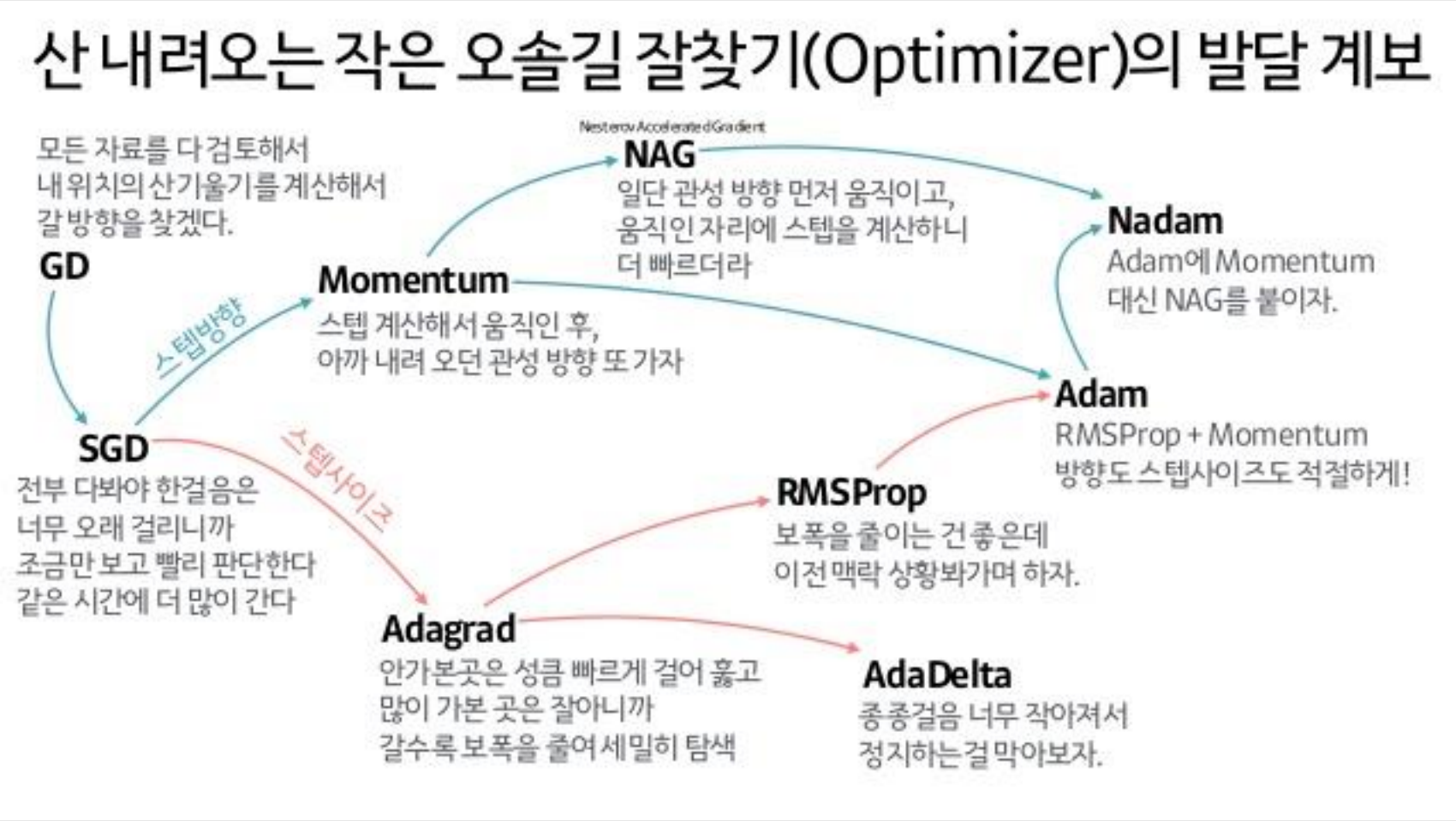
Definition

- 기존의 BM에서 유닛들간의 연결에 특별한 제한을 걸어 학습 시간을 크게 줄여 실질적인 사용이 가능하도록 변형한 신경망
- visible layer 1개, hidden layer 1개로 구성된 완전 이분 그래프 모델로 visible-visible, hidden-hidden 유닛들간의 연결이 없음
  - BM의 식에서 U와 V가 0벡터




〈https://www.slideshare.net/blaswan/energy-based-models-and-boltzmann-machines〉

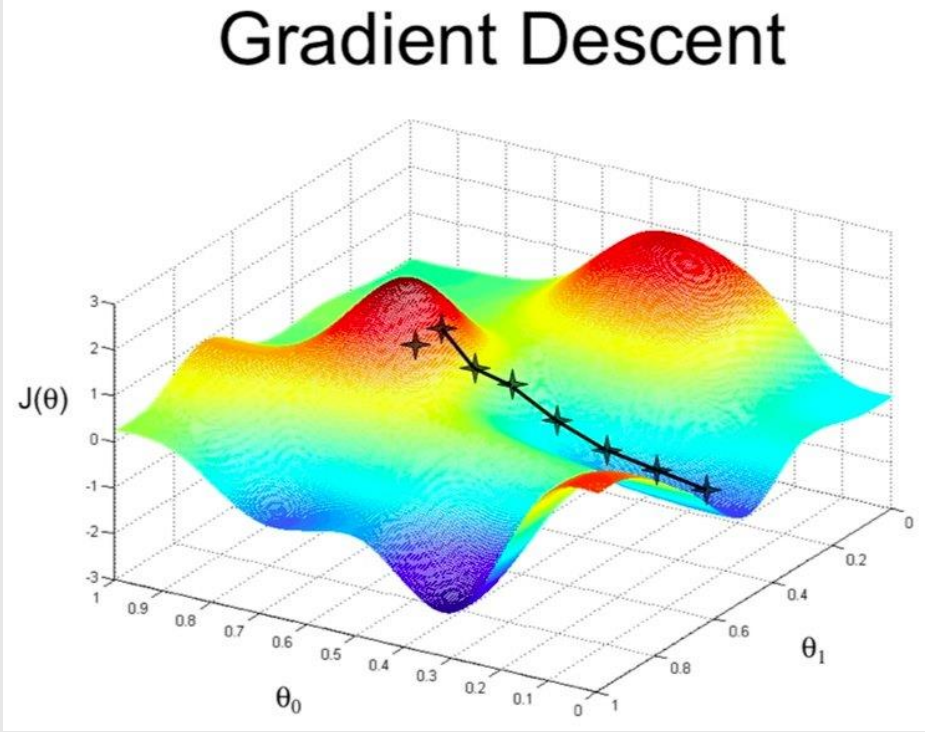
# Optimizer




〈https://www.slideshare.net/yongho/ss-79607172〉

# Gradient Descent

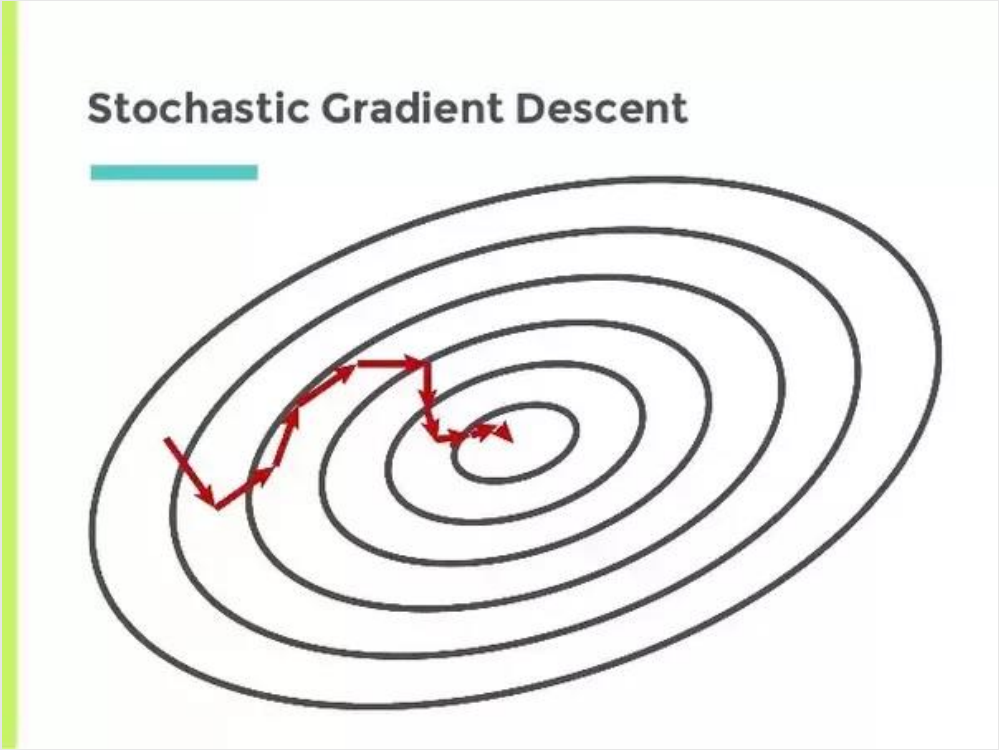
기울기를 따라서 내려가자




〈https://www.ahmednasr.at/machine-learning-02/〉

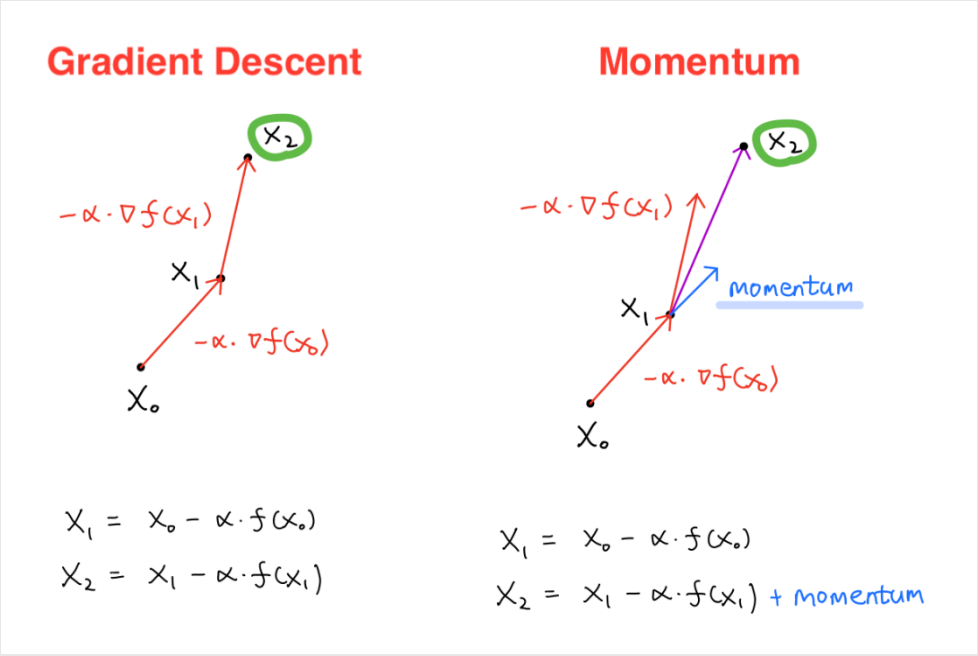
# Gradient Descent

데이터 전체를 써서 한걸음은 느리니까
데이터 일부만 써서 빠르게 내려가자



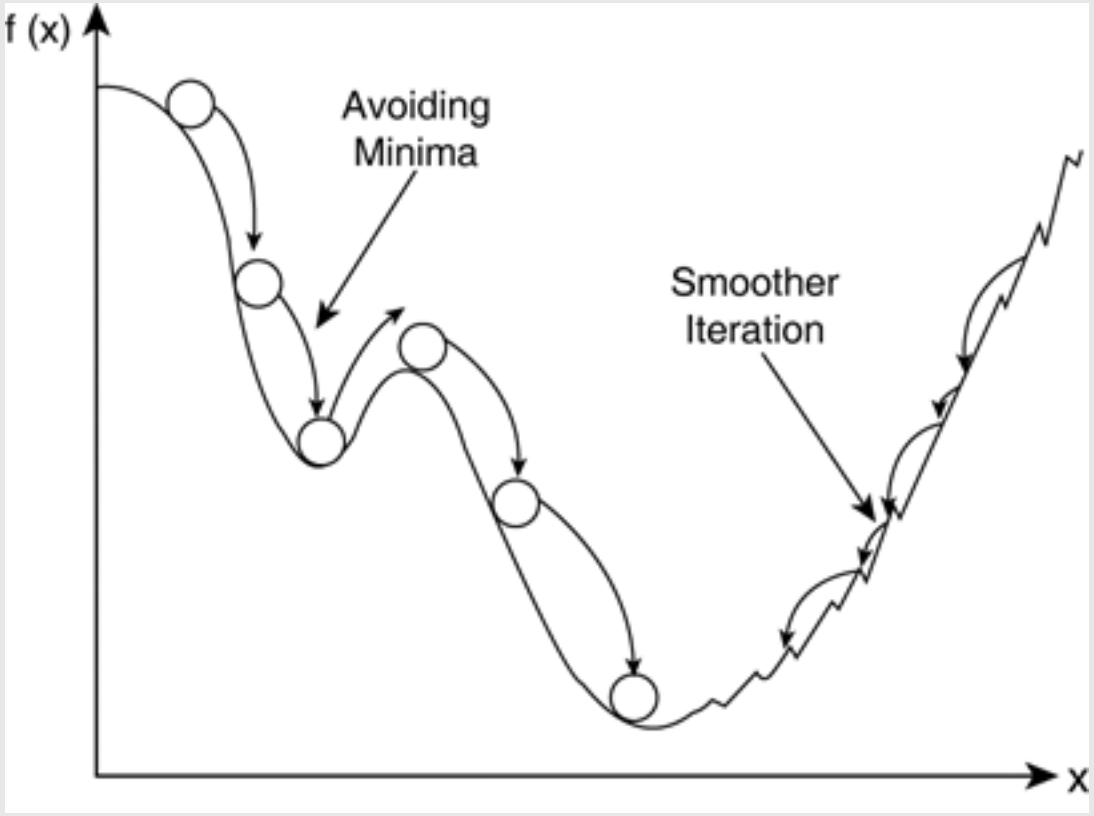
# Momentum Optimizer

실제로 이동하는 것 처럼
관성을 줘서 이동해보자




〈https://icim.nims.re.kr/post/easyMath/428〉

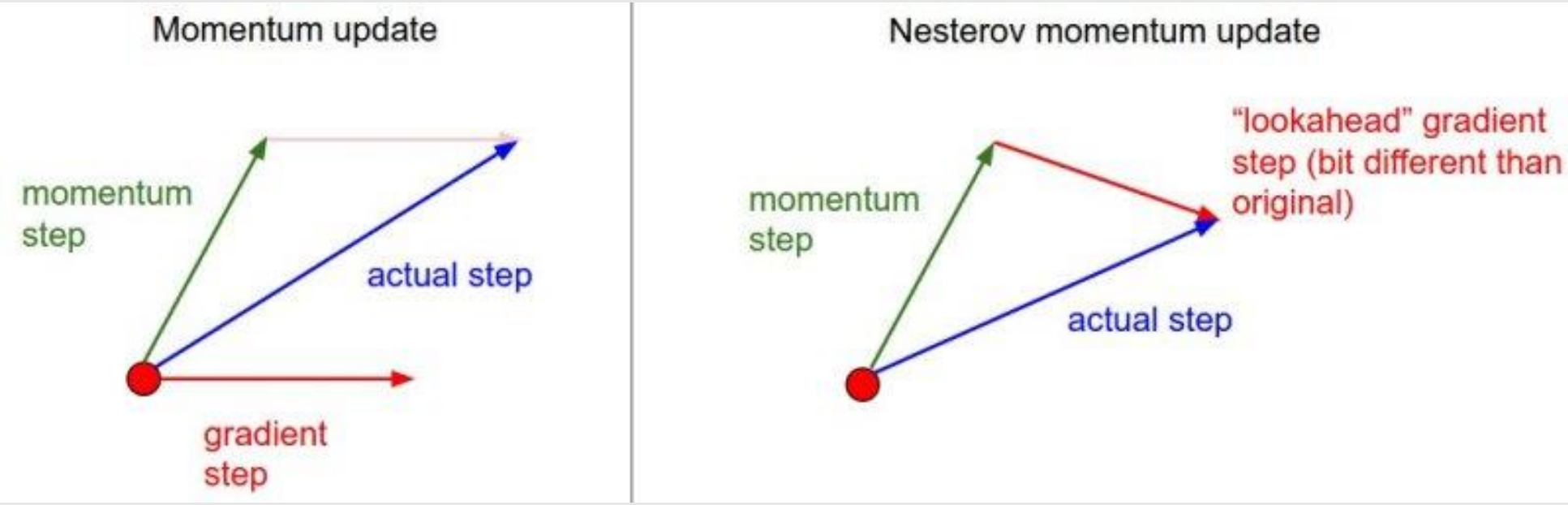
# Momentum Optimizer




<http://shuuki4.github.io/deep%20learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html>

# NAG Optimizer

Momentum쪽으로 이동하고 나서 Gradient방향으로 가니까 속도와 성능이 더 좋네?

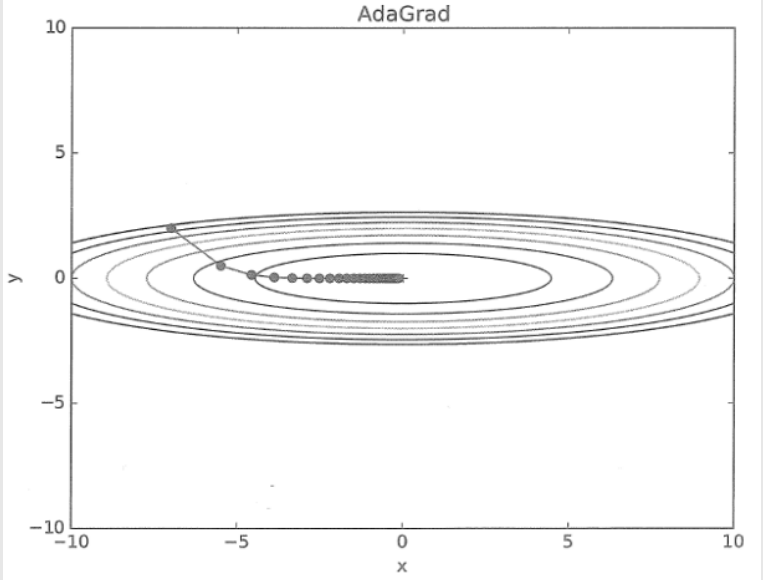



<http://shuuki4.github.io/deep%20learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html>

# Adagrad

지금까지 많이 변화한 변수는 step size가 작게,
지금까지 조금 변화한 변수는 step size가 크게.

그림 6-6 AdaGrad에 의한 최적화 갱신 경로




<http://ling.snu.ac.kr/class/cl_under1801/DL06-NNLearningConcepts.pdf>

# RMSProp

Adagrad는 너무 값의 비약이 너무 심하니까
적당한 Step size를 유지해보자

RMSProp은 딥러닝의 대가 제프리 힌톤이 제안한 방법으로서, Adagrad의 단점을 해결하기 위한 방법이다. Adagrad의 식에서 gradient의 제곱값을 더해나가면서 구한 $G_t$ 부분을 합이 아니라 지수평균으로 바꾸어서 대체한 방법이다. 이렇게 대체를 할 경우 Adagrad처럼 $G_t$가 무한정 커지지는 않으면서 최근 변화량의 변수간 상대적인 크기 차이는 유지할 수 있다. 식으로 나타내면 다음과 같다.

$$G = \gamma G + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

$$\theta = \theta - \frac{\eta}{\sqrt{G+\epsilon}} \cdot \nabla_\theta J(\theta_t)$$

<http://shuuki4.github.io/deep%20learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html>

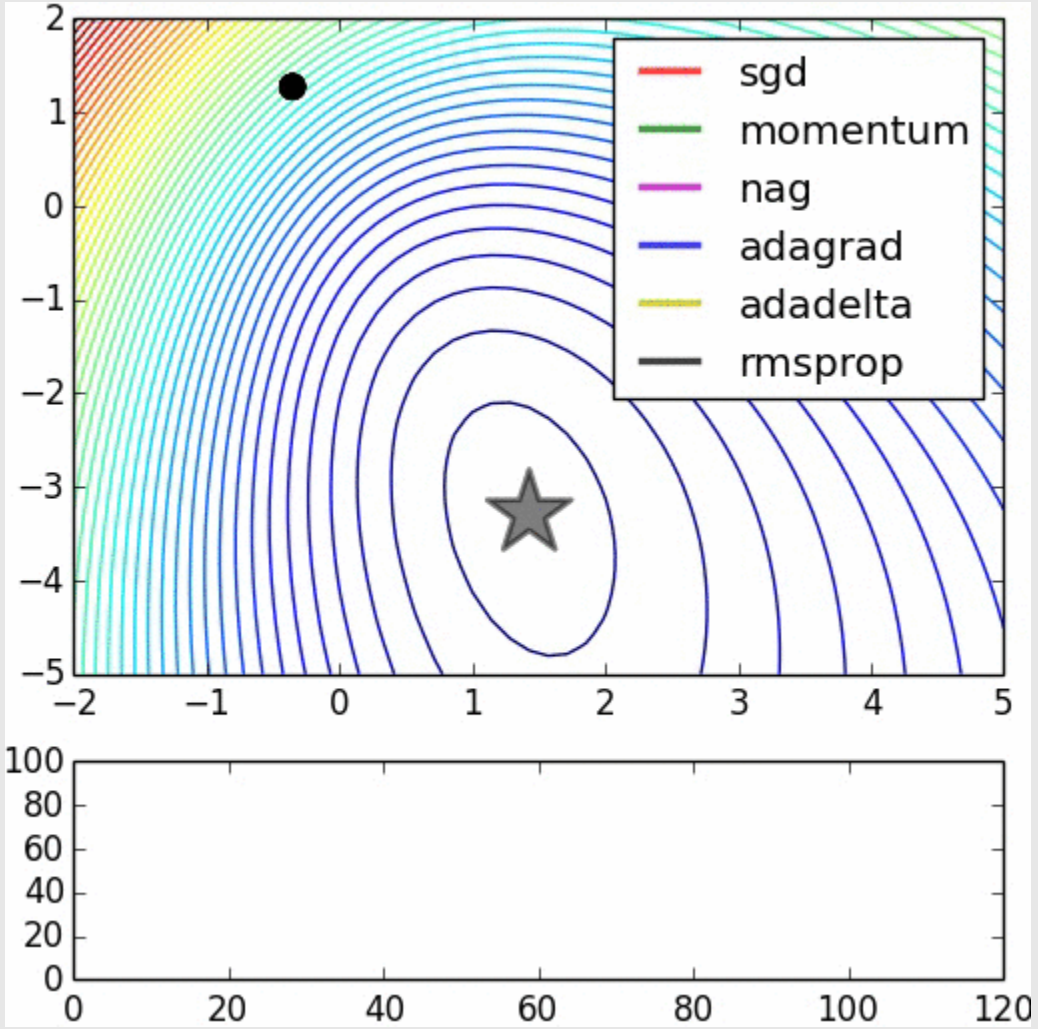
# Adam Optimizer

RMSProp도 좋고, Momentum도 좋으니까
둘 다 섞자.

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. $g_t^2$ indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With $\beta_1^t$ and $\beta_2^t$ we denote $\beta_1$ and $\beta_2$ to the power $t$.

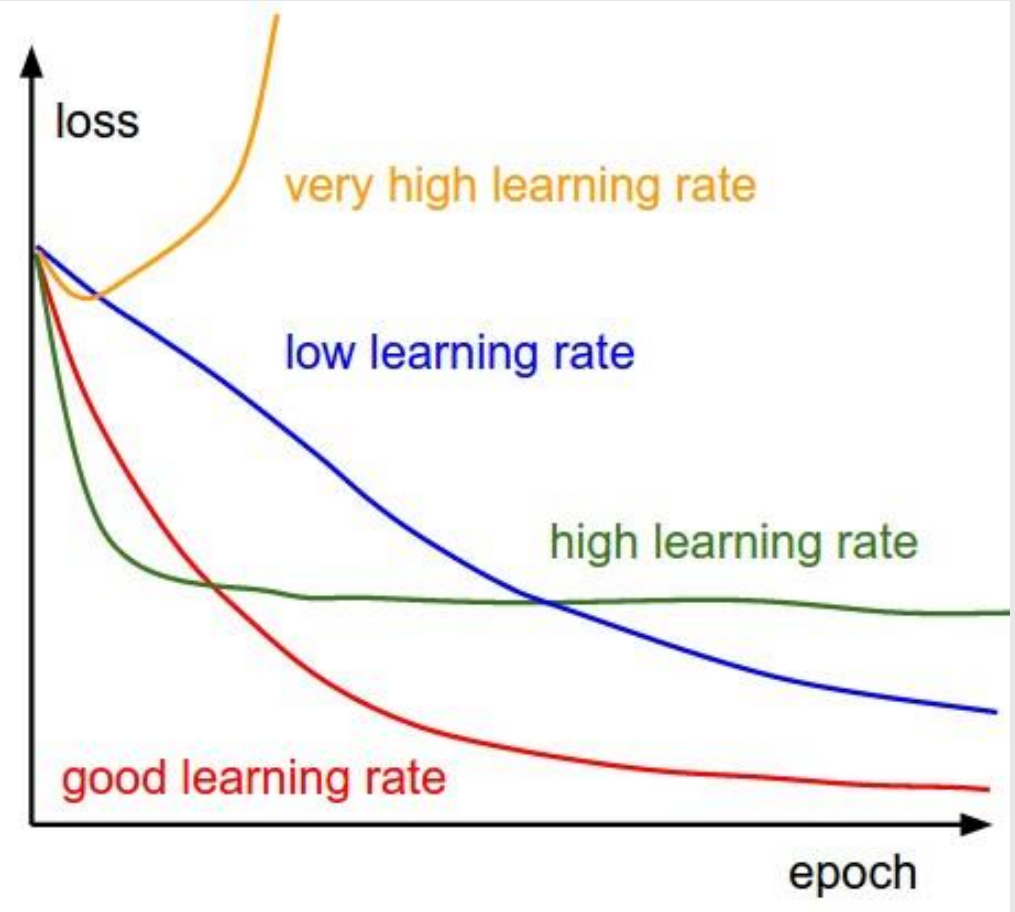
**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
**while** $\theta_t$ not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
**end while**
**return** $\theta_t$ (Resulting parameters)

# All Optimizer




<http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

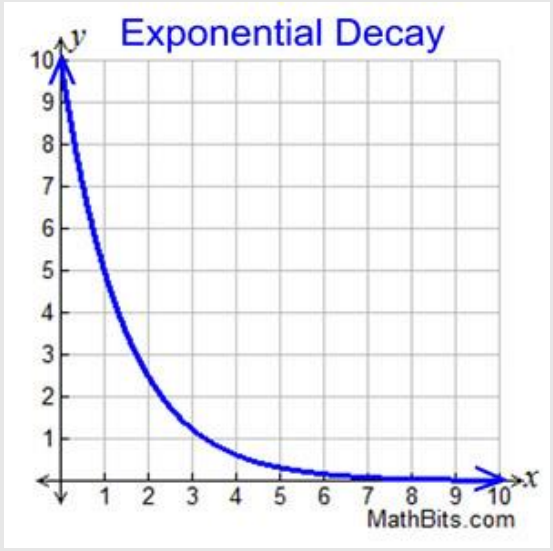
# Learning Scheduling




〈https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10〉

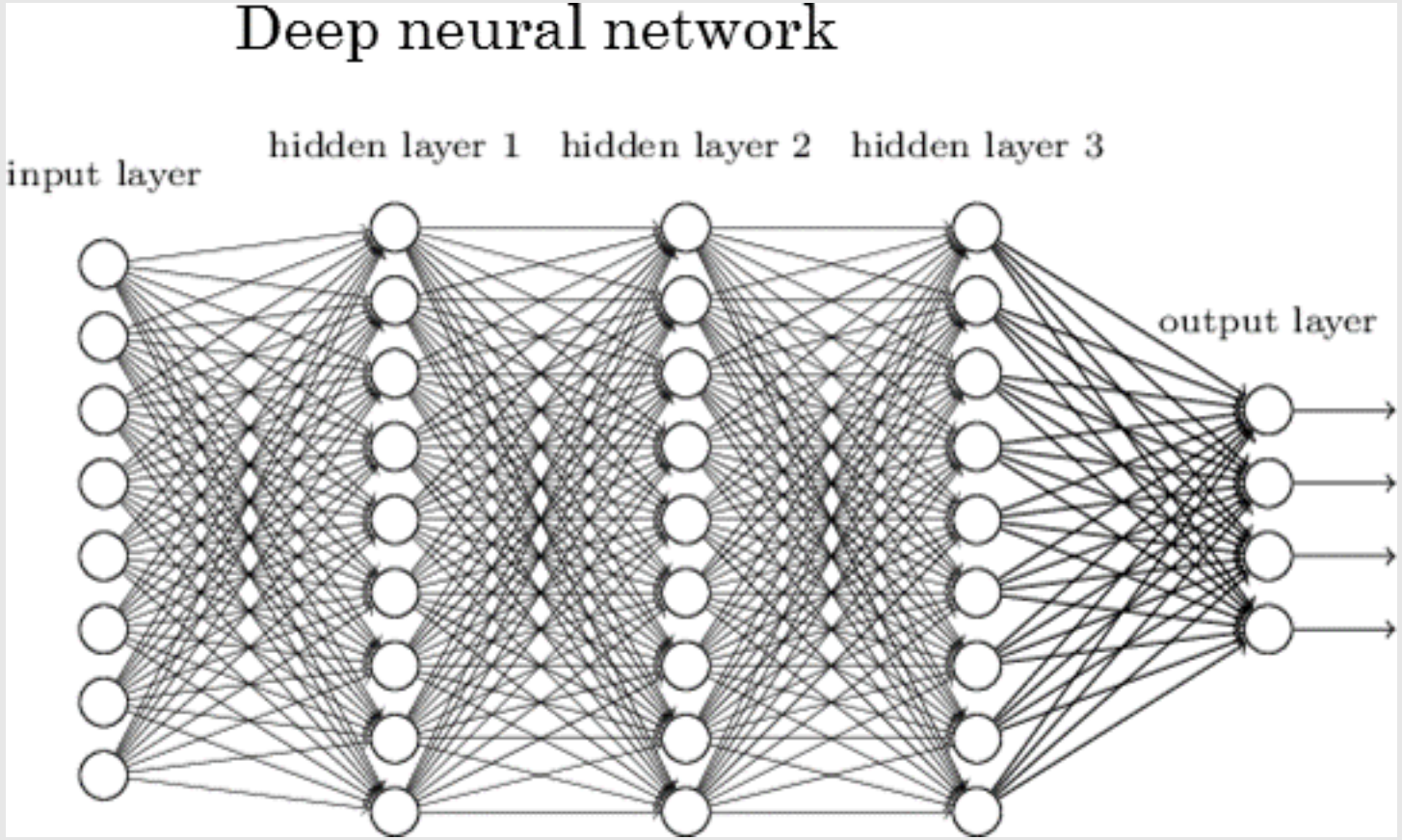
# Learning Scheduling

- 미리 정의된 개별적인 고정 학습률
- 성능 기반 스케줄링
- 지수 기반 스케줄링 (exponential decay)
- 거듭제곱 기반 스케줄링 (exponentiation decay)




〈https://stackoverflow.com/questions/48210579/exponential-decrease〉

# Regularization




〈https://datawarrior.wordpress.com/2017/10/31/interpretability-of-neural-networks/〉

# Regularization

> Everything should be made simple as possible, but not simpler
> – Albert Einstein

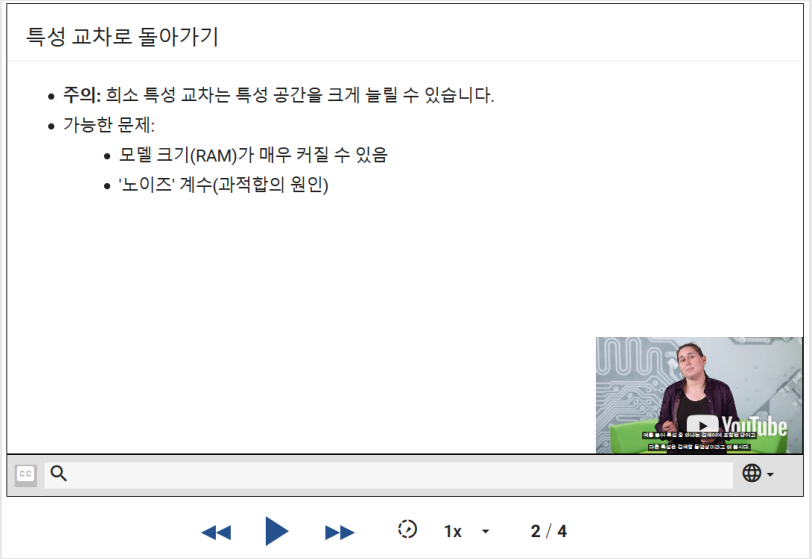
우리가 했던 것은 'simpler'였고, 모두가 할 수 있는 것입니다. 하지만 이번 시간에는 'making it simple'에 대해 알아봅시다. 이것이 우리가 정규화를 통해 코드를 최적화하려고 하는 이유입니다.

〈https://brunch.co.kr/@itschloe1/11〉

# Regularization

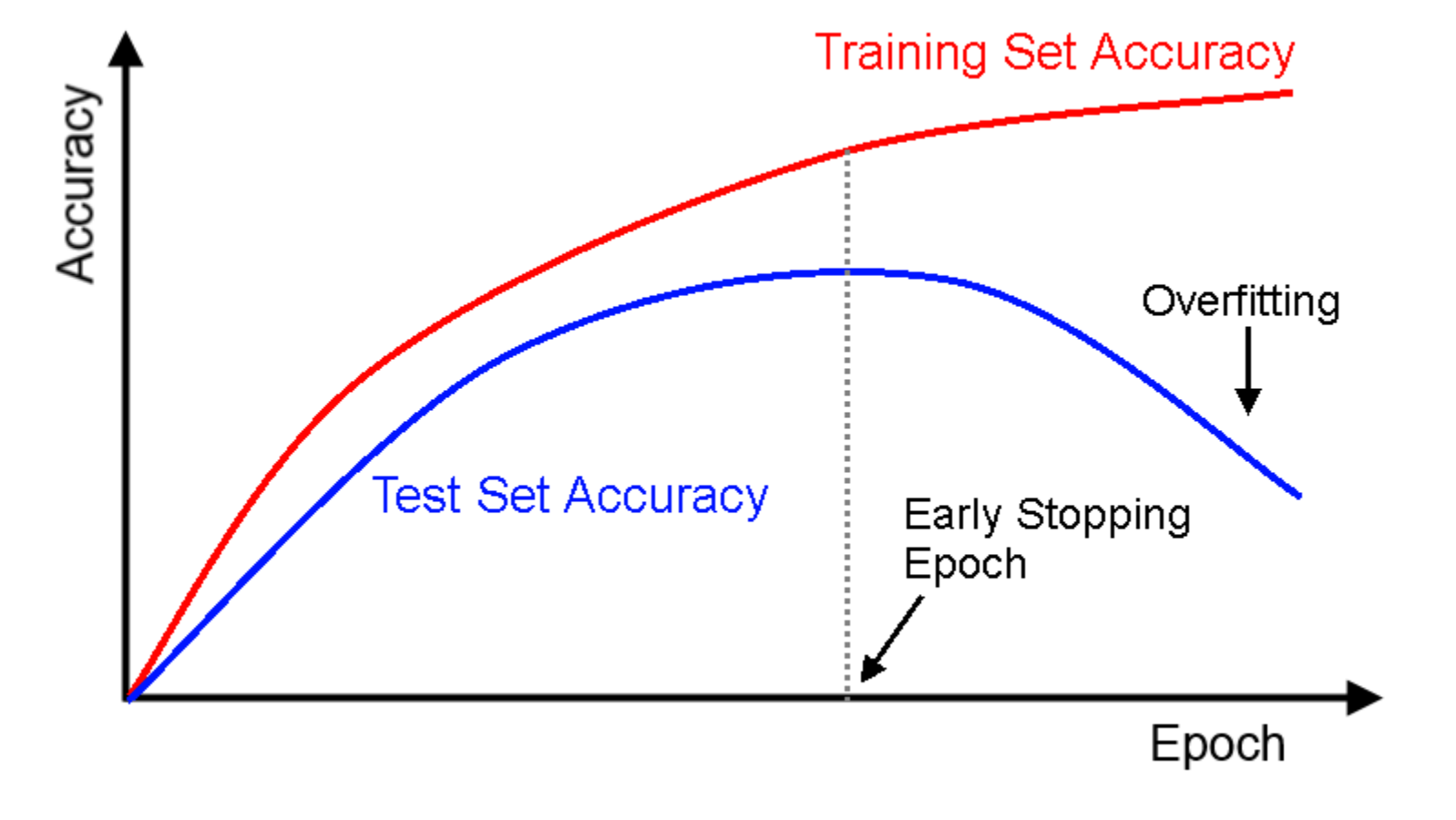
- 이전에는 과적합에 초점을 맞춤
  (지금도 과적합 방지가 메인 이긴 하다.)
- 모형이 복잡하면 RAM사용량이 증가
- 노이즈에 민감하게 반응
- 작은 Signal을 찾는 것이 딥러닝이기 때문에 Robust해야함

➔ 이러한 문제 해결을 위해
Regularization Term을 사용




https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/video-lecture?hl=ko

# Early Stopping




〈https://deeplearning4j.org/docs/latest/deeplearning4j-nn-early-stopping〉

# Norm

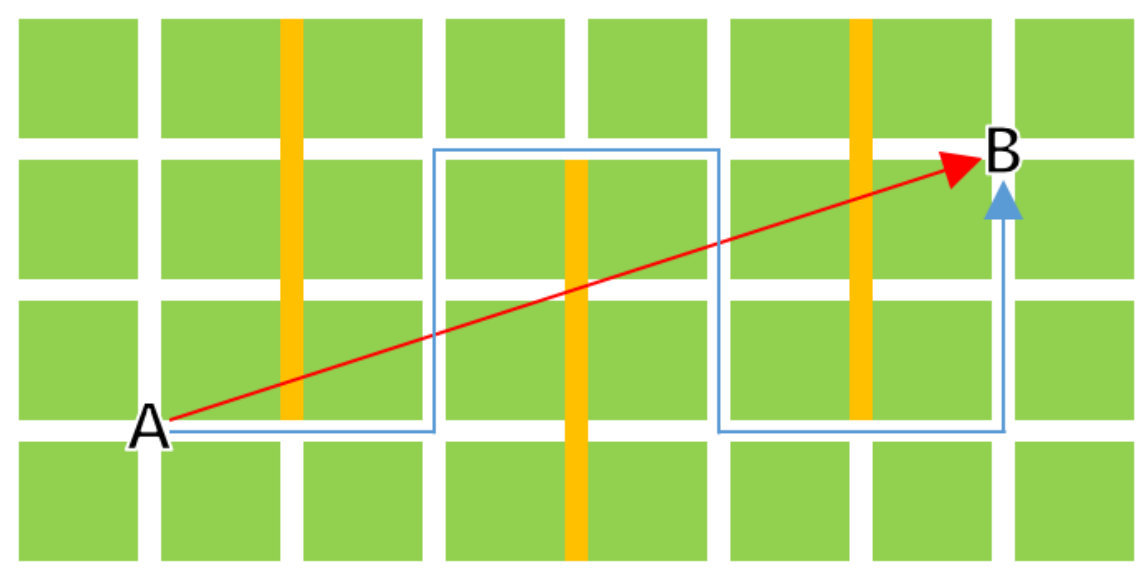
$V$를 $\mathbb{F}$ 상에서의 벡터공간이라고 하자. $||\cdot|| : V \to \mathbb{F}$ 가 $\mathbf{u}, \mathbf{v} \in V$와 $k \in \mathbb{F}$ 에 대해서 다음 세 조건을 만족시키면 $||\cdot||$ 을 $V$ **상에서의 놈**이라고 정의한다.

**(1) 정부호** : $||\mathbf{u}|| \geq 0$ 이고 $\mathbf{u} = 0 \iff ||\mathbf{u}|| = 0$

**(2) 동질성** : $||k\mathbf{u}|| = |k|||\mathbf{u}||$

**(3) 삼각부등식** : $||\mathbf{u} + \mathbf{v}|| \leq ||\mathbf{v}|| + ||\mathbf{u}||$

〈http://freshrimpsushi.tistory.com/257〉



# Norm

Going a bit further, we define $||x||_p$ as a "p-norm". Given $x$, a vector with $i$ components, a p-norm is defined as:

$$||x||_p = \left( \sum_i |x_i|^p \right)^{1/p}$$

〈https://www.kaggle.com/residentmario/l1-norms-versus-l2-norms〉

## L2 Norm ➔ Euclidean Norm

$$||x||_2 = \left( \sum_i x_i^2 \right)^{1/2} = \sqrt{x_1^2 + x_2^2 + \cdots + x_i^2}$$

## L1 Norm ➔ Manhattan Norm

$$||x||_1 = \sum_i |x_i| = |x_1| + |x_2| + \cdots + |x_i|$$

# Ridge & Lasso

## Overview

Let's look at the equations. In ordinary least squares, we solve to minimize the following cost function:

$$\text{Cost} = (y - X\beta)^T (y - X\beta)$$

This term is the RSS, residual sum of squares. In ridge regression we instead solve:

$$\text{Cost} = (y - X\beta)^T (y - X\beta) + \lambda \beta^T \beta$$

The $\lambda \beta^T \beta$ term is an L2 norm.

In lasso regression we instead solve:

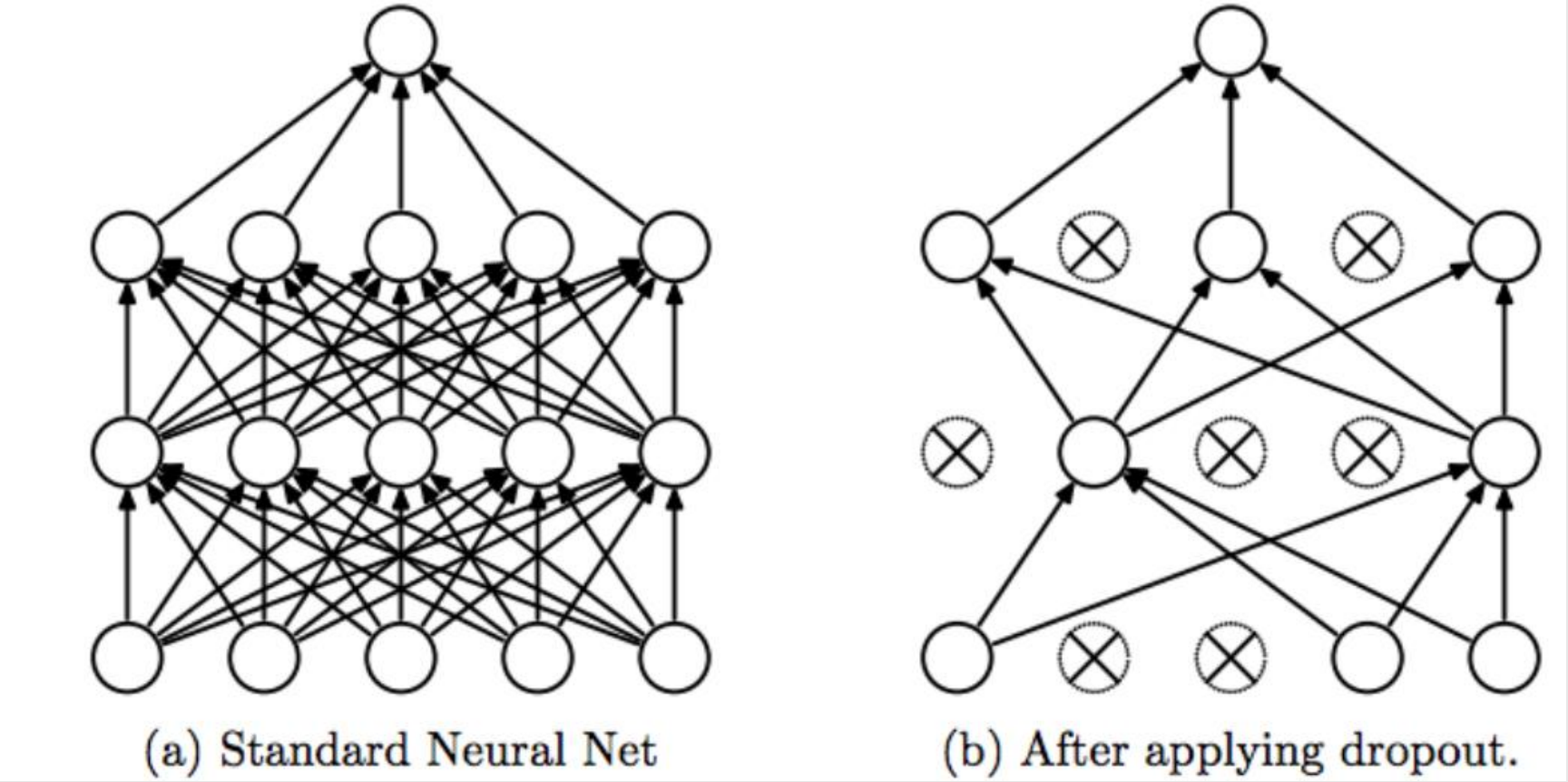
$$\text{Cost} = (y - X\beta)^T (y - X\beta) + \lambda |\beta|$$

The $\lambda |\beta|$ term is an L1 norm.

〈https://www.kaggle.com/residentmario/l1-norms-versus-l2-norms〉

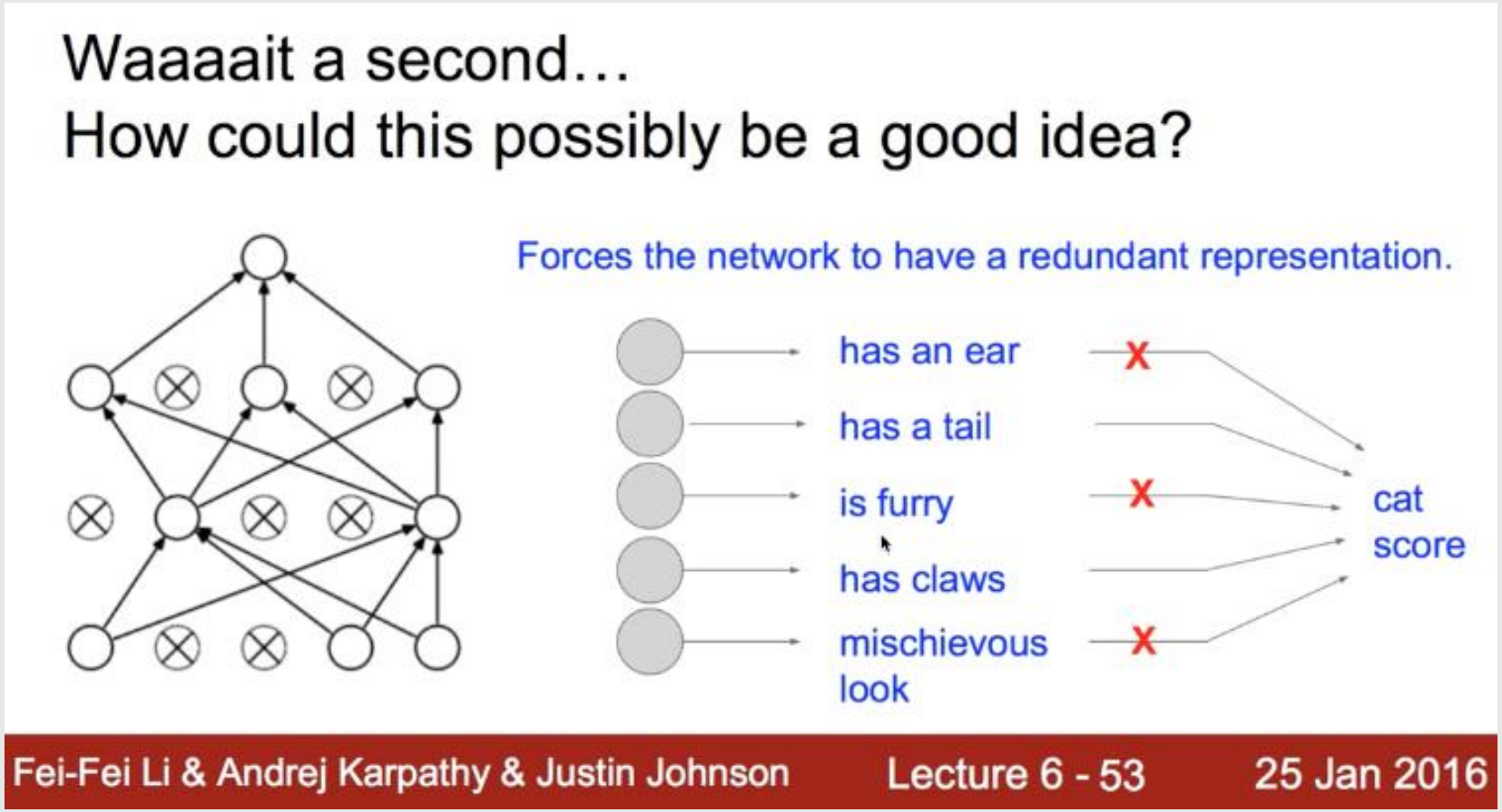
➔ MLP에서도 같은 방식으로 Cost에 Regularization Term 추가

# Dropout



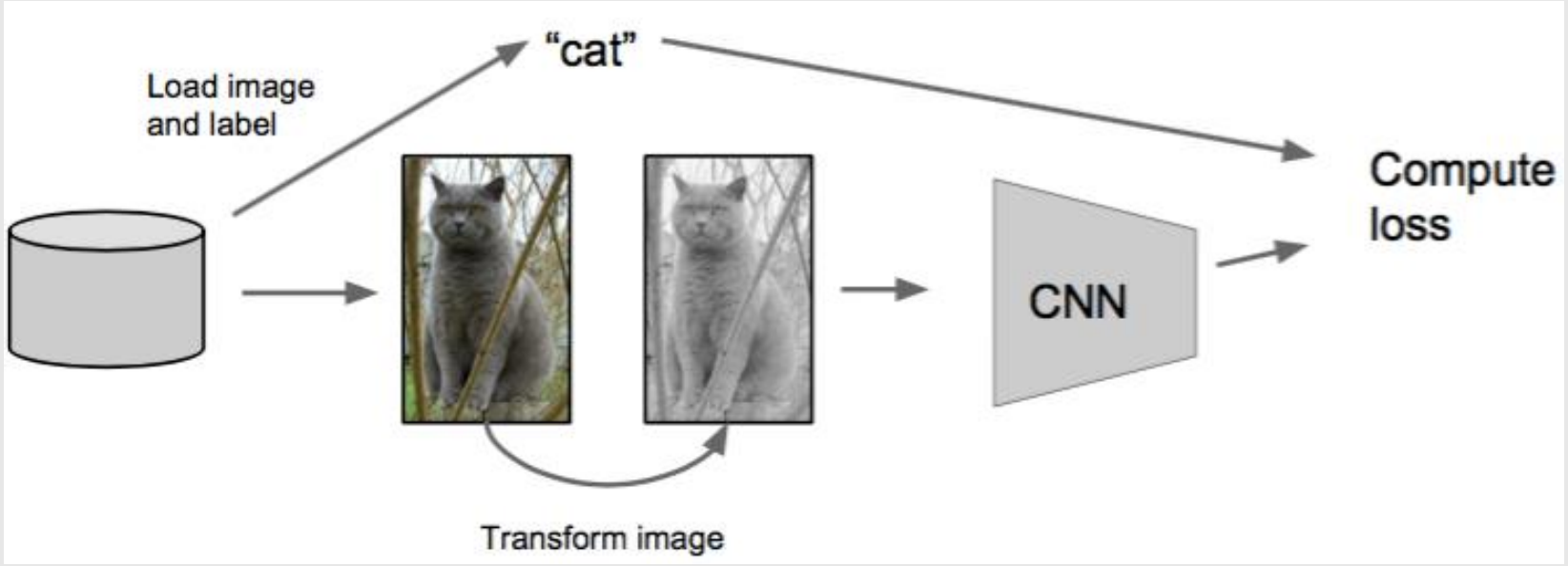

<https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5>

# Dropout




<http://pythonkim.tistory.com/42>

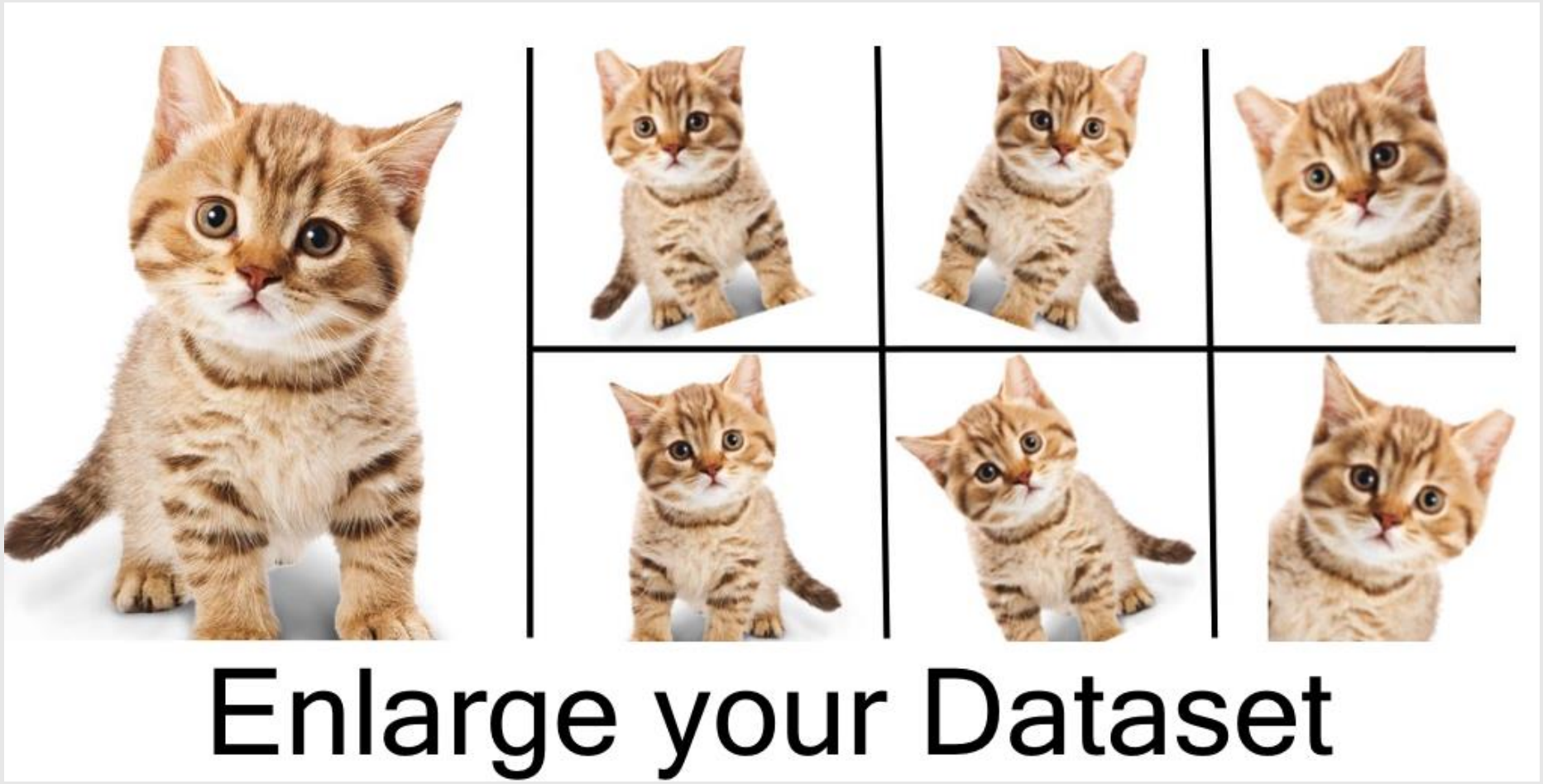
# Data Augmentation




〈http://nmhkahn.github.io/CNN-Practice〉

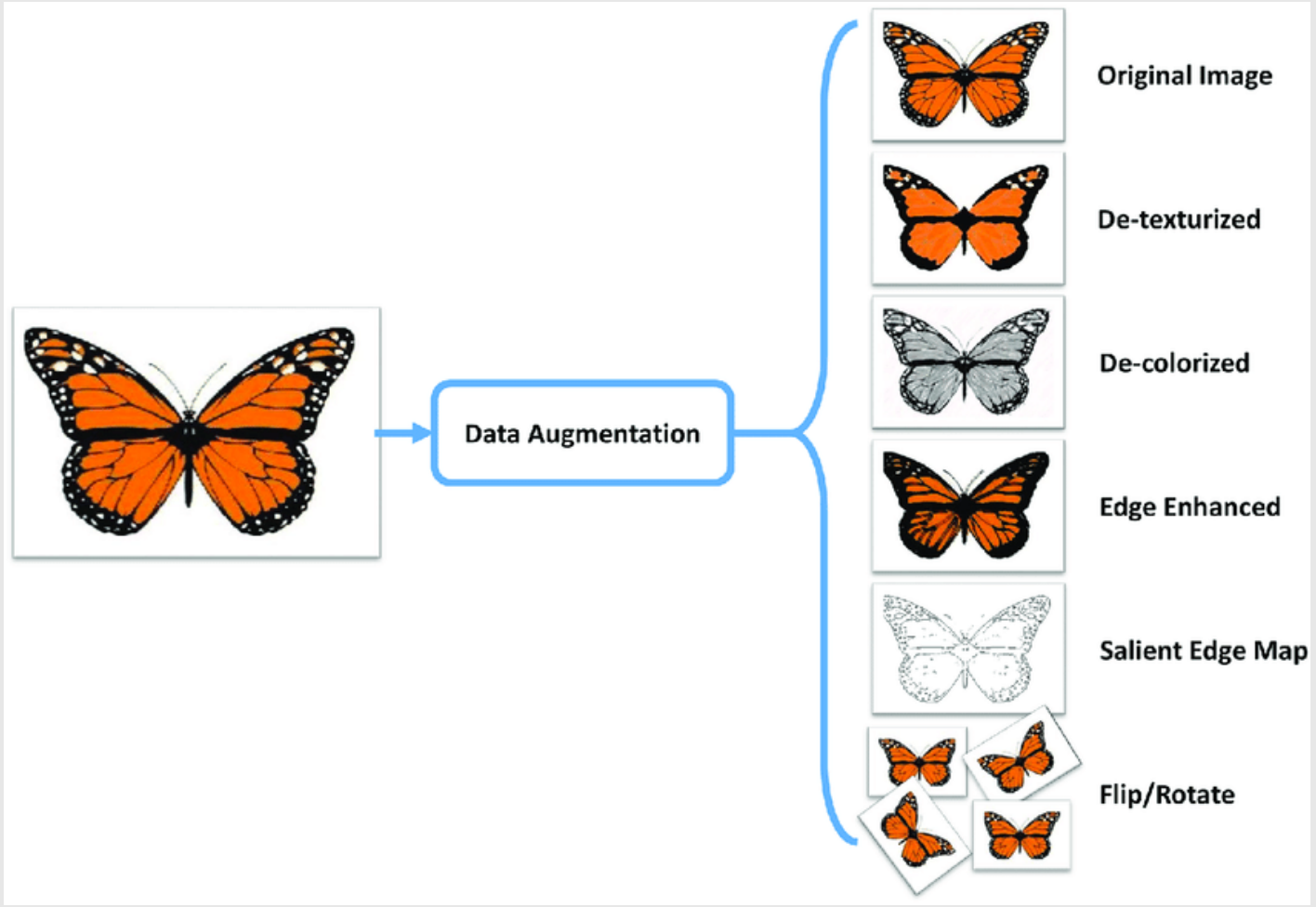
이미지의 레이블을 변경하지 않고 픽셀을 변화시키는 방법이며
변환된 데이터를 이용하여 학습을 진행

# Data Augmentation




〈https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced〉

# Data Augmentation




〈https://www.researchgate.net/figure/Data-augmentation-using-semantic-preserving-transformation-for-SBIR_fig2_319413978〉

# Adversarial Attack

**Adversarial Attacks on Neural Network Policies**

**Sandy Huang**[†], **Nicolas Papernot**[‡], **Ian Goodfellow**[§], **Yan Duan**[†§], **Pieter Abbeel**[†§]
[†] University of California, Berkeley, Department of Electrical Engineering and Computer Sciences
[‡] Pennsylvania State University, School of Electrical Engineering and Computer Science
[§] OpenAI

**Abstract**

Machine learning classifiers are known to be vulnerable to inputs maliciously constructed by adversaries to force misclassification. Such adversarial examples have been extensively studied in the context of computer vision applications. In this work, we show adversarial attacks are also effective when targeting neural network policies in reinforcement learning. Specifically, we show existing adversarial example crafting techniques can be used to significantly degrade test-time performance of trained policies. Our threat model considers adversaries capable of introducing small perturbations to the raw input of the policy. We characterize the degree of vulnerability across tasks and training algorithms, for a subclass of adversarial-example attacks in white-box and black-box settings. Regardless of the learned task or training algorithm, we observe a significant drop in performance, even with small adversarial perturbations that do not interfere with human perception. Videos are available at http://rll.berkeley.edu/adversarial.

**1 Introduction**

Recent advances in deep learning and deep reinforcement learning (RL) have made it possible to learn end-to-end policies that map directly from raw inputs (e.g., images) to a distribution over actions to take. Deep RL algorithms have trained policies that achieve superhuman performance on Atari games [15, 19, 16] and Go [21], perform complex robotic manipulation skills [13], learn locomotion tasks [19, 14], and drive in the real world [7].

These policies are parametrized by neural networks, which have been shown to be vulnerable to adversarial attacks in supervised learning settings. For example, for convolutional neural networks trained to classify images, perturbations added to the input image can cause the network to classify the adversarial image incorrectly, while the two images remain essentially indistinguishable to humans [22]. In this work, we investigate whether such adversarial examples affect neural network *policies*, which are trained with deep RL. We consider a fully trained policy at test time, and allow the adversary to make limited changes to the raw input perceived from the environment before it is passed to the policy.

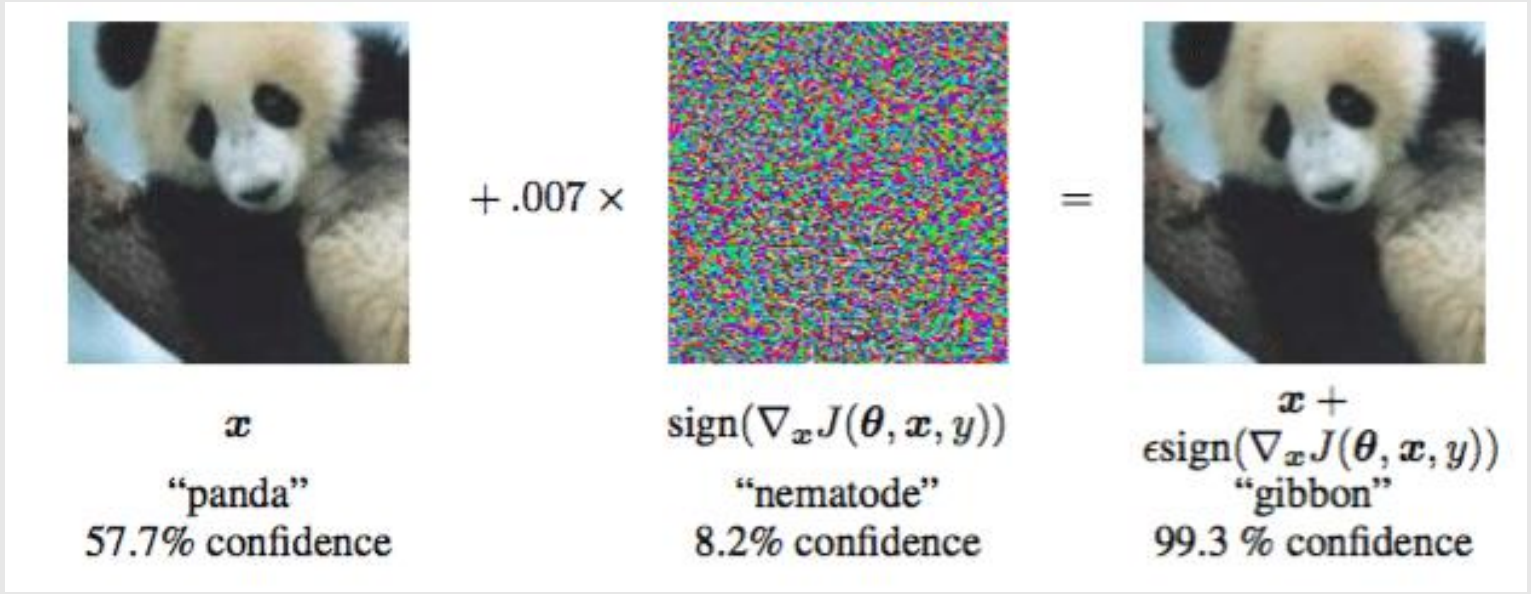
Unlike supervised learning applications, where a fixed dataset of training examples is processed during learning, in reinforcement learning these examples are gathered throughout the training process. In other words, the algorithm used to train a policy, and even the random initialization of the policy network's weights, affects the states and actions encountered during training. Policies trained to do the same task could conceivably be significantly different (e.g., in terms of the high-level features they extract from the raw input), depending on how they were initialized and trained. Thus, particular learning algorithms may result in policies more resistant to adversarial attacks. One could also imagine that the differences between supervised learning and reinforcement learning might prevent an adversary from mounting a successful attack in the black-box scenario, where the attacker does not have access to the target policy network.

## Adversarial Attack?

➔ Adversarial examples are inputs to machine learning models that an attacker has intentionally designed to cause the model to make a mistake; they're like optical illusions for machines.
(Adversarial examples는 공격자가 모델이 실수 만들도록 고안한 기계 학습 모델에 대한 입력이다. 그들은 기계에 대한 착시 현상과 같다.)

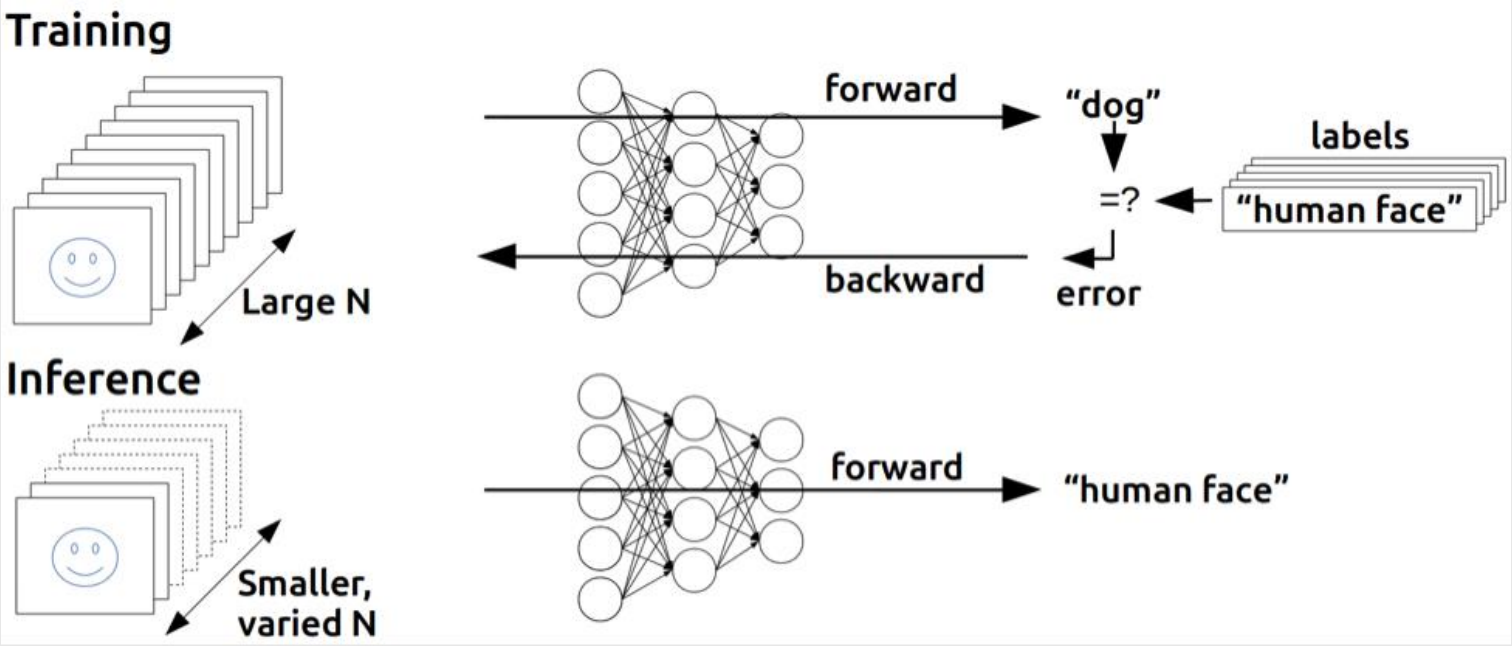
➔ 노이즈를 생성하여 사람의 눈에는 같아 보이지만 머신러닝 모델을 헷갈리게 만드는 것이 핵심

# Adversarial Attack

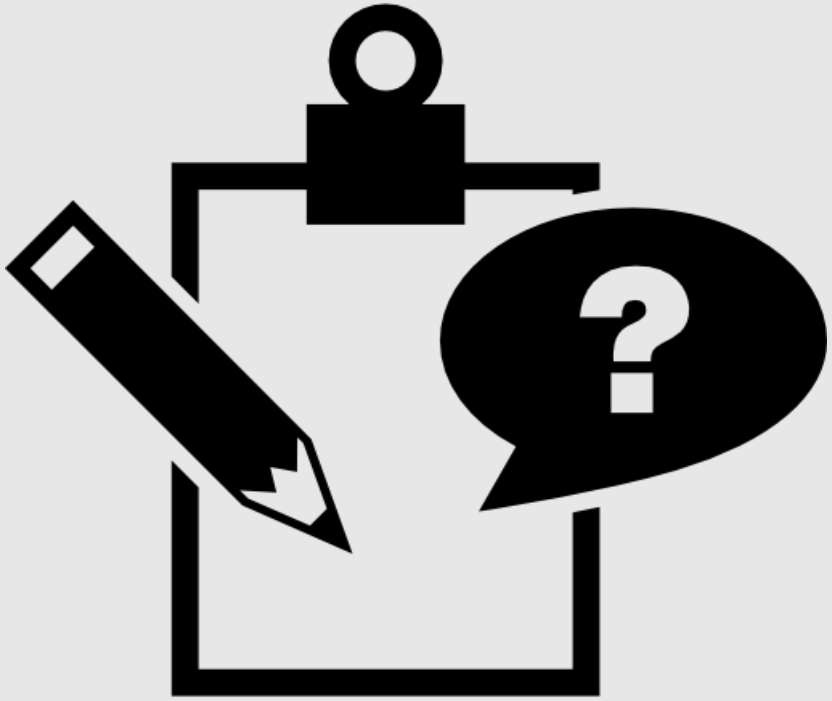



〈https://blog.openai.com/adversarial-example-research/〉

# Adversarial Attack

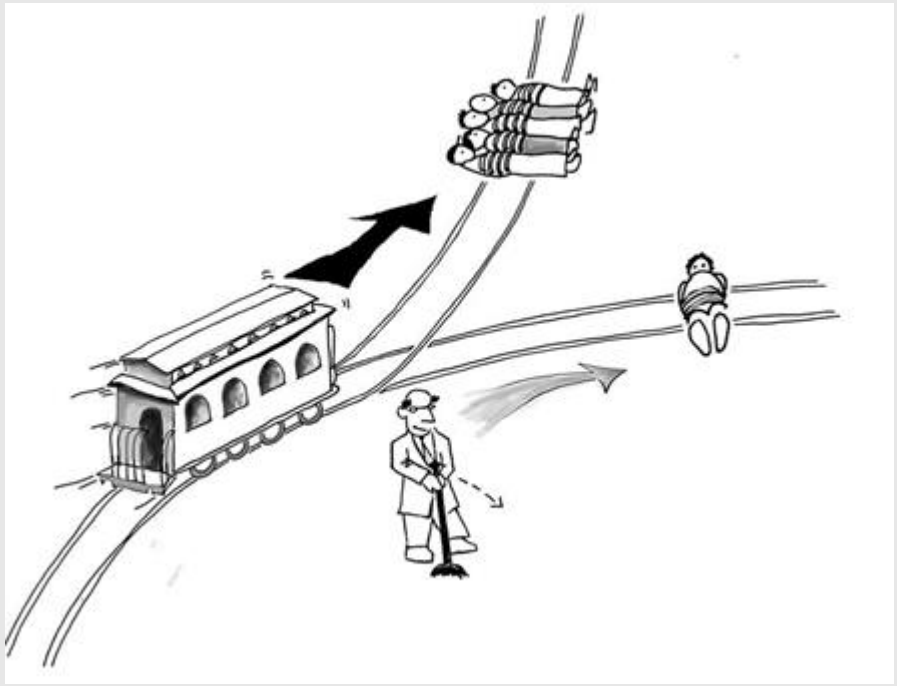



〈https://devblogs.nvidia.com/inference-next-step-gpu-accelerated-deep-learning/〉



# Before Finish Class

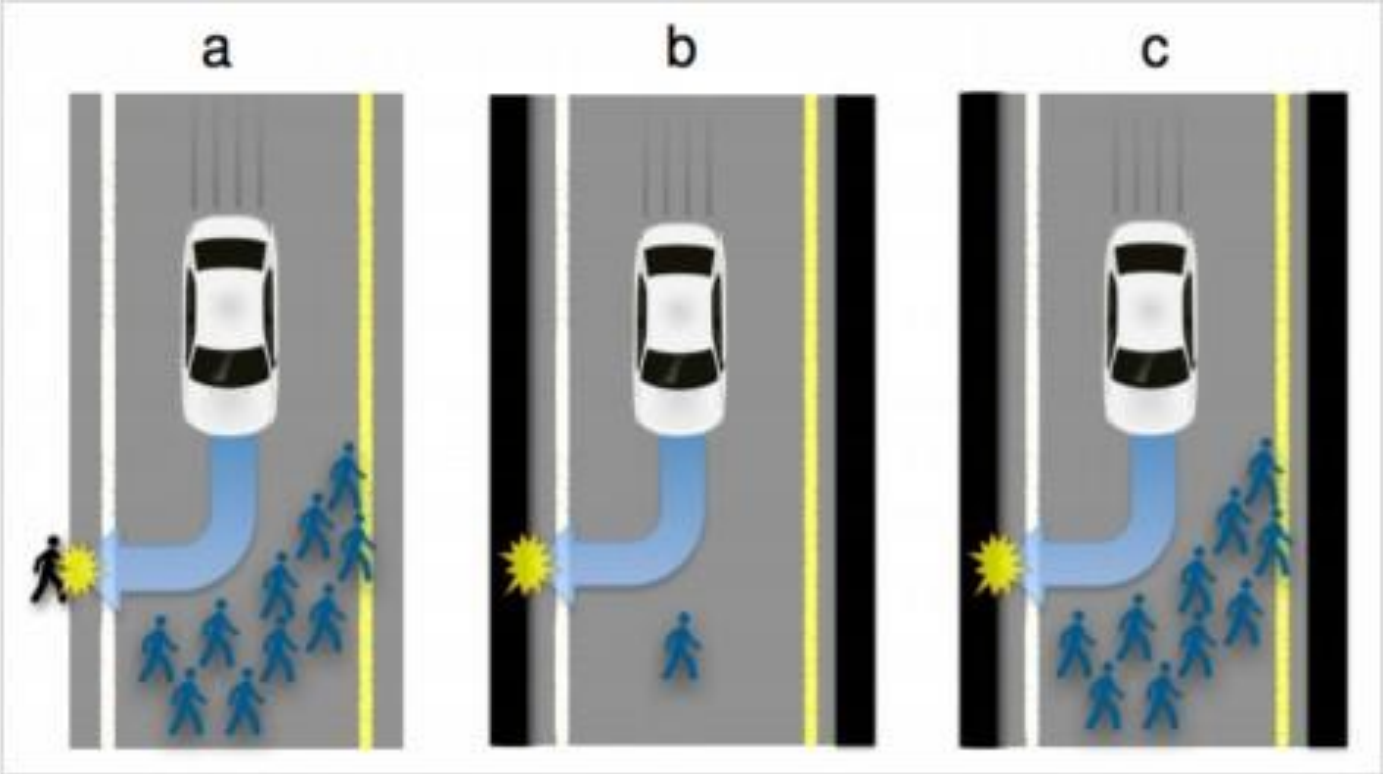
# Trolley Problem



- 열차가 선로를 따라 달리고 있고 선로 중간에 서는 인부 다섯 명이 작업을 하고 있다.
- 당신의 손에는 열차의 선로를 바꿀 수 있는 전환기가 있다.
- 다섯 사람을 구하기 위해서 선로를 바꾸는 전환기를 당기면 된다.
- 불행하게도 다른 선로에는 인부 한 명이 작업을 하고 있다.
- **다섯 명을 살리기 위해 한 명을 희생시키는 행위가 도덕적으로 허용될 수 있는가?**

+ 추가적으로, 그 한 명이 나의 가족이라면?

# Trolley Problem




〈http://blog.rightbrain.co.kr/?p=9003&fbclid=IwAR01IAnvp66T7hjwcGds_1zVNexIYHkVJXIWAm7pIYWJ3RqS9IpovObhIek〉

# Schedule

12장 Skip!
11월 30일: 13장 (CNN)
12월 07일: 14장 (RNN)
12월 14일: 15장 (Autoencoder)
12월 21일: 총 정리 + 망년회


Ideas worth spreading
– TED Talks

# 고생하셨습니다.