Machine Learning Tutorial
Part 2
[Neural Network Theory and Optimization]

Inha University

Prof. Sang-Jo Yoo

sjyoo@inha.ac.kr

Contents (Part 2)

- **Neural Network Basics**
  - Linear Regression
  - Gradient Descent Algorithm
  - Logistic Regression (sigmoid)
  - SoftMax Regression
- **Deep Neural Network**
  - Multi-Layer Perceptron
  - XOR Problem
  - Feed Forwarding
  - Backpropagation
  - Gradient Vanishing Problem
  - Tanh/ReLu/Maxout
- **Optimization for Training Deep Models**
  - Data Normalization
  - Parameter Updating (SGD, Batch GD, Mini-batch GD)
  - Gradient Descent with Momentum
  - Gradient Descent with Adaptive Learning Rate
  - Regularization and Dropout
  - Parameter Initialization
  - Hyperparameters for Learning

Neural Network Basics

Linear Regression

Gradient Descent Algorithm

Logistic Regression

SoftMax Regression

Machine Learning Tasks

The task

predicting t from x

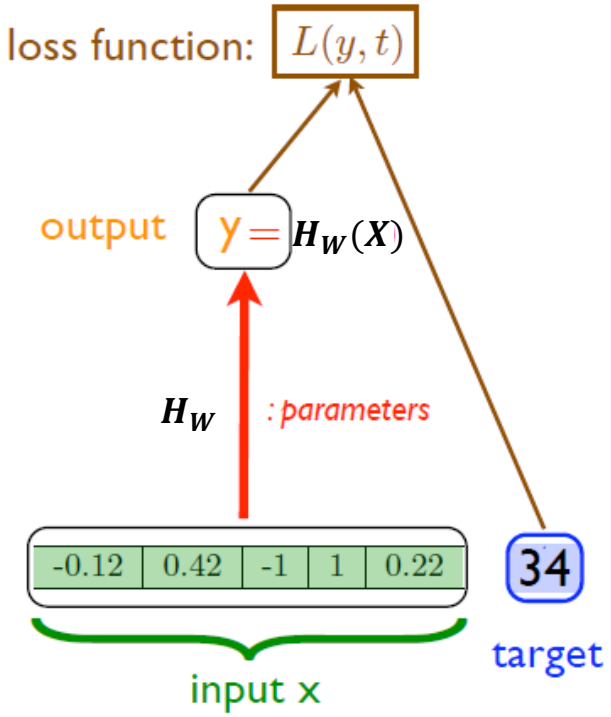
input $x \in \mathbb{R}^d$ target t

n examples

x_1	x_2	x_3	x_4	x_5	t
0.32	-0.27	+1	0	0.82	113
-0.12	0.42	-1	1	0.22	34
0.06	0.35	-1	1	-0.37	56
0.91	-0.72	+1	0	-0.63	77
...	...	...	...	...	...

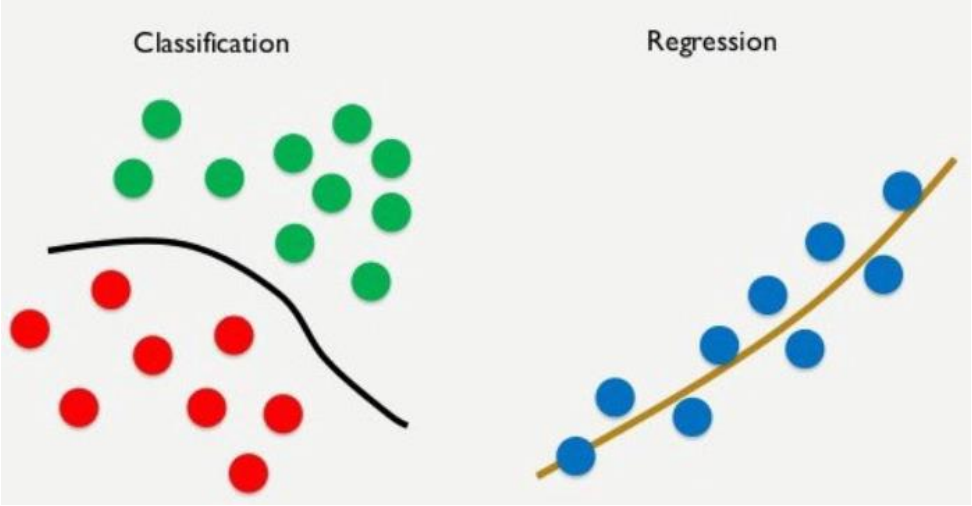
Training Set D_n

Learning a parameterized function H_W that minimizes a loss.



Regression and Classification

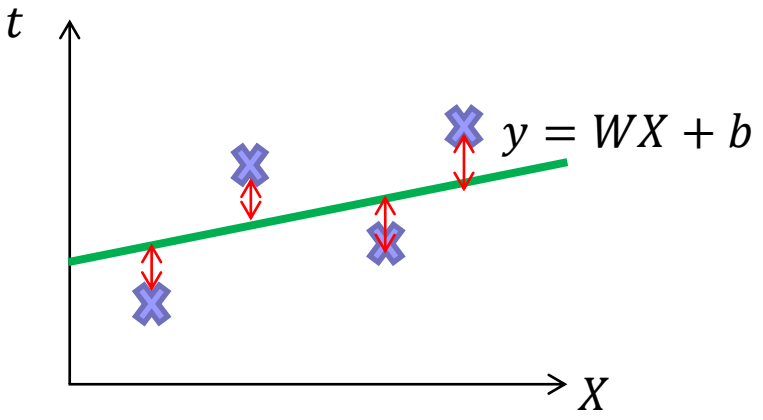
- **A regression model predicts continuous values.**
  - What is the value of a house in California?
  - What is the probability that a user will click on this ad?
- **A classification model predicts discrete values.**
  - Is a given email message spam or not spam?
  - Is this an image of a dog, a cat, or a hamster?



Linear Regression

- **Linear Hypothesis**

$$H_W(X) = WX + b$$



- **Cost function : L2 loss function**
  - For m training examples

$$cost = \frac{1}{m}\sum_{i=1}^{m}\left[H\left(X^{(i)}\right) - t^{(i)}\right]^2$$

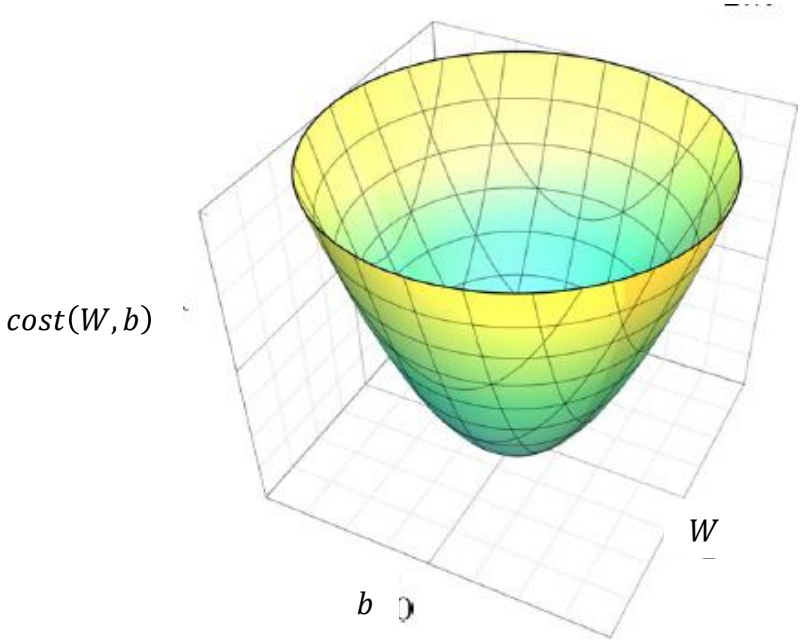
$$cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}\left[WX^{(i)} + b - t^{(i)}\right]^2$$

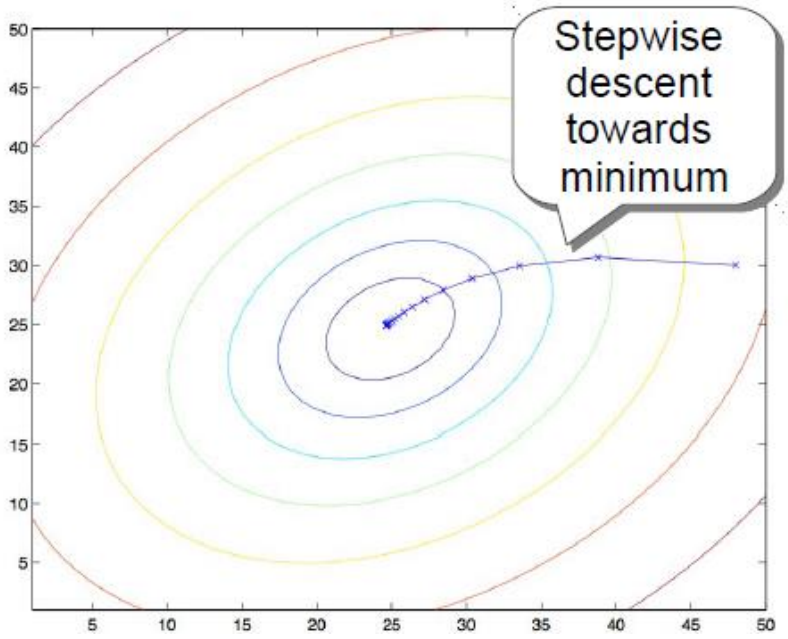
- **Learning objective: Finding optimal parameters (W, b)**

$$(W^*, b^*) = \min_{W,b} cost(w,b)$$

Gradient Descent Algorithm

- **How to minimize cost?**
  - Little modified cost function
    - $cost(W,b) = \frac{1}{2m}\sum_{i=1}^{m}\left[WX^{(i)} + b - t^{(i)}\right]^2$





Gradient Descent Algorithm

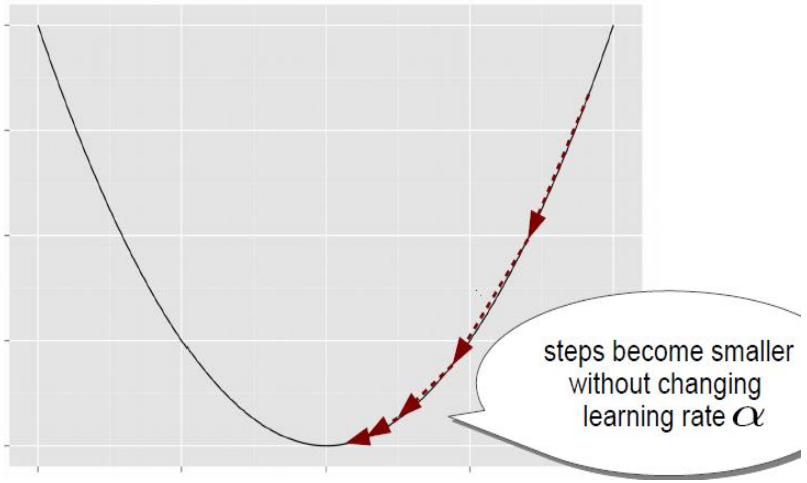
$$WX + b = \overbrace{[b \;\; w_1 \; w_2 \;\; \cdots w_d]}^{W} \overbrace{\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}}^{X} = WX$$

$$W := W - \alpha \frac{\partial}{\partial W} cost(W) = W - \alpha \frac{\partial}{\partial W} \left\{ \frac{1}{2m} \sum_{i=1}^{m} \left[WX^{(i)} - t^{(i)} \right]^2 \right\}$$

$$W := W - \alpha \frac{1}{m} \sum_{i=1}^{m} \left[WX^{(i)} - t^{(i)} \right] X^{(i)}$$

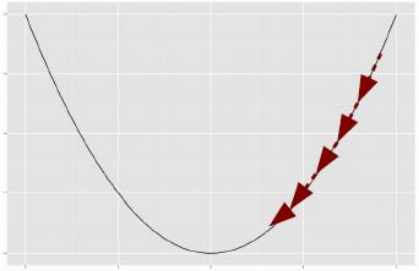
α: learning rate

The most commonly used rates are :
0.001, 0.003, 0.01, 0.03, 0.1, 0.3.

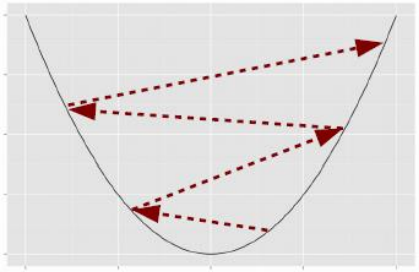


Gradient Descent Algorithm

- **Learning rate considerations**
  - Small learning rate leads to slow convergence

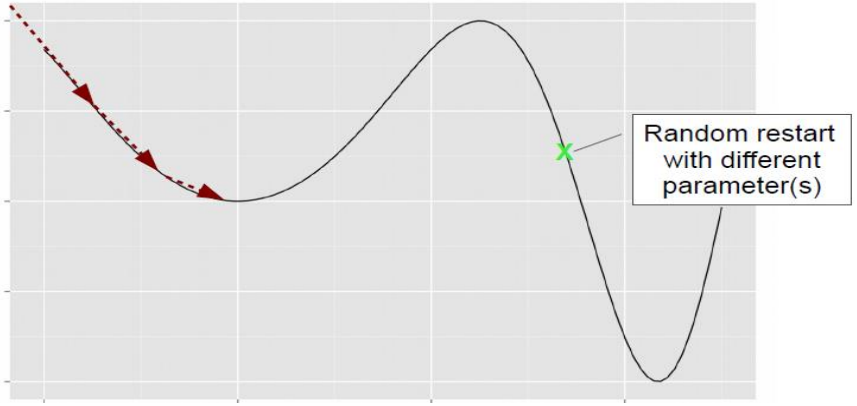


  - Overly large learning rate may not lead to converge or divergence



  - Open $\alpha \in [0.001,1]$

- **Gradient descent can get stuck at local minima.**

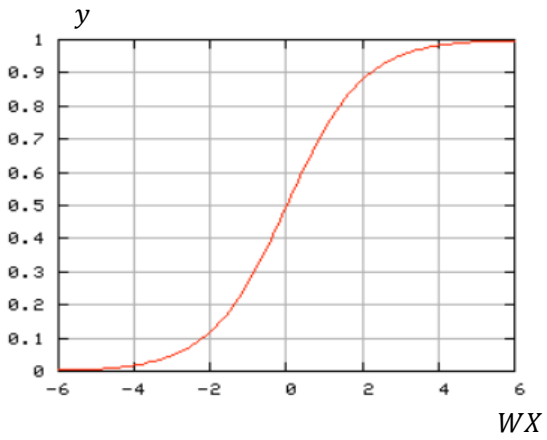


Logistic Regression (Sigmoid)

- **If we have a binary classification task:**
  - We want to estimate conditional probability:

$$t \in \{0,1\}$$

$$y \cong P(t = 1|X) \quad y \in [0,1]$$



  - We choose a non-linear mapping: **Sigmoid**

$$H(X) = WX + b = \boldsymbol{WX}$$

$$y = f(X) = \frac{1}{1+e^{-H(X)}} = \frac{1}{1+e^{-\boldsymbol{WX}}}$$

  - New cost function for logistic sigmoid
    - Cross-entropy loss function

$t = 1$		$t = 0$	
$y = 1$	$L = 0$	$y = 0$	$L = 0$
$y = 0$	$L = \infty$	$y = 1$	$L = \infty$

$$L(y,t) = -t\log(y) - (1-t)\log(1-y)$$

$$= -t\log\left(\frac{1}{1+e^{-\boldsymbol{WX}}}\right) - (1-t)\log\left(1-\frac{1}{1+e^{-\boldsymbol{WX}}}\right)$$

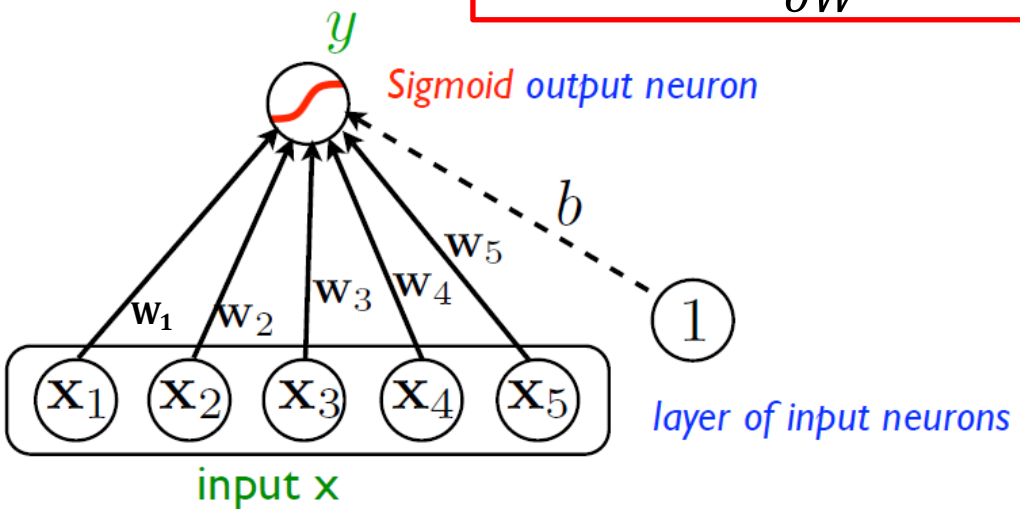
Logistic Regression (Sigmoid)

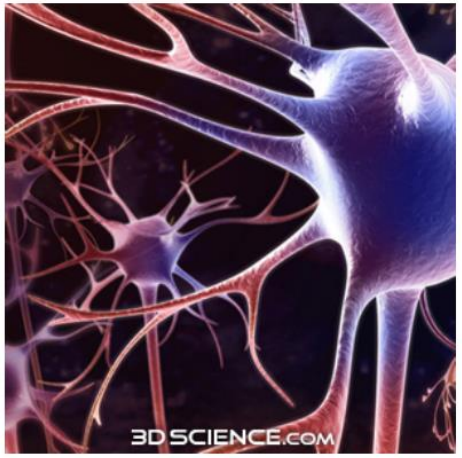
- **Minimize loss function: Using Gradient descent algorithm**

$$Loss(W) = \frac{1}{m}\sum_{i=1}^{m} L\left(y^{(i)}, t^{(i)}\right)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left\{-t^{(i)} log\left(\frac{1}{1+e^{-WX^{(i)}}}\right) - \left(1-t^{(i)}\right) log\left(1-\frac{1}{1+e^{-WX^{(i)}}}\right)\right\}$$

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





TensorFlow Example

- **TensorFlow**
  - is an open source software library for machine learning in various kinds of perceptual and language understanding tasks originally developed by the Google Brain team.

```
h=tf.matmul(W,X)
hypothesis=tf.div(1., 1.+tf.exp(-h))
cost=tf.reduce_mean(-t*tf.log(hypothesis)-(1-t)*tf.log(1-hypothesis))
a=tf.Variable(0,1)
optimizer=tf.train.GradientDescentoptimizer(a)
train=optimizer.minimize(cost)

for step in xrange(2001):
    sess.run(train.feed_dict={X:x_data, Y:y_data})
```

Softmax Regression

Multinomial classification

$$Z = H(X) = WX$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

- In general, the output of network can be any value and may not be easy to interpret.
- Sigmoid is used for binary classification.
- For output layer for multiclass classifier, Softmax is generally used.

$$y_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

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

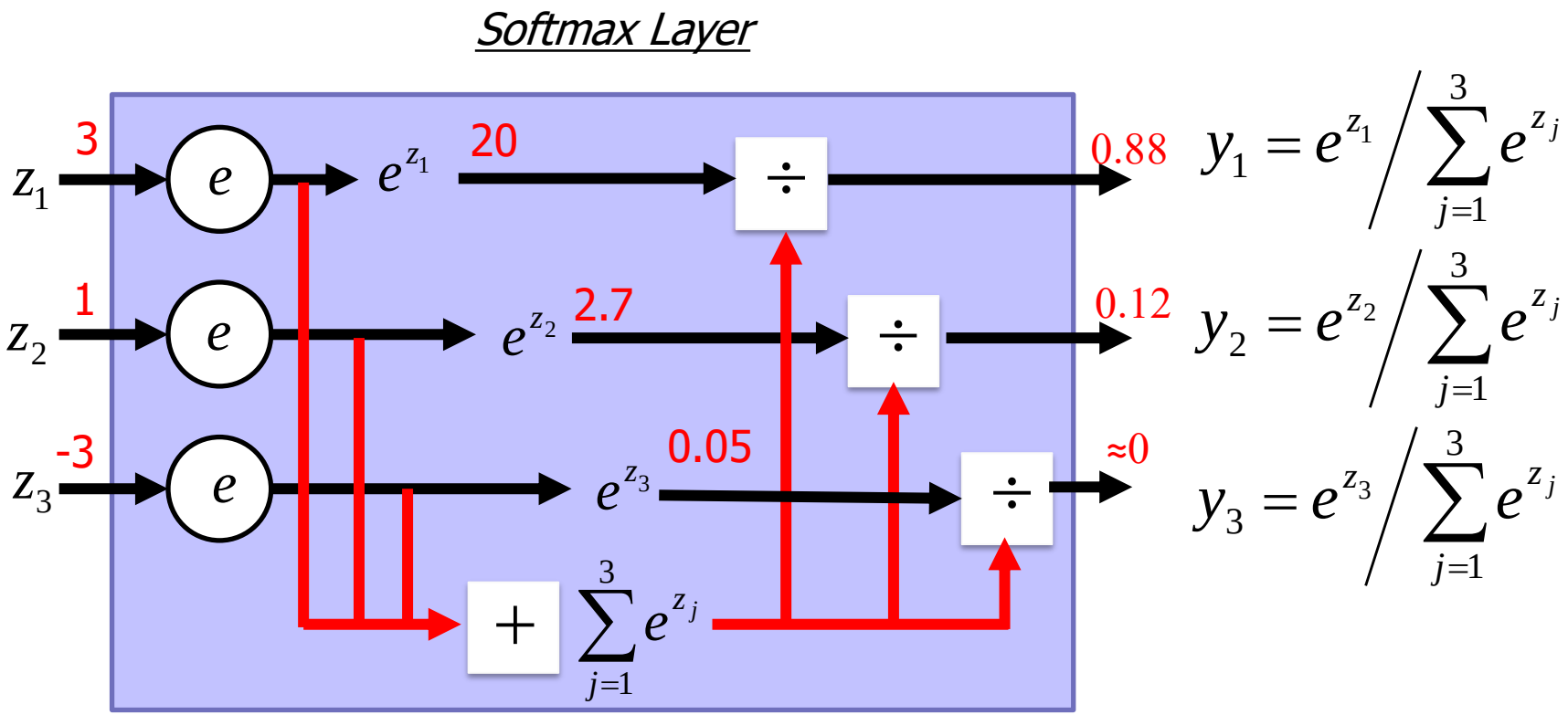
Probability:

$$1 > y_k > 0$$

$$\sum_j y_j = 1$$

Softmax Regression

- **Softmax** layer as the output layer



Softmax Regression

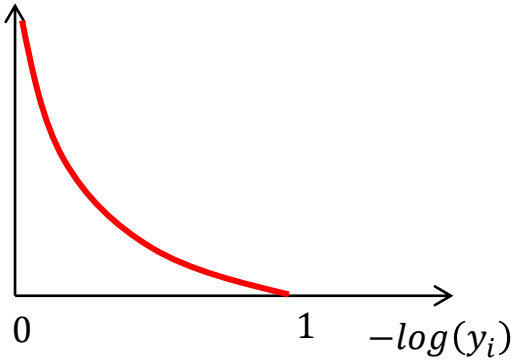
- **New cost function for Softmax Regression**
  - loss function

$$L(y,t) = -\sum_{i=1} t_i log(y_i) = -\sum_{i=1} t_i log\left(\frac{e^{z_i}}{\sum_j e^{z_j}}\right)$$

$t = \begin{bmatrix}0\\1\end{bmatrix}$, target

$y = \begin{bmatrix}0\\1\end{bmatrix}$, then $L(y,t) = -[0\ 1]\begin{bmatrix}\log(0)\\\log(1)\end{bmatrix} = 0$

$y = \begin{bmatrix}1\\0\end{bmatrix}$, then $L(y,t) = -[0\ 1]\begin{bmatrix}\log(1)\\\log(0)\end{bmatrix} = \infty$



  - W update

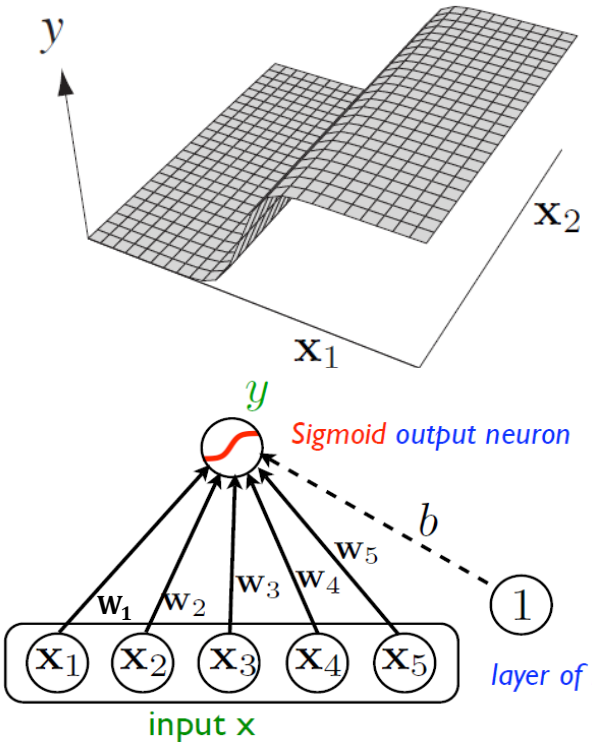
$$W := W - \alpha \frac{\partial}{\partial W} Loss(W)$$

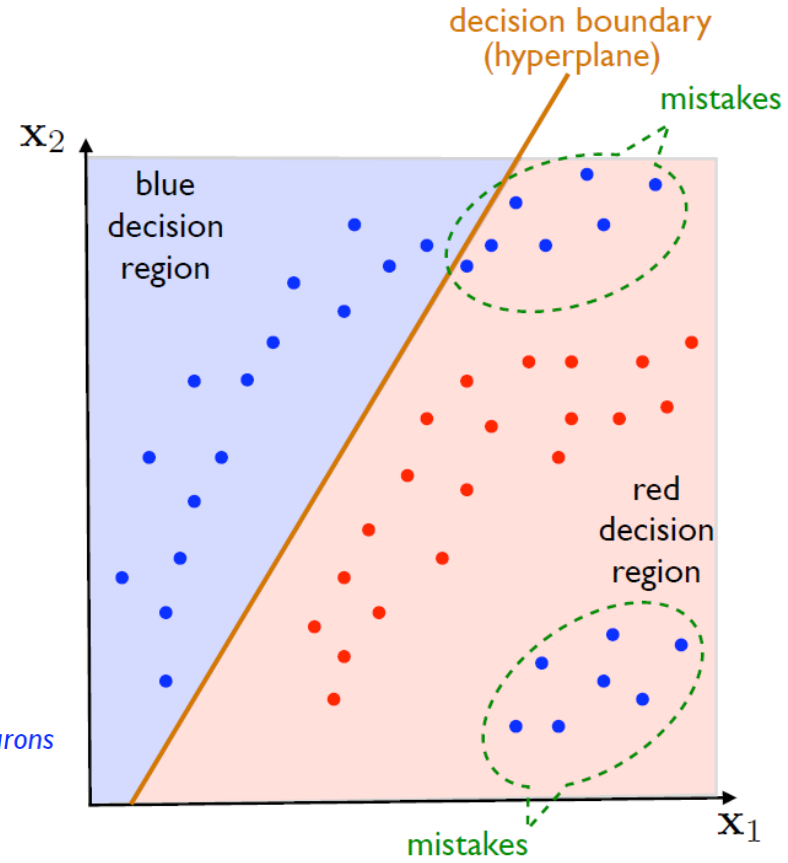
Deep Neural Network

Multi-Layer Perceptron
XOR Problem
Feed Forwarding
Backpropagation
ReLu/Maxout

Limitations of Logistic Regression

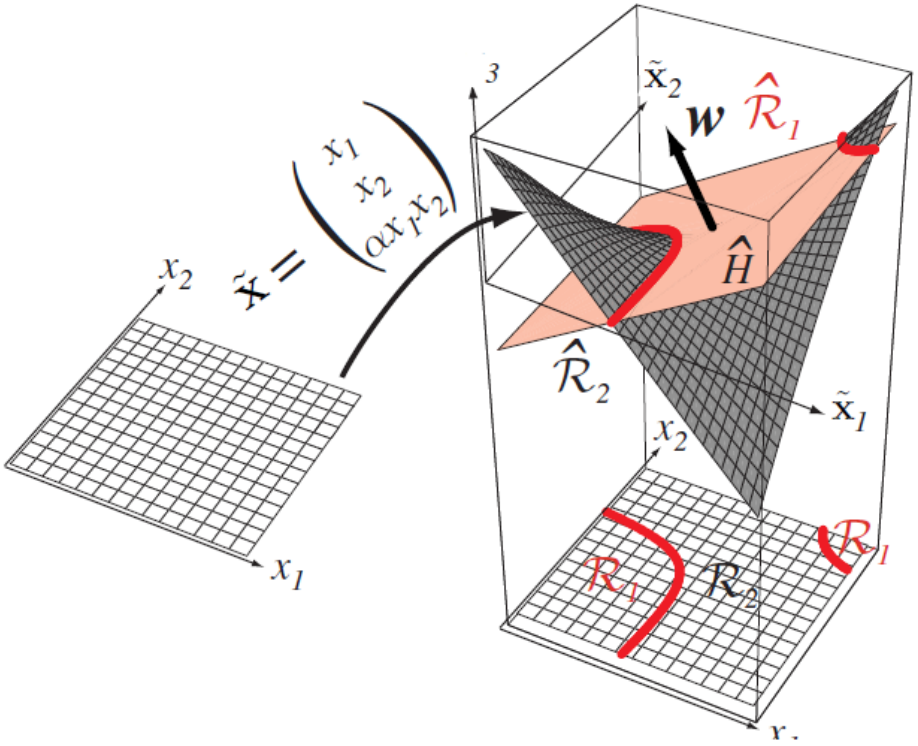
- **Only yields "linear" decision boundary: a hyperplane**
  - Inappropriate if classes not linearly separable





Non Linear Classification

- How to obtain non-linear decision boundaries?
  - An old technique: **Fixed mapping**
    - map X non-linearly to feature space: $\tilde{X} = \emptyset(X)$
    - find separating hyperplane in new space
    - hyperplane in new space corresponds to non-linear decision surface in initial X space.

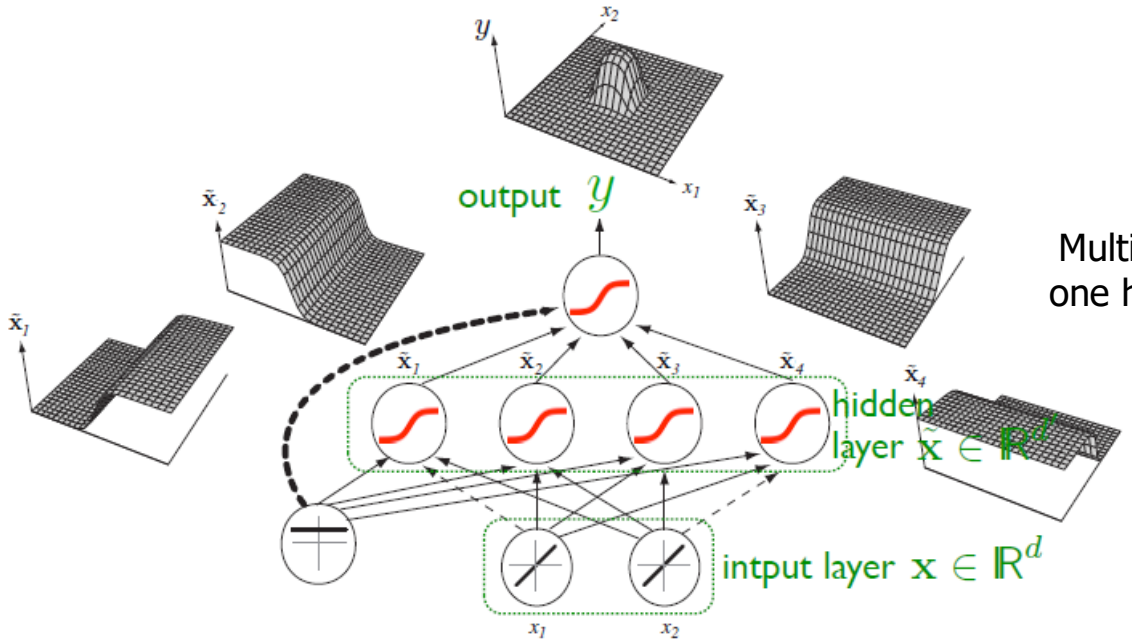


Non Linear Classification

- **How to obtain non-linear decision boundaries?**
  - Three ways to map $X \;\; to \; \tilde{X} = \emptyset(X)$
  - Use an explicit fixed mapping (previous example)
  - Use an implicit fixed mapping
    - Kernel Methods (SVMs, Kernel Logistic Regression, …)
  - Learn a parameterized mapping:
    - Multilayer feed-forward Neural Networks such as Multilayer Perceptron (MLP)
    - MLP: a feed forward artificial neural network model that maps sets of input data onto a set of appropriate outputs.

Multi-Layer Perceptron (MLP)

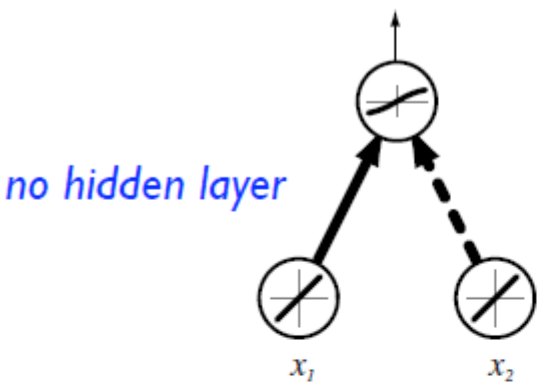
- An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron with a nonlinear activation function.
- MLP utilizes a supervised learning technique called back-propagation for training the network. MLP is a modification of the standard linear perceptron and can distinguish data that are not linearly separable.

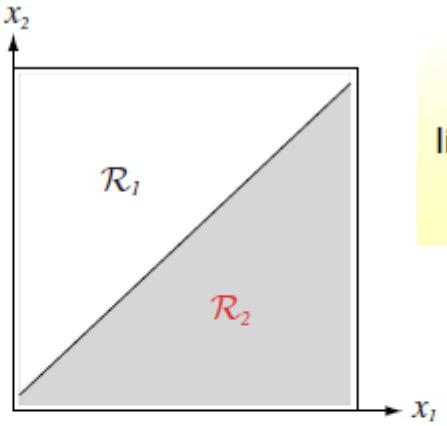


Multi-Layer Perceptron (MLP) with one hidden layer of size 4 neurons.

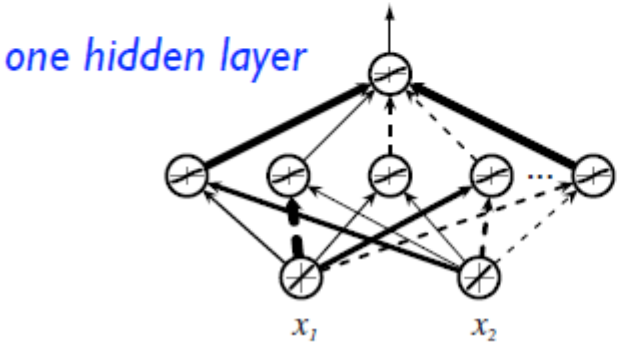
Multi-Layer Perceptron

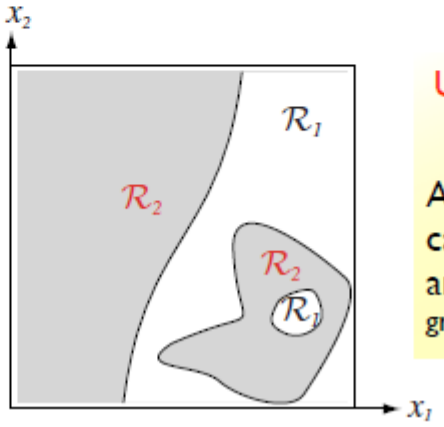
- **Expressive power of Neural Networks with one hidden layer**





== Logistic regression limited to representing a separating hyperplane



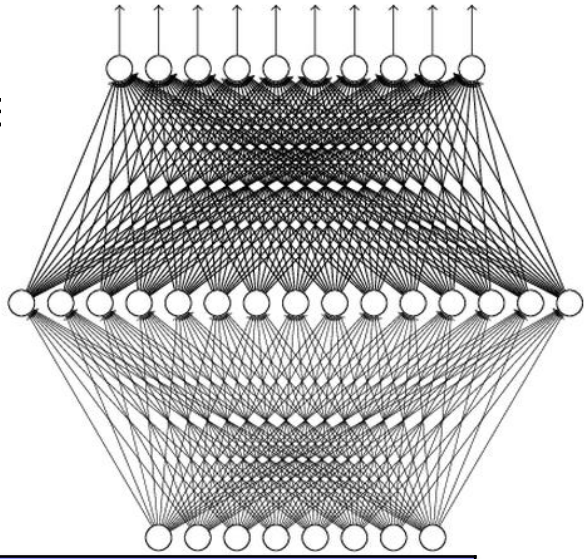


Universal approximation property

Any continuous function can be approximated arbitarily well (with a growing number of hidden unis)

Universality Theorem

- **Any continuous function** $f : R^N \rightarrow R^M$
  - can be realized by a network with one hidde layer: given enough hidden neurons.
- **Why "Deep" neural network not "Fat" neural network?**

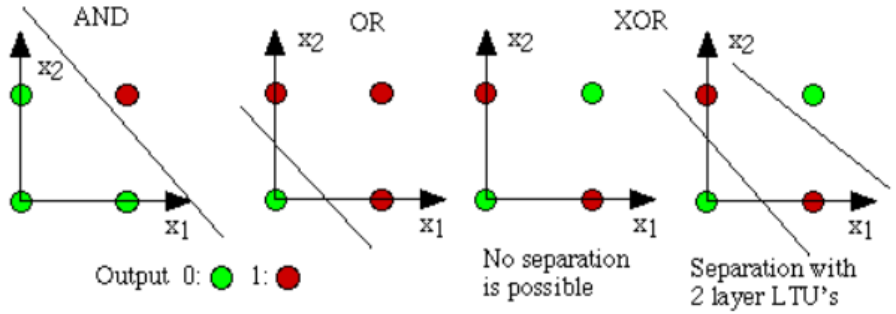


Seide, Frank, Gang Li, and Dong Yu. "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks." *Interspeech*. 2011.

Layer X Size	Word Error Rate (%)	Layer X Size	Word Error Rate (%)
1 X 2k	24.2		
2 X 2k	20.4		
3 X 2k	18.4		
4 X 2k	17.8		
5 X 2k	17.2	1 X 3772	22.5
7 X 2k	17.1	1 X 4634	22.6
		1 X 16k	22.1

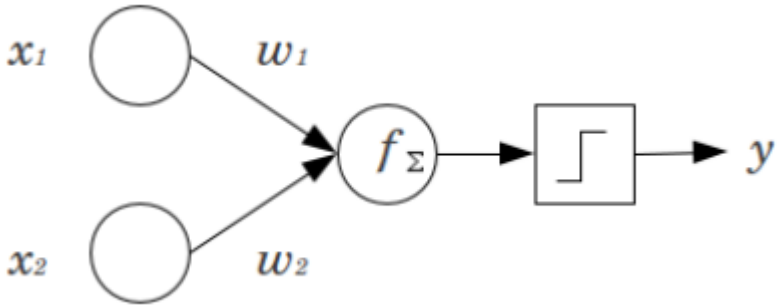
XOR Problem in Neural Network

- A typical example of non-lineally separable function is the XOR.
  - This function takes two input arguments with values in {0,1} and returns one output in {0,1}:



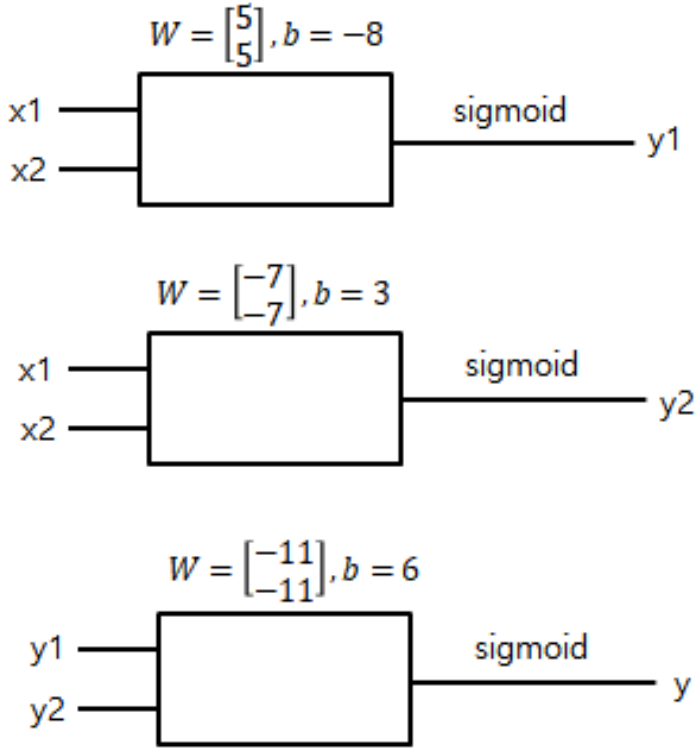
x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

  - Single layer perceptron is not able to implement XOR operation.



XOR Problem in Neural Network

- **XOR using a hidden layer**



x_1	x_2	y_1	y_2	y	XOR
0	0	0	1	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	0	0	0

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

XOR Problem in Neural Network

- **XOR using a hidden layer**

x_1	x_2	y_1	y_2	y	XOR
0	0	0	1	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	0	0	0

$$(0 \quad 0)\begin{pmatrix}5\\5\end{pmatrix} - 8 = -8, y_1 = s(-8) = 0$$

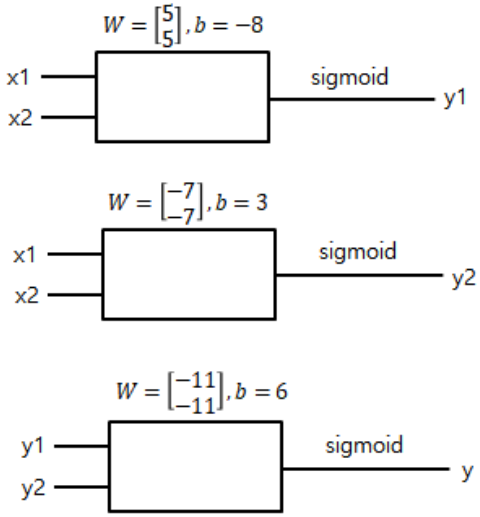
$$(0 \quad 0)\begin{pmatrix}-7\\-7\end{pmatrix} + 3 = +3, y_2 = s(+3) = 1$$

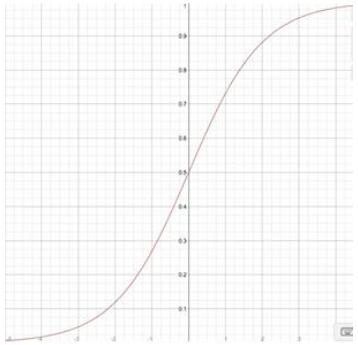
$$(0 \quad 1)\begin{pmatrix}-11\\-11\end{pmatrix} + 6 = -11 + 6 = -5, y = s(-5) = 0$$

$$(0 \quad 1)\begin{pmatrix}5\\5\end{pmatrix} - 8 = -3, y_1 = s(-3) = 0$$

$$(0 \quad 1)\begin{pmatrix}-7\\-7\end{pmatrix} + 3 = -4, y_2 = s(-4) = 0$$

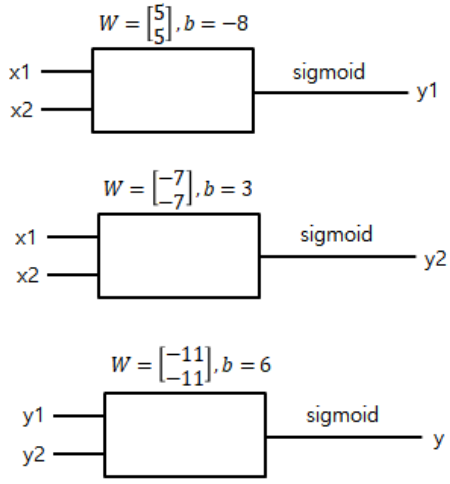
$$(0 \quad 0)\begin{pmatrix}-11\\-11\end{pmatrix} + 6 =, +6, y = s(+6) = 1$$

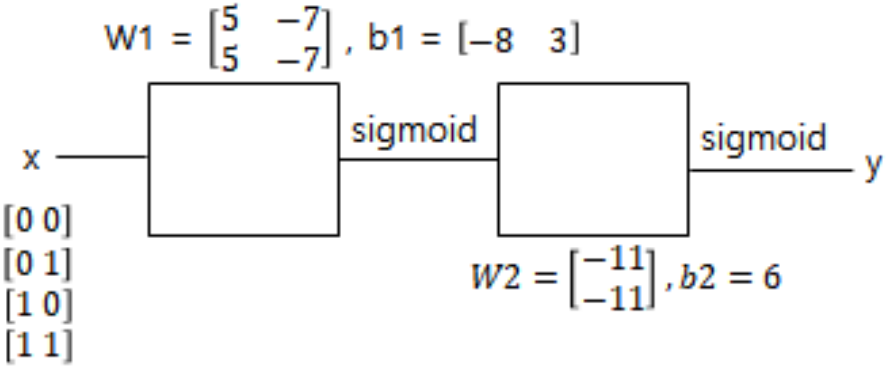


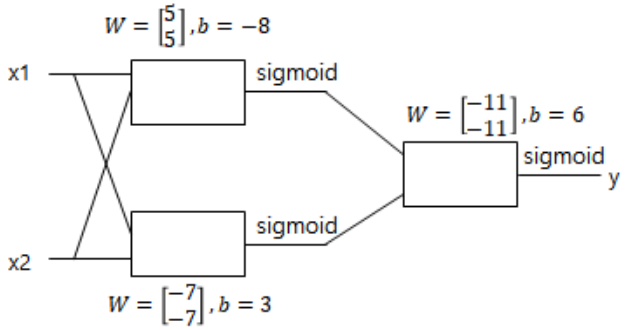


XOR Problem in Neural Network

❑ XOR using a hidden layer







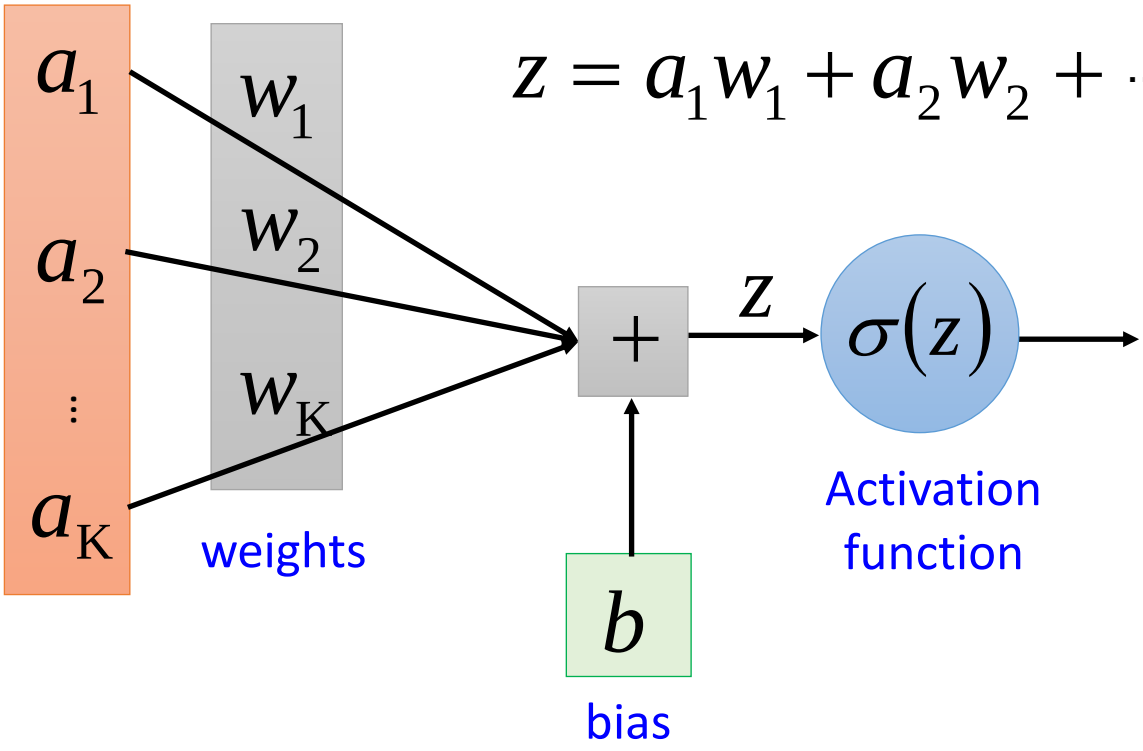

```
K=tf.sigmoid(tf.matmul(X, W1)+b1)
hypothesis = tf.sigmoid(tf.matmul(K, W2)+b2)
```

Feed Forwarding

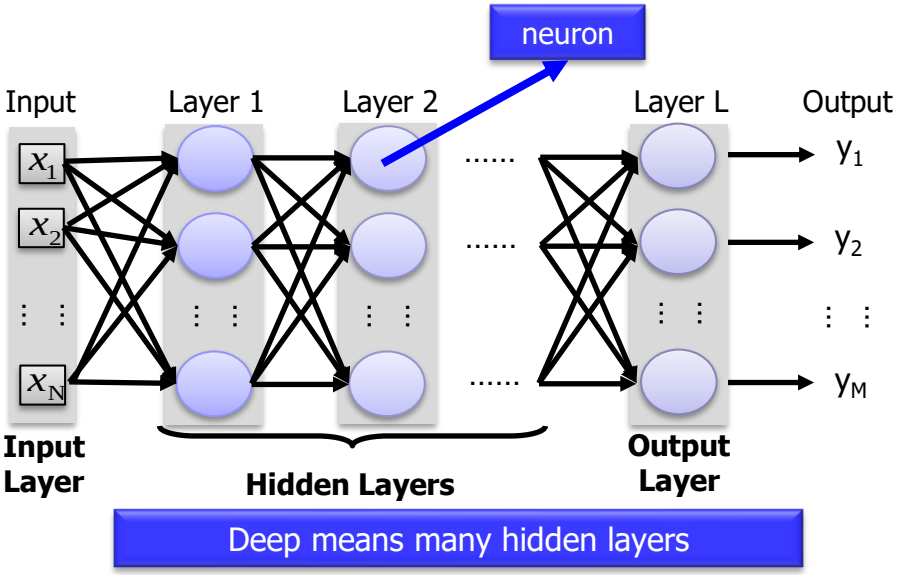
- **Element of Neural network**

Neuron $f: R^K \rightarrow R$

$$z = a_1 w_1 + a_2 w_2 + \cdots + a_K w_K + b$$

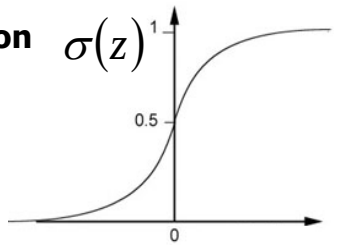


Feed Forwarding for Fully Connected Neural Network



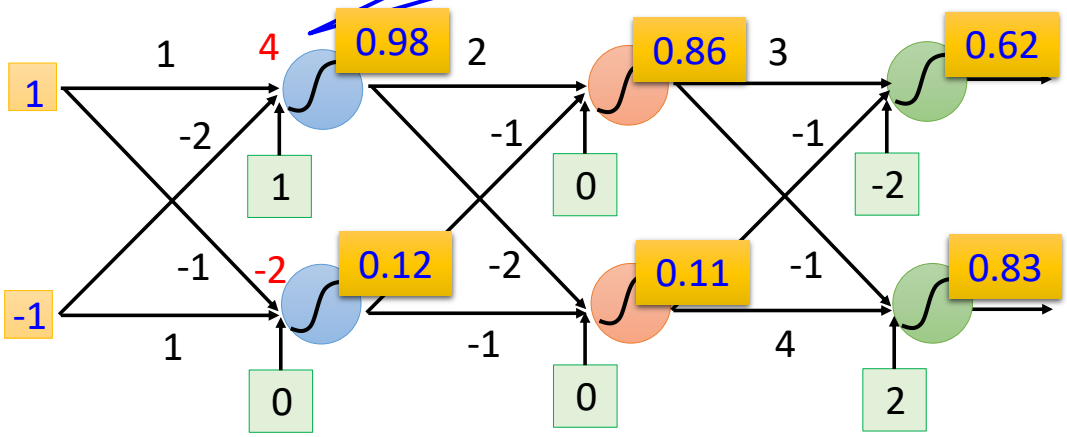
Sigmoid Function

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



$$\sigma\left(\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

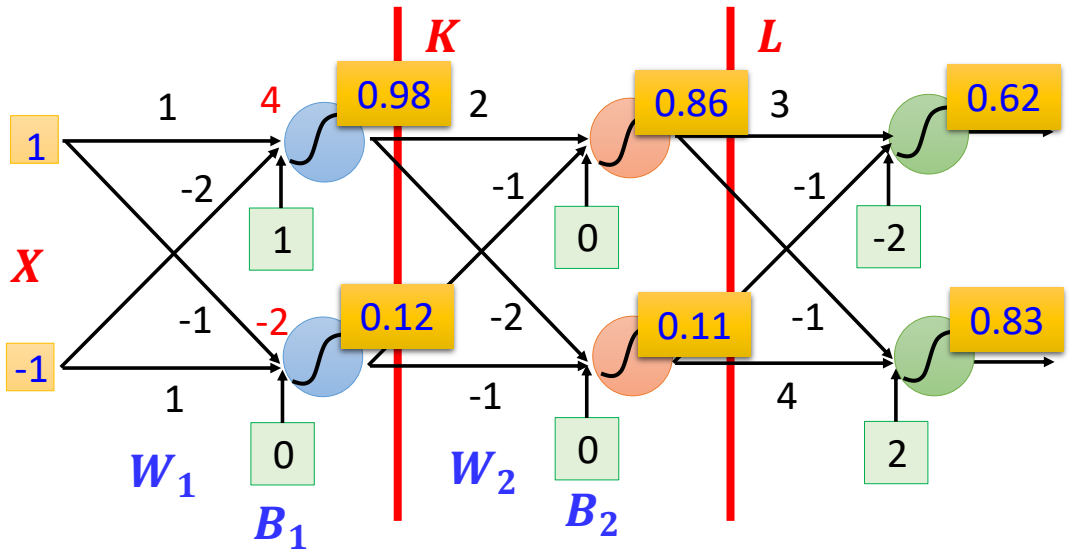
$$\begin{bmatrix} 4 \\ -2 \end{bmatrix}$$



Example of Neural Network

Feed Forwarding

❑ Forward Propagation



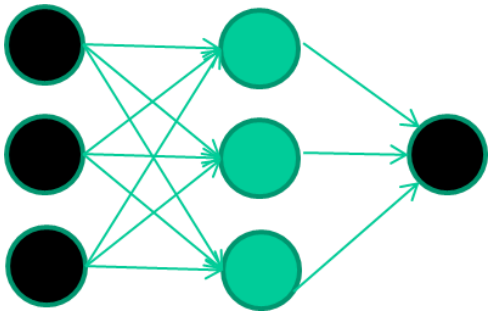
$$K=\text{tf.sigmoid}(\text{tf.matmul}(X,W_1)+B_1)$$
$$L=\text{tf.sigmoid}(\text{tf.matmul}(K,W_2)+B_2)$$

How Do We Train A Multi-Layer Network?

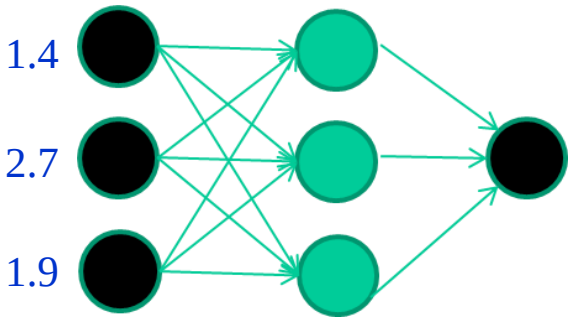
- **Backpropagation**
  - Backpropagation learns by iteratively processing a set of training data (samples).
  - For each sample, weights are modified to minimize the error between network's classification and actual classification.
    - It performs gradient descent to try to minimize the sum squared error between the network's output values and the given target values.

Backpropagation

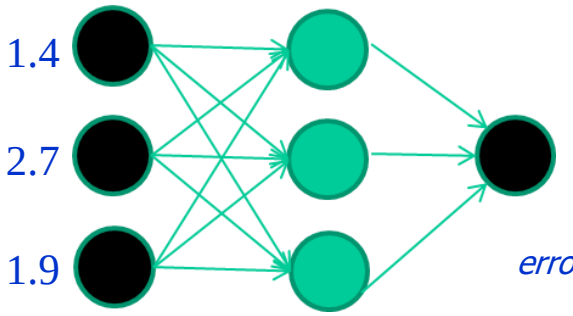
Initialise with random weights



Present a training pattern

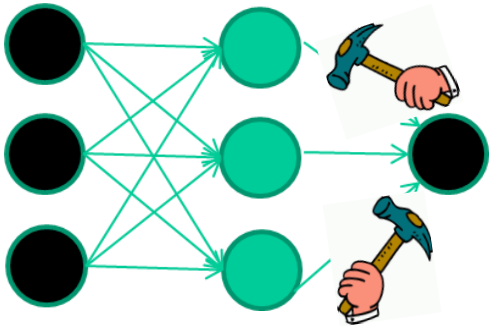


Feed it through to get output

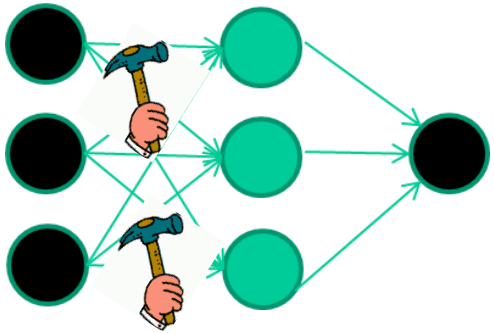


Compare with target output

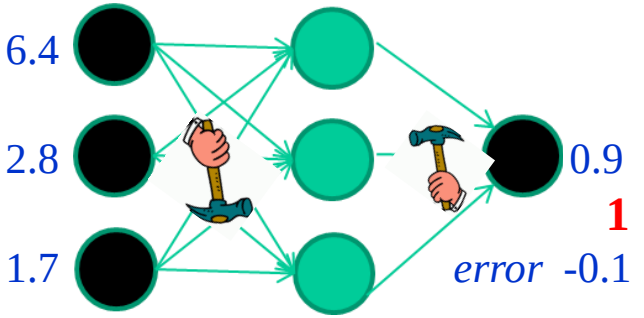
Adjust weights based on error



Adjust weights based on error



Repeat this thousand, maybe millions of times



Backpropagation

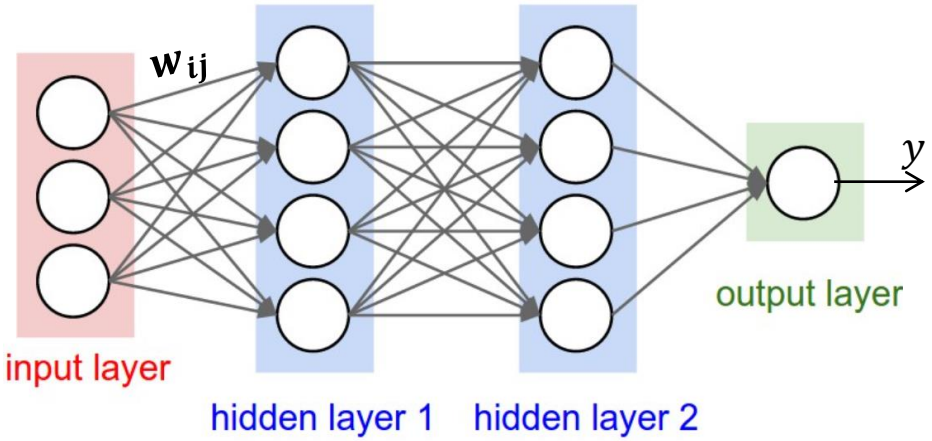
Backpropagation

- **How to optimize w_{ij}?**
  - Choose w_{ij} which minimize the decision (classification) error.

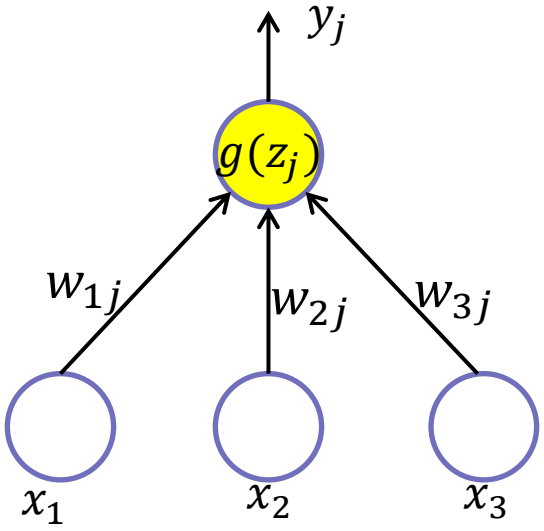
$$E = \frac{1}{2}(y - t)^2, \qquad y = output, t = target$$

  - Updatew_{ij}

$$w_{ij} := w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}} \qquad \alpha = learning\ rate$$



Backpropagation



$$z = WX = \sum_i w_{ij} x_i$$

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

$$g(z)' = g(z)\big(1-g(z)\big) = \frac{e^{-z}}{(1+e^{-z})^2}$$

- **How to calculate $\frac{\partial E}{\partial w_{ij}}$?**

$$E = \frac{1}{2}(y_j - t)^2$$

$$\frac{\partial E}{\partial y_j} = (y_j - t),$$

$$\frac{\partial y_j}{\partial z_j} = \frac{\partial g(z_j)}{\partial z_j} = g(z_j)\big(1-g(z_j)\big)$$

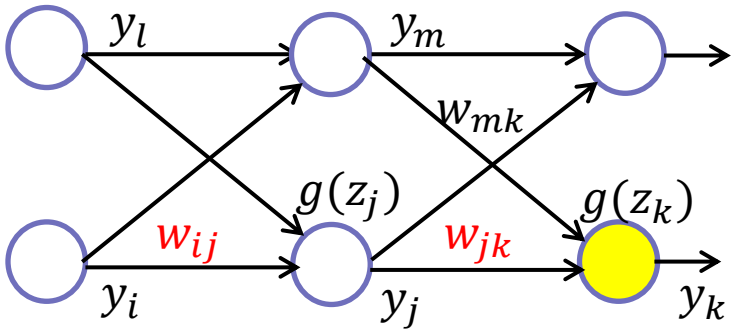
$$\frac{\partial z_j}{\partial w_{ij}} = x_i$$

chain rule

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$
$$= (y_j - t_j)g(z_j)\big(1-g(z_j)\big)x_i$$

All values can be obtained after a sample feed forwarding.

Backpropagation



$$E_k = \frac{1}{2}(y_k - t_k)^2$$

$$\frac{\partial E_k}{\partial y_k} = (y_k - t_k)$$

$$\frac{\partial E_k}{\partial z_k} = \frac{\partial E_k}{\partial y_k}\frac{\partial y_k}{\partial z_k} = \frac{\partial E_k}{\partial y_k}\frac{\partial g(z_k)}{\partial z_k} = (y_k - t_k)g(z_k)'$$

$$\frac{\partial E_k}{\partial w_{jk}} = \frac{\partial E_k}{\partial y_k}\frac{\partial y_k}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}} = (y_k - t_k)g(z_k)'y_j$$

$$\boldsymbol{w_{jk} := w_{jk} - \alpha\frac{\partial E_k}{\partial w_{jk}}} = \boldsymbol{w_{jk} - \alpha(y_k - t_k)g(z_k)'y_j}$$

$$\frac{\partial E_k}{\partial y_j} = \frac{\partial E_k}{\partial z_k}\frac{\partial z_k}{\partial y_j} = (y_k - t_k)g(z_k)'w_{jk}$$

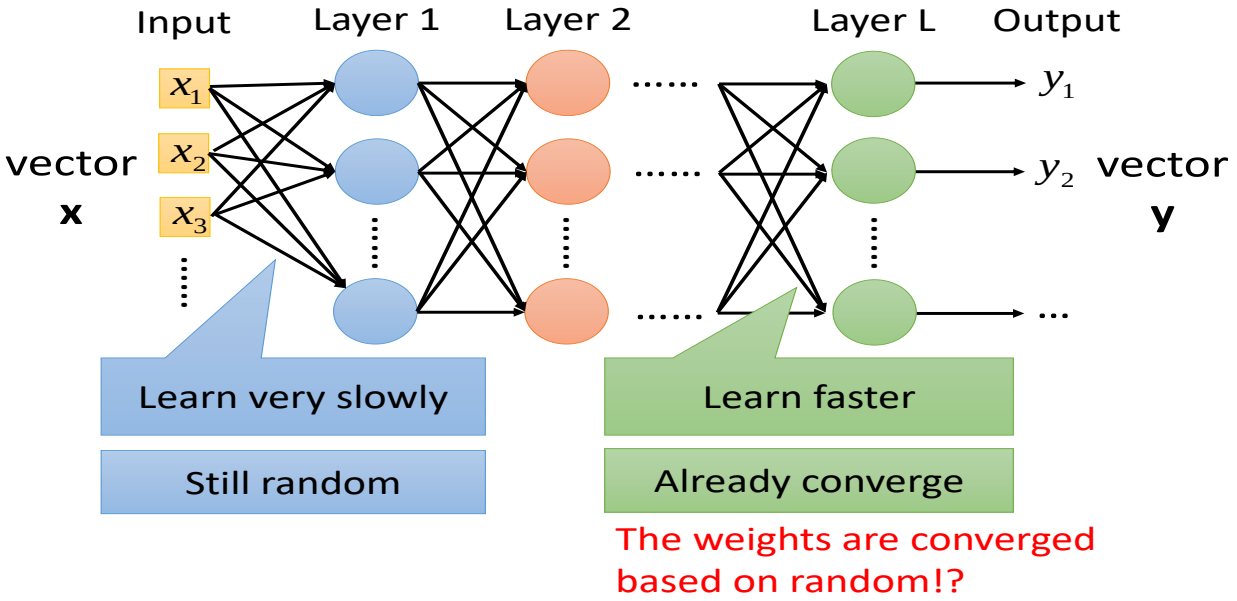
$$\frac{\partial E_k}{\partial z_j} = \frac{\partial E_k}{\partial y_j}\frac{\partial y_j}{\partial z_j} = \frac{\partial E_k}{\partial y_j}\frac{\partial g(z_j)}{\partial z_j} = (y_k - t_k)g(z_k)'w_{jk}g(z_j)'$$

$$\frac{\partial E_k}{\partial w_{ij}} = \frac{\partial E_k}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = (y_k - t_k)g(z_k)'w_{jk}g(z_j)'y_i$$

$$\boldsymbol{w_{ij} := w_{ij} - \alpha\frac{\partial E_k}{\partial w_{ij}}}$$

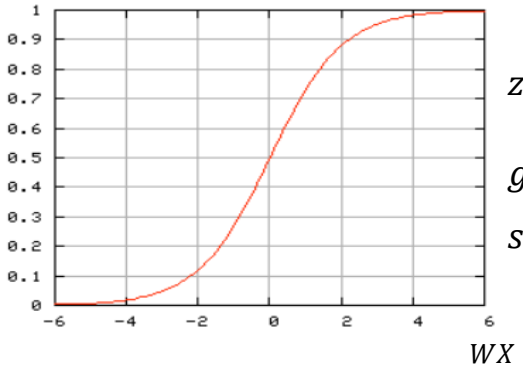
Gradient Vanishing Problems

- **As the neural network has deeper hidden layers, gradient vanishing problem can happen.**
  - During the backpropagation, for sigmoid function, $\sigma'(z)$ always smaller than 1.
  - Error signal is getting smaller and smaller.
  - Input impact on the output very little.



Gradient Vanishing Problems

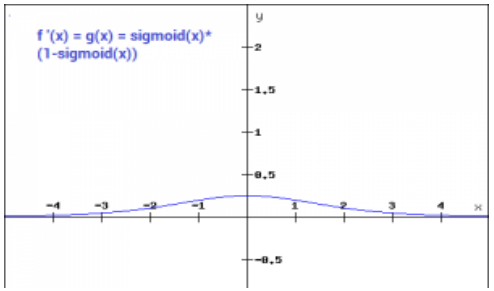
- **Gradient of sigmoid function**



$$z = WX = \sum_i w_{ij} x_i$$

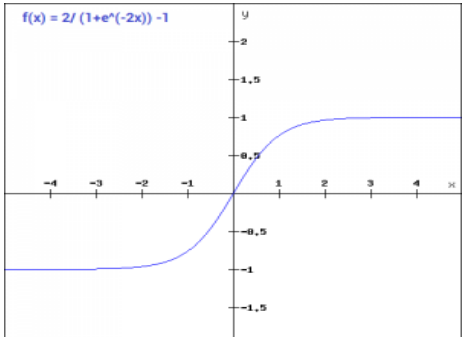
$$g(z) = \frac{1}{1 + e^{-z}}$$

$$sigmoid$$

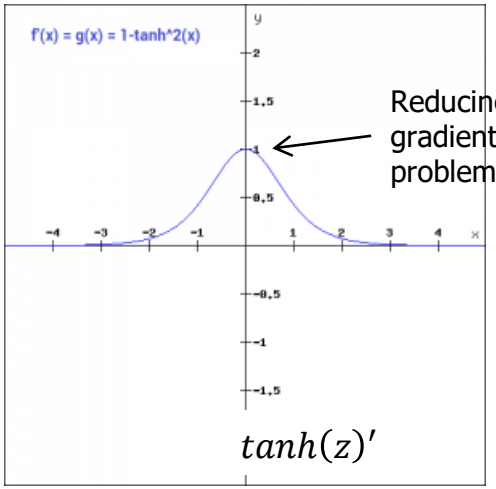


$$g(z)' = g(z)\big(1 - g(z)\big) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

- **tanh() hyperbolic tangent function**
  - It is actually just a scaled version of the sigmoid function.

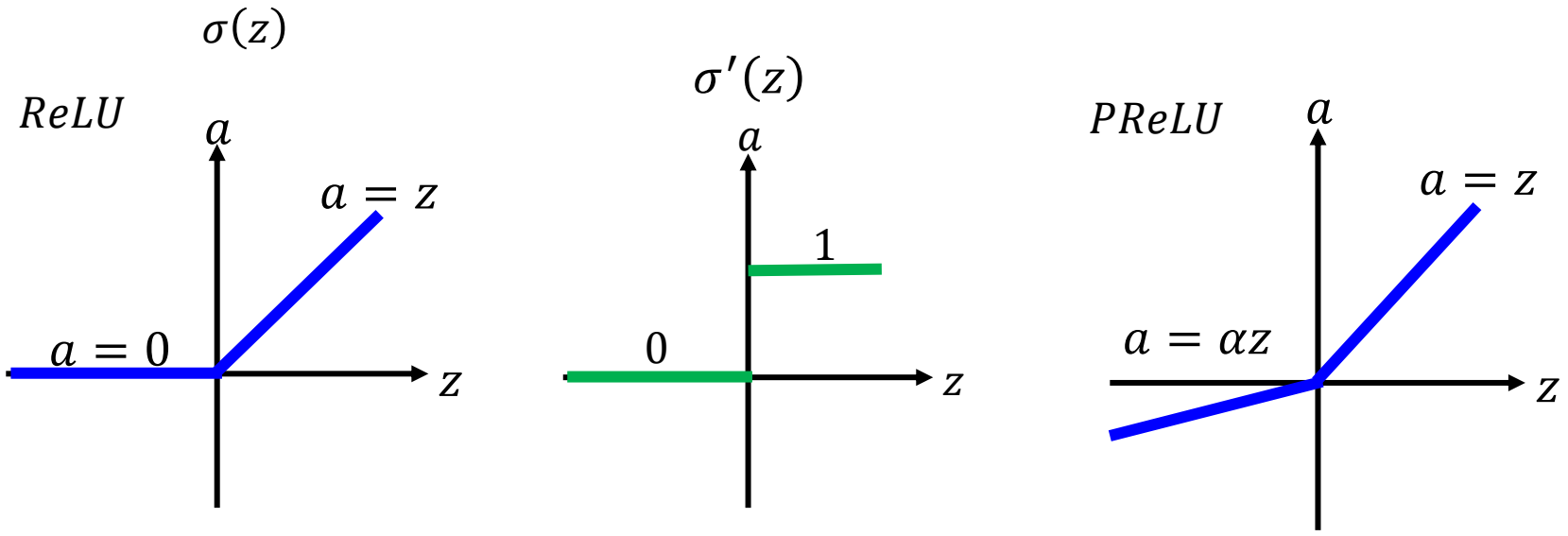


$$tanh(z) = 2sigmoid(2z) - 1$$



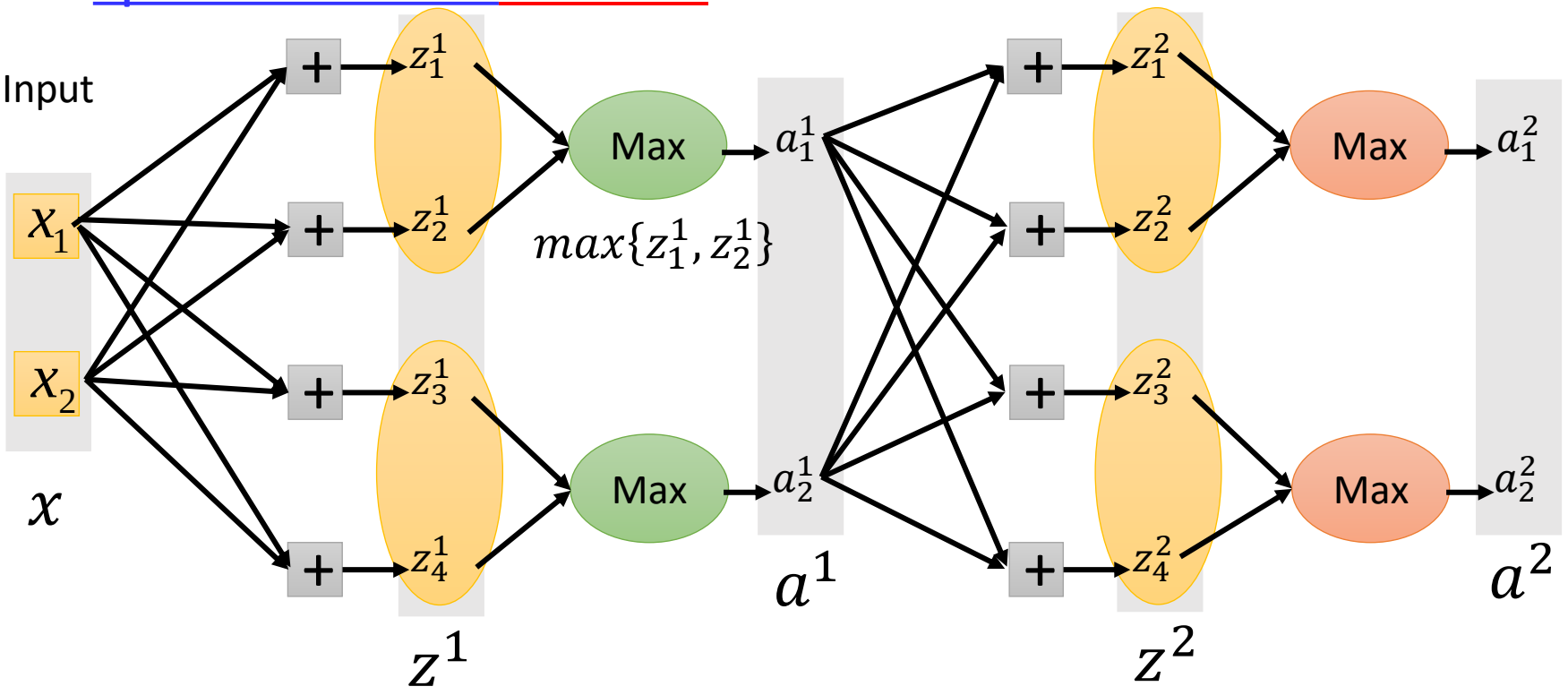
ReLU : Against Vanishing Gradient

- **Rectified Linear Unit (ReLU)**: [Hinton 2010]
  - To avoid vanishing gradient problem
  - Guarantee faster computation for computing derivation.
  - It offers the better performance and generalization in deep learning compared to the Sigmoid and tanh activation functions

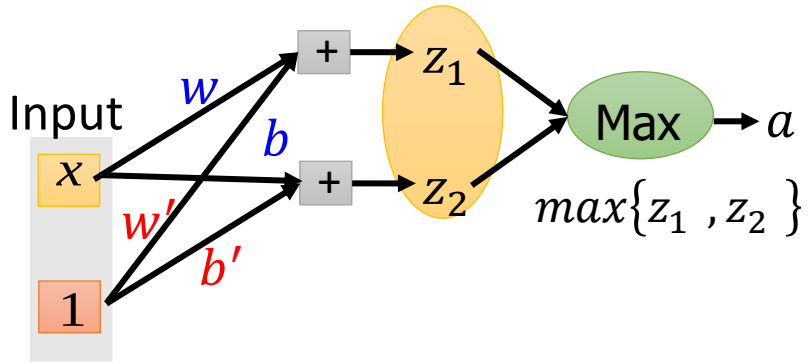


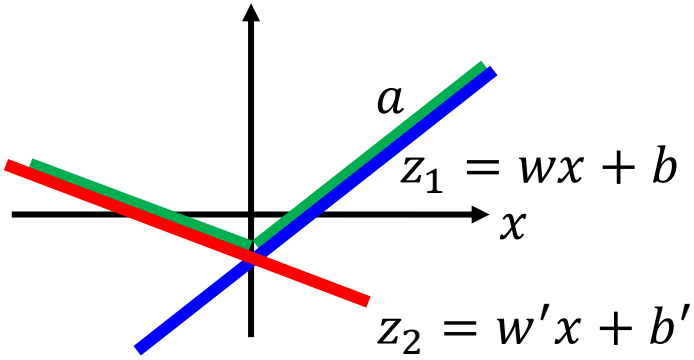
Maxout : against Vanishing Gradient

- Maxout [Goodfellow 2013]: ReLU or PReLu is just a special case of Maxout

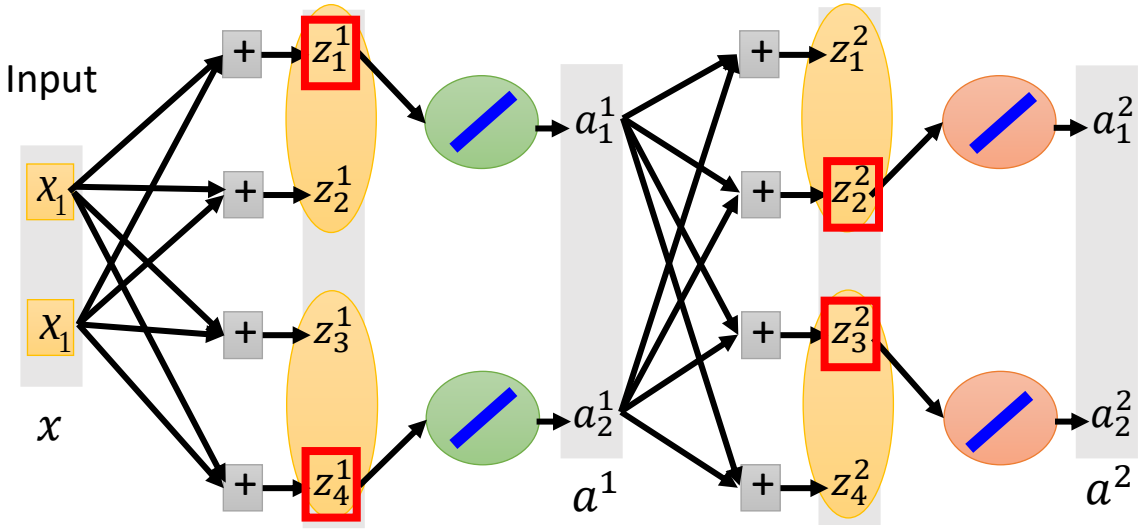


Maxout : against Vanishing Gradient



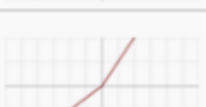


Training by Backpropagation



- In forward pass, we know which z would be the max.
- In backward pass, treat it as network whose neurons having linear activation functions .

Activation Functions

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Why Non-linear Activation Function

- **ReLU is non-linear activation function**
  - Looks like linear but for $z < 0$, $\sigma(z) = 0$ so that it is non-linear
- **If a multilayer perceptron has a linear activation function in all neurons,**
  - That is, a linear function that maps the weighted inputs to the output of each neuron,
  - Then linear algebra shows that any number of layers can be reduced to a two-layer input-output model.
  - There is no meaning having multi-layer perceptron.

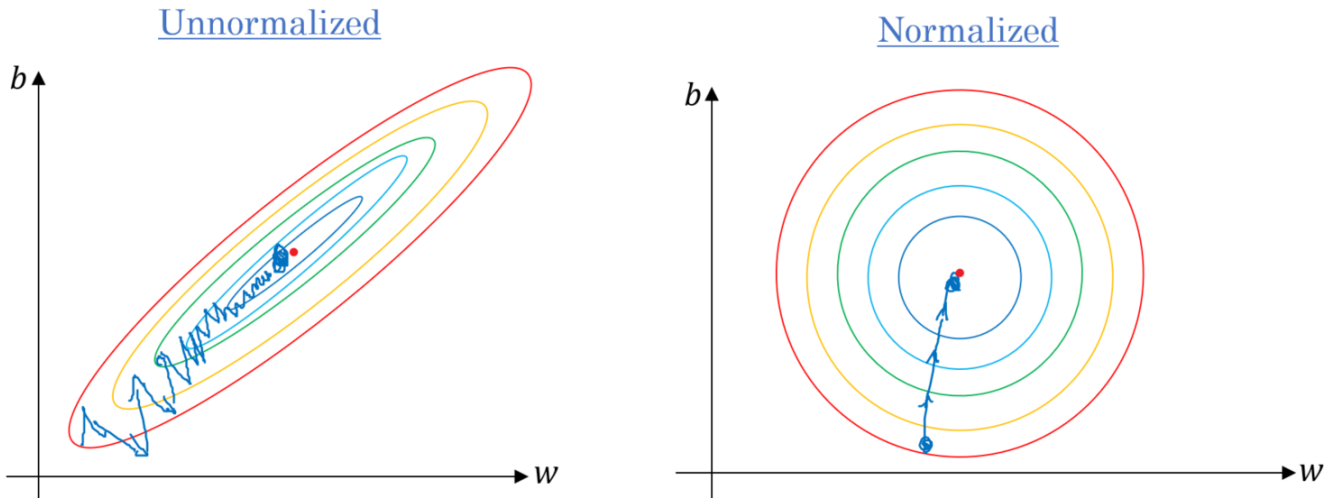
$$f(x) = ax + b$$
$$g(y) = cy + d$$
$$g(f(x)) = c(ax + b) + d = (ca)x + (cb + d)$$

Optimization for Training Deep Models

- Data Normalization
- Parameter Updating (SGD, Batch GD, Mini-batch GD)
- Gradient Descent with Momentum
- Gradient Descent with Adaptive Learning Rate
- Regularization and Dropout
- Parameter Initialization
- Hyperparameters for Learning

Data Normalization

- **Make sure to scale the data if it's on a very different scales.**
  - Normalization is a technique often applied as part of data preparation for machine learning.
  - If we don't scale the data, the level curves (contours) would be narrower and taller which means it would take longer time to converge.



Data Normalization

- **Feature scaling**
  - A method used to normalize the range of independent variables or features of data. In data processing, it is also known as data normalization and is generally performed during the data preprocessing step.
- **Normalization: rescales the values into a range of [0,1]**
  - Min-max normalization

[0,1] $$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

[a,b] $$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

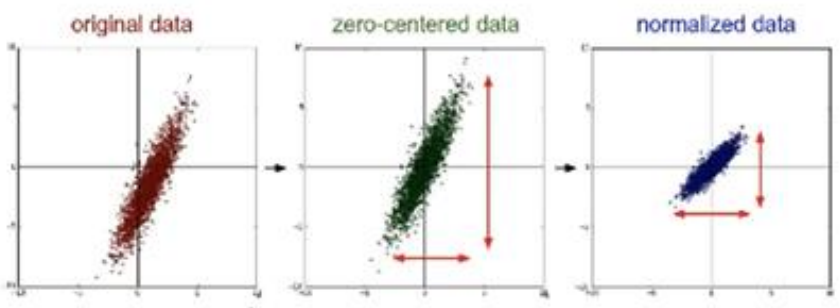
  - Mean normalization

$$x' = \frac{x - \text{average}(x)}{\max(x) - \min(x)}$$

Data Normalization

Standardization

- rescales data to have a mean of 0 and a standard deviation of 1 (unit variance).



$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Standardization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Parameter Updating: Stochastic Gradient Descent (SGD)

- Parameter updating with gradient descent

$$W := W - \alpha \frac{\partial}{\partial W} J_W(W) = W - \alpha \frac{\partial}{\partial W} cost(W, \boldsymbol{x}, \boldsymbol{y})$$

- **Stochastic Gradient Descent (SGD)**
  - Performs the parameters update on each example (x_i, y_i). Therefore, learning happens on every data example:
  - Shuffle the training data set to avoid pre-existing order of examples.
  - $W := W - \alpha \frac{\partial}{\partial W} cost(W, x_i, y_i)$

```
for i in range(num_epochs):
    np.random.shuffle(data)
    for example in data:
        grad = compute_gradient(example, params)
        params = params — learning_rate * grad
```

Parameter Updating: Batch Gradient Descent

Batch Gradient Descent (BGD)

- Batch Gradient Descent is when we sum up over all examples on each iteration when performing the updates to the parameters.

$$W := W - \alpha \frac{\partial}{\partial W} \left\{ \frac{1}{N} \sum_{i=1}^{N} cost(W, x_i, y_i) \right\}$$

- **Advantages:**
  - It has straight trajectory towards the minimum and it is guaranteed to converge in theory to the global minimum if the loss function is convex and to a local minimum if the loss function is not convex.
  - It has unbiased estimate of gradients. The more the examples, the lower the standard error.

Parameter Updating: Batch Gradient Descent

- **Disadvantages:**
  - Even though we can use vectorized implementation, it may still be slow to go over all examples especially when we have large datasets.
  - Each step of learning happens after going over all examples where some examples may be redundant and don't contribute much to the update.
- Could compute gradient over entire data set on each step, but this turns out to be unnecessary.
- Computing gradient on small data samples works well.

```
for i in range(num_epochs):
    grad = compute_gradient(data, params)
    params = params — learning_rate * grad
```

Parameter Updating: Min-batch Gradient Descent

Mini-batch Gradient Descent

- Instead of going over all examples, Mini-batch Gradient Descent sums up over lower number of examples based on the batch size.
  Therefore, learning happens on each mini-batch of *m* examples:

$$W := W - \alpha \frac{\partial}{\partial W} \left\{ \frac{1}{m} \sum_{i=1}^{m} cost(W, x_i, y_i) \right\}$$

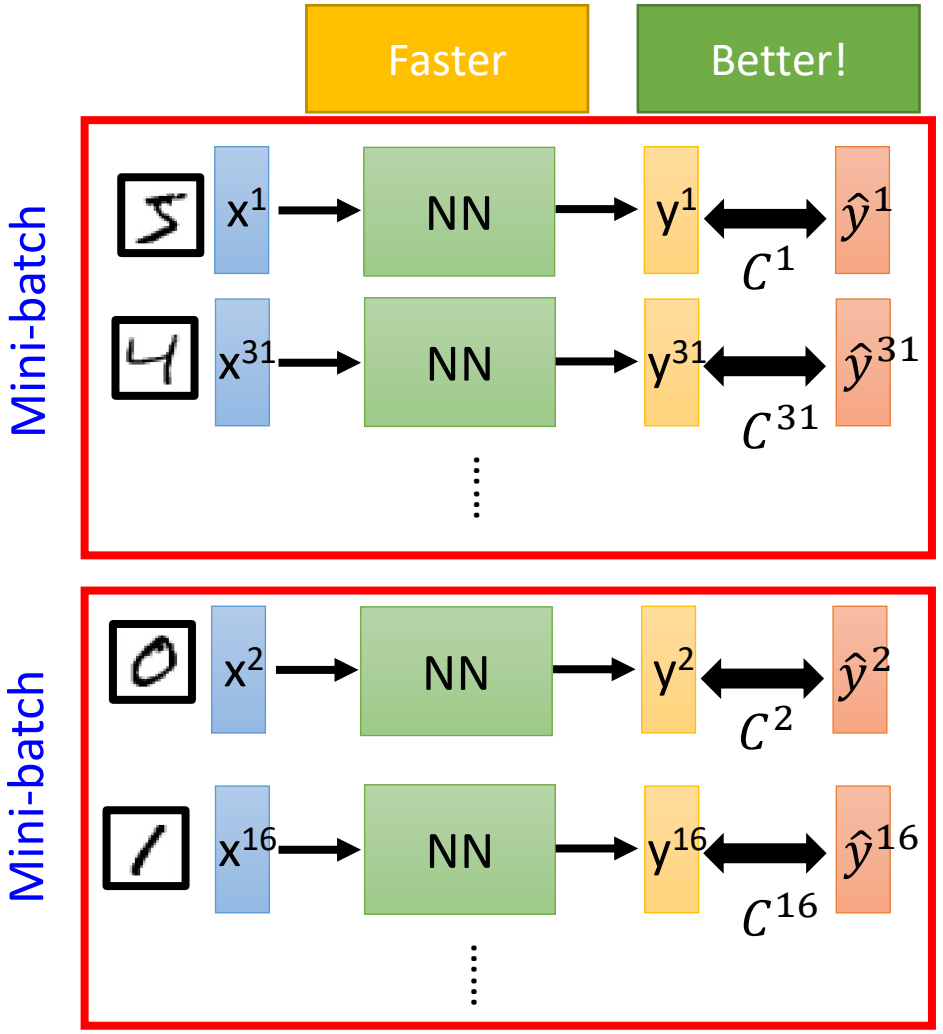
- Shuffle the training data set to avoid pre-existing order of examples.
- Partition the training data set into *b* mini-batches based on the batch size.
  - The batch size is something we can tune. It is usually chosen as power of 2 such as 32, 64, 128, 256, 512, etc. The reason behind it is because some hardware such as GPUs achieve better run time with common batch sizes such as power of 2.

Parameter Updating: Min-batch Gradient Descent

- **Advantages:**
  - Faster than Batch version.
  - Randomly selecting examples will help avoid redundant examples or examples that are very similar that don't contribute much to the learning.
  - With batch size < size of training set, it adds noise to the learning process that helps improving generalization error.
- **Disadvantages:**
  - It won't converge. On each iteration, the learning step may go back and forth due to the noise. Therefore, it wanders around the minimum region but never converges.

```
for i in range(num_epochs):
    np.random.shuffle(data)
    for batch in radom_minibatches(data, batch_size=32):
        grad = compute_gradient(batch, params)
        params = params — learning_rate * grad
```

Parameter Updating: Min-batch Gradient Descent



- Randomly initialize θ^0
- Pick the 1st batch

  $C = C^1 + C^{31} + \cdots$

  $\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$
- Pick the 2nd batch

  $C = C^2 + C^{16} + \cdots$

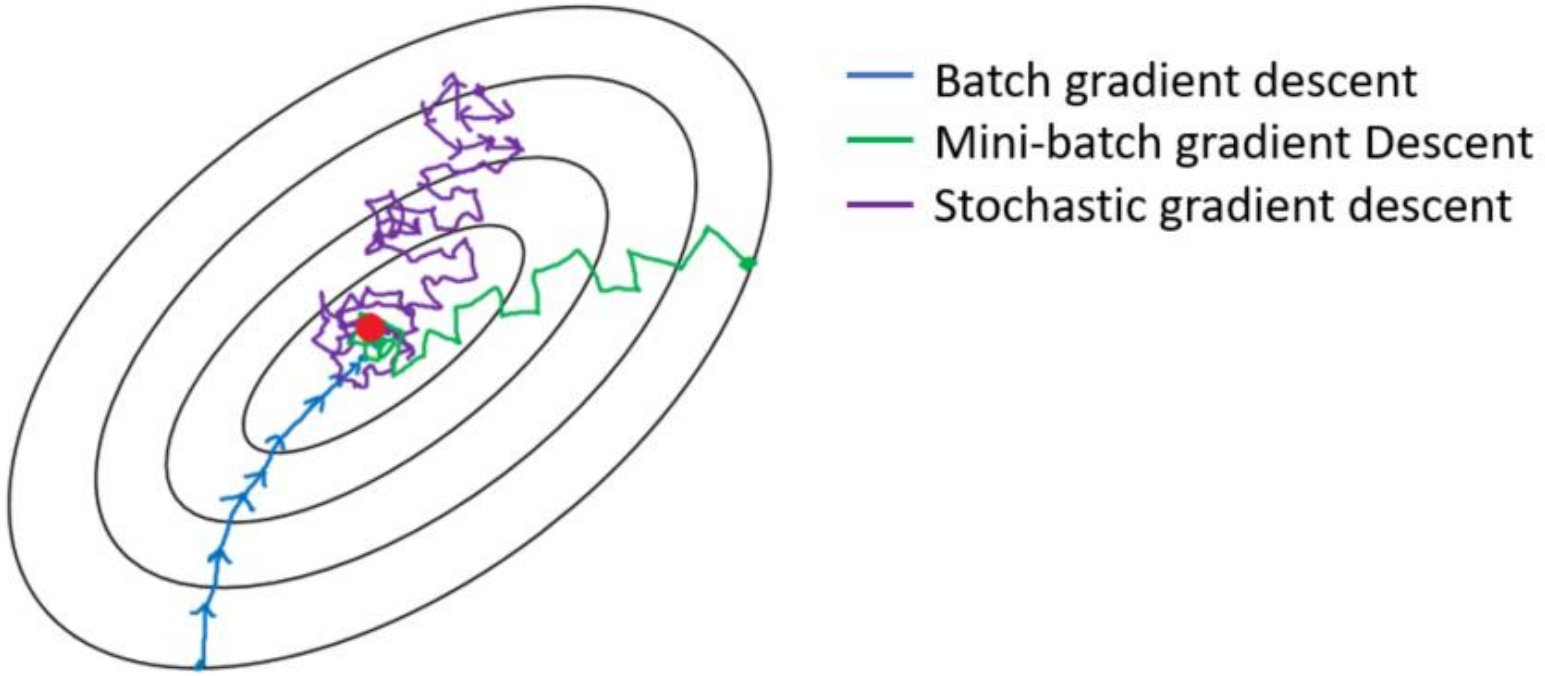
  $\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$

  ⋮
- Until all mini-batches have been picked

one epoch
Repeat the above process

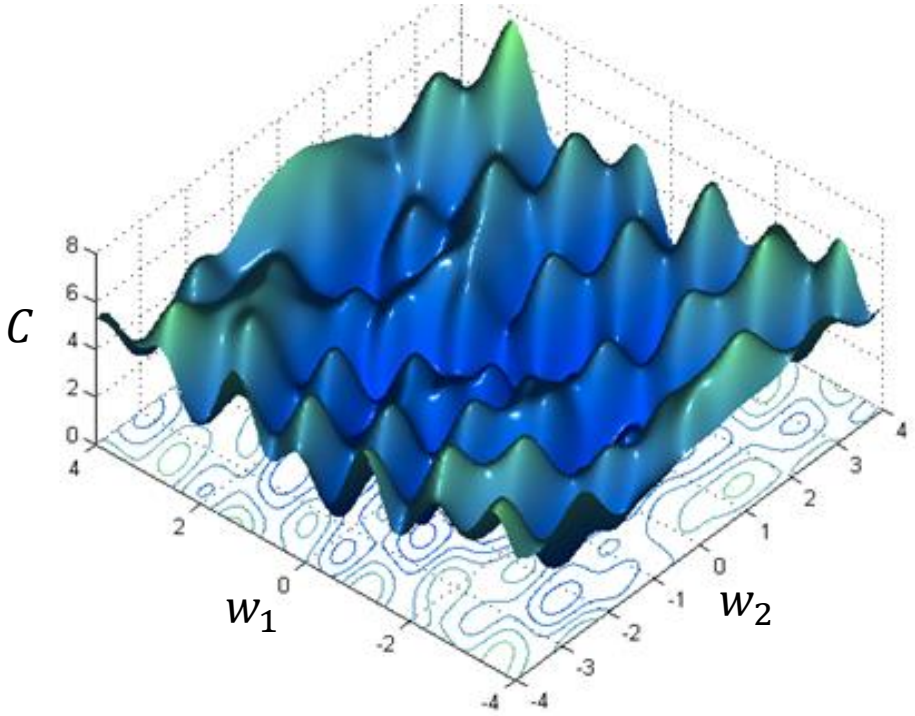
Parameter Updating

- SGD, Batch GD, Mini-batch GD



Gradient Descent: Local Minimum

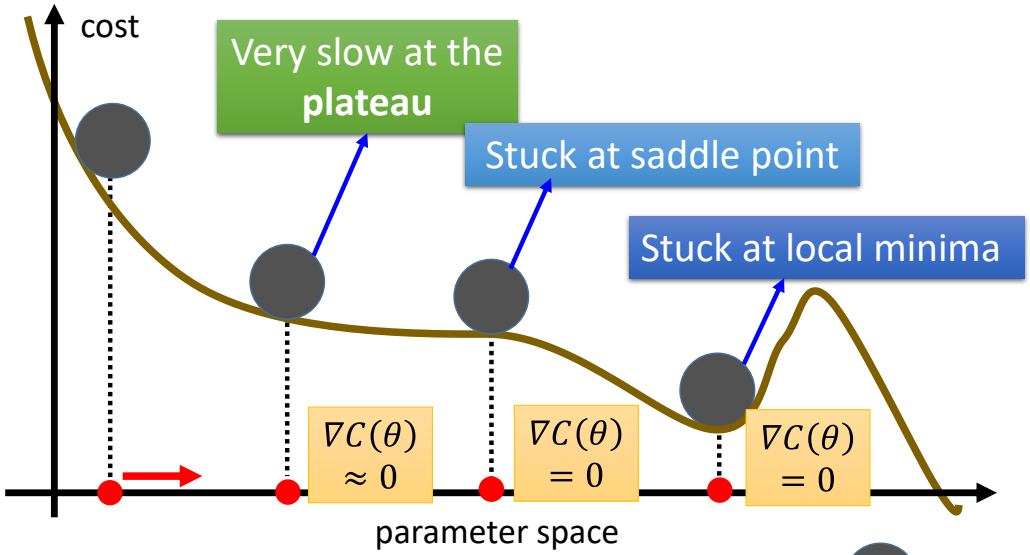
- **Gradient descent never guarantees global minimum.**



Different initial point θ^0

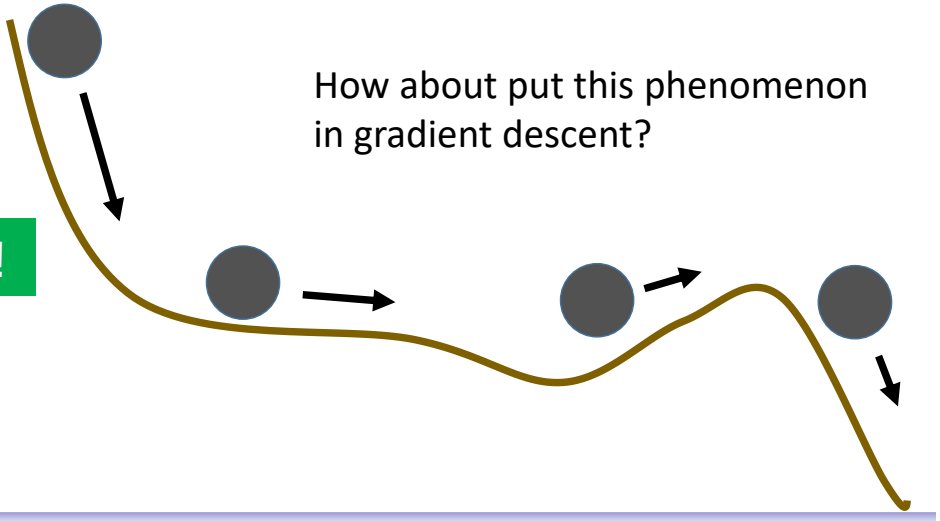
Reach different minima, so different results

Gradient Descent: Local Minimum



How about put this phenomenon in gradient descent?

Momentum!

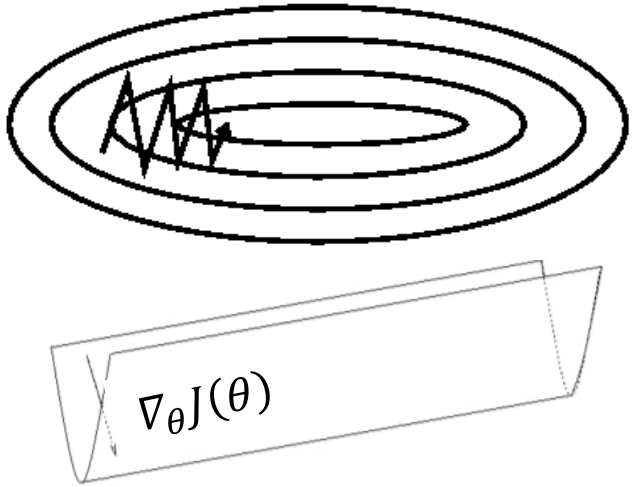


Stochastic Gradient Descent: Problems

- **Motivation:**
  - SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another.

$$\boldsymbol{\theta = \theta - \eta \nabla_\theta J(\theta)}$$

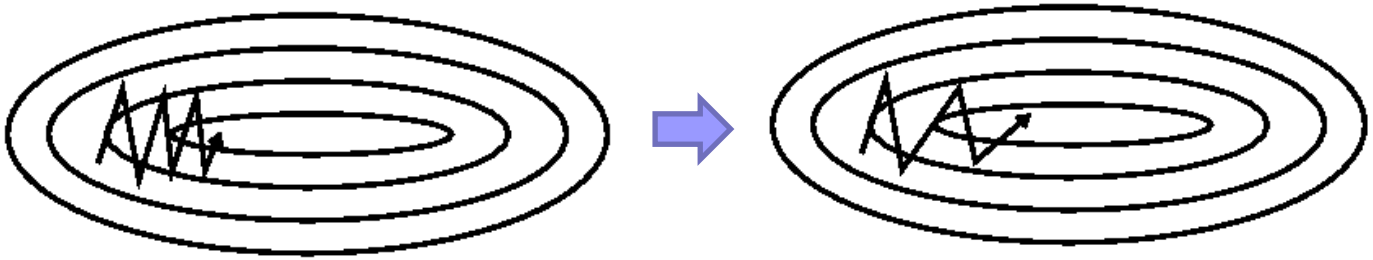
$\theta: parameters\ (deep\ NN\ weights)$
$\eta: learning\ rate$
$J(\theta): loss\ function$



  - Gradient descent does:
    - very slow progress along shallow dimension, jitter along steep direction.
  - SGD may reach at local minima or saddle-point.
  - SGD may gets stuck with zero gradient.
  - Saddle point is much more common in high dimensions.

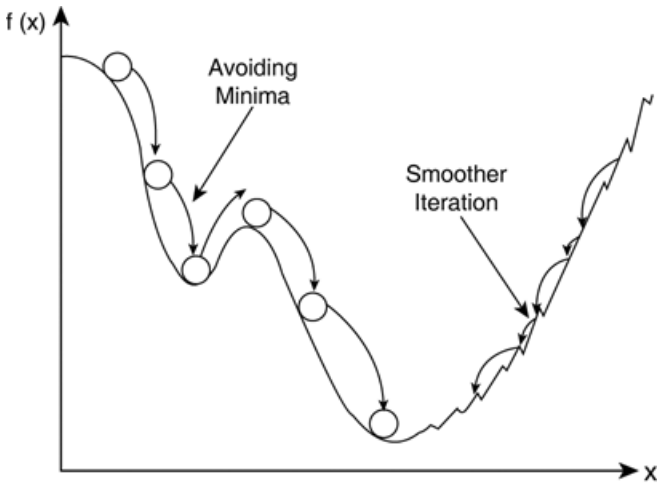
GD Optimization: SGD with Momentum

- **Adding momentum:**
  - With Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient (same directon).



θ: $parameters\ (deep\ NN\ weights)$
η: $learning\ rate$
$J(\theta)$: $loss\ function$
γ: $momentum\ term\ (typically\ 0.9)$

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

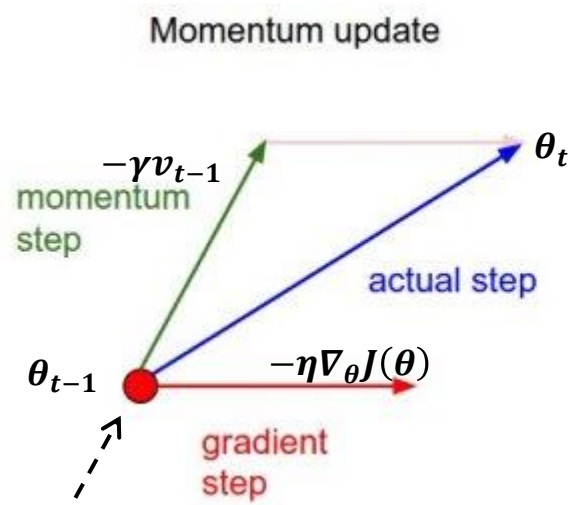


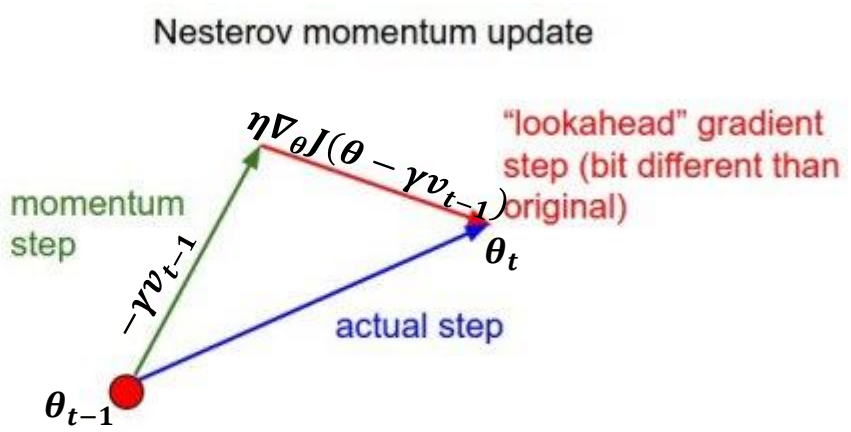
GD Optimization: NAG

Nesterov accelerate gradient (NAG) method

- We know that we will use our momentum term (γv_{t-1}) to move the parameters.
- Computing ($\theta - \gamma v_{t-1}$) gives us a rough idea where our parameters are going to be.
- Computing the gradient w.r.t. the approximate future position of our parameters.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$





GD Optimization: AdaGrad

- **Adaptive Gradient (AdaGrad)** [Duchi et al., 2011]
  - **Adaptively determine the learning rate.**
    - SGD, momentum, NAG use a constant learning rate.
    - Step size (learning rate) is equal for every iteration and for every parameters ($(\theta_1, \theta_2, \cdots, \theta_p)$
  - Divide the fixed learning rate by "***average" gradient***
  - At $(t+1)$th update time
    - $\boldsymbol{G_t} = average\ squred\ gradient\ of\ parameter \boldsymbol{\theta}\ at\ t$

$$\boldsymbol{G_t} := \boldsymbol{G_{t-1}} + \left(\boldsymbol{\nabla_\theta J(\theta_t)}\right)^2$$

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta_t} - \frac{\eta}{\sqrt{\boldsymbol{G_t} + \varepsilon}} \boldsymbol{\nabla_\theta J(\theta_t)}$$

  - If $\boldsymbol{\theta}$ has small average gradient → Larger learning rate
  - If $\boldsymbol{\theta}$ has large average gradient → Smaller learning rate

GD Optimization: AdaGrad

- **Advantages**
  - Does not need to carefully concern about the learning rate.
  - Typically set the fixed learning rate η=0.01.
- **Disadvantages**
  - In deep learning, AdaGrad can result in premature and excessive decrease in the effective learning rate.
    - Because the effective learning rate $\frac{\eta}{\sqrt{G_t+\varepsilon}}$ is inversely proportional to G_t and G_t is monotonically increasing function.
    - $G_t := G_{t-1} + \left(\nabla_\theta J(\theta_t)\right)^2$, the squared gradient is always positive and added to the previous value.
    - Therefore, eventually the effective learning rate is very close to zero.

GD Optimization: RMSProp

- **RMSProp** [Hinton, 2012]
  - Modifies AdaGrad using exponential decay in accumulation (exponential average)
  - Better than Adagrad in non-convex problems

$$G_t := \gamma G_{t-1} + (1-\gamma)\left(\nabla_\theta J(\theta_t)\right)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \nabla_\theta J(\theta_t)$$

GD Optimization: Adam

- **Idea: Take advantages of momentum and adaptive learning rate approaches**
  - Estimate the first moment and the second moment of gradients to do the update.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\,\nabla_\theta J(\theta_t)$$

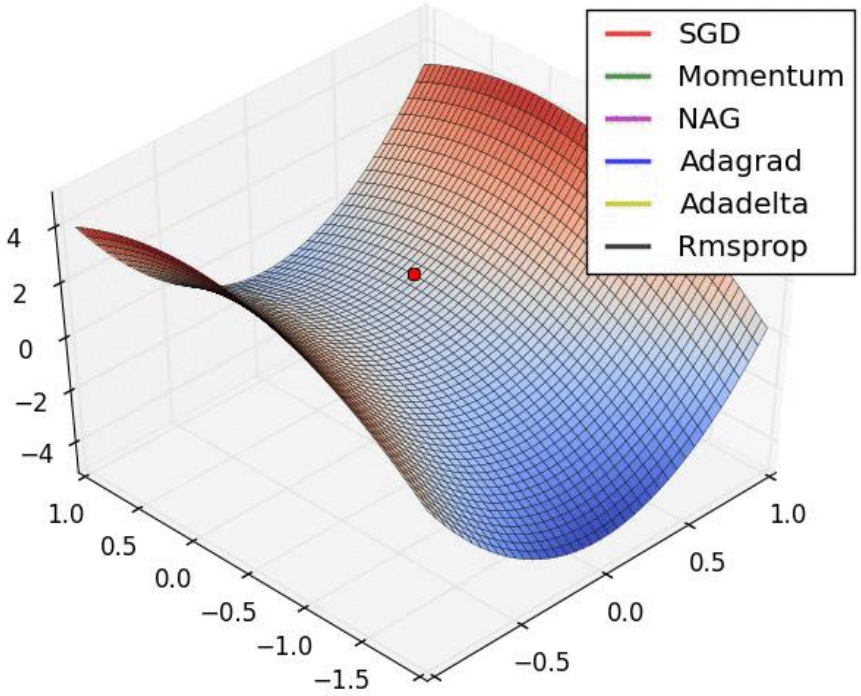
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_\theta J(\theta_t)\right)^2$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t},\ \widehat{v_t} = \frac{v_t}{1-\beta_2^t}$$

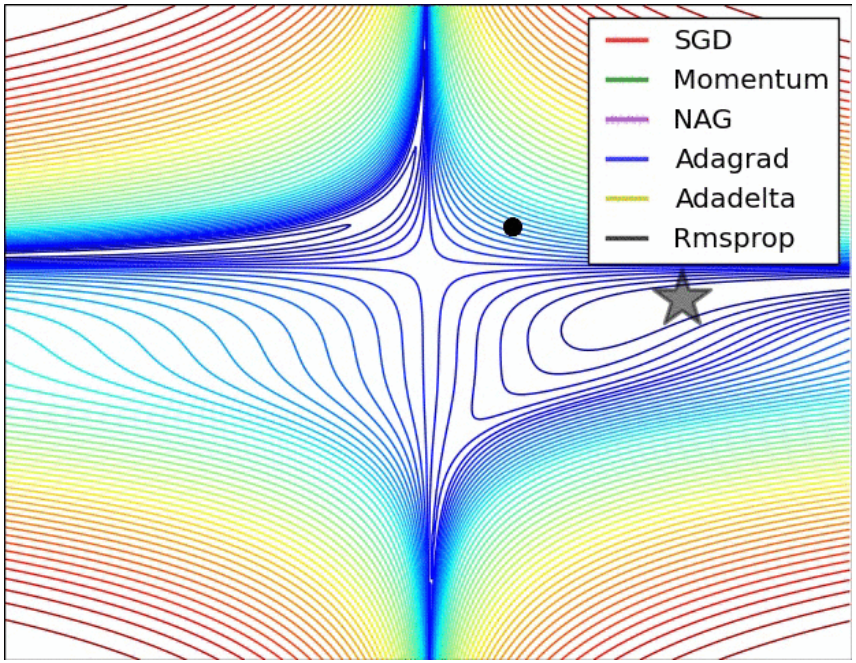
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v_t} + \varepsilon}}\widehat{m_t}$$

  - m_t: to capture the exponential average of momentum
  - v_t : to change the learning rate dynamically
  - Usually β_1=.9, β_2=.999, ε=10^{-8} ,η=10^{-3}

Gradient Descent Optimizations



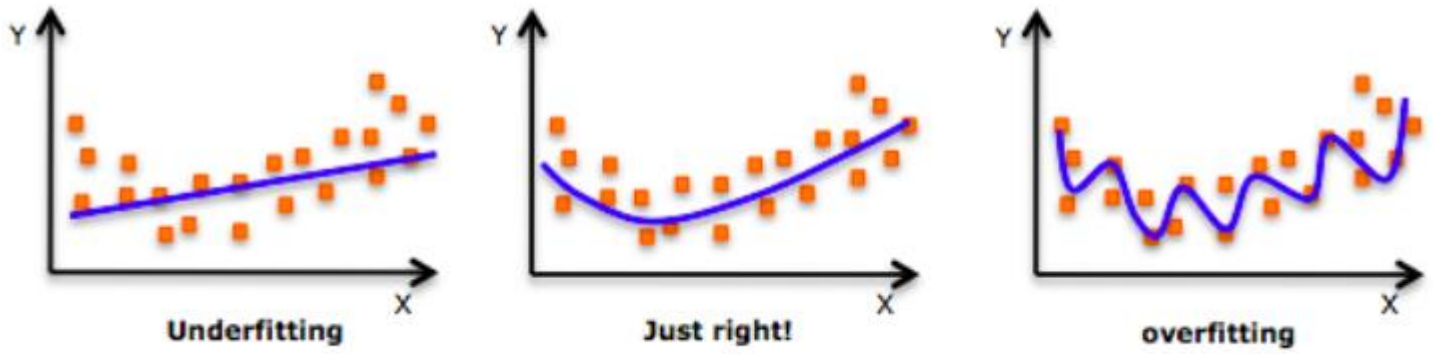
Gradient Descent Optimization Algorithms
at Long Valley



Gradient Descent Optimization Algorithms
at Beale's Function

Overfitting

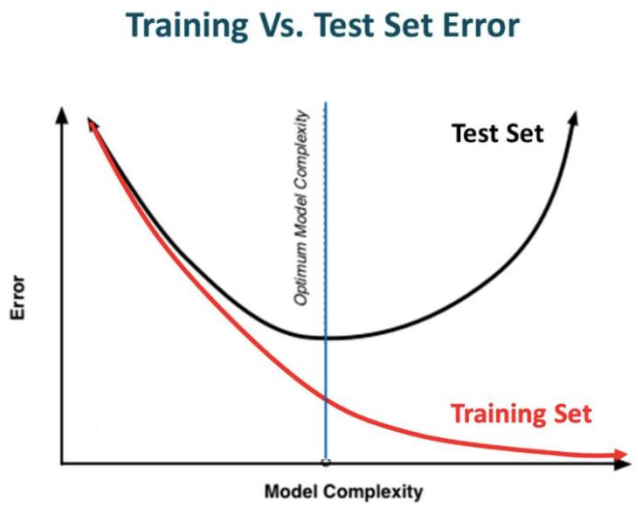
- Overfitting problem?

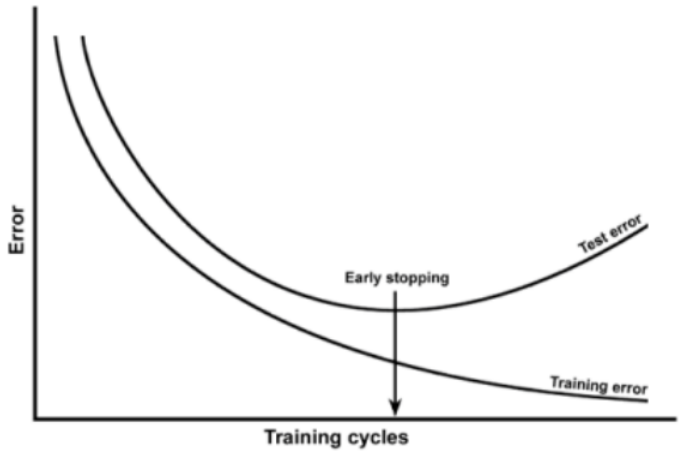


- One of the most common problem data science professionals face is to avoid overfitting.
  - A situation where your model performed exceptionally well on train data, but was not able to predict test data.
- As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, which *ultimately results in poor performance on the unseen data*.

Overfitting

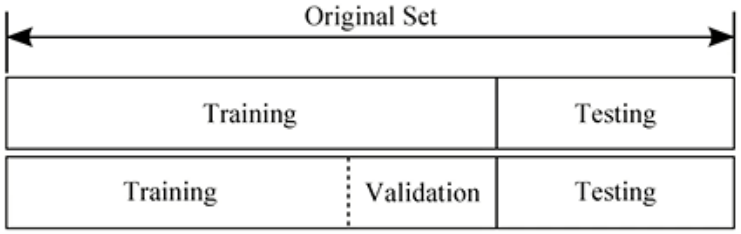
- In other words, while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't.
- Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.

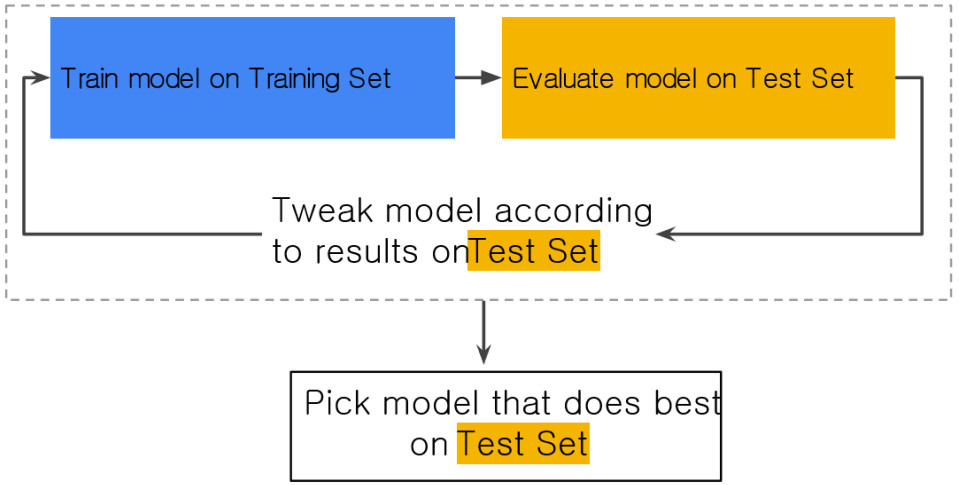




Validation Set

Training and test sets





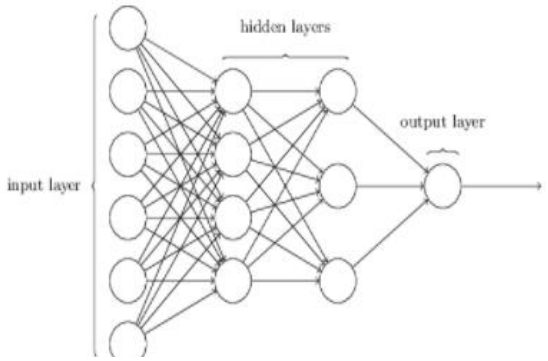
- First, it splits the data into 70% training data and the rest. Then, the rest is again splitted into a validation data set (33%, 10% of the total data) and the test data (66%, 20% of the total data).
- A **validation dataset** is a sample of data held back from training your model that is used to give an estimate of model skill while tuning model's hyperparameters.

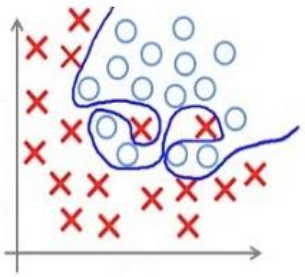
Validation Set

- **Early stopping** is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model. This is known as early stopping.
  - In the above image, we will stop training at the dotted line since after that our model will start overfitting on the training data.

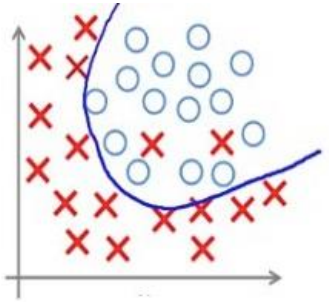
Regularization

- **Regularization penalizes the coefficients. In deep learning, it actually penalizes the weight matrices of the nodes.**
  - Assume that our regularization coefficient is so high that some of the weight matrices are nearly equal to zero.

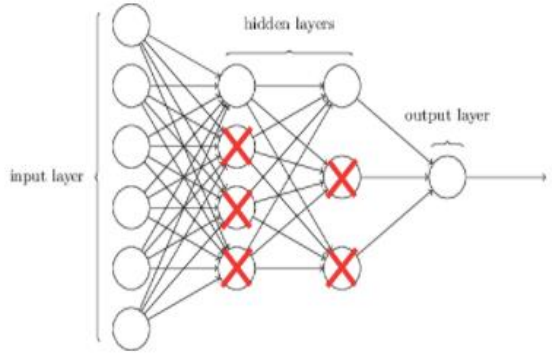


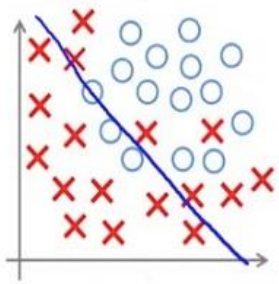


Over-fitting



Appropriate-fitting





Under-fitting

Such a large value of the regularization coefficient is not that useful. We need to optimize the value of regularization coefficient in order to obtain a well-fitted model as shown in the image below.

Regularization

Regularization

- Large values of W makes big fluctuations.
  - If weights are large, a small change in a feature can result in a large change in the prediction also gives too much weight to any one feature.
- A regularizer is an additional criteria to the loss function to make sure that we don't overfit.
  - Learning objective: Finding optimal parameters (W, b)

$$Cost\ function = Loss\ function + Regularization\ term$$

$$(W^*, b^*) = \min_{W,b}\{Loss(W, b) + \lambda \cdot Penalty(W)\}$$

λ : regularization parameter

Regularization

- L_p regularizer: $\left(\sum_i |w_i|^p\right)^{\frac{1}{p}}$
- L2 Regularization

$$(W^*, b^*) = \min_{W,b} \{Loss(W, b) + \lambda \|W\|^2\}$$

$$Loss(W, b) = \frac{1}{m} \sum_{i=1}^{m} \left[WX^{(i)} + b - t^{(i)}\right]^2$$

$$\|W\|^2 = w_1^2 + w_2^2 + \cdots +$$

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

$$W := W - \alpha \frac{\partial}{\partial W} \{Loss(W) + \lambda \|W\|^2\}$$

$$W := W(1 - 2\alpha\lambda) - \alpha \frac{\partial}{\partial W} Loss(W)$$

Since $W(1 - 2\alpha\lambda) < W$, L2 regularization results in "**weight decay**"

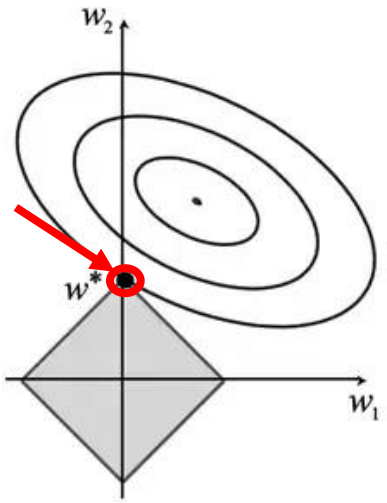
Regularization

- **L1 regularization**

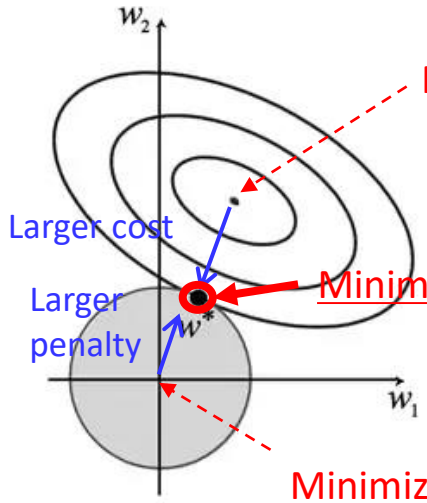
$$(W^*, b^*) = \min_{W,b}\{Loss(W, b) + \lambda \|W\|\}$$

  - Unlike L2, the weights may be reduced to zero here. **Hence, it is very useful when we are trying to compress our model (reduce feature dimension, feature selection). Otherwise, we usually prefer L2 over it.**

The optimal W that minimizes combination makes $w_1 = 0$, so that the feature x_1 is not considered.



L1



L2

Minimizes cost term

Minimizes combination

Minimizes regularization penalty term

L2 Regularization Example

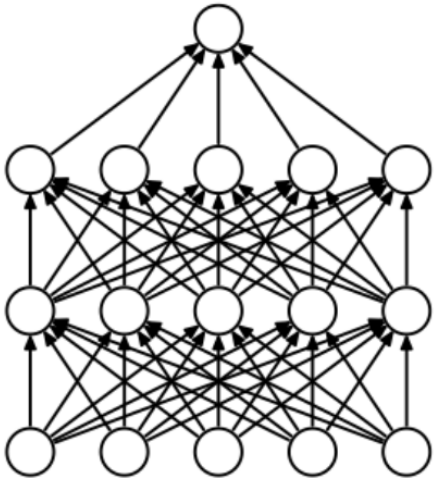
Tensorflow

```
out_layer = tf.matmul(hidden_layer, out_weights) + out_biases
#our real output is a softmax of prior result


#and we also compute its cross-entropy to get our loss
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(out_layer,
tf_train_labels) + 0.01*tf.nn.l2_loss(out_weights))


#now we just minimize this loss to actually train the network
optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
```

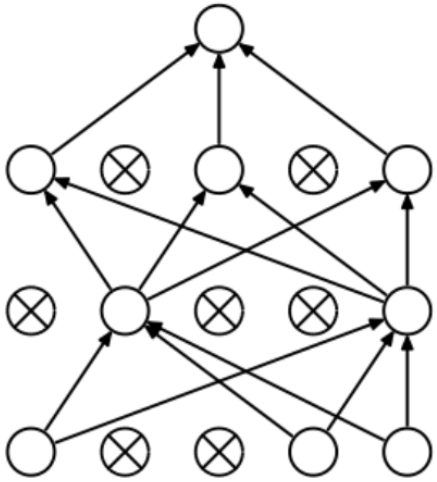
Dropout: Against Overfitting

- At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections.
- So each iteration has a different set of nodes and this results in a different set of outputs. **It can also be thought of as an ensemble technique in machine learning.**
  - Ensemble models usually perform better than a single model as they capture more randomness. Similarly, dropout also performs better than a normal neural network model.
  - Whenever, training data is applied some nodes of neural networks randomly removed with $P_{dropout}$.
  - Typical $P_{dropout}$ is 0.5.

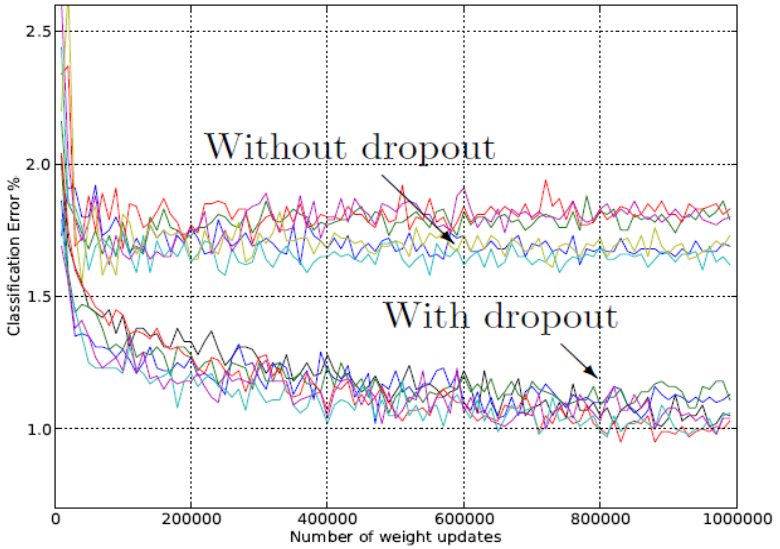
Dropout: Against Overfitting

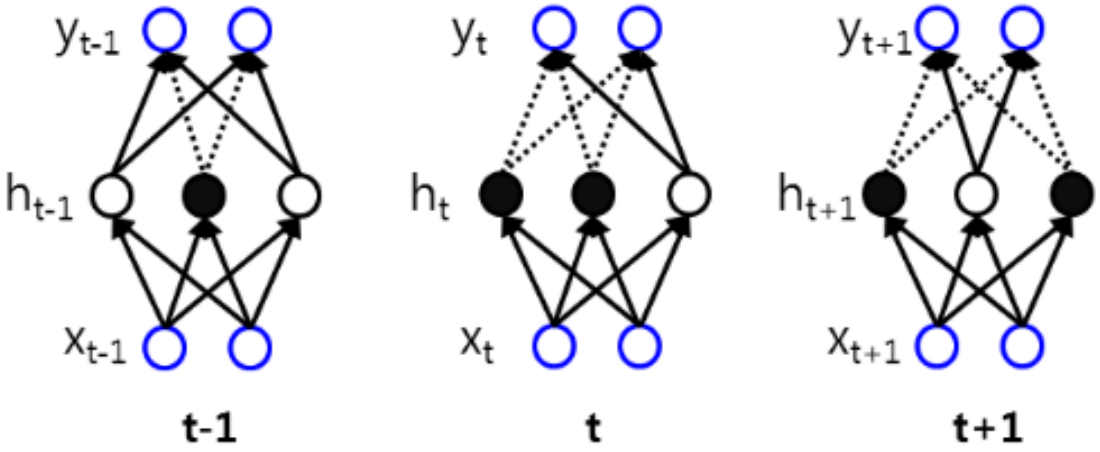


(a) Standard Neural Net



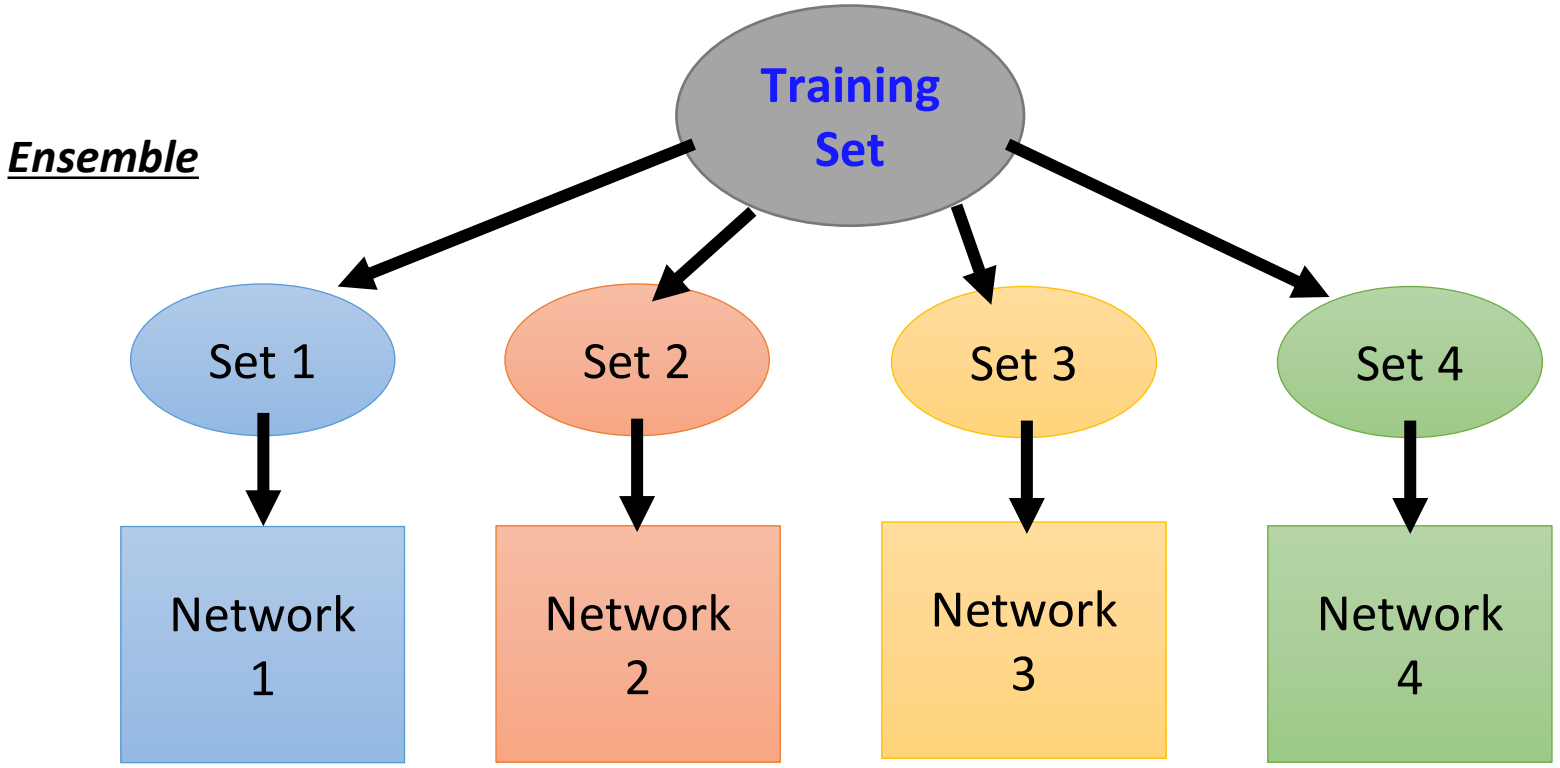
(b) After applying dropout.





Dropout: Against Overfitting

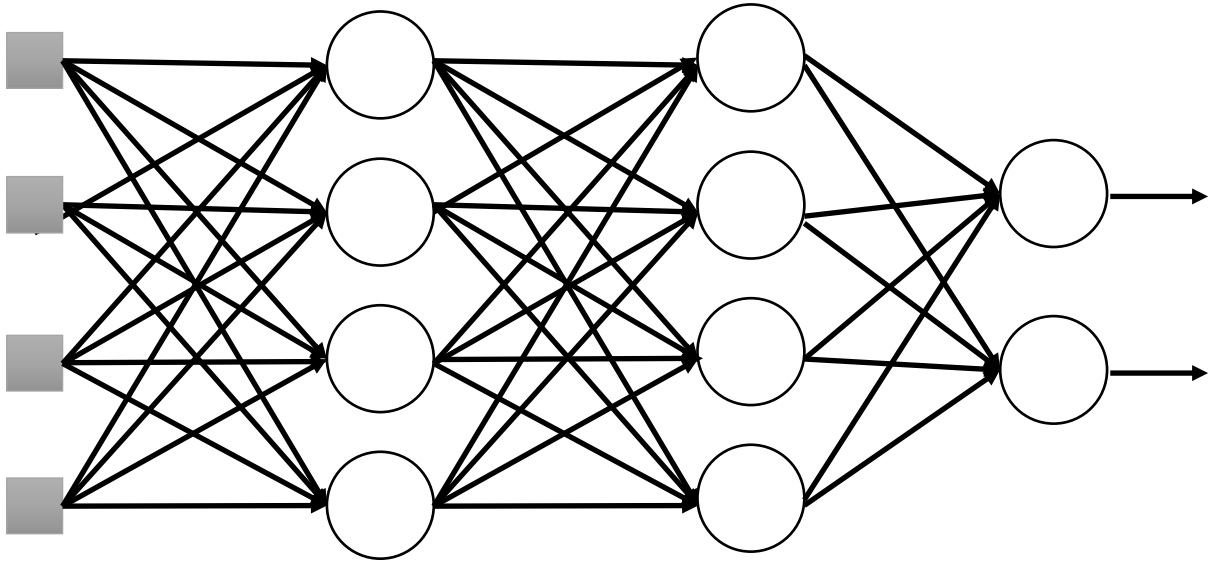
- **Dropout is a kind of ensemble.**



Train a bunch of networks with different structures

Dropout: Against Overfitting

Testing:



- **No dropout**
  - If the dropout rate at training is p%, all the weights times (1-p)%
  - Assume that the dropout rate is 50%.
    If a weight $w = 1$ by training, set $w = 0.5$ for testing.

Data Augmentation

- The simplest way to reduce overfitting is to increase the size of the training data. In machine learning, we were not able to increase the size of training data as the labeled data was too costly.
  - Now let's consider we are dealing with images. In this case, there are a few ways of increasing the size of the training data – rotating the image, flipping, scaling, shifting, etc. In the below image, some transformation has been done on the handwritten digits dataset.

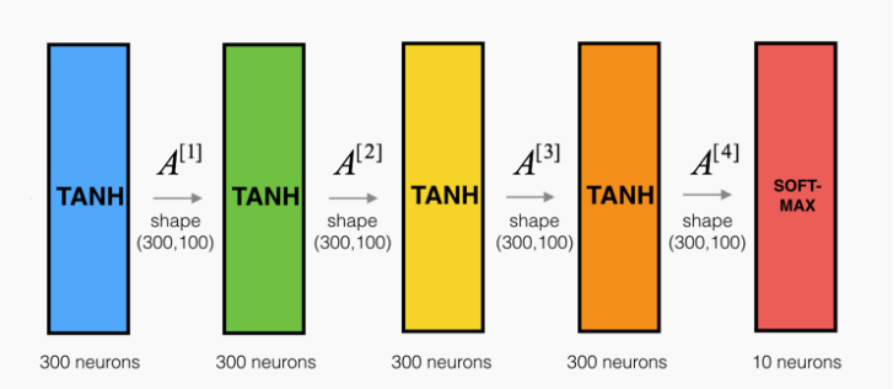
Weight Parameter Initialization

- **Weight parameter initialization**
  - To initialize the weights of the network so that the neuron activation functions are not starting out in saturated (z=WX is too large, it results in saturated constant value (for sigmoid) or explosion for ReLU) or dead regions (z=WX is too small, it results in linear operation (for sigmoid) or vanishing values).
  - In other words, we want to initialize the weights with random values that are not "too small" and not "too large."
- **If W=0 (or constant) initialization**
  - For all units at each hidden layer, the outputs will be constant and the same so that the units have identical influence on the cost, which will lead to identical gradients. Thus neurons will evolve symmetrically throughout training, effectively preventing different neurons from learning different things.

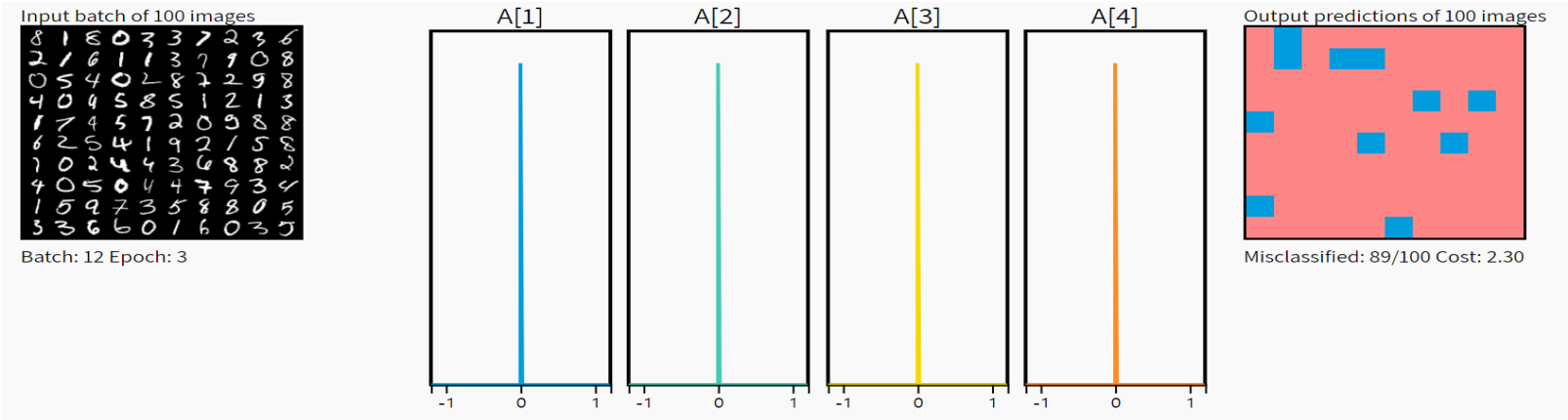
Weight Parameter Initialization

- **If W=large random number initialization**
  - Sigmoid(or Tanh) case: output will converge to 0 (or -1) to 1
  - ReLU case: large negative output causes ‘dead ReLU’

Example case study:
https://deeplearning.ai/ai-notes/initialization/

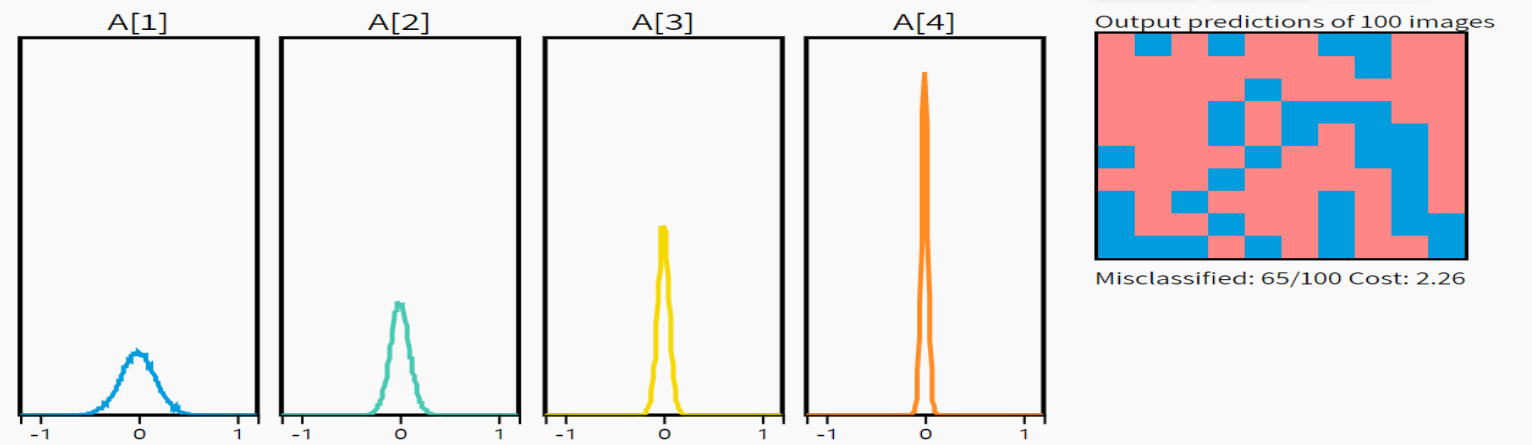


[zero initialization]

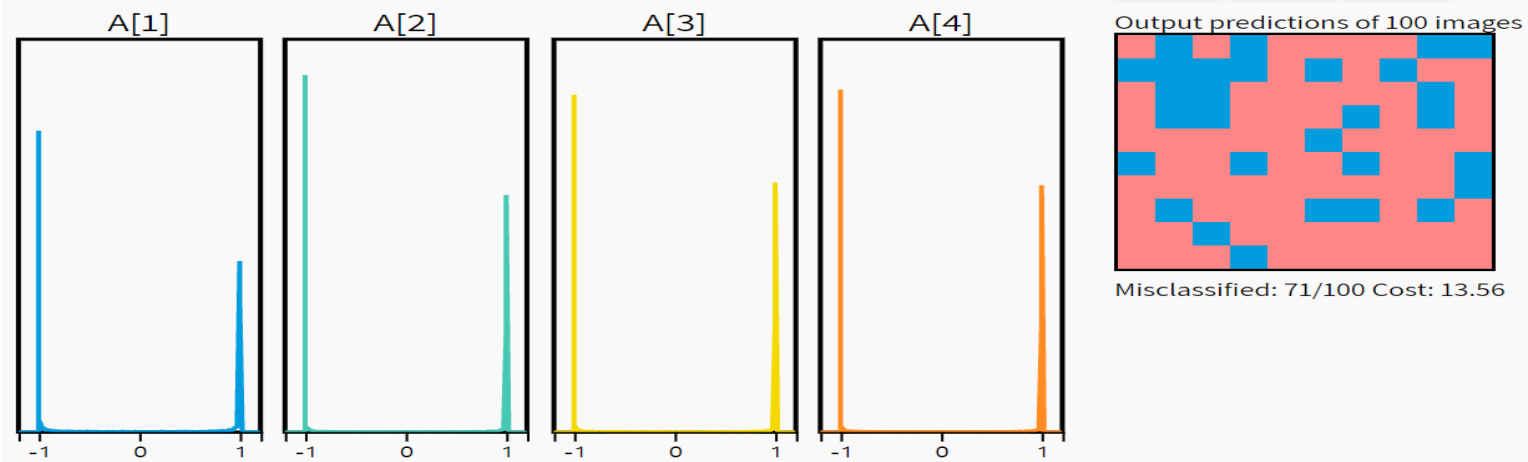


Weight Parameter Initialization

[Uniform initialization]



[Standard Normal initialization]



Weight Parameter Initialization

❑ Xavier initialization

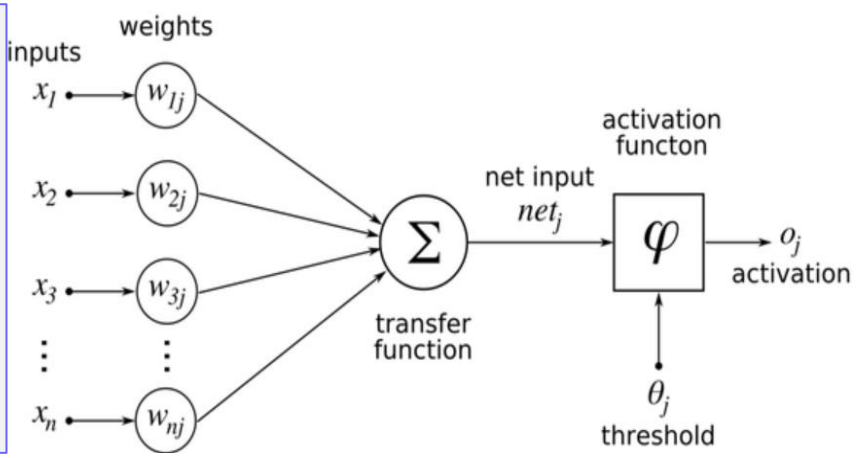
- suggests initializing the weights $\boldsymbol{w_j}$ with a variance so that the variance of net_j, $Var(net_j) = 1$.

$$Var(net_j) = Var(w_{1j}x_1 + w_{2j}x_2 + \cdots + w_{nj}x_n)$$

Assume that our input data at each layer is zero mean, unit variance (e.g., data normalization) and data features are independent. Then,

$$Var(net_j) = nVar(w_{ij})$$

$$nVar(w_{ij}) = 1 \rightarrow Var(w_{ij}) = 1/n$$



- Therefore, we initialize the weights from an IID normal so that

$$\boldsymbol{w_{ij} \sim N\left(0, \frac{1}{n}\right)} \quad n = input\ dimension\ of\ j-th\ layer$$

Weight Parameter Initialization

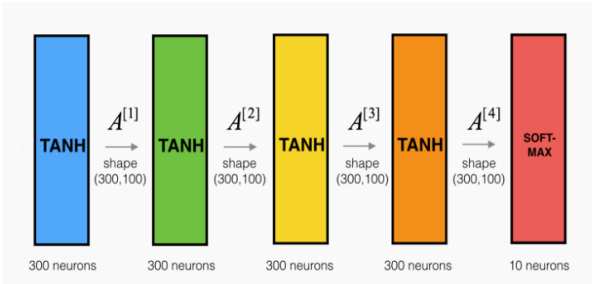
- **Xavier initialization:** $w \sim N\left(0, \frac{1}{n}\right)$
  - generate standard normal weights and then multiply with $\frac{1}{\sqrt{n}}$

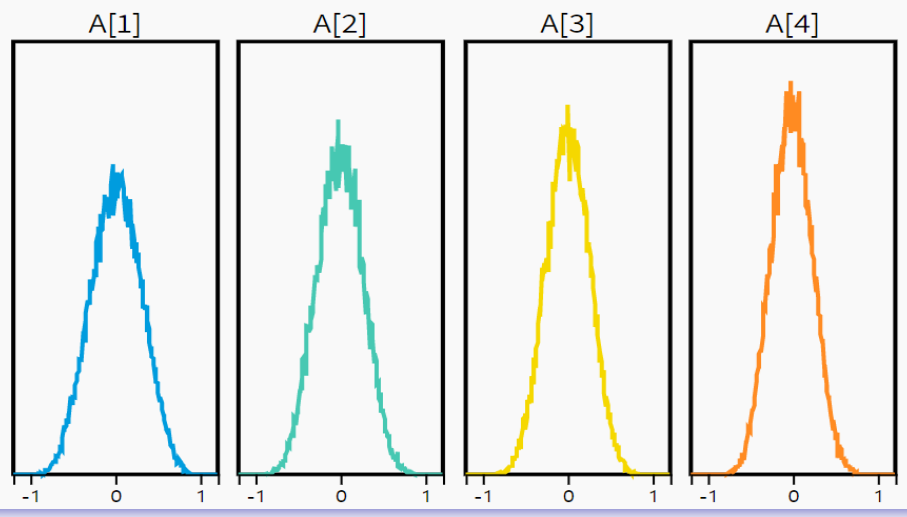
$$Y = cX \rightarrow \quad VAR(Y) = c^2 VAR\,(X)$$

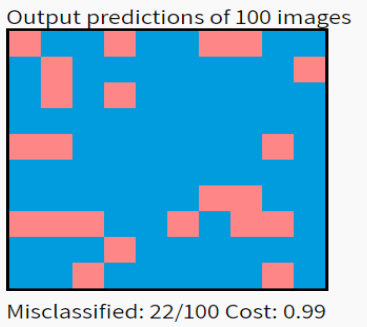
  - TensorFlow

```
W = tf.get_variable("W", shape=[784, 256],
                    initializer=tf.contrib.layers.xavier_initializer())
```

[Xavier initialization]

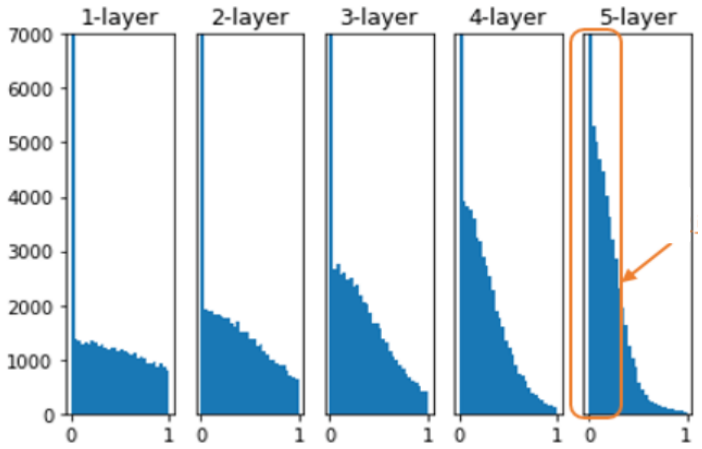




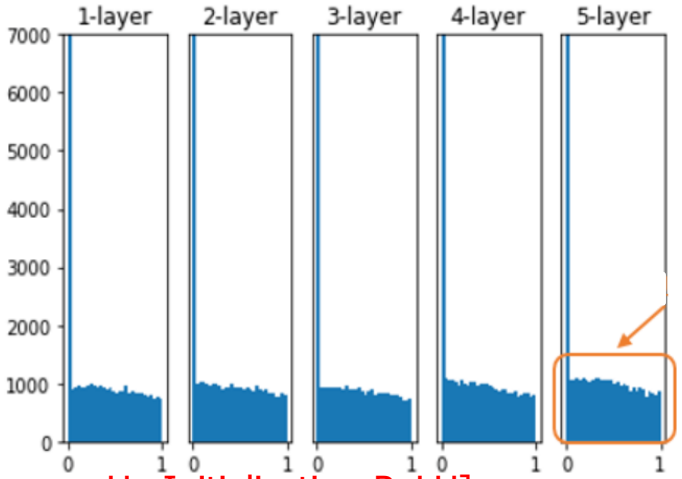


Weight Parameter Initialization

- **He Initialization:** $W \sim N\left(0, \frac{2}{n}\right)$
  - When ReLU activation function is used instead of 'sigmoid' or 'tanh', 'He' initialization is one of the methods you can choose to bring the variance of outputs to approximately one.



[Xavier Initialization-ReLU]



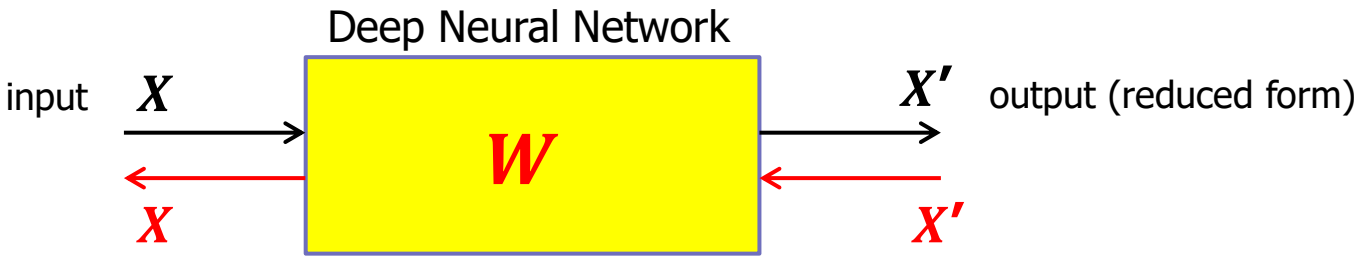
He Initialization-ReLU]

```
W=tf.get_variable("W", shape=[784, 256],
                  initializer=tf.initializers.he_normal())
```

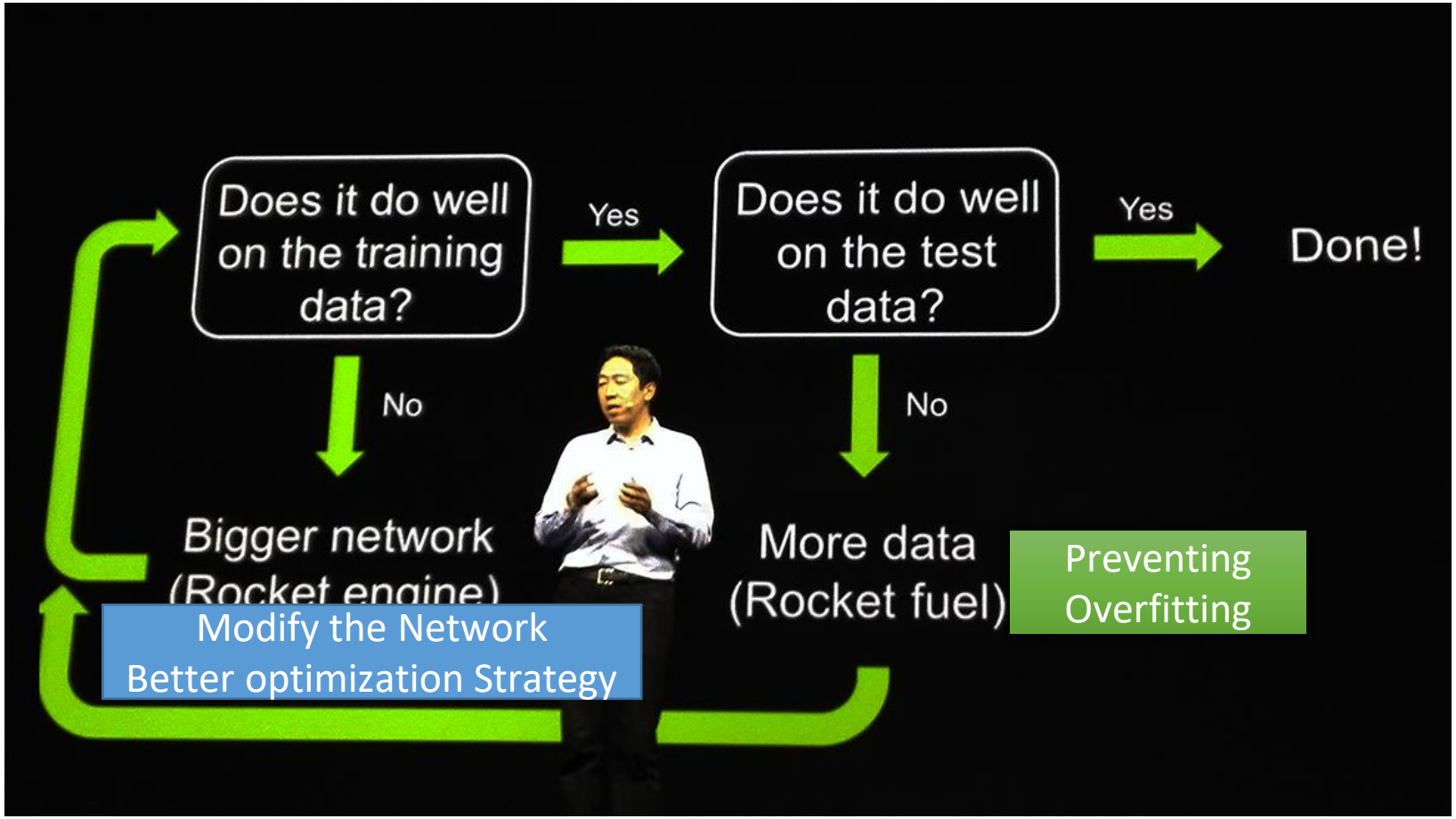
Weight Parameter Initialization

Deep Belief Network

- Designed for unsupervised learning (No data labels).
- To determine W matrix use RBM (Restricted Boltzmann Machine)
- *DBN can be used for parameter initialization for supervised learning*.
  - Good parameter initialization can increase classification performance 3-5%.
- Objective: adjust W to minimize $|X - X'|$



Recipe for Learning



http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

Hyperparameters for Learning

- **NN model related**
  - MLP, CNN, RNN, GAN,…
  - Input/output dimensions
  - Number of hidden layers
    - MLP: 1~10 layers
    - CNN: ~152, ~1000 layers
  - Number of hidden units
    - 10 ~1024 units/layer
  - Activation function
    - Sigmoid, Tanh, ReLu, LeakyReLu, GeLU,…
  - Loss function, Initialization methods
- **Optimization related**
  - Optimization methods
    - GD, SGD, BGD, RMSProp, NAG, AdaGrad, Adam,…)
  - Learning rate: log scale → linear scale
    - 0.0001, 0.001, 0.01, 0.1
    - 0.01, 0.02, 0.03, …
  - Regularization?, Regularization weight
    - 0.0001, 0.001, 0.01, 0.1, 1, 10, 100 …
    - Then, linear scale
  - dropout?, dropout rate
    - 0.1 ~0.5
- **Training and evaluation related**
  - Training/Validation Test/ samples (70%/10%/20%)
  - Normalization method
  - Batch size
  - Number of epochs
    - If validation loss is not reduced more than N epoch, then stop training