DASC7606– 3A
Deep Learning

Training Neural Networks

Dr Bethany Chan
Professor Francis Chin

2024

Revision - More on Logistic Regression

- Metrics for accuracy of predictions:
  - Precision (accurate), Recall (positive), Specificity (reject)
  - F1 (Harmonic mean)
  - Receiver Operating Characteristic ROC (AUC)
- Binary classification (sigmoid $\frac{1}{1+e^{-s}}$)

  with binary cross-entropy cost function

$$J(\boldsymbol{\theta}) = -\frac{1}{M}\sum_{i=1}^{M}\left[y_i \log h(\boldsymbol{x}_i) + (1-y_i)\log(1-h(\boldsymbol{x}_i))\right]$$

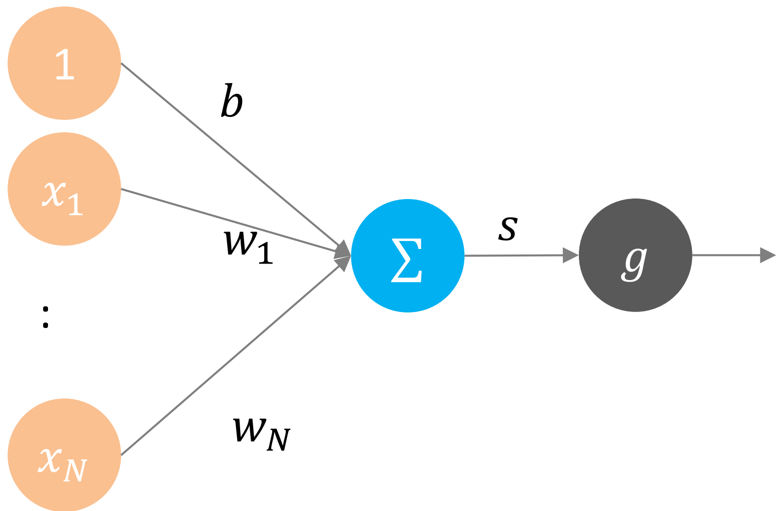
- Multi-classification - *K* classes (softmax $\frac{e^{s_k}}{\sum e^{s_i}}$)

  with categorical cross-entropy cost function

$$J(\boldsymbol{\theta}) = -\frac{1}{M}\sum_{i=1}^{M}\sum_{K}\left[\boldsymbol{y}_i \log \boldsymbol{h}(\boldsymbol{x}_i)\right] = -\frac{1}{M}\sum_{i=1}^{M}\sum_{k=1}^{K}\left[y_{i,k}\log h_k(\boldsymbol{x}_i)\right]$$

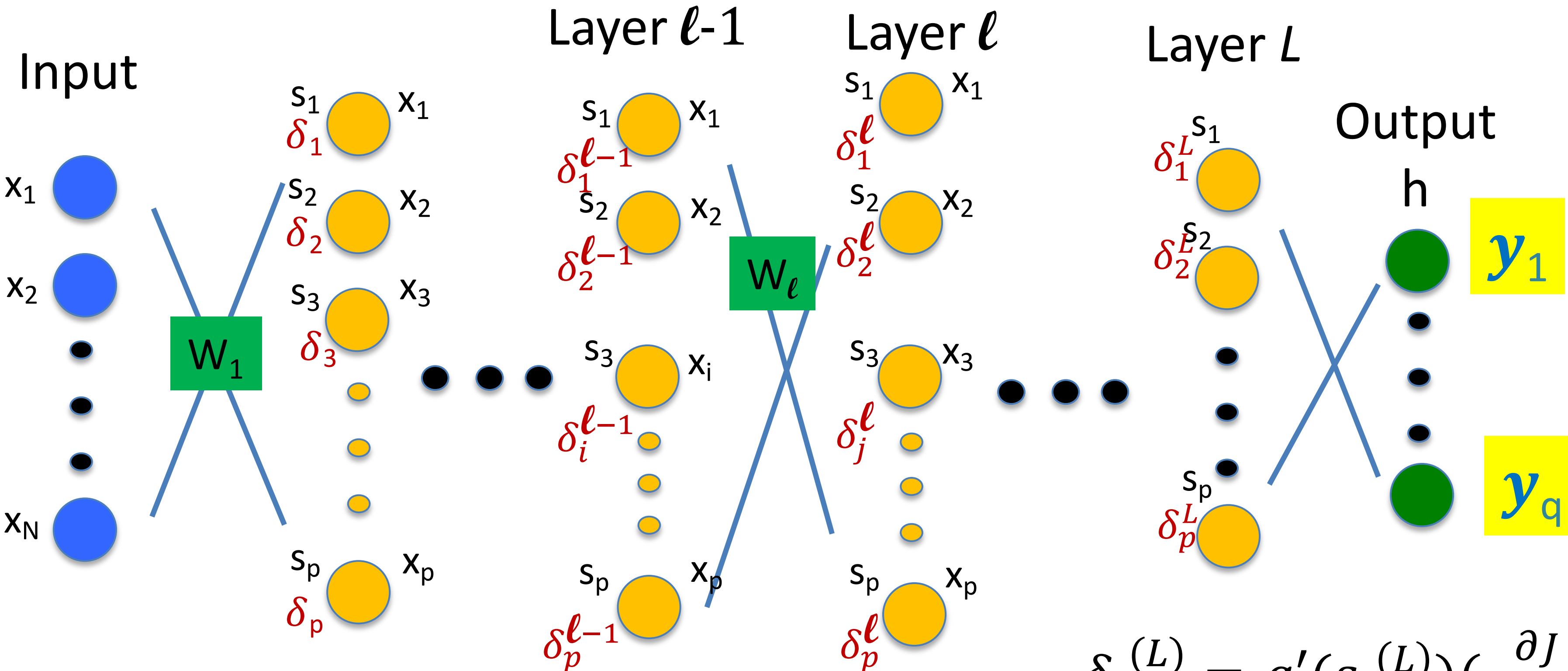
Revision: Perceptron and Neural Network

- Logistic Regression
  ≈ **Perceptron with Sigmoid**
  as the activation function
- Perceptron with **nonlinear** activation functions *g* (Sigmoid, TanH, ReLu, Step) as the building block of neural networks
- **Any logic functions, classification and Regression problems can be approximated** by the Neural Network with sufficient complexity, i.e., number of layers and number of neurons



Perceptron

Revision: Feedforward and Backpropagation

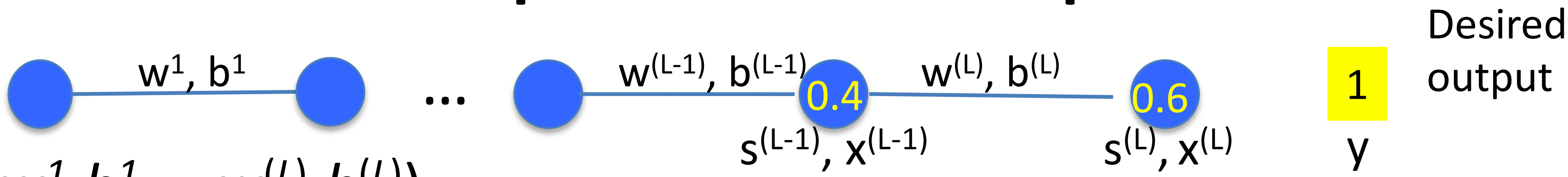


$$\delta_i^{(L)} = g'(s_i^{(L)})\left(\frac{\partial J}{\partial h(x)}\right)$$

$$s_j^{(l)} = \sum (w_{ij}^{(l)} x_i^{(l-1)} + b_i) \qquad x_j^{(l)} = g(s_j^{(l)})$$

$\delta_i^{(l-1)} = g'(s_i^{(l-1)})(\sum w_{ij}^{(l)} \delta_j^{(l)})$; $l = L, L\text{-}1, \ldots,$ where $\delta_i^{(l-1)} = \frac{\partial J}{\partial s_i^{(l-1)}}$

A simple BP example



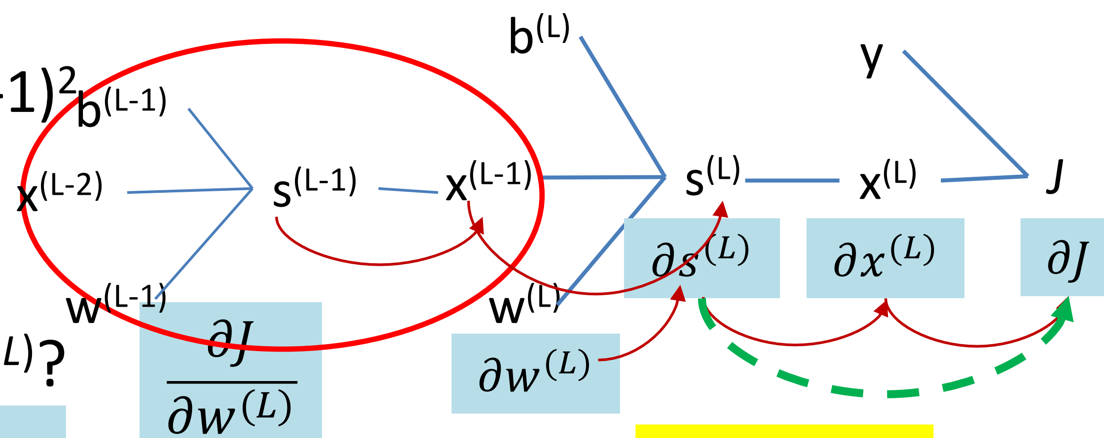
$J(w^1,b^1,\dots w^{(L)},b^{(L)})$

$= \frac{1}{2}(x^{(L)} - y)^2 = \frac{1}{2}(0.6 - 1)^2$

$x^{(L)} = g(s^{(L)})$

$s^{(L)} = w^{(L)}x^{(L-1)} + b^{(L)}$

How J changes w.r.t. $w^{(L)}$?



chain rule

$$\frac{\partial J}{\partial w^{(L)}} = \frac{\partial s^{(L)}}{\partial w^{(L)}}\frac{\partial x^{(L)}}{\partial s^{(L)}}\frac{\partial J}{\partial x^{(L)}}$$

$$= x^{(L-1)} g'(s^{(L)})(x^{(L)} - y)$$

Gradient Descent

$$= x^{(L-1)}\delta^{(L)}$$

key value for BP

$$\delta^{(L)} = \frac{\partial J}{\partial s^{(L)}} = g'(s^{(L)})(x^{(L)} - y)$$

$x^{(L-1)} = g(s^{(L-1)})$

$s^{(L-1)} = w^{(L-1)}x^{(L-2)} + b^{(L-1)}$;

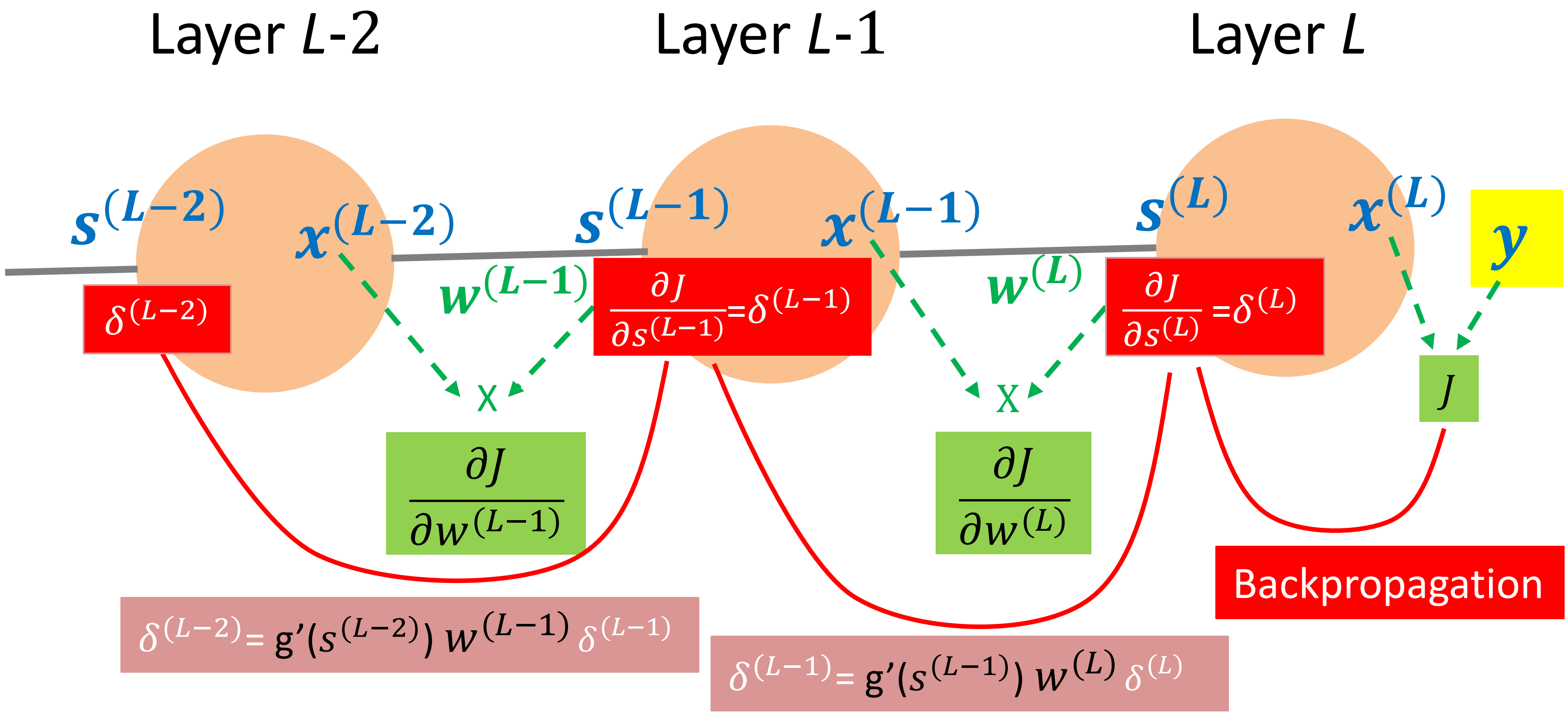
$$\frac{\partial J}{\partial w^{(L-1)}} = \frac{\partial s^{(L-1)}}{\partial w^{(L-1)}}\frac{\partial J}{\partial s^{(L-1)}}$$

$$= x^{(L-2)}\frac{\partial J}{\partial s^{(L-1)}}$$

$$\frac{\partial J}{\partial s^{(L-1)}} = \frac{\partial x^{(L-1)}}{\partial s^{(L-1)}}\frac{\partial s^{(L)}}{\partial x^{(L-1)}}\frac{\partial J}{\partial s^{(L)}}$$

$$\delta^{(L-1)} = g'(s^{(L-1)})\, w^{(L)}\delta^{(L)}$$

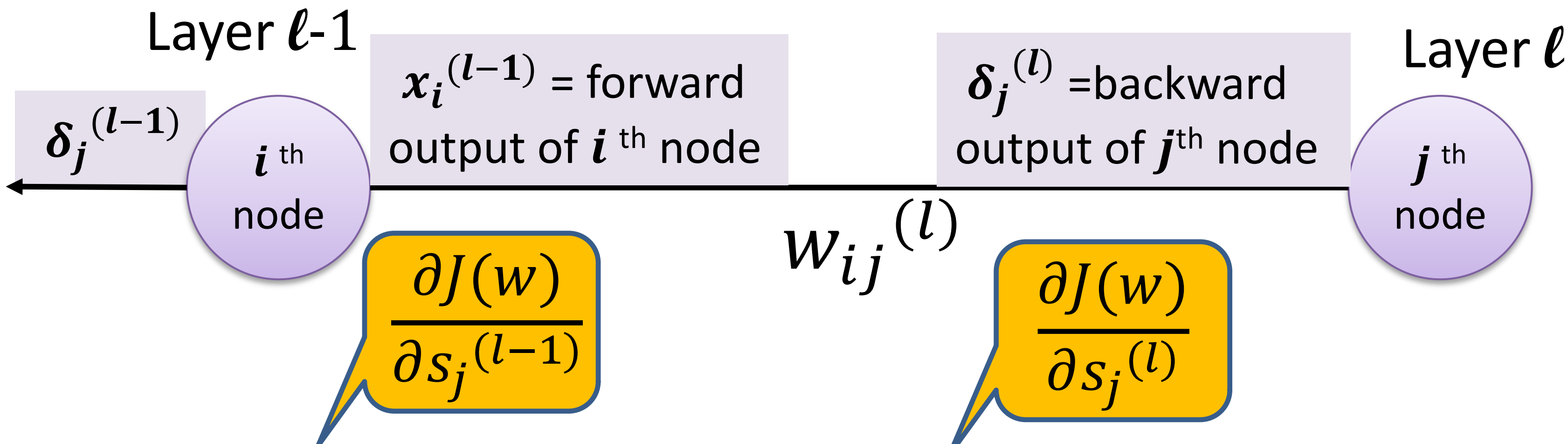
How w's are adjusted?



Backpropagate $\frac{\partial J}{\partial s^{(l)}} = \delta^{(l)}$

$\frac{\partial J}{\partial w^{(l)}} = x^{(l-1)}\delta^{(l)}$ based on output $x^{(l-1)}$ and BP value $\frac{\partial J}{\partial s^{(l)}} = \delta^{(l)}$

Backpropagation



$\delta_i^{(l-1)} = g'(s_i^{(l-1)})(\sum w_{ij}^{(l)} \delta_j^{(l)})$; $l = L, L\text{-}1, L\text{-}2\ldots$

As $J(w)$ depends on { $h(x)$, y } and $h(x) = g(s)$

where $\delta_i^{(L)} = \frac{\partial J(w)}{\partial s_i^{(L)}} = \frac{\partial h(x)}{\partial s_i^{(L)}} \frac{\partial J}{\partial h(x)} = g'(s_i^{(L)})(\frac{\partial J}{\partial h(x)})$

$w_{ij}^{(l)} \to w_{ij}^{(l)} - \alpha\, x_i^{(l-1)} \delta_j^{(l)}$

need to prove

$\frac{\partial J(\mathbf{w})}{\partial w_{ij}^{(l)}} = x_i^{(l-1)} \delta_j^{(l)}$ and $\delta_j^{(l)} = \frac{\partial J(w)}{\partial s_j^{(l)}}$

Training NN with backpropagation

- Backpropagation
- Explanation through examples
- **Showing gradient $\frac{\partial \mathrm{J(W)}}{\partial w_{ij}{}^{(l)}}$ is indeed equal to $x_i{}^{(l-1)}\delta_j{}^{(l)}$**
- BP with different activation and cost functions
- Coding backpropagation

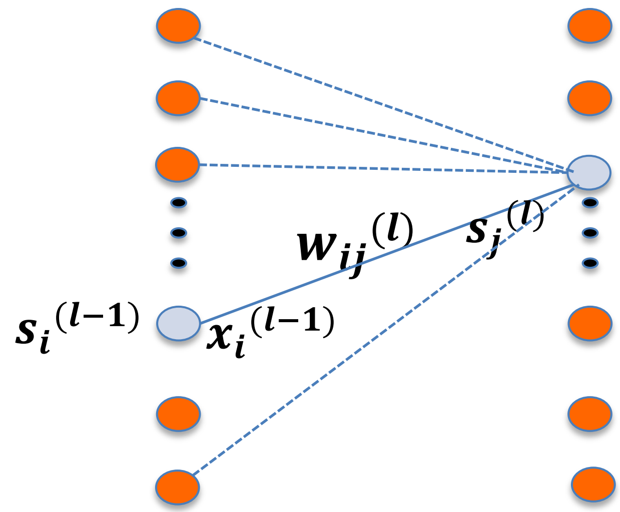
GOAL: Want to show: $\frac{\partial J(\mathbf{w})}{\partial w_{ij}{}^{(l)}} = \delta_j{}^{(l)} \times x_i{}^{(l-1)}$

By induction (next few slides)

$$\frac{\partial J(\mathbf{w})}{\partial w_{ij}{}^{(l)}} = \frac{\partial J(\mathbf{w})}{\partial s_j{}^{(l)}} \times \frac{\partial s_j{}^{(l)}}{\partial w_{ij}{}^{(l)}} = \frac{\partial J(\mathbf{w})}{\partial s_j{}^{(l)}} \times x_i{}^{(l-1)}$$

$$s_j{}^{(l)} = \left(\sum_i w_{ij}{}^{(l)} x_i{}^{(l-1)} \right) + b$$

$$\Longrightarrow \frac{\partial s_j{}^{(l)}}{\partial w_{ij}{}^{(l)}} = x_i{}^{(l-1)}$$

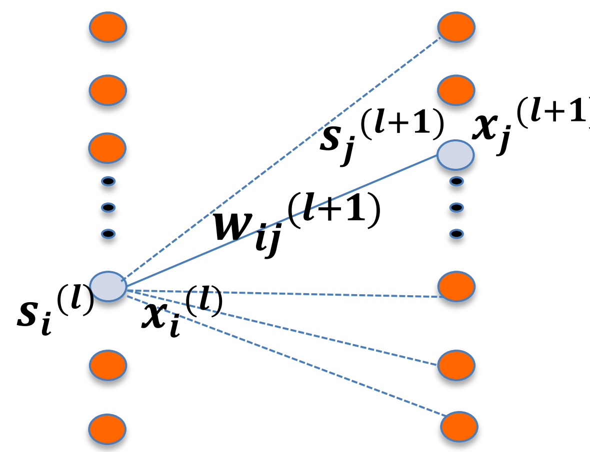


GOAL now becomes: want to show that $\delta_j{}^{(l)} = \frac{\partial J(\mathbf{w})}{\partial s_j{}^{(l)}}$

NEW GOAL : want to show that $\delta_j^{(l)} = \frac{\partial J(\mathbf{w})}{\partial s_j^{(l)}}$

$$\frac{\partial J(\mathbf{w})}{\partial s_i^{(l)}} = \sum_j \boxed{\frac{\partial J(\mathbf{w})}{\partial s_j^{(l+1)}}} \times \frac{\partial s_j^{(l+1)}}{\partial x_i^{(l)}} \times \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$

$$= \sum_j \delta_j^{(l+1)} \times \frac{\partial s_j^{(l+1)}}{\partial x_i^{(l)}} \times \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$



By Induction (Backward)

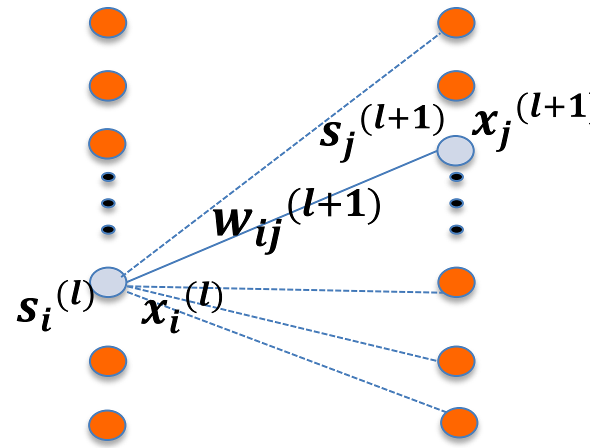
with $\delta_j^{(k)} = \frac{\partial J(\mathbf{w})}{\partial s_j^{(k)}}$

for $k = L, L\text{-}1, \ldots\, l+1$

NEW GOAL : want to show that $\delta_j^{(l)} = \frac{\partial J(\mathbf{w})}{\partial s_j^{(l)}}$

$$\frac{\partial J(\mathbf{w})}{\partial s_i^{(l)}} = \sum_j \boxed{\frac{\partial J(\mathbf{w})}{\partial s_j^{(l+1)}}} \times \frac{\partial s_j^{(l+1)}}{\partial x_i^{(l)}} \times \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$

$$= \sum_j \delta_j^{(l+1)} \times \boxed{\frac{\partial s_j^{(l+1)}}{\partial x_i^{(l)}}} \times \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$



$$s_j^{(l+1)} = \left(\sum_i w_{ij}^{(l+1)} x_i^{(l)}\right) + b$$

$$\Longrightarrow \frac{\partial s_j^{(l+1)}}{\partial x_i^{(l)}} = w_{ij}^{(l+1)}$$

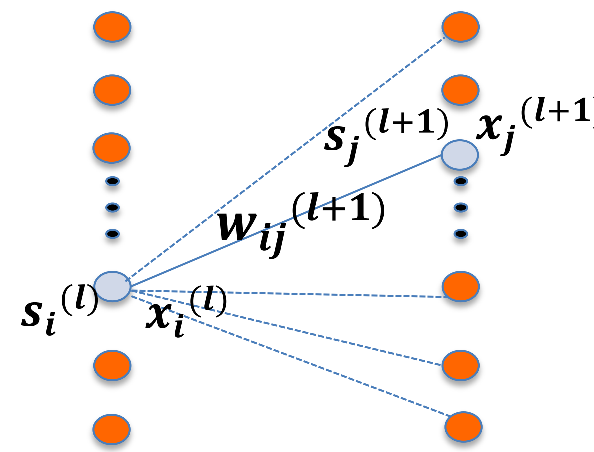
NEW GOAL : want to show that $\delta_j^{(l)} = \dfrac{\partial J(\mathbf{w})}{\partial s_j^{(l)}}$

$$\frac{\partial J(\mathbf{w})}{\partial s_i^{(l)}} = \sum_j \frac{\partial J(\mathbf{w})}{\partial s_j^{(l+1)}} \times \frac{\partial s_j^{(l+1)}}{\partial x_i^{(l)}} \times \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$

$$= \sum_j \delta_j^{(l+1)} \times \frac{\partial s_j^{(l+1)}}{\partial x_i^{(l)}} \times \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$

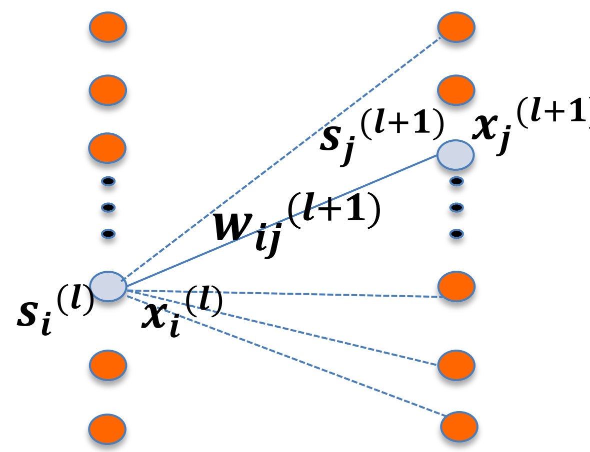
$$= \sum_j \delta_j^{(l+1)} \times w_{ij}^{(l+1)} \times \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}}$$

$$= \sum_j \delta_j^{(l+1)} \times w_{ij}^{(l+1)} \times g'(s_i^{(l)}) \quad \text{[since } x_i^{(l)} = g(s_i^{(l)})\text{]}$$



NEW GOAL : want to show that $\delta_j{}^{(l)} = \frac{\partial J(\mathbf{w})}{\partial s_j{}^{(l)}}$

$$\frac{\partial J(\mathbf{w})}{\partial s_i{}^{(l)}} = \sum_j \frac{\partial J(\mathbf{w})}{\partial s_j{}^{(l+1)}} \times \frac{\partial s_j{}^{(l+1)}}{\partial x_i{}^{(l)}} \times \frac{\partial x_i{}^{(l)}}{\partial s_i{}^{(l)}}$$



$$= \sum_j \delta_j{}^{(l+1)} \times \frac{\partial s_j{}^{(l+1)}}{\partial x_i{}^{(l)}} \times \frac{\partial x_i{}^{(l)}}{\partial s_i{}^{(l)}}$$

$$= \sum_j \delta_j{}^{(l+1)} \times w_{ij}{}^{(l+1)} \times \frac{\partial x_i{}^{(l)}}{\partial s_i{}^{(l)}}$$

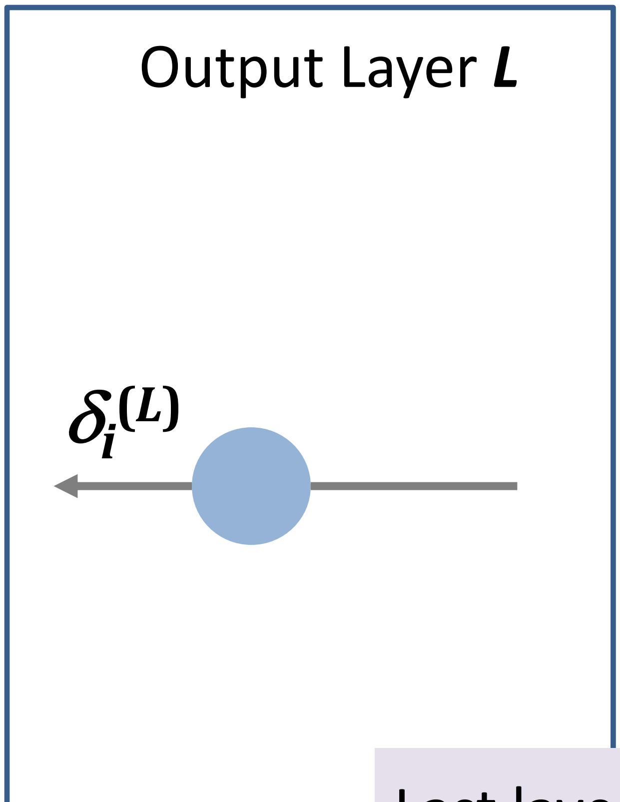
$$= \sum_j \delta_j{}^{(l+1)} \times w_{ij}{}^{(l+1)} \times g'(s_i{}^{(l)}) \quad = \delta_i{}^{(l)}$$

Recall: $\delta_i{}^{(l)} = g'(s_i{}^{(l)})(\sum_j w_{ij}{}^{(l+1)} \delta_j{}^{(l+1)})$

Training NN with backpropagation

- Backpropagation
- Explanation through examples
- Showing gradient $\frac{\partial \mathrm{J}(\mathbf{W})}{\partial {w_{ij}}^{(l)}}$ is indeed equal to ${x_i}^{(l-1)}{\delta_j}^{(l)}$
- **BP with different activation and cost functions**
- Coding backpropagation

For the special case of output layer



For earlier layers:

$$\delta_i{}^{(l)} = g'(s_i{}^{(l)})(\sum w_{ij}{}^{(l+1)}\delta_j{}^{(l+1)})$$

For output layer *L*:

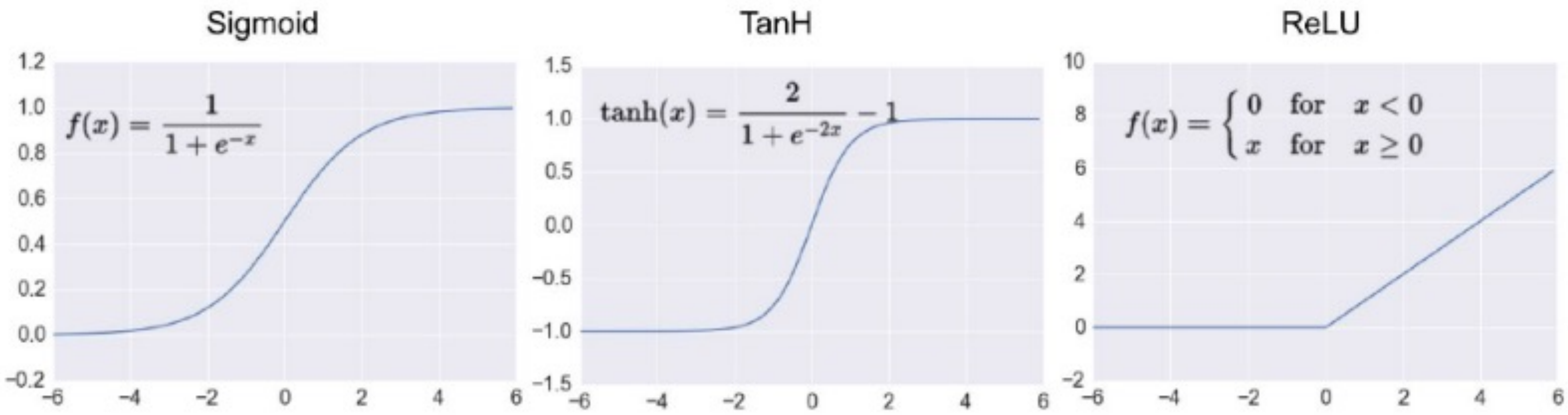
$$\delta_i{}^{(L)} = g'(s_i{}^{(L)})(\frac{\partial J}{\partial h(x)})$$

Note: J(h(x),y), h(x)=h(g(s)) at output layer

Last layer $\delta_i{}^{(L)}$ depends on (the derivative of) :

- activation function g in the last layer
- cost function J

g : Common Activation Functions



Sigmoid: $g(s) = \sigma(s) = \dfrac{1}{1+e^{-s}}$

$$\rightarrow g'(s) = \frac{dg(s)}{ds} = \frac{e^{-s}}{(1+e^{-s})^2}$$

$$= \left(\frac{1+e^{-s}-1}{1+e^{-s}}\right)\left(\frac{1}{1+e^{-s}}\right)$$

$$= \big(1-g(s)\big)g(s)$$

g : Common Activation Functions

Tanh:

$$g(s) = \frac{2}{1+e^{-2s}} - 1 = \frac{1-e^{-2s}}{1+e^{-2s}}$$

$$= \frac{1-e^{-s}/e^{s}}{1+e^{-s}/e^{s}} = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$$

Quotient rule

$$\left(\frac{u}{v}\right)' = \frac{u'v-uv'}{v^2}$$

$$g'(s) = \frac{dg(s)}{ds} = \frac{(e^{s}+e^{-s})(e^{s}+e^{-s}) - (e^{s}-e^{-s})(e^{s}-e^{-s})}{(e^{s}+e^{-s})^2}$$

$$= 1 - \frac{(e^{s}-e^{-s})^2}{(e^{s}+e^{-s})^2} = 1 - g(s)^2$$

RelU:

$$g(s) = \begin{cases} 0 & \text{if } s < 0 \\ s & \text{if } s \geq 0 \end{cases} \quad \rightarrow g'(s) = \begin{cases} 0 & \text{if } s < 0 \\ 1 & \text{if } s \geq 0 \end{cases}$$

Output Layer

	Size	Loss Function J	Activation Function g
Regression	1	Mean square error	g

Mean square error: $J = \frac{1}{2}(h(x) - y)^2$

Binary cross entropy loss: $J = -[y \log h(x) + (1 - y) \log(1 - h(x)]$

Cross entropy loss: $J = -\sum y_k \log h(x)$

Output Layer $\delta^{(L)}$- Mean Square Error

	Size	Loss Function J	Activation Function g	$\delta^{(L)}$
Regression	1	Mean square error	g	$g'(s)(h(x) - y)$
Binary Classification	1	Binary cross entropy	Sigmoid	$h(x) - y$
Multi Classification	K	Cross entropy	Softmax	$\boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y}$

$$J = \frac{1}{2}(h(x) - y)^2$$

For output layer *L* (with one output):

$$\delta^{(L)} = g'\left(s^{(L)}\right)\left(\frac{\partial J}{\partial h(x)}\right)$$

$$= g'\left(s^{(L)}\right)(h(x) - y)$$

Output Layer $\delta^{(L)}$- Binary Classification

	Size	Loss Function J	Activation Function g	$\delta^{(L)}$
Regression	1	Mean square error	g	$g'(s)(h(x) - y)$
Binary Classification	1	Binary cross entropy	Sigmoid	$h(x) - y$
Multi Classification	K	Cross entropy	Softmax	$\boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y}$

Note that $h(x) = g(s) = \sigma(s)$

$J = -[y \log h(x) + (1 - y) \log(1 - h(x))]$

For output layer L (with one output):

$$\delta^{(L)} = g'\left(s^{(L)}\right)\left(\frac{\partial J}{\partial h(x)}\right)$$

$$= \sigma'\left(s^{(L)}\right)\left(-\frac{y}{h(x)} + \frac{1-y}{1-h(x)}\right)$$

$$= \left(1 - \sigma(s)\right)\sigma(s)\left(-\frac{y-h(x)}{h(x)(1-h(x))}\right) = h(x) - y$$

Output Layer $\delta^{(L)}$ - Multi Classification

	Size	Loss Function J	Activation Function g	δ(L)
Regression	1	Mean square error	g	$g'(s)(h(x) - y)$
Binary Classification	1	Binary cross entropy	Sigmoid	$h(x) - y$
Multi Classification	K	Cross entropy	Softmax	$\boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y}$

Cross entropy loss: $J = -\sum y_k \log p_k$

Note that $\boldsymbol{h}(\boldsymbol{x}), \boldsymbol{y}$ and $\boldsymbol{\delta}^{(L)}$ are matrices

$\boldsymbol{y} = (y_1, \ldots, y_K)$

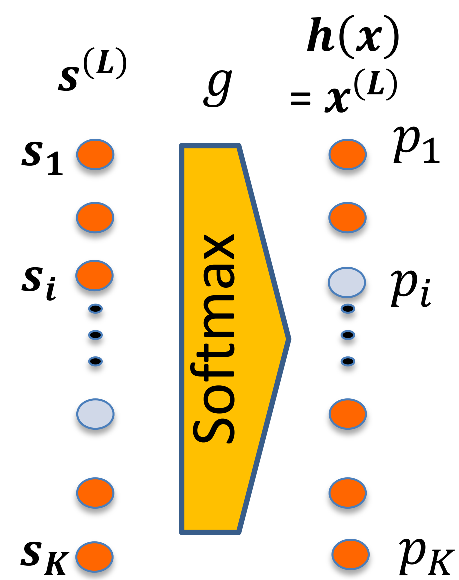
$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{x}^{(L)} = (p_1, \ldots, p_K)$

$p_i = g(s_i)$ (note that g is softmax)

$$\boldsymbol{\delta}^{(L)} = g'\left(\boldsymbol{s}^{(L)}\right)\left(\frac{\partial J}{\partial \boldsymbol{h}(\boldsymbol{x})}\right)$$

$$\frac{\partial J}{\partial \boldsymbol{h}(\boldsymbol{x})} = \left(\frac{\partial J}{\partial p_1}, \frac{\partial J}{\partial p_2}, \ldots \frac{\partial J}{\partial p_K}\right)$$

$\boldsymbol{g}'\left(\boldsymbol{s}^{(L)}\right) = \left(\frac{\partial p_i}{\partial s_j}\right)$, for $1 \le i, j \le K$ is a matrix



Output Layer $\delta^{(L)}$ - Multi Classification

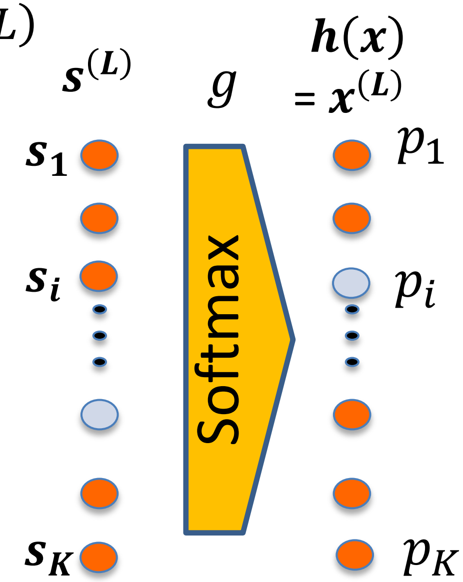
As $\boldsymbol{\delta}^{(L)} = \boldsymbol{g}'\left(\boldsymbol{s}^{(L)}\right)\left(\frac{\partial J}{\partial \boldsymbol{h}(\boldsymbol{x})}\right)$; let s_i=$s_i{}^{(L)}$ and p_i= $x_i{}^{(L)}$

$\boldsymbol{h}(\boldsymbol{x})$=($p_1$,… p_K) where $g(s_i)$ = p_i = $\frac{e^{s_i}}{\sum e^{s_k}}$

$\boldsymbol{g}'\left(\boldsymbol{s}^{(L)}\right)$ = ($\frac{\partial p_i}{\partial s_j}$), for $1 \leq i,j \leq K$

If $i = j$; $\frac{\partial p_i}{\partial s_i}$ = $(e^{s_i} \sum e^{s_k} - e^{s_i} e^{s_i})/(\sum e^{s_k})^2 = p_i(1\text{-}p_i)$

If $i \neq j$; $\frac{\partial p_i}{\partial s_j}$ = $-(e^{s_j} e^{s_i})/(\sum e^{s_k})^2 = -p_j p_i$



$$\boldsymbol{g}'\left(\boldsymbol{s}^{(L)}\right) = \begin{pmatrix} p_1(1\text{-}p_1) & -p_1p_2 & -p_1p_3 \ldots & -p_1p_K \\ -p_2p_1 & p_2(1\text{-}p_2) & -p_2p_3 \ldots & -p_2p_K \\ & \ldots & & \\ -p_Kp_1 & -p_Kp_2 & -p_Kp_3 \ldots & p_K(1\text{-}p_K) \end{pmatrix}$$

K x K matrix

PROOF – SKIPPED

Output Layer $\delta^{(L)}$- Multi Classification

	Size	Loss Function J	Activation Function g	$\delta^{(L)}$
Regression	1	Mean square error	g	$g'(s)(h(x)-y)$
Binary Classification	1	Binary cross entropy	Sigmoid	$h(x)-y$
Multi Classification	K	Cross entropy	Softmax	$\boldsymbol{h}(\boldsymbol{x})-\boldsymbol{y}$

Cross entropy loss: $J=-\sum y_k \log p_k$

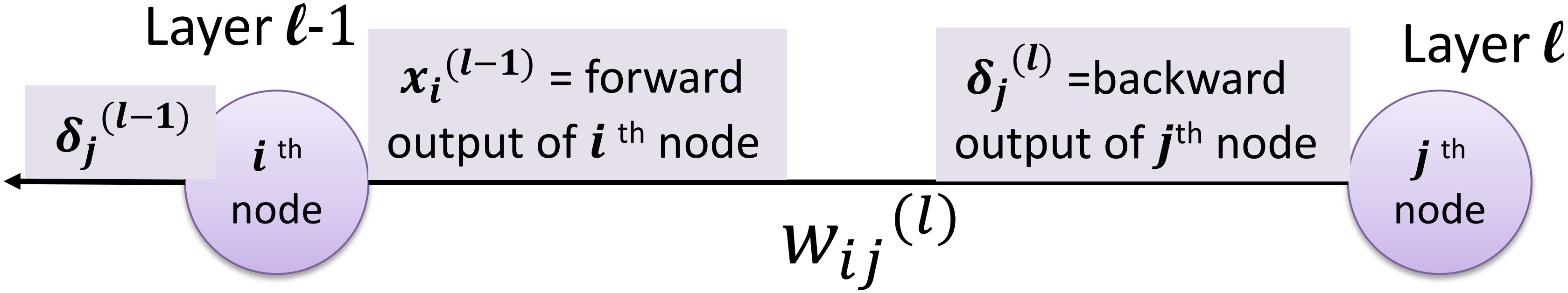
$$\frac{\partial J}{\partial \boldsymbol{h}(\boldsymbol{x})} = \left(\frac{\partial J}{\partial p_1}, \frac{\partial J}{\partial p_2}, \ldots \frac{\partial J}{\partial p_C}\right) \text{ where } \frac{\partial J}{\partial p_i} = -\frac{y_i}{p_i}$$

$$\boldsymbol{\delta}^{(\boldsymbol{L})} = \boldsymbol{g}'\left(\boldsymbol{s}^{(\boldsymbol{L})}\right)\left(\frac{\partial J}{\partial \boldsymbol{h}(\boldsymbol{x})}\right) = \begin{pmatrix} p_1(1\text{-}p_1) & -p_1p_2 & -p_1p_3 \ldots & -p_1p_K \\ -p_2p_1 & p_2(1\text{-}p_2) & -p_2p_3 \ldots & -p_2p_K \\ & & & \ldots \\ -p_Kp_1 & -p_K p_2 & -p_Kp_3 \ldots & p_K(1\text{-}p_K) \end{pmatrix} \begin{pmatrix} -y_1/p_1 \\ -y_2/p_2 \\ \ldots \\ -y_K/p_K \end{pmatrix}$$

$$= \left(-y_1+p_1\sum y_k, -y_2+p_2\sum y_k, \ldots -y_K+p_K\sum y_k\right)$$

$$= \left(-y_1+p_1, -y_2+p_2, \ldots -y_K+p_K\right) \text{ as } \sum y_k=1$$

$$= \boldsymbol{h}(\boldsymbol{x})-\boldsymbol{y}$$



- $w_{ij}{}^{(l)} \rightarrow w_{ij}{}^{(l)} - \alpha\, x_i{}^{(l-1)} \delta_j{}^{(l)}$ where $\frac{\partial J(\mathbf{w})}{\partial w_{ij}{}^{(l)}} = \delta_j{}^{(l)} \times x_i{}^{(l-1)}$
- $\delta_i{}^{(l-1)} = g'(s_i{}^{(l-1)})(\sum w_{ij}{}^{(l)} \delta_j{}^{(l)})$; $l = L, L\text{-}1, L\text{-}2 \ldots$

sigmoid	$\boldsymbol{g}(s) = \frac{1}{1+e^{-s}}$	$\boldsymbol{g}'(s) = \big(1 - g(s)\big)g(s)$
Tanh	$\boldsymbol{g}(s) = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}}$	$\boldsymbol{g}'(s) = 1 - g(s)^2$
Relu	$\boldsymbol{g}(s) = \begin{cases} 0 & \text{if } s < 0 \\ s & \text{if } s \geq 0 \end{cases}$	$\boldsymbol{g}'(s) = \begin{cases} 0 & \text{if } s < 0 \\ 1 & \text{if } s \geq 0 \end{cases}$

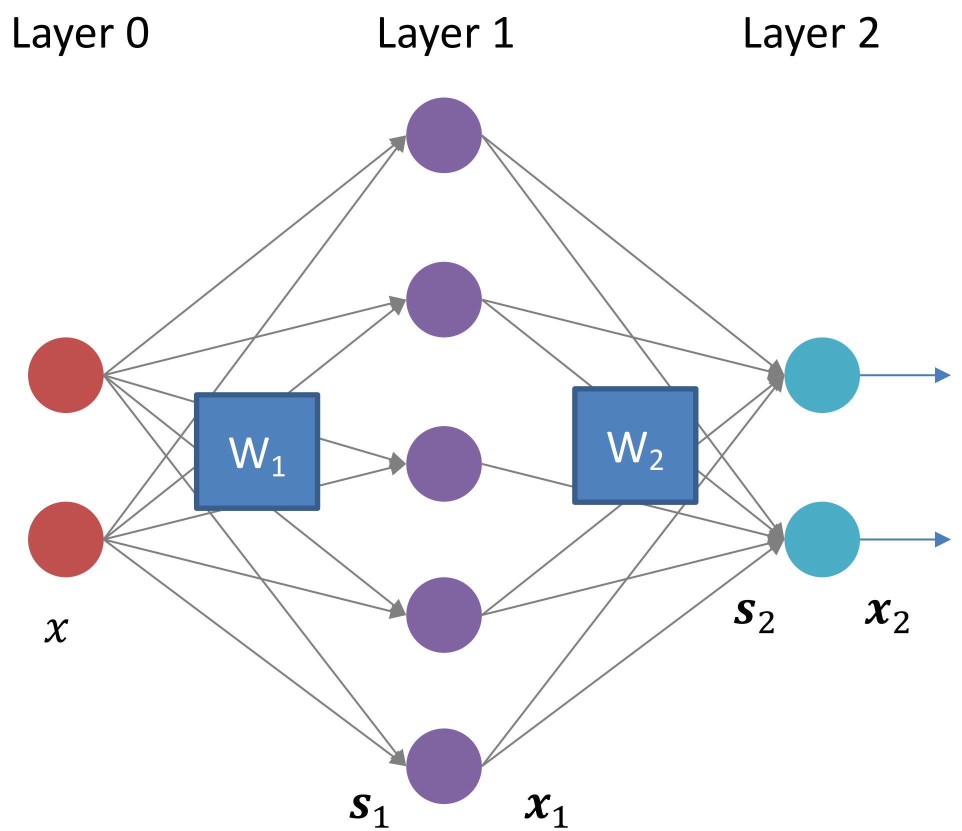
BP Summary

g	$\delta^{(L)} = g'(s)(h(x) - y)$
Sigmoid	$\delta^{(L)} = h(x) - y$
Softmax	$\delta^{(L)} = \boldsymbol{h}(\boldsymbol{x}) - \boldsymbol{y}$

Training NN with backpropagation

- Backpropagation
- Explanation through examples
- Showing gradient $\frac{\partial \mathrm{J}(\mathbf{W})}{\partial {w_{ij}}^{(l)}}$ is indeed equal to ${x_i}^{(l-1)}{\delta_j}^{(l)}$
- BP with different activation and cost functions
- **Coding backpropagation**

NN Classifier with one hidden layer



Forward pass equations:

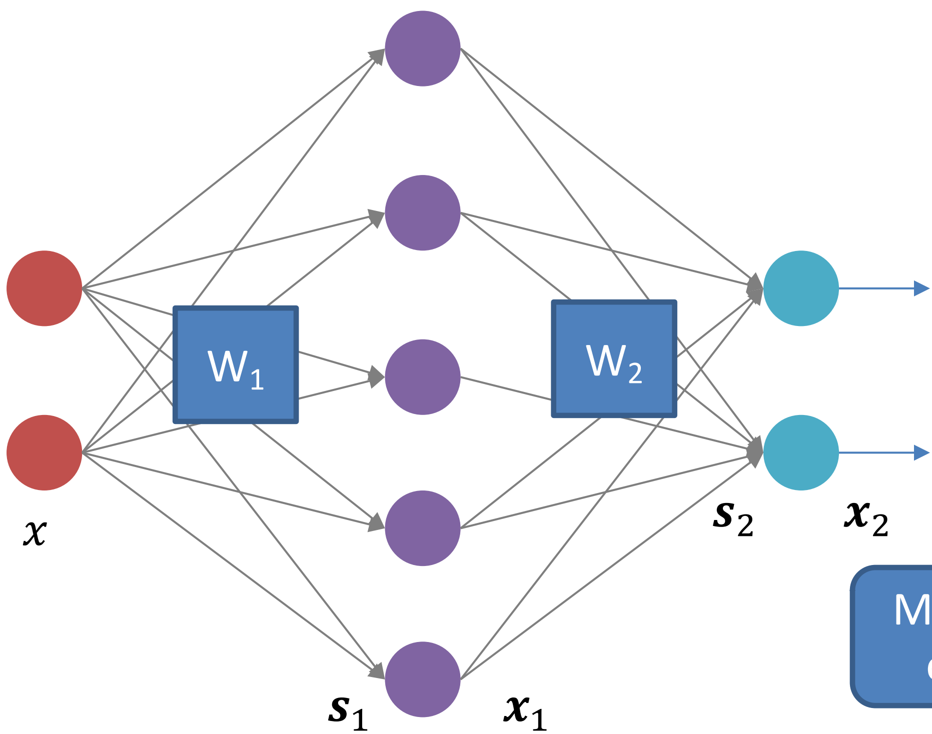
$$s_1 = xW_1 + b_1$$
$$x_1 = \tanh(s_1)$$
$$s_2 = x_1 W_2 + b_2$$
$$h = x_2 = \text{softmax}(s_2)$$

Dimensions of matrices:

x: 1×2
W_1: 2×5
b_1: 1×5
s_1, x_1: 1×5
W_2: 5×2
b_2: 1×2
s_2, x_2: 1×2

Let x_i is the forward output of layer i (here x_1, x_2)
Let δ_i is the backward output of layer i (here δ_1, δ_2)

NN Classifier with one hidden layer



Forward pass equations:

$$s_1 = xW_1 + b_1$$
$$x_1 = \tanh(s_1)$$
$$s_2 = x_1 W_2 + b_2$$
$$h = x_2 = \text{softmax}(s_2)$$

Dimensions of matrices:

x: **M** × 2

W_1: 2 × 5

b_1: 1 × 5

s_1, x_1: **M** × 5

W_2: 5 × 2

b_2: 1 × 2

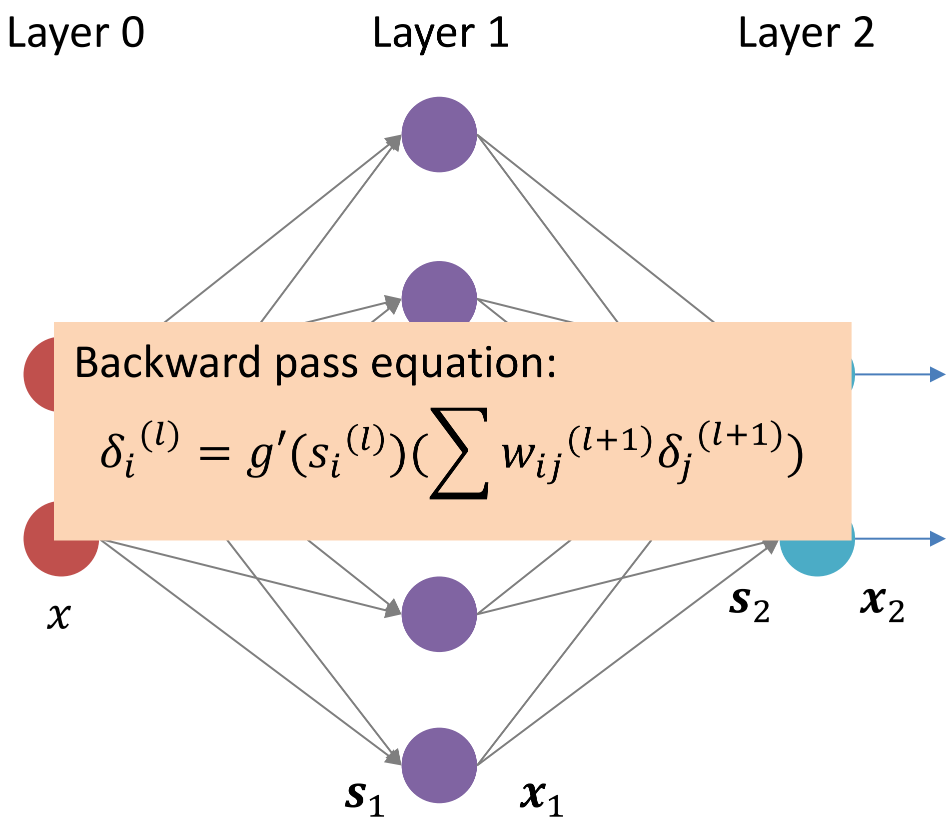
s_2, x_2: **M** × 2

M = the number of input data

Let x_i is the forward output of layer i (here x_1, x_2)

Let δ_i is the backward output of layer i (here δ_1, δ_2)

NN Classifier with one hidden layer



Backward pass equation:

$$\delta_i{}^{(l)} = g'(s_i{}^{(l)})(\sum w_{ij}{}^{(l+1)}\delta_j{}^{(l+1)})$$

Forward pass equations:

$$s_1 = xW_1 + b_1$$
$$x_1 = \tanh(s_1)$$
$$s_2 = x_1W_2 + b_2$$
$$h = x_2 = \mathrm{softmax}(s_2)$$

Backward pass equations:

$$\delta_2 = h - y$$

$$\delta_1 = g'(s_1)\delta_2 W_2^T$$
$$= (1 - \tanh^2 s_1)\circ\delta_2 W_2^T$$

Dimensions of matrices:

δ_2, x : M × 2

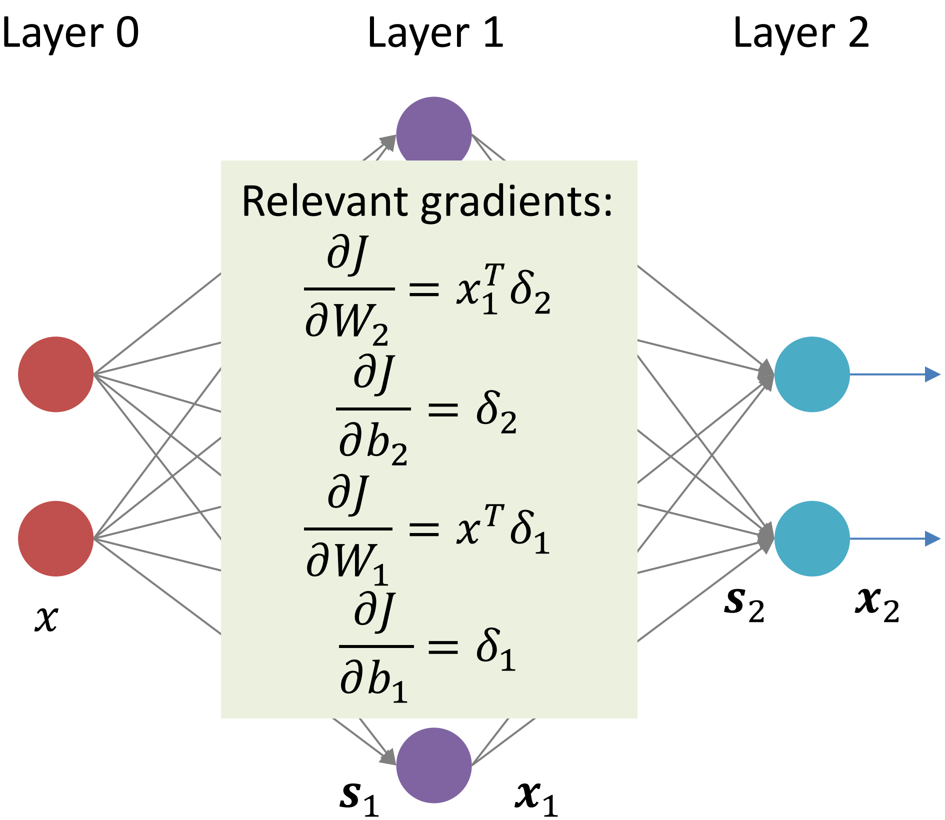
W_2 : 5 × 2

δ_1, s_1: M × 5

Let x_i is the forward output of layer i (here x_1, x_2)

Let δ_i is the backward output of layer i (here δ_1, δ_2)

NN Classifier with one hidden layer



Relevant gradients:

$$\frac{\partial J}{\partial W_2} = x_1^T \delta_2$$
$$\frac{\partial J}{\partial b_2} = \delta_2$$
$$\frac{\partial J}{\partial W_1} = x^T \delta_1$$
$$\frac{\partial J}{\partial b_1} = \delta_1$$

Forward pass equations:

$$s_1 = xW_1 + b_1$$
$$x_1 = \tanh(s_1)$$
$$s_2 = x_1 W_2 + b_2$$
$$h = x_2 = \mathrm{softmax}(s_2)$$

Backward pass equations:

$$\delta_2 = h - y$$
$$\delta_1 = (1 - \tanh^2 s_1) \circ \delta_2 W_2^T$$

Adjust weights:

$$W_2 = W_2 - \alpha x_1^T \delta_2$$
$$b_2 = b_2 - \alpha \delta_2$$
$$W_1 = W_1 - \alpha x^T \delta_1$$
$$b_1 = b_1 - \alpha \delta_1$$

Let x_i is the forward output of layer i (here x_1, x_2)
Let δ_i is the backward output of layer i (here δ_1, δ_2)

Framework for training Neural Network

1. Initialize all weights ${w_{ij}}^{(l)}$ at random
2. Repeat until convergence:
3. Pick a random example to feed into layer 0
4. *Forward*: Compute all ${x_j}^{(l)}$
5. *Backward*: Compute all ${\delta_j}^{(l)}$
6. Update weights: ${w_{ij}}^{(l)} \rightarrow {w_{ij}}^{(l)} - \alpha\, \boxed{{x_i}^{(l-1)} {\delta_j}^{(l)}}$
7. Return the final weights ${w_{ij}}^{(l)}$

$$\frac{\partial J(\boldsymbol{W})}{\partial {w_{ij}}^{(l)}}$$

$$\frac{\partial J}{\partial {s_j}^{(l)}}$$

Layer $\boldsymbol{\ell}$-1

Layer $\boldsymbol{\ell}$

$\boldsymbol{i}$ th node

${\boldsymbol{x}_{\boldsymbol{i}}}^{(\boldsymbol{l}-\boldsymbol{1})}$ = forward output of $\boldsymbol{i}$ th node

${\boldsymbol{\delta}_{\boldsymbol{j}}}^{(\boldsymbol{l})}$ =backward output of $\boldsymbol{j}$th node

$\boldsymbol{j}$ th node

${\boldsymbol{w}_{\boldsymbol{ij}}}^{(\boldsymbol{l})}$

Techniques to Improve Training

- **Choice of activation function**
- Training by mini-batches
- Finding the right learning rate
- Overfitting and techniques to avoid it
  1. Regularization
  2. Dropout (Minimal effort backpropagation)
  3. Early stopping

Key equations for backpropagation: $\boldsymbol{\delta_i}^{(\boldsymbol{l})}$

For output layer L: $\delta_i{}^{(L)} = g'(s_i{}^{(L)})(\frac{\partial J}{\partial h(x)})$

For earlier layers for l = L-1, …., 1 :

$$\delta_i{}^{(l)} = g'(s_i{}^{(l)})(\sum w_{ij}{}^{(l+1)} \delta_j{}^{(l+1)})$$

$$= g'(s_i{}^{(l)})(\sum w_{ij}{}^{(l+1)} g'(s_j{}^{(l+1)})(\sum w_{jk}{}^{(l+2)} \delta_k{}^{(l+2)}))$$

Key equations for backpropagation: $\boldsymbol{\delta_i}^{(\boldsymbol{l})}$

For output layer *L*: $\delta_i{}^{(L)} = g'(s_i{}^{(L)})(\frac{\partial J}{\partial h(x)})$

For earlier layers for l = *L*-1, …., 1 :

$$\delta_i{}^{(l)} = g'(s_i{}^{(l)})(\sum_j w_{ij}{}^{(l+1)}\delta_j{}^{(l+1)})$$

$$= g'(s_i{}^{(l)})(\sum w_{ij}{}^{(l+1)} g'(s_j{}^{(l+1)})(\sum w_{jk}{}^{(l+2)}\delta_k{}^{(l+2)}))$$

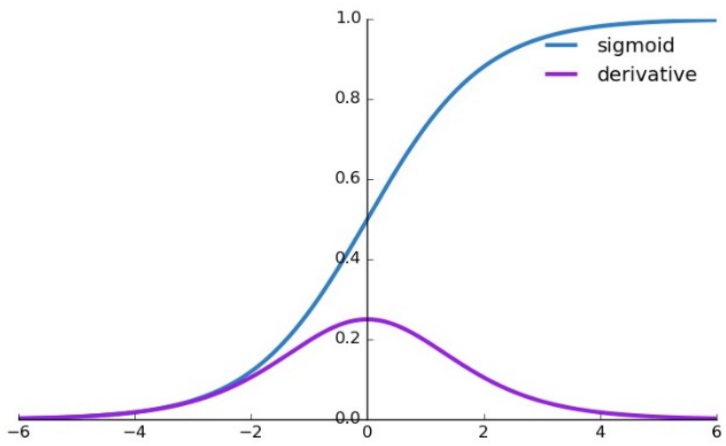
$= g'(\ )\ \ldots g'(\ )\ldots g'(\ )\ldots$ [*up to L occurrences*]

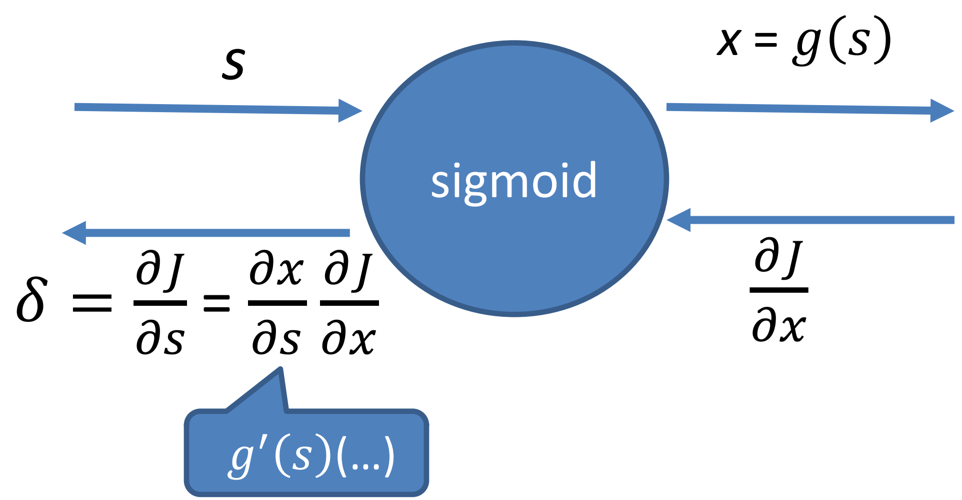
Gradient with **Sigmoid**

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

$$\rightarrow\ g'(s) = \big(1 - g(s)\big)g(s)$$

When $g'(s)$ is very small,
$\frac{\partial J}{\partial s}$ is very small
=> tiny steps in gradient descent!





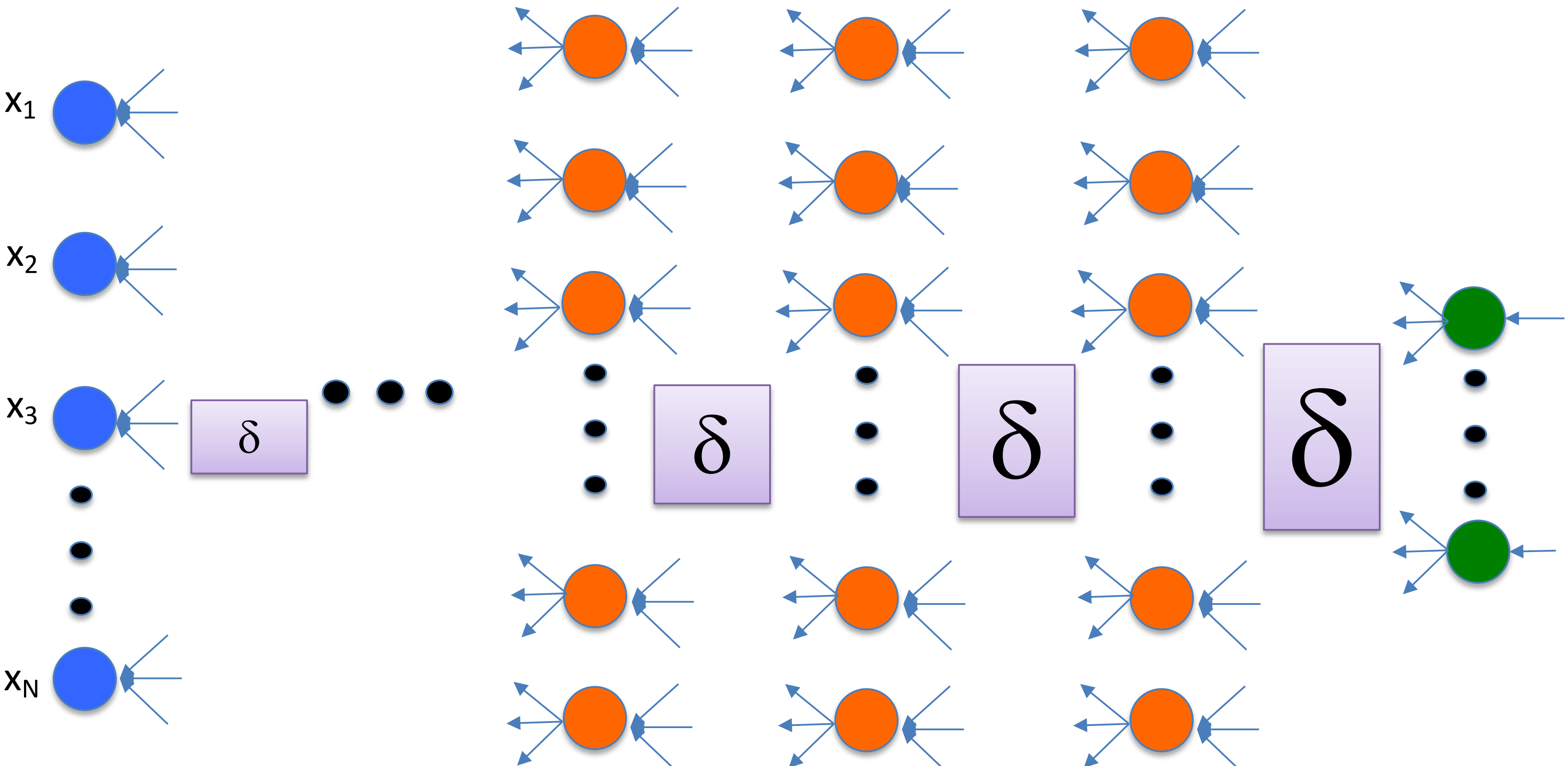
- When s is very small or very large, $g'(s)$ is very small.
- Even when s isn't, $g'(s) \leq 0.25$.

Vanishing Gradient Problem

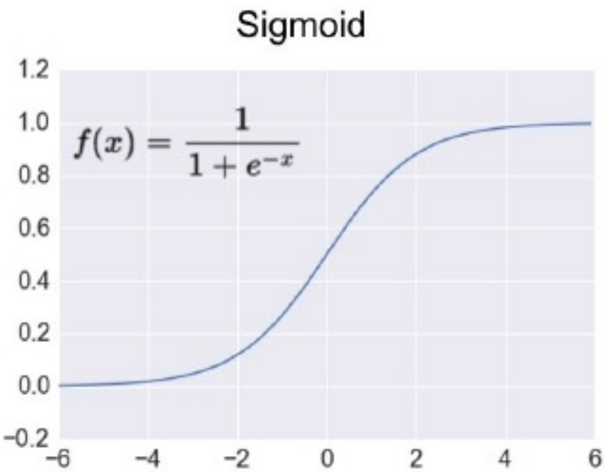
δ's get smaller and smaller as you go back because

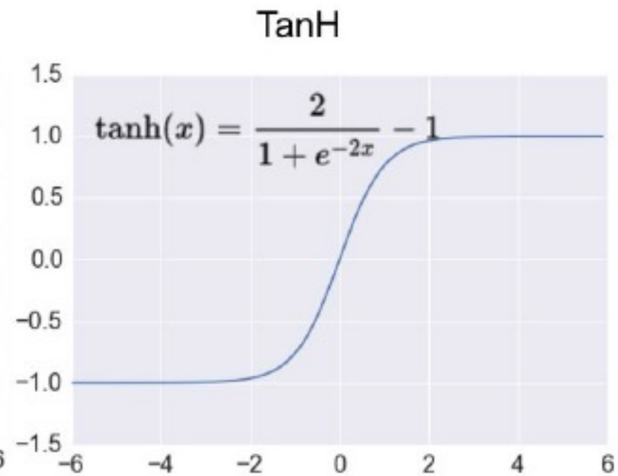
$\delta = g'(\,)\ldots g'(\,)\ldots g'(\,)\ldots$ and $g'(\,) \leq 0.25$

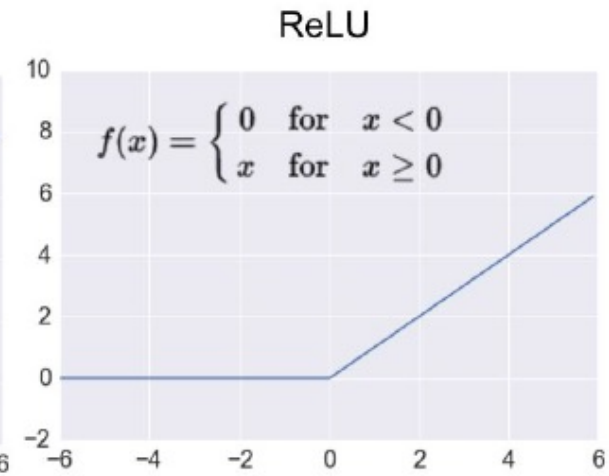
$\Rightarrow$ smaller changes to weights as you go back

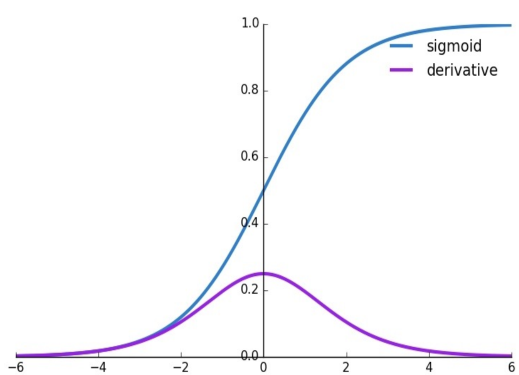


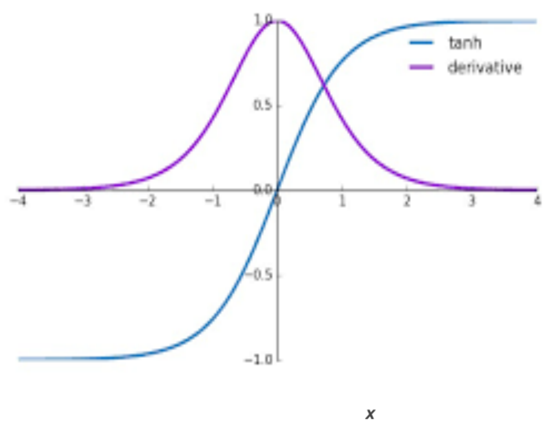
What about other activation functions?

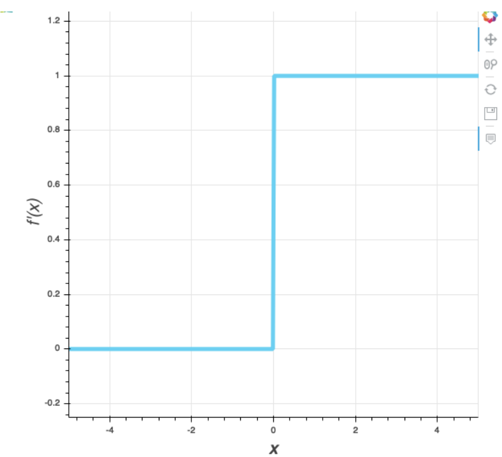












$$\boldsymbol{g}'(s) = \big(1 - g(s)\big)g(s)$$

$$\boldsymbol{g}'(s) = 1 - g(s)^2$$

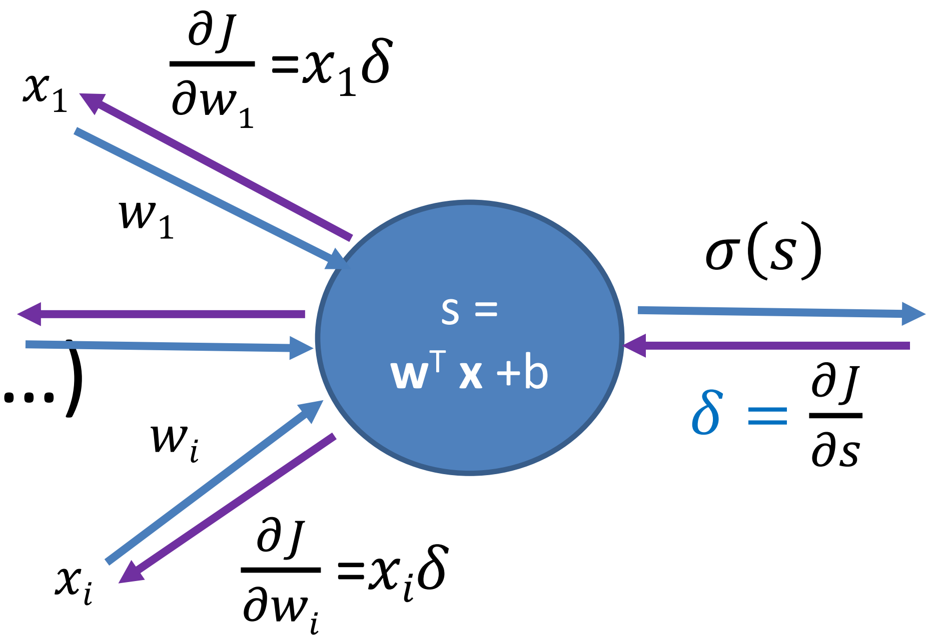
$$\boldsymbol{g}'(s) = \begin{cases} 0 & \text{if } s < 0 \\ 1 & \text{if } s \geq 0 \end{cases}$$

Not Zero-centered

$$\frac{\partial J}{\partial \boldsymbol{w}} = \left(\frac{\partial J}{\partial w_1}, \dots \frac{\partial J}{\partial w_i}, \dots \right)$$

$$= \left(\frac{\partial s}{\partial w_1}\frac{\partial J}{\partial s}, \dots \frac{\partial s}{\partial w_i}\frac{\partial J}{\partial s}, \dots \right)$$

$$= \left(x_1\delta, \dots, x_i\,\delta, \dots \right)$$



What happens when we have only positive values $x = \sigma(s)$ (i.e., sigmoid output)? What about $\frac{\partial J}{\partial \boldsymbol{w}}$?

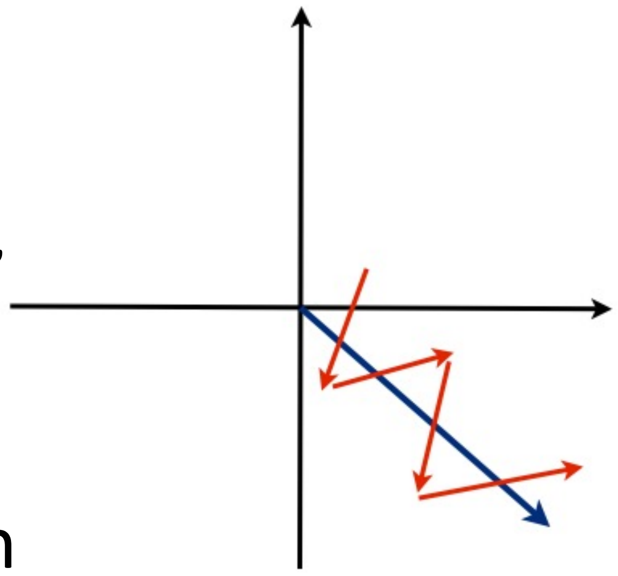
$(x_1\delta, \dots, x_i\,\delta, \dots)$ will be either all positive or negative depending on δ, i.e., all in the same direction.

Zig-Zag Path for Gradient Descent

Suppose the blue vector is the optimal direction for the gradient (assume a simple case with 2 weights).

Since the weights either all increase or decrease (same direction), then we get a zig-zag path, not good for gradient descent.

Similarly, non zero-centered activation functions are not ideal, they produce more positive or more negative output values.



Thus, we want activation function to be zero-centered.

Sigmoid Activation Function

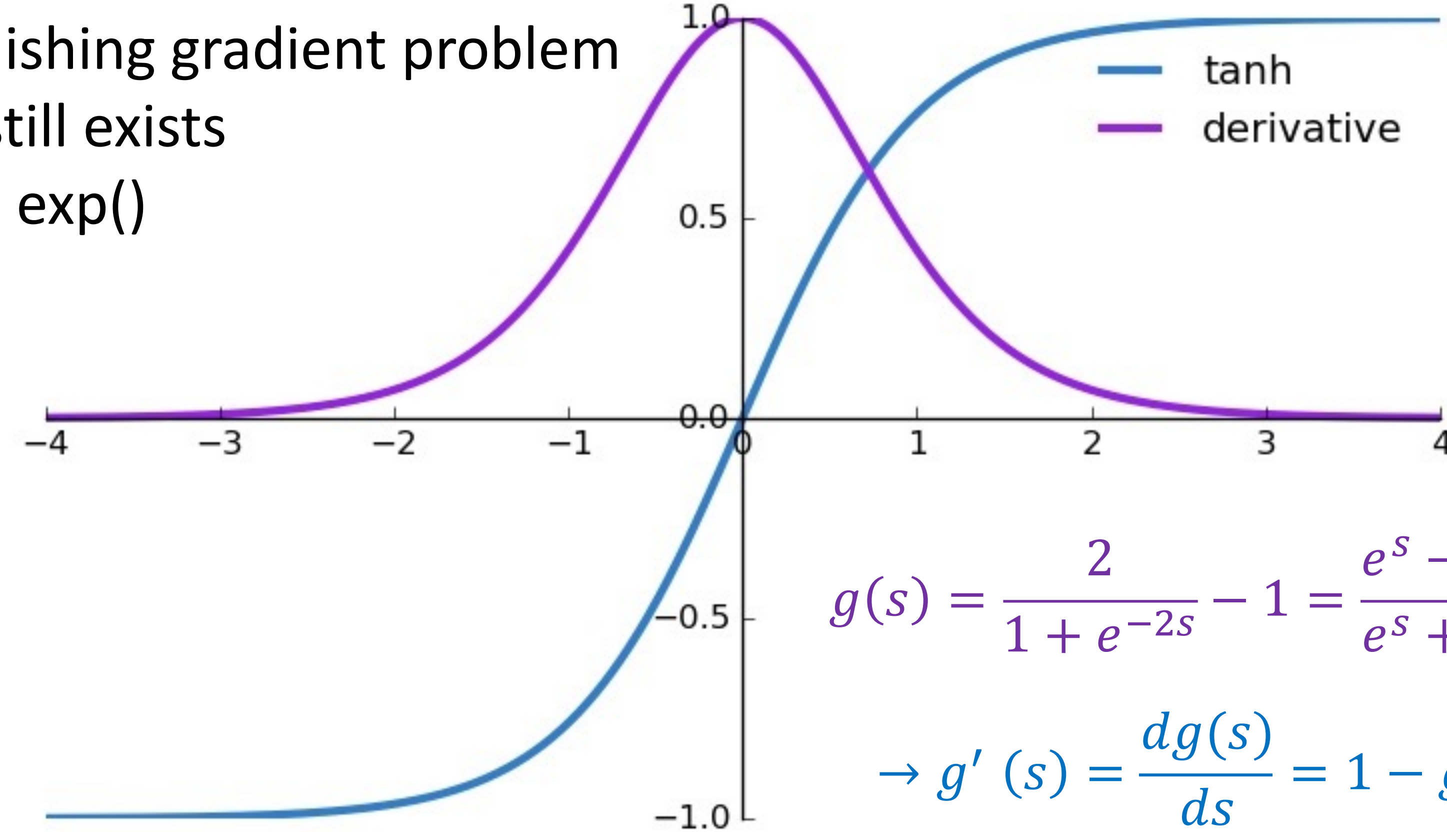
Historically popular because of nice interpretation of probability by squeezing numbers in [0, 1] for classification.

Shortcomings:

- Vanishing gradient
- Sigmoid outputs are not zero-centered (mean > 0)
- exp() is expensive to compute

Tanh and its derivative

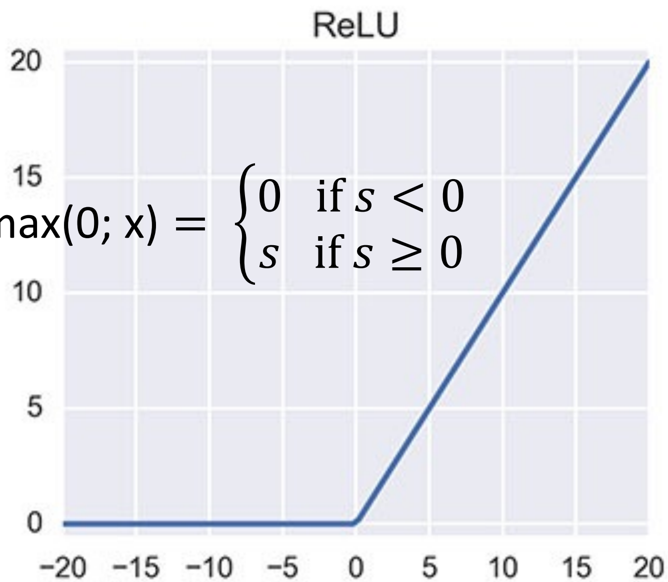
Squashes numbers in [-1, 1]
Zero centered
Vanishing gradient problem
    still exists
Still exp()



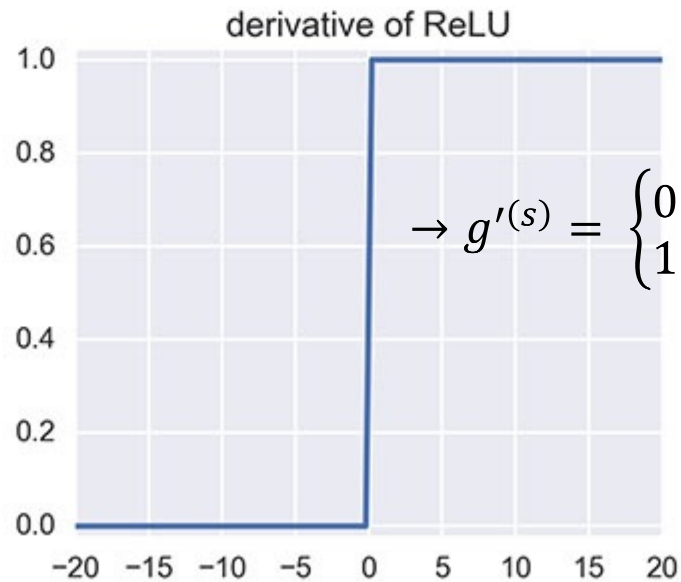
$$g(s) = \frac{2}{1+e^{-2s}} - 1 = \frac{e^{s} - e^{-s}}{e^{s} + e^{-s}}$$

$$\rightarrow g'(s) = \frac{dg(s)}{ds} = 1 - g(s)^2$$

ReLU (Rectified Linear Unit)



$$g(s) = \max(0; x) = \begin{cases} 0 & \text{if } s < 0 \\ s & \text{if } s \geq 0 \end{cases}$$



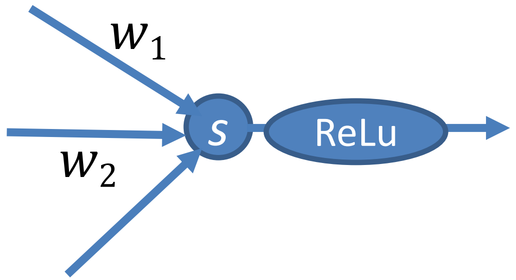
$$\rightarrow g'^{(s)} = \begin{cases} 0 & \text{if } s < 0 \\ 1 & \text{if } s > 0 \end{cases}$$

- No vanishing gradient problem when positive, but still exists when negative (gradient = 0 at 0)
- Converges much faster than sigmoid/tanh (a factor of 6)
- Computationally efficient
- Non-zero centered
- Widely used in CNN (since Alexnet) – to be learned later
- Seems to model better (neurons has the dropout effect)

ReLu-like Activation Functions

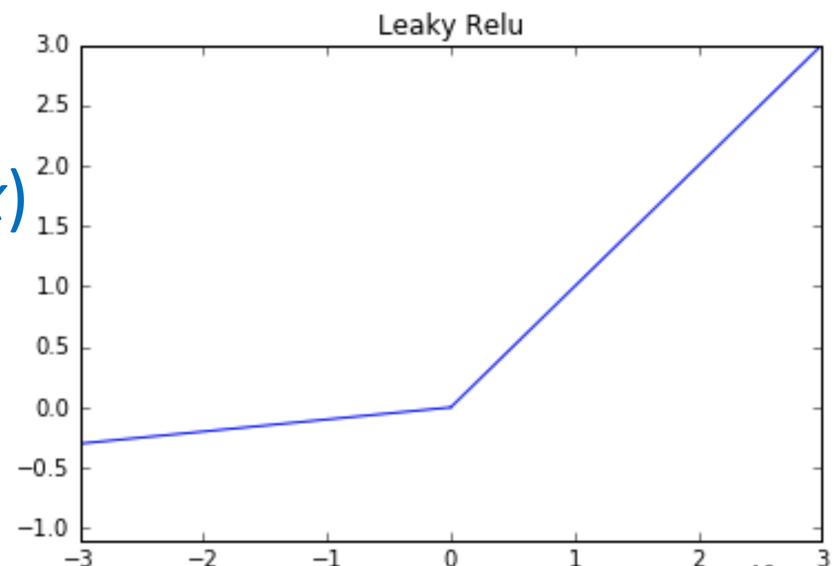
Problem for ReLu:

- Gradient is 0 for negative *s*
- Weights will not get adjusted
- Dying ReLu problem: neurons will stop (die) simply because gradient is 0, nothing changes (*s* remains –ve and gradient remains 0).
- As much as 40% of your network can be "dead" (i.e., neurons never activate across the entire training dataset)



Leaky Relu - $f(x) = \max(0.01x, x)$

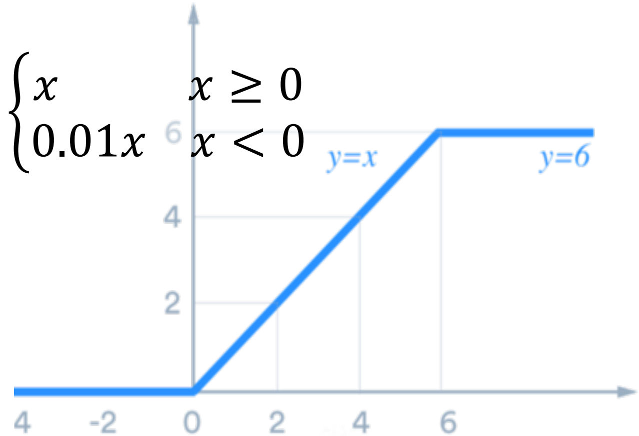
- All benefits of ReLU
- closer to zero-centered outputs
- nonzero gradient when negative



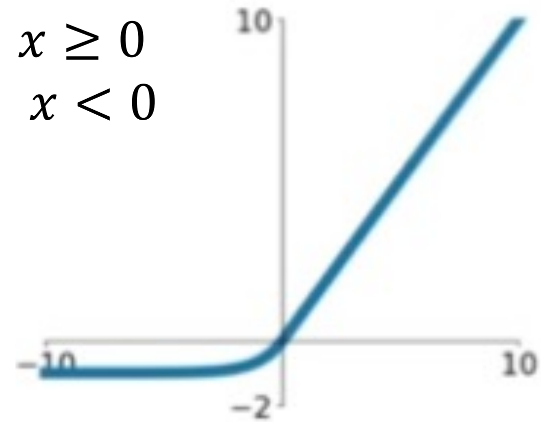
ReLu-like Activation Functions

Variations in ReLu

- Leaky ReLu - $f(x)$ = max $(0.01x, x)$ $\quad f(x) = \begin{cases} x & x \geq 0 \\ 0.01x & x < 0 \end{cases}$
- Parametric ReLu - $f(x)$ = max (ax, x)
- ReLu6 - $f(x) = \min(\max(x, 0), 6)$
- clipping the gradient at some value n



Exponential Linear Units (ELU): $f(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$

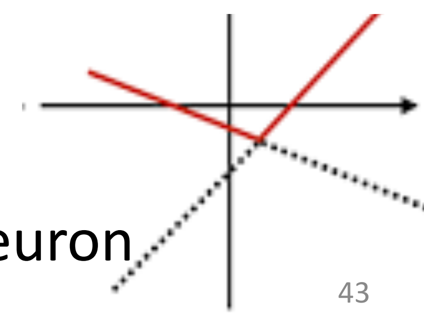


- Similar to leaky ReLu
- Differentiable at 0 when α=1
- Drawback: Higher computation for exp()

Maxout "Neuron"

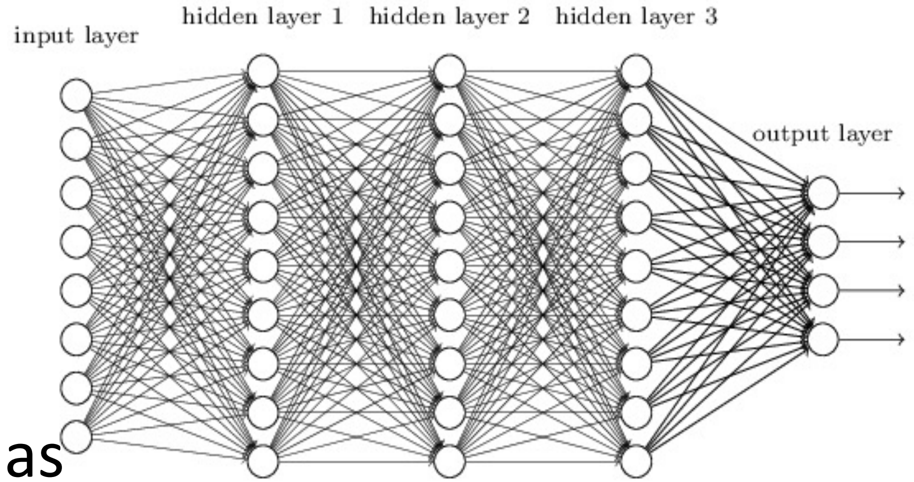
Max $(w_1x + b_1, w_2x + b_2)$

- Generalizes ReLu and Leaky ReLu
- Drawback: doubles number of parameters per neuron



Maxout (n=2)

Issue with Symmetry



Network Initialization?

Neural Network with

- same activation function
- same no. of neurons/layer
- same initial weights and bias

Problem?

- Same activation values in each layer
- Same gradient at every neuron in each layer

Limited number of directions for gradient descent

Solution: weight initialization –
sample from the normal distribution $N(0,10^{-2})$
with bias=0

Techniques

- Choice of activation function
- **Training by mini-batches**
- Finding the right learning rate
- Overfitting and techniques to avoid
  1. regularization
  2. Dropout (Minimal effort backpropagation)
  3. Early stopping
- Error Analysis

Gradient Descent (GD)

- Minimize $J(\theta) = \frac{1}{M}\sum_{i=1}^{M} \mathrm{e}\left(h(\boldsymbol{x}_i), y_i\right)$
- By taking iterative steps along $-\nabla J(\theta)$:

$$\Delta\theta = -\alpha \nabla J(\theta)$$

- ∇J computed based on all M examples

Stochastic Gradient Descent (SGD)

- Minimize $J(\theta) = \frac{1}{M}\sum_{i=1}^{M} \mathrm{e}\,(h(\boldsymbol{x}_i), y_i)$
- By taking iterative steps along $-\nabla J(\theta)$:

$$\Delta\theta = -\alpha\nabla J(\theta)$$

- ∇J computed based on all M examples
- Stochastic GD: randomly pick one $(\boldsymbol{x}_i, y_i)$
- Expected value of negative gradient of $(\boldsymbol{x}_i, y_i)$:

$$E[-\nabla \mathrm{e}(h(\boldsymbol{x}_i), y_i)] = \frac{1}{M}\sum_{i=1}^{M} -\nabla\, \mathrm{e}(h(\boldsymbol{x}_i), y_i) = -\nabla J$$

Stochastic Gradient Descent (SGD)

- Minimize $J(\theta) = \frac{1}{M}\sum_{i=1}^{M} \mathrm{e}\left(h(\boldsymbol{x}_i), y_i\right)$
- By taking iterative steps along $-\nabla J(\theta)$:

$$\Delta\theta = -\alpha\nabla J(\theta)$$

- ∇J computed based on all M examples
- Stochastic GD : randomly pick one $(\boldsymbol{x}_i, y_i)$
- Expected value of negative gradient of $(\boldsymbol{x}_i, y_i)$:

$$E[-\nabla \mathrm{e}(h(\boldsymbol{x}_i), y_i)] = \frac{1}{M}\sum_{i=1}^{M} -\nabla\, \mathrm{e}(h(\boldsymbol{x}_i), y_i) = -\nabla J$$

Alternatively: SGD can pick few samples.

Mini-batch SGD

- Minimize $J(\theta) = \frac{1}{B}\sum_{i=1}^{B} \mathrm{e}\left(h(x_i), y_i\right)$
- By taking iterative steps along $-\nabla J(\theta)$:

$$\Delta\theta = -\alpha\nabla J(\theta)$$

- ∇J computed based on ~~all~~ B examples

SGD Algorithm

1. Initialize all weights ${w_{ij}}^{(l)}$ at random
2. Repeat until convergence:
3. Pick a mini-batch of (can be one) examples to feed into layer 0
4. *Forward*: Compute all ${x_j}^{(l)}$
5. Compute error
6. *Backward*: Compute all ${\delta_j}^{(l)}$
7. Update weights: ${w_{ij}}^{(l)} \rightarrow {w_{ij}}^{(l)} - \alpha \, {x_i}^{(l-1)} {\delta_j}^{(l)}$
8. Return the final weights ${w_{ij}}^{(l)}$

Mini-batches in SGD

- Instead of updating weights based on single random example as input, use a **batch** of random examples
- Advantages:
  - More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates
  - Mini-batches lead to fast training!
  - Can parallelize computation + achieve significant speed increases on GPU’s

Techniques

- Choice of activation function
- Training by mini-batches
- **Finding the right learning rate**
- Overfitting and techniques to avoid
    1. regularization
    2. Dropout (Minimal effort backpropagation)
    3. Early stopping

Navigating a Complex Terrain

https://towardsdatascience.com/the-math-behind-adam-optimizer-c41407efe59b

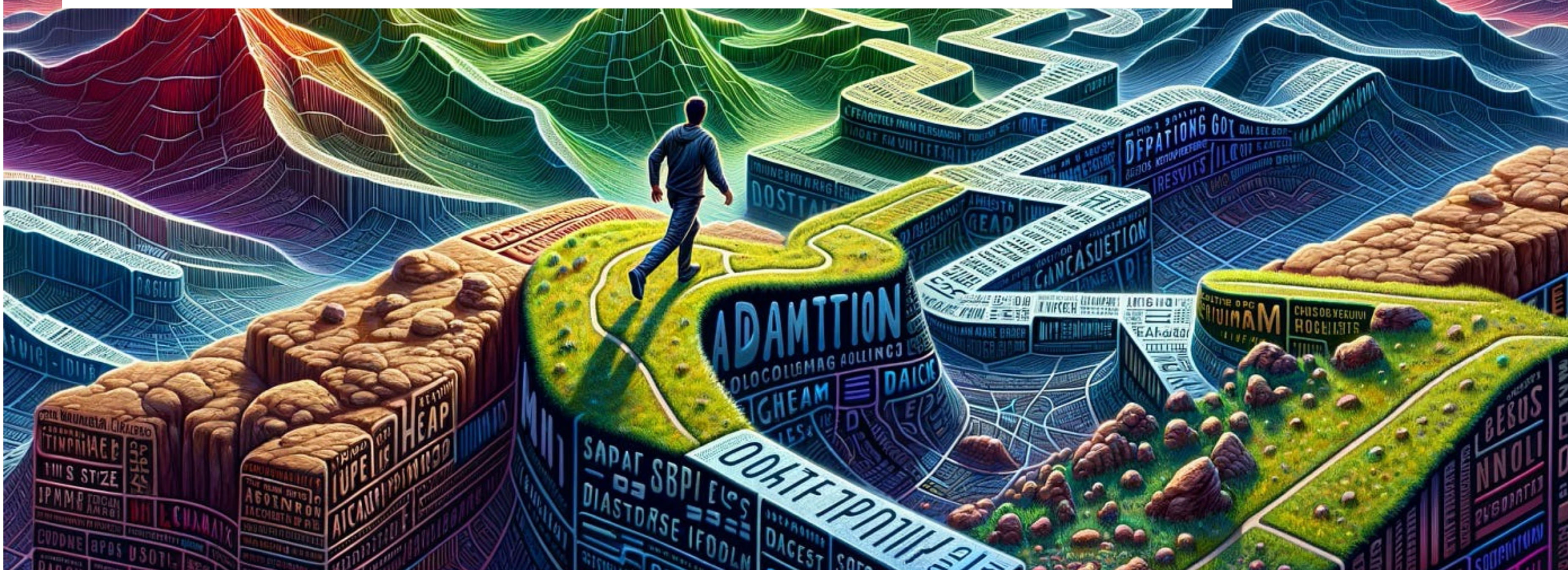
Large strides for some area and cautious steps for others

Take small steps when gradient changes rapidly and unpredictably

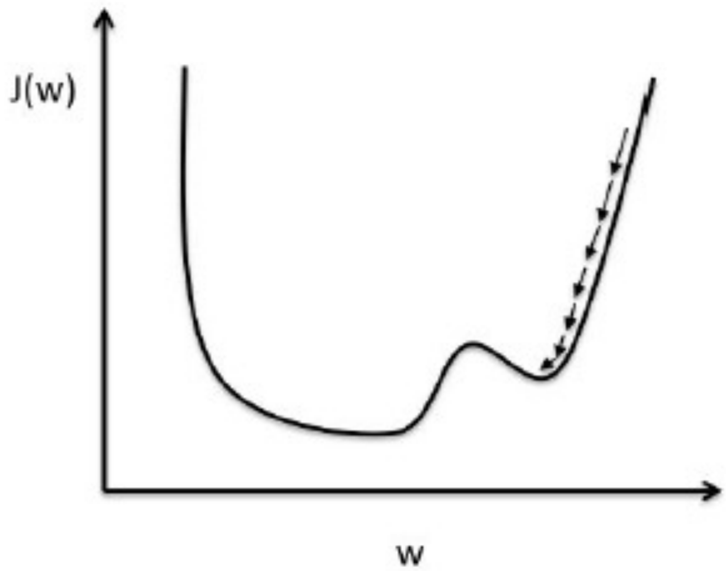
large when gradient changes slowly or predictably

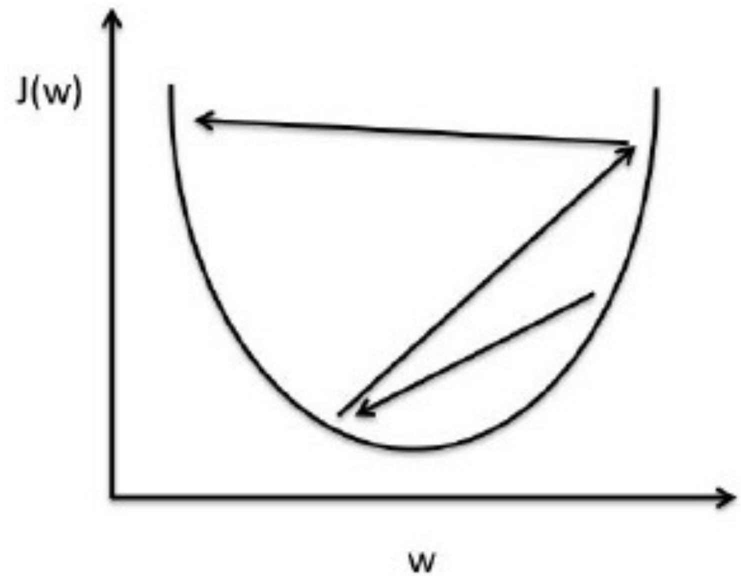
Maintain individual learning rates for each parameter in your model

Rates adjusted as training progress, based on the data encountered



Problem: Learning Rate α





Small learning rate:

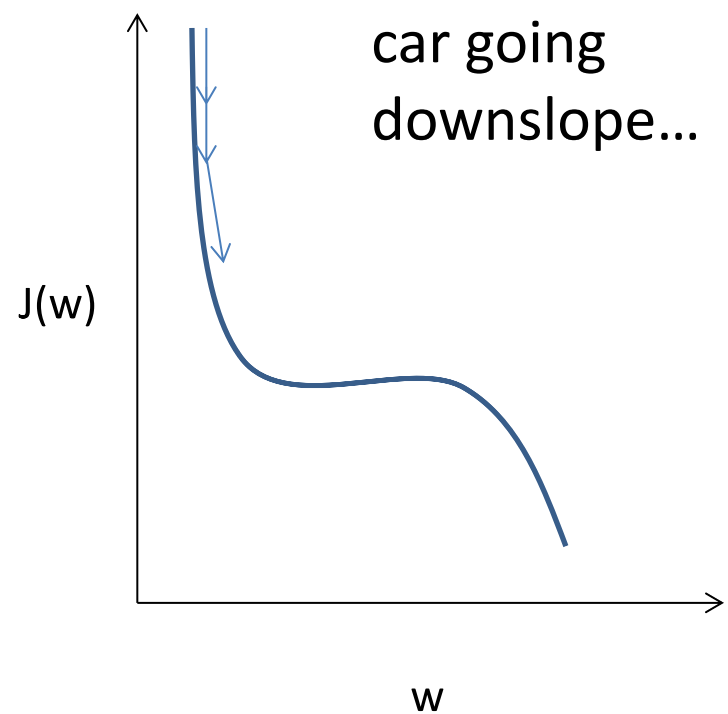
- Many iterations till convergence
- Trapped in local minimum

Large learning rate:

- Overshooting
- No convergence

Momentum

- Trapped in a local minimum: What if you get stuck in a flattish region where gradient is small?
- Let past steps affect current step so that you keep on moving in the direction you were heading
- Weights updated by a function of past steps and current gradient



Momentum

- Weights updated by a function of past steps and current gradient
- Introduce new variable to capture past steps:

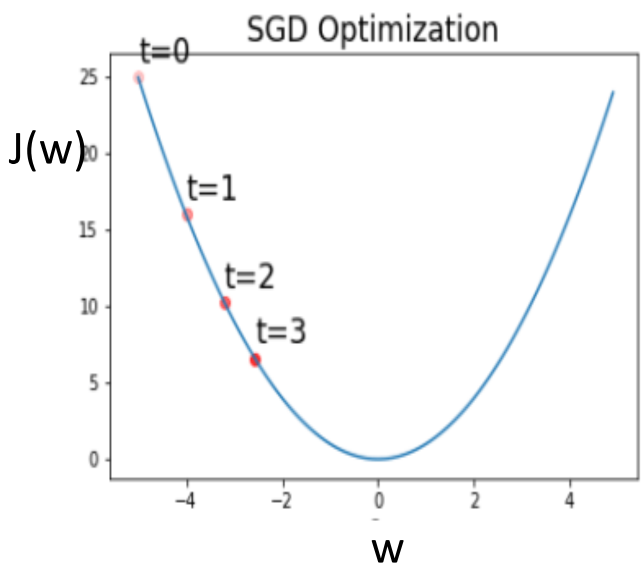
$$v_t = \text{velocity at time } t$$

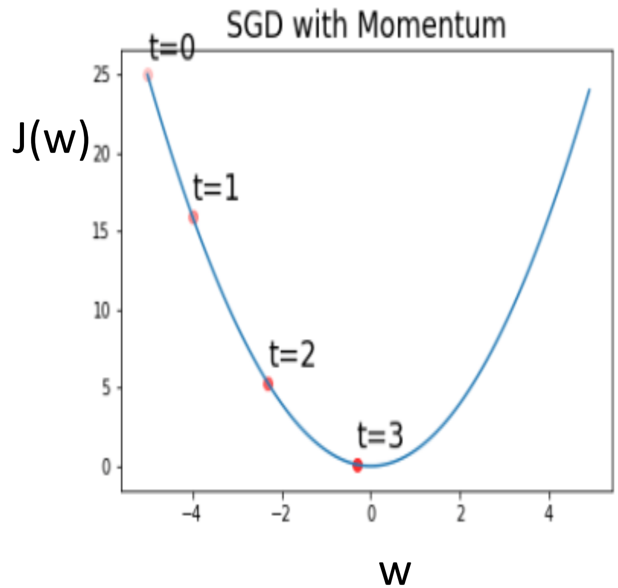
- Update of weights: $w_t \leftarrow w_{t-1} + v_t$
- Update of velocity:

$$v_t \leftarrow \mu v_{t-1} - \alpha \nabla J(w_{t-1})$$

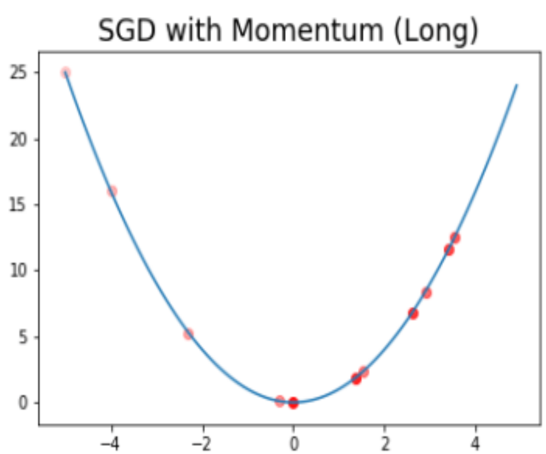
$$\left[\text{with } v_1 = -\alpha \nabla J(w_0)\right]$$

  - Momentum step: μv_{t-1} [μ *is typically set to 0.9*]
  - Gradient step: $-\alpha \nabla J(w_{t-1})$





Momentum gets to minimum faster



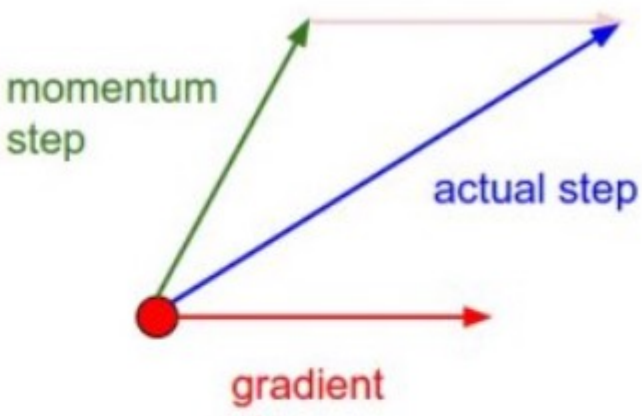
But overshoots the minimum if run too long

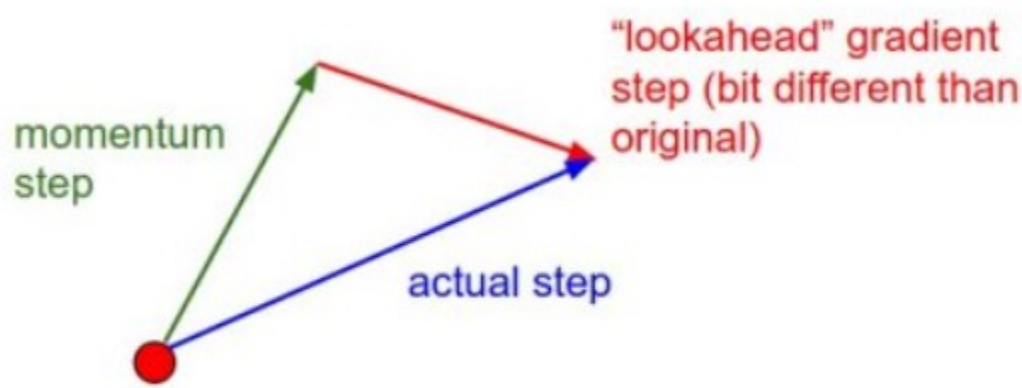
Wouldn't it be great if we could: detect the overshooting, i.e. change in the direction of the gradient earlier?

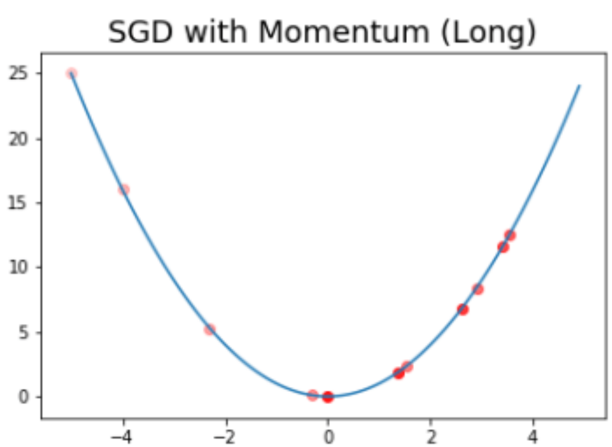
Momentum vs. Nesterov Momemtum

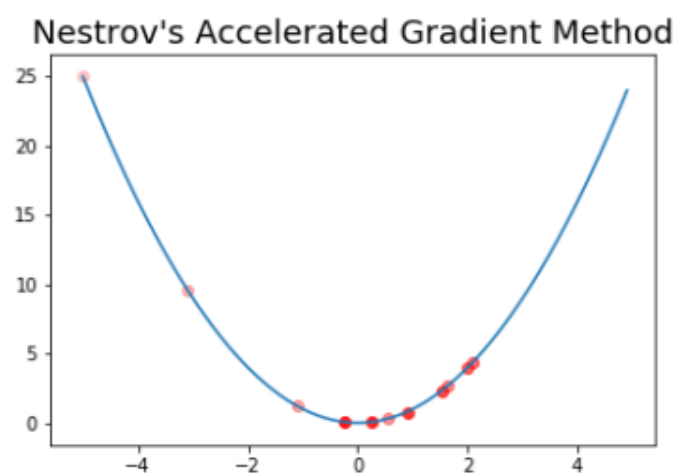
- $v_t \leftarrow \mu v_{t-1} - \alpha \nabla J(w_{t-1})$
- $w_t \leftarrow w_{t-1} + v_t$

- $v_t \leftarrow \mu v_{t-1} - \alpha \nabla J(w_{t-1} + \mu v_{t-1})$
- $w_t \leftarrow w_{t-1} + v_t$









https://pytorch.org/docs/stable/generated/torch.optim.SGD.html?highlight=sgd#torch.optim.SGD

SGD

CLASS torch.optim.SGD(*params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False, *, maximize=False, foreach=None, differentiable*

Implements stochastic gradient descent (optionally with momentum).

$$
\begin{aligned}
&\textbf{input} : \gamma \text{ (lr)}, \ \theta_0 \text{ (params)}, \ f(\theta) \text{ (objective)}, \ \lambda \text{ (weight decay)}, \\
&\hspace{3em} \mu \text{ (momentum)}, \ \tau \text{ (dampening)}, \ \textit{nesterov}, \ \textit{maximize} \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{if } \lambda \neq 0 \\
&\quad\quad g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\quad \textbf{if } \mu \neq 0 \\
&\quad\quad \textbf{if } t > 1 \\
&\quad\quad\quad \mathbf{b}_t \leftarrow \mu \mathbf{b}_{t-1} + (1-\tau) g_t \\
&\quad\quad \textbf{else} \\
&\quad\quad\quad \mathbf{b}_t \leftarrow g_t \\
&\quad\quad \textbf{if } \textit{nesterov} \\
&\quad\quad\quad g_t \leftarrow g_t + \mu \mathbf{b}_t \\
&\quad\quad \textbf{else} \\
&\quad\quad\quad g_t \leftarrow \mathbf{b}_t \\
&\quad \textbf{if } \textit{maximize} \\
&\quad\quad \theta_t \leftarrow \theta_{t-1} + \gamma g_t \\
&\quad \textbf{else} \\
&\quad\quad \theta_t \leftarrow \theta_{t-1} - \gamma g_t \\
&\textbf{return } \theta_t
\end{aligned}
$$

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

• NOTE

The implementation of SGD with Momentum/Nesterov subtly differs from Sutskever et. al. and implementations in some other frameworks.

Considering the specific case of Momentum, the update can be written as

$$
\begin{aligned}
v_{t+1} &= \mu * v_t + g_{t+1}, \\
p_{t+1} &= p_t - \text{lr} * v_{t+1},
\end{aligned}
$$

where p, g, v and μ denote the parameters, gradient, velocity, and momentum respectively.

This is in contrast to Sutskever et. al. and other frameworks which employ an update of the form

$$
\begin{aligned}
v_{t+1} &= \mu * v_t + \text{lr} * g_{t+1}, \\
p_{t+1} &= p_t - v_{t+1}.
\end{aligned}
$$

The Nesterov version is analogously modified.

- $v_t \leftarrow \mu v_{t-1} - \alpha \nabla J(w_{t-1})$
- $w_t \leftarrow w_{t-1} + v_t$

AdaGrad (separate learning rate for each w)

- **Ada**ptive **Grad**ient
- Idea:

Historical gradients for w	Learning rate for w
High	Low
Low	High

- Introduce variable **G** to capture historical gradient information (second moment), with the following update rule :

$$G_t \leftarrow G_{t-1} + (\nabla J(w_{t-1}))^2$$

- Update weights:

$$w_t \leftarrow w_{t-1} - \frac{\alpha}{\sqrt{G_{t-1}+\epsilon}} \nabla J(w_{t-1})$$

ϵ (epsilon) is a small scalar (e.g., 10^{-8}) added to prevent division by zero and maintain numerical stability.

- Learning rate diminishes quickly as G gets bigger and bigger

AdaGrad vs. RMSProp

AdaGrad	
$G \leftarrow G + (\nabla J(w))^2$	G gets bigger and bigger
$w \leftarrow w - \frac{\alpha}{\sqrt{G+\epsilon}} \nabla J(w)$	coefficient gets smaller and smaller

RMSProp	
$G \leftarrow \beta G + (1-\beta)\,(\nabla J(w))^2$	β is the decay rate (typically 0.9, 0.99,..) preventing G from getting too big
$w \leftarrow w - \frac{\alpha}{\sqrt{G+\epsilon}} \nabla J(w)$	Same update rule for w but problem goes away

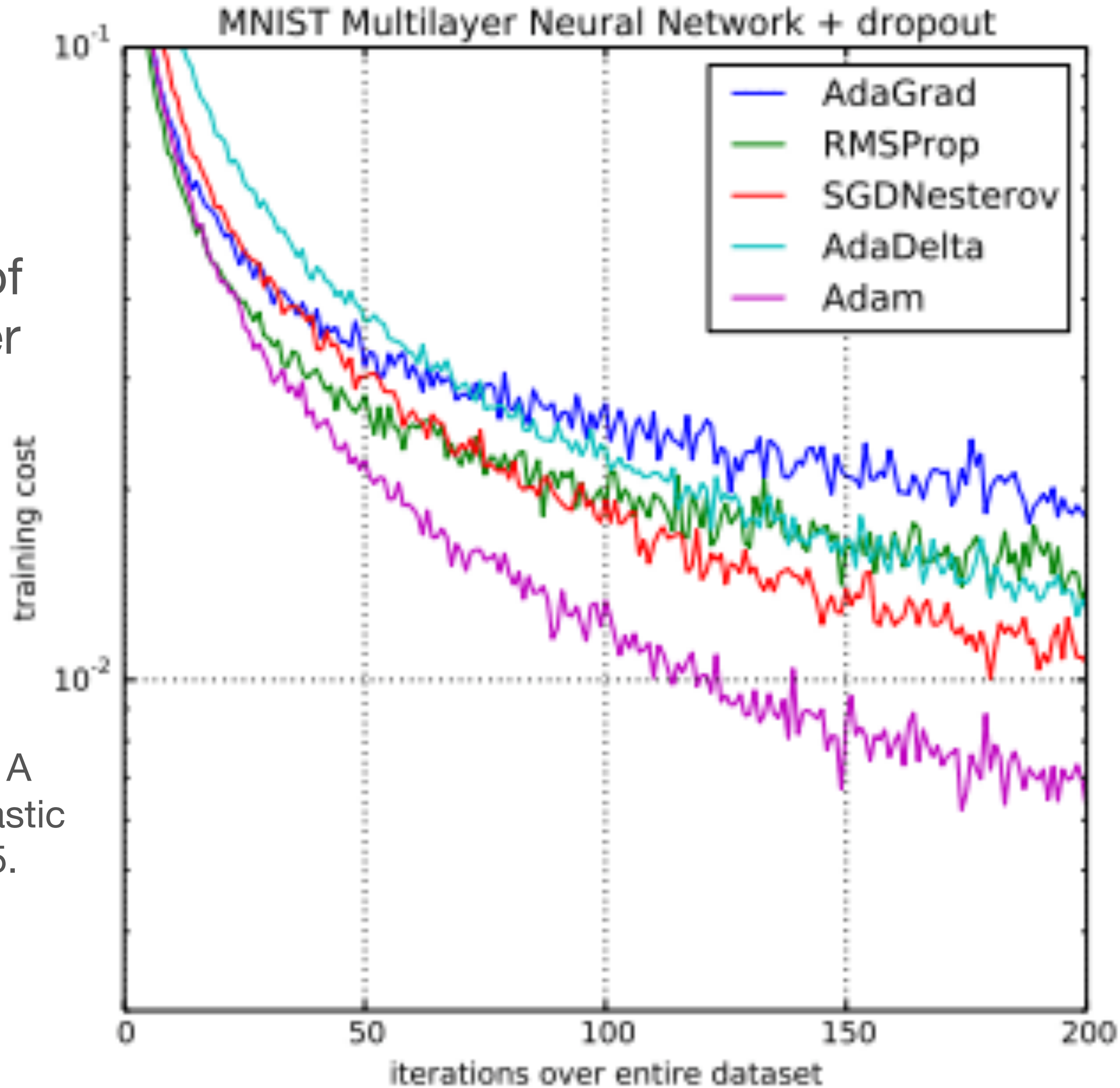
RMSProp vs. Adam

RMSProp	
$G \leftarrow \beta G + (1-\beta)\,(\nabla J(w))^2$	G does not get too big
$w \leftarrow w - \frac{\alpha}{\sqrt{G+\epsilon}} \nabla J(w)$	Same update rule as AdaGrad

Adam	
$v \leftarrow \beta v + (1-\beta)\,(\nabla J(w))^2$	Same idea as RMSProp (except called v instead of G)
$m \leftarrow \gamma m + (1-\gamma)\,(\nabla J(w))$	New first order variable m introduced to "smooth" gradient
$w \leftarrow w - \frac{\alpha}{\sqrt{v+\epsilon}} m$	Both v and m used to update w

Parameter values: β=0.999, γ=0.9, α=0.001 (10^{-5}, 0.3), $\epsilon=10^{-8}$

Comparison of Adam to Other Optimization Algorithms Training a Multilayer Perceptron



Taken from Adam: A Method for Stochastic Optimization, 2015.

https://pytorch.org/docs/stable/generated/torch.optim.Adam.html?highlight=adam#torch.optim.Adam

ADAM

CLASS torch.optim.Adam(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, ***, *foreach=None*, *maximize=False*, *capturable=False*, *differentiable=False*, *fused=False*) [SOURCE]

Implements Adam algorithm.

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
$\quad\lambda$ (weight decay), $amsgrad$, $maximize$

initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** $\ldots$ **do**
- **if** $maximize$:
  - $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
- **else**
  - $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
- **if** $\lambda \neq 0$
  - $g_t \leftarrow g_t + \lambda\theta_{t-1}$
- $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
- $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
- $\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$
- $\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$
- **if** $amsgrad$
  - $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
  - $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t}/(\sqrt{\widehat{v_t}^{max}} + \epsilon)$
- **else**
  - $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t}/(\sqrt{\widehat{v_t}} + \epsilon)$

---

return θ_t

Adam

$v \leftarrow \beta v + (1-\beta)\,(\nabla J(w))^2$

$m \leftarrow \gamma m + (1-\gamma)\,(\nabla J(w))$

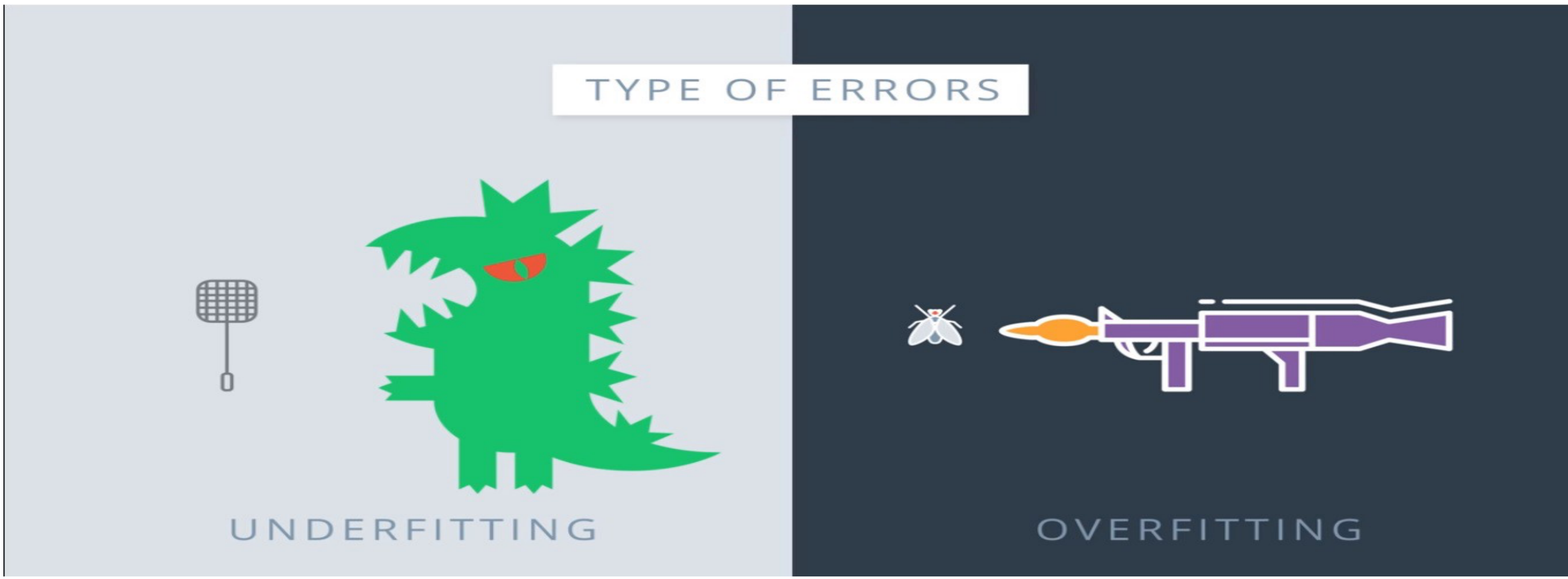
$w \leftarrow w - \frac{\alpha}{\sqrt{v}+\epsilon}\, m$

Techniques

- Choice of activation function
- Training by mini-batches
- Finding the right learning rate
- **Overfitting and techniques to avoid**
  1. regularization
  2. Dropout (Minimal effort backpropagation)
  3. Early stopping

Which NN is suitable for my problem? # of Neurons and Layers?

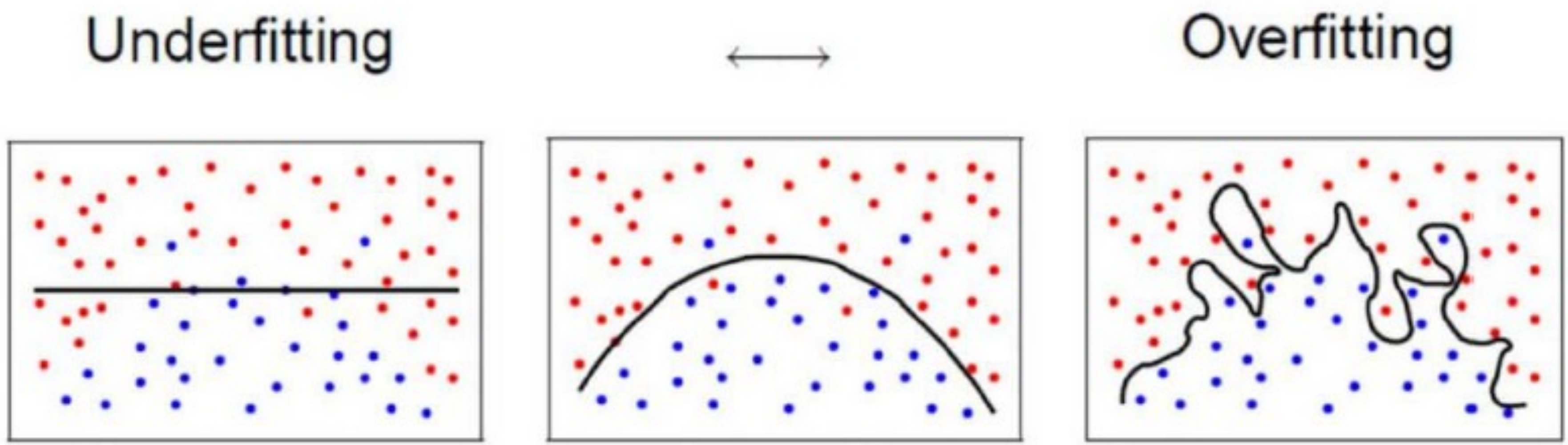
Mistakes in Life



Kill Godzilla using a flyswatter

Kill a fly using a machine gun

Problem of Overfitting



NN with a few neurons

NN with many layers and neurons

Difficult to find the right model

Approach:

Start with an overly complicated model

Then apply certain techniques to prevent overfitting

(Analogy: buying clothing without knowing the size)

Overfitting is a familiar concept

Memorization vs Generalization

Solution 1: Weight Regularization

Cost function *J* includes another term known as the regularization term

Cost function = Loss (say, binary cross entropy)
+ Regularization term

- This regularization term is to suppress the values of weight matrices because a neural network with smaller weights leads to simpler models (reduce overfitting).

Solution 1: Weight Regularization

- Observation: Large weights $\rightarrow$ overfitting
- Add the size of the weights to loss function

$$\min \ \frac{1}{M}\sum_i J(h_\theta(\boldsymbol{x}_i), y_i) + \lambda \sum_j |\theta_j|$$

L1 regularization
(Lasso Regression)

$$\min \ \frac{1}{M}\sum_i J(h_\theta(\boldsymbol{x}_i), y_i) + \lambda \sum_j {\theta_j}^2$$

L2 regularization
(Ridge Regression)

Solution 1: Weight Regularization

- Observation: Large weights $\rightarrow$ overfitting
- Add the size of the weights to loss function

$$\min \ \frac{1}{M}\sum_i J(h_\theta(\boldsymbol{x}_i), y_i) + \lambda \sum_j |\theta_j|$$

L1 regularization (Lasso Regression)

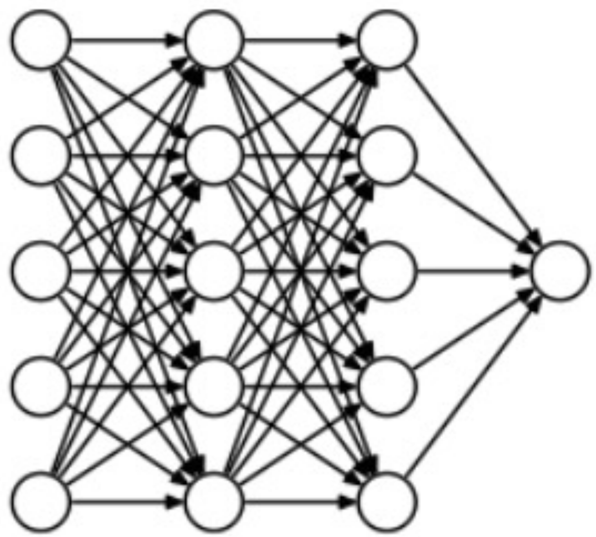
Sparse output (lots of zeros) $\rightarrow$ feature selection

$$\min \ \frac{1}{M}\sum_i J(h_\theta(\boldsymbol{x}_i), y_i) + \lambda \sum_j {\theta_j}^2$$

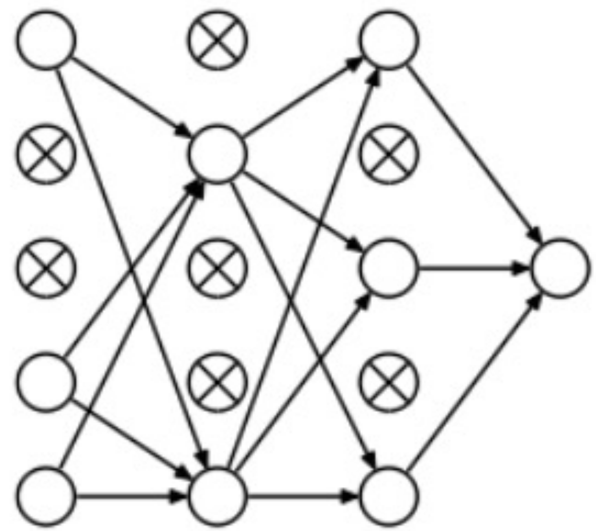
L2 regularization (Ridge Regression)

Computationally efficient (analytical solutions)

Solution 2: Dropout



Standard Neural Network

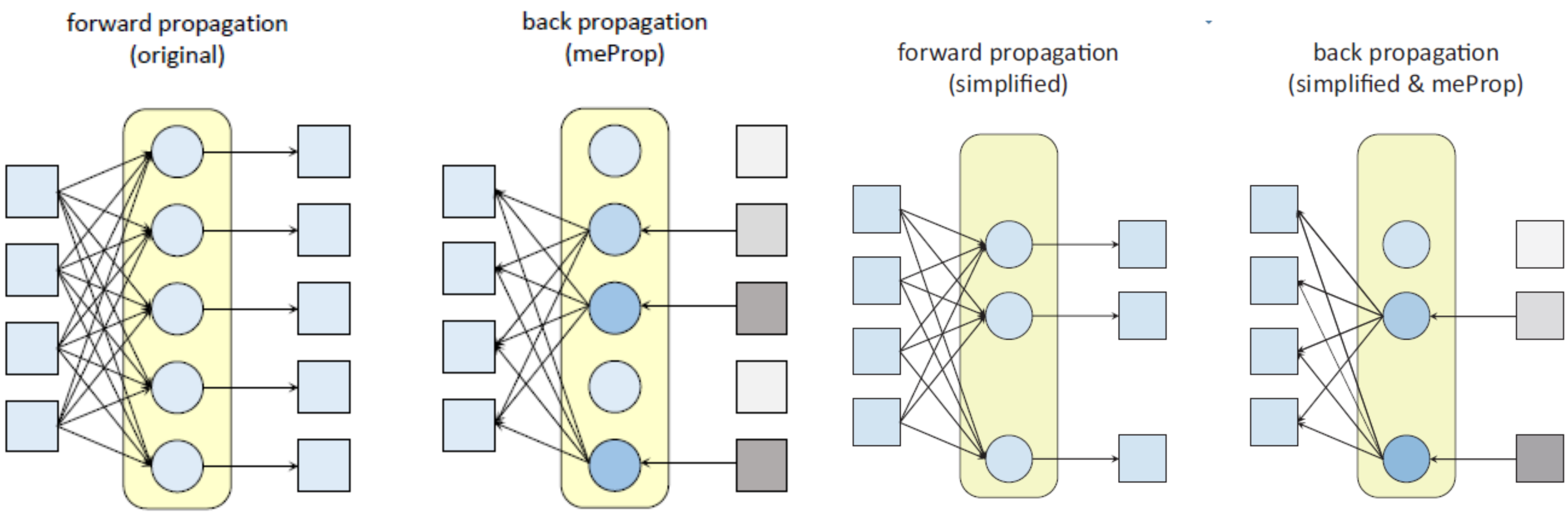


After applying dropout – Randomly select 50% to drop out, prevents reliance on any one neuron

Minimal Effort Back Propagation Method

- Work of Xu SUN et. al. of Peking University (Nov 2017)
- Motivation 1: Reduce cost of costly backprop
- Only a **small subset** of full gradient is computed to update weights (only *k* rows or columns of weight matrix modified)
- Most of the time only **need to update < 5%** of the weights at each back propagation pass
- **Accuracy improved** rather than degraded
- Motivation 2: Network size reduction
- Size of network could be reduced by up to **9x**
- **Accuracy** of simplified model **improved** rather than degraded

Only keep essential features



Model is trained for several iterations. Keep track of the "activeness" of path – indicated by shade of neurons.

Model simplification – inactive neurons are eliminated, repeat as needed

Solution 3: Early Stopping

- The choice of the number of training epochs is important for training neural networks
- Too many epochs can lead to overfitting, whereas too few may result in an underfit model.
- Early stopping is a method that allows you to stop training once the model performance stops improving (how to detect this?).

How did we learn in school?

Asked **Questions**

given **Answers**

Questions & Answers

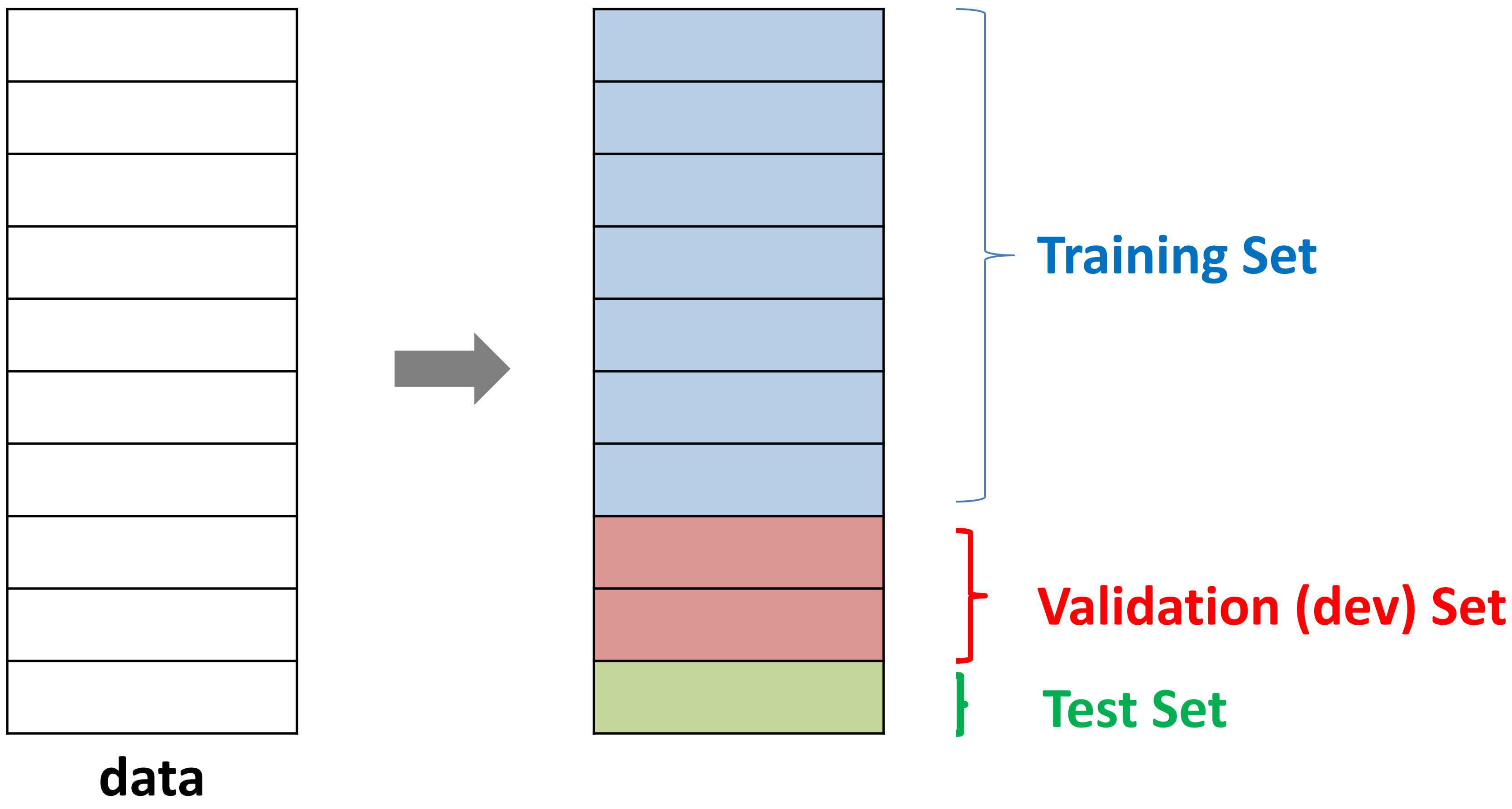
→ Practice exercises (**training data**)

→ Mock exams (**validation**)

→ Final exams (**testing**)

Supervised Learning

Divide up the data



Standard split ratios: (50,25,25) or (60,20,20)

Iterative Process

1. Train the model with the **training** set.
2. Evaluate the performance on the **training** set and **validation** set.
3. Repeat Step 1 & 2 until performance (accuracy) stops improving. [EARLY STOPPING]
4. If performance is unsatisfactory, improve model and repeat Steps 1-4.
5. Evaluate performance on **test** set.

Early Stopping by Learning Curves

- Don't give the network time to over-fit

  ...

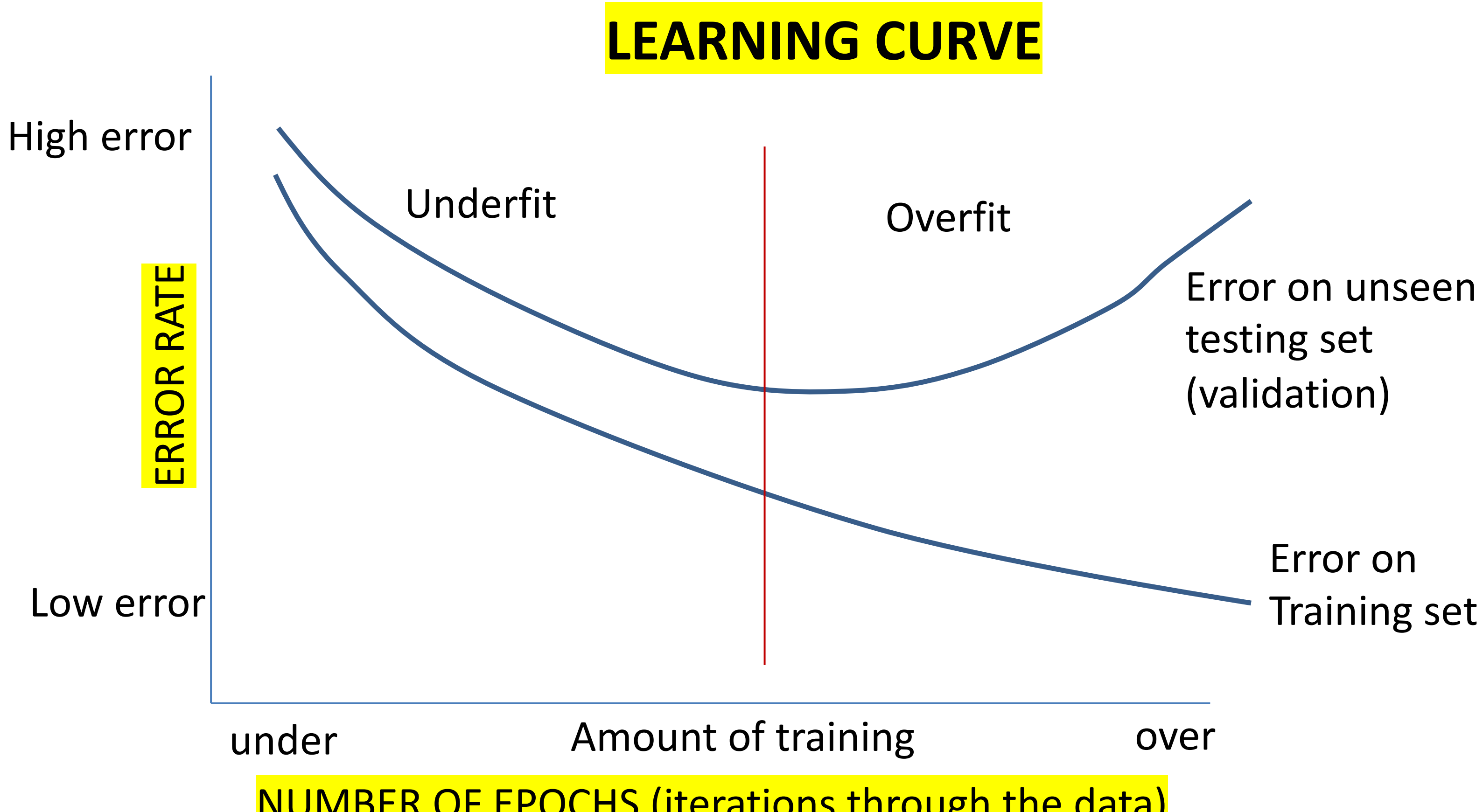
  - Epoch 15: Train: 85% Validation: 80%
  - Epoch 16: Train: 87% Validation: 82%
  - Epoch 17: Train: 90% Validation: 85%
  - Epoch 18: Train: 95% Validation: 83%
  - Epoch 19: Train: 97% Validation: 78%
  - Epoch 20: Train: 98% Validation: 75%

STOP!!!
when accuracy stops improving

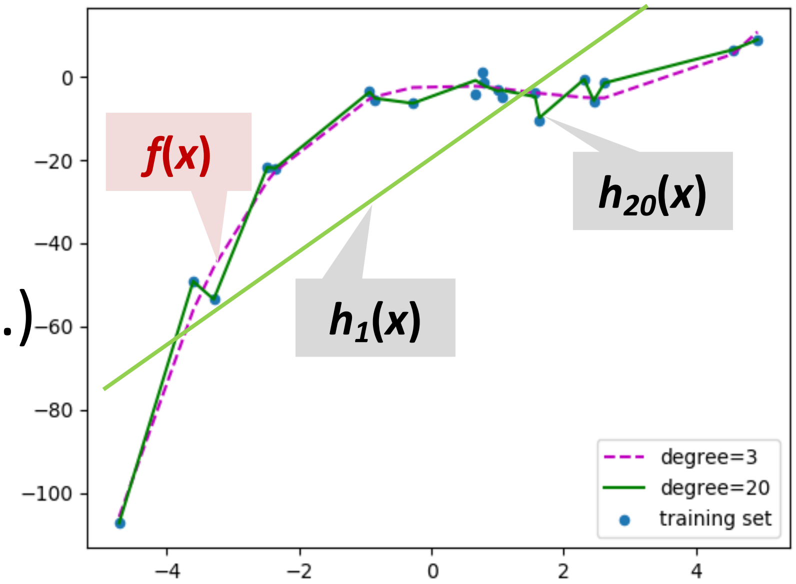
accuracy

Phenomenon in training

More training is equivalent to better fitting with a higher degree of polynomial



Two Types of Errors - (Bias and Variance)



Given data set of (x, y), where

$y = f(x) + \varepsilon$; ε is the noise

f is the unknown true function

Let $h(x)$ = trained function of fixed complexity (i.e, linear, quad, cubic..)

MSE error = $(f(x) + \varepsilon - h(x))^2$

Let $\bar{h}$ = E{h} of different training sets

Bias² = $(f - \bar{h})^2$

Variance = $E[(h - \bar{h})^2]$ = error which $h(x)$ differs from $\bar{h}$

Expected prediction error (MSE) = Bias² + Variance

$$MSE = E\{ (f(x) + \varepsilon - h(x))^2 \} = E((f - \bar{h})^2) + E[(h - \bar{h})^2]$$

Variance - between expected model $\bar{h}$ and predicted model h

Bias² - between expected model $\bar{h}$ and f

When h is of low complexity; **Bias²** is high due to model's simplistic assumptions

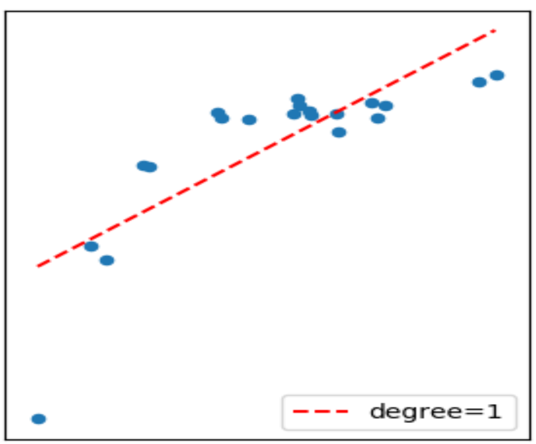
When h is of high complexity; **Variance** is high

- $\bar{h}$ approximates f
- $h(x)$ fits the noise and deviates from f

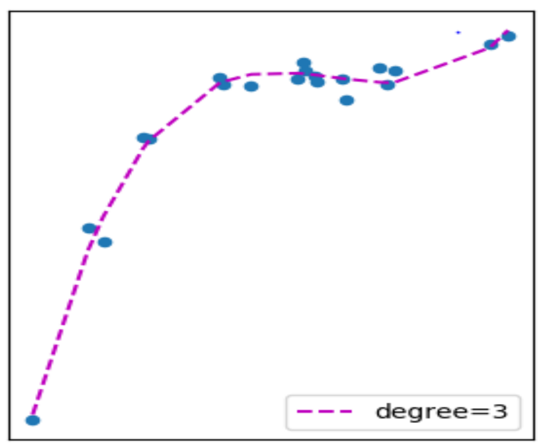
Bias vs Variance Trade-off

Expected prediction error (MSE) = $\text{Bias}^2 + \text{Variance}$

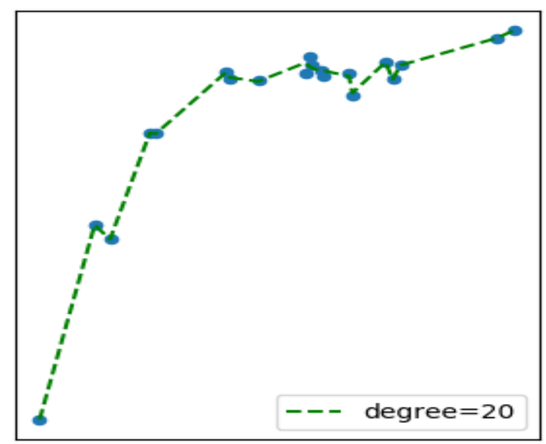
- A **high bias** means that the model is unable to capture the patterns in the data, and results in **under-fitting**.
- **High variance** means the model passes through most of the data points, and results in **over-fitting** the data.
- In order to achieve a good model that performs well both on the training and unseen data, a **trade-off** is made.



Underfit
High Bias, Low Variance

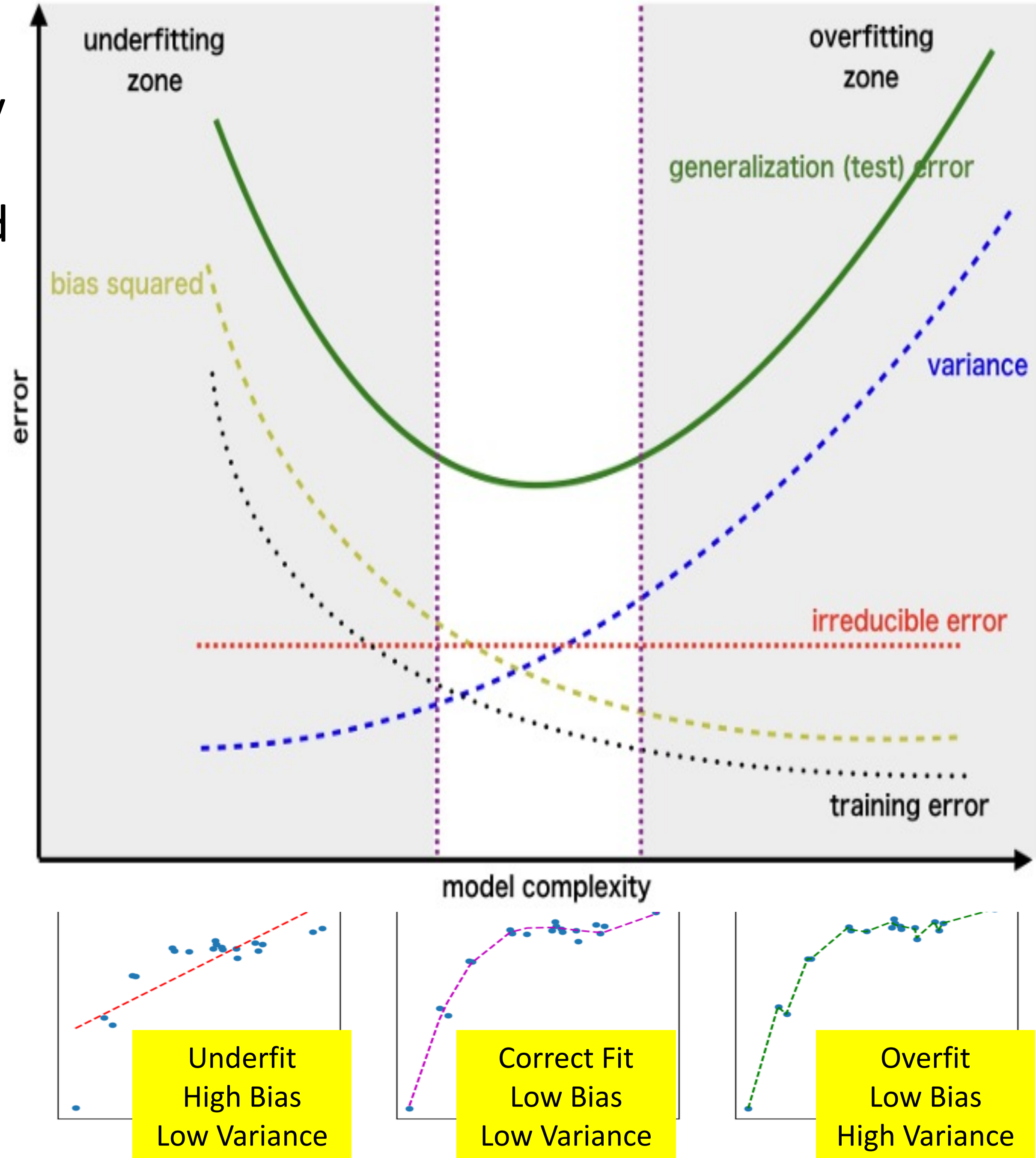


Correct Fit
Low Bias, Low Variance

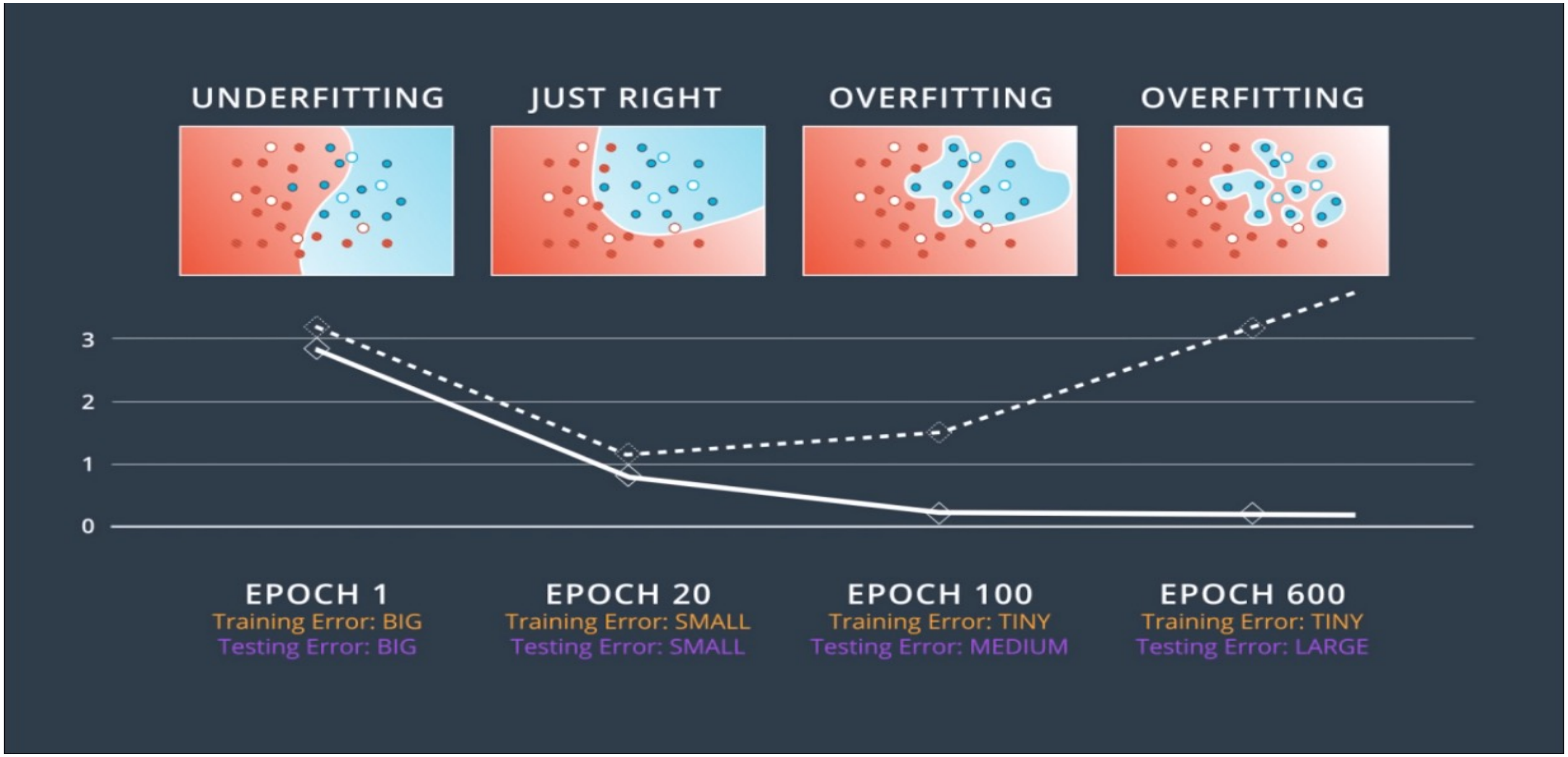


Overfit
Low Bias, High Variance

- As model complexity increases, “bias” decreases and “variance” increases and vice-versa.
- The combined test error is U-shape.
- A machine learning model should have **low variance and low bias**



- Start with a complicated network architecture
- Evaluate these models by plotting the error in the training and testing set with respect to each epoch



- Testing set error decreases as the model generalizes well until it gets to a ***minimum point — the Goldilocks spot*** (stops under-fitting and starts over-fitting)