Mahidol University
Faculty of Medicine Ramathibodi Hospital
Section for Clinical Epidemiology and Biostatistics

Neural Network

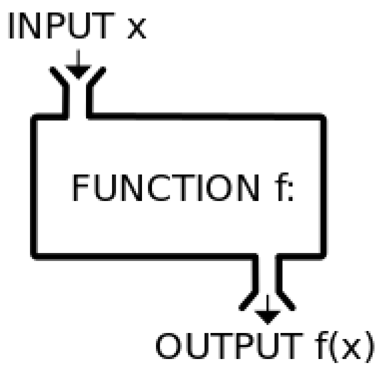
Ratchainant Thammasudjarit, Ph.D.

Prerequisite Mathematics

Recall your linear algebra

Linear Algebra

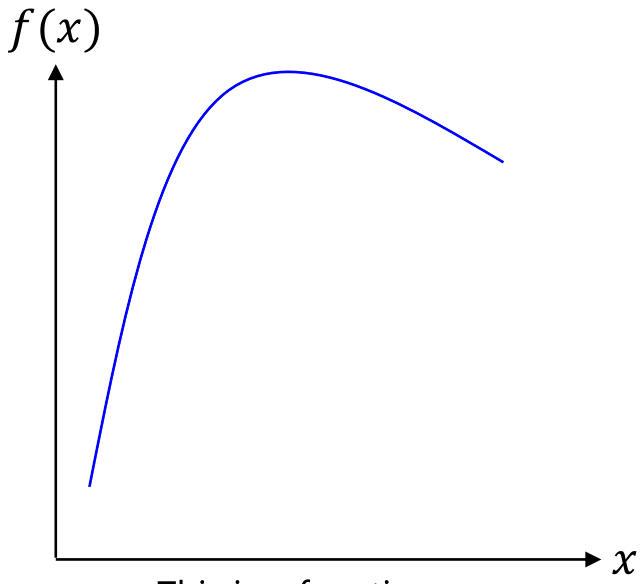
Function



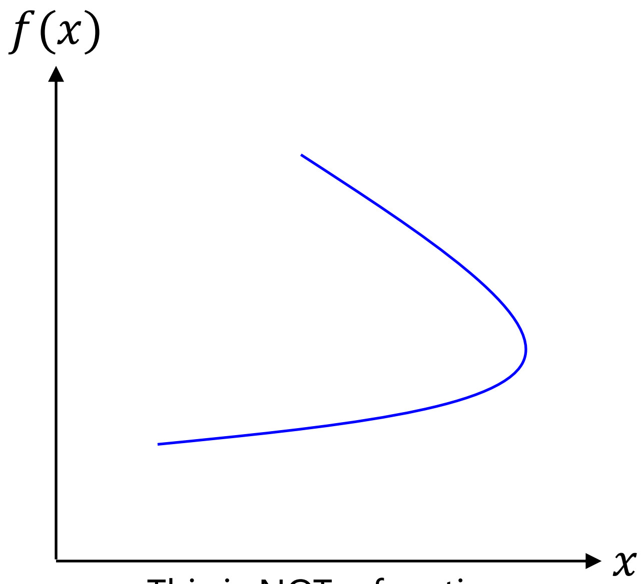
Each input must have only one output

- A function of univariate is defined as

$$x \mapsto f(x)$$



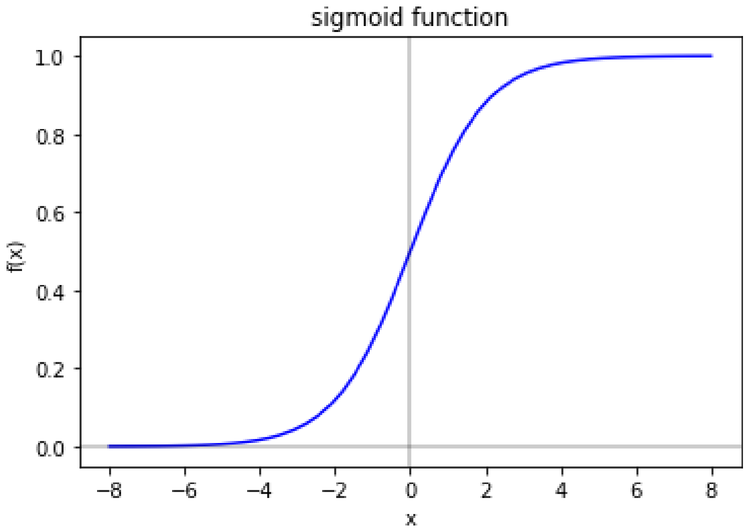
This is a function



This is NOT a function

Linear Algebra

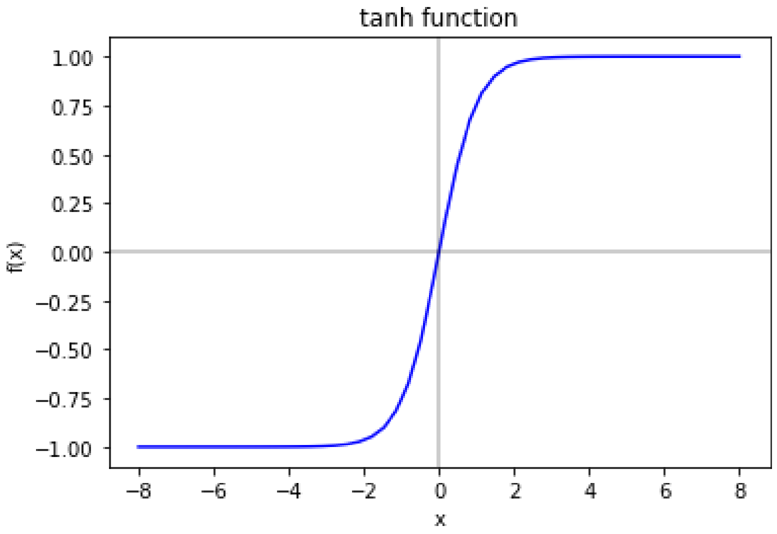
sigmoid



- Properties
  - $f(x) \in [0, 1]$
  - $x \in [-\infty, \infty]$
  - $x \to 0$, $f(x)$ becomes linear
  - $abs(x) > 4$, $f(x)$ changes slowly

Linear Algebra

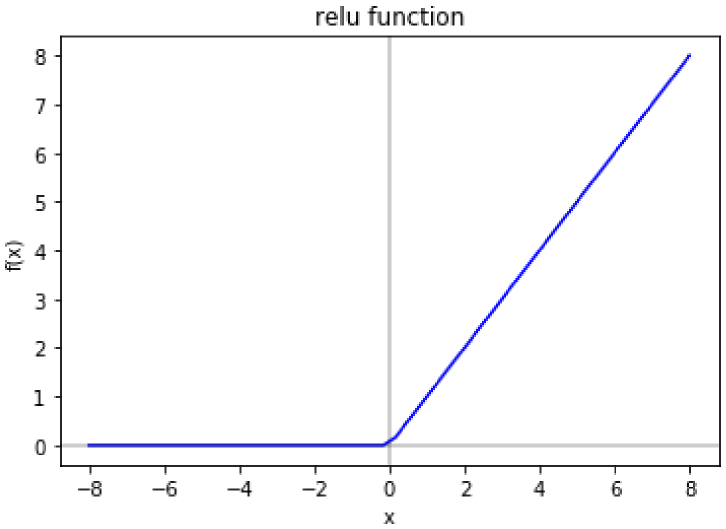
tanh



- Properties
  - f(x) ∈ [-1, 1]
  - x ∈ [-∞, ∞]
  - x → 0, f(x) becomes linear
  - abs(x) > 2, f(x) changes slowly

Linear Algebra

ReLu (Rectifier Linear Unit)

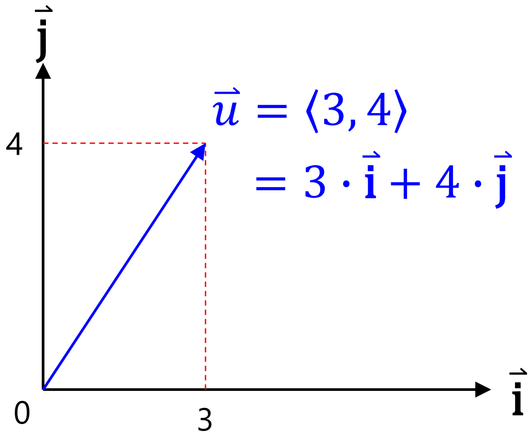



```python
def relu(x):
    y = np.maximum(0, x)
    return y
```

- Properties
  - f(x) ∈ [0, ∞]
  - x ∈ [-∞, ∞]
  - x ≤ 0, f(x) = 0
  - x > 0, f(x) = x

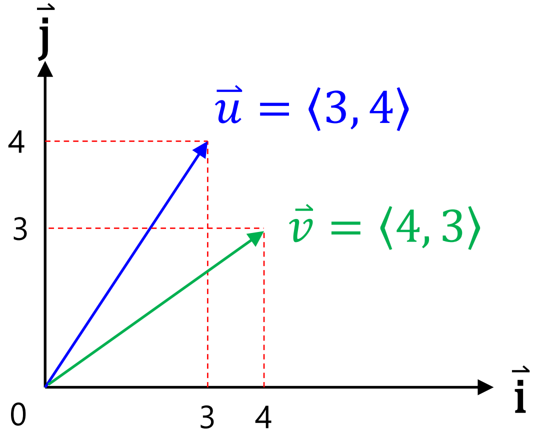
Linear Algebra

Vector



$$\|\vec{u}\| = \sqrt{3^2 + 4^2} = 5$$

- Dot product



$$\vec{u} \cdot \vec{v} = (3 \times 4) + (4 \times 3) = 24$$

- Transpose

if

$$\mathbf{u} = \begin{bmatrix} 3 & 4 \end{bmatrix}$$

then

$$\mathbf{u}^T = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

Linear Algebra

Matrix

A stack of vectors

- Pairwise multiplication

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

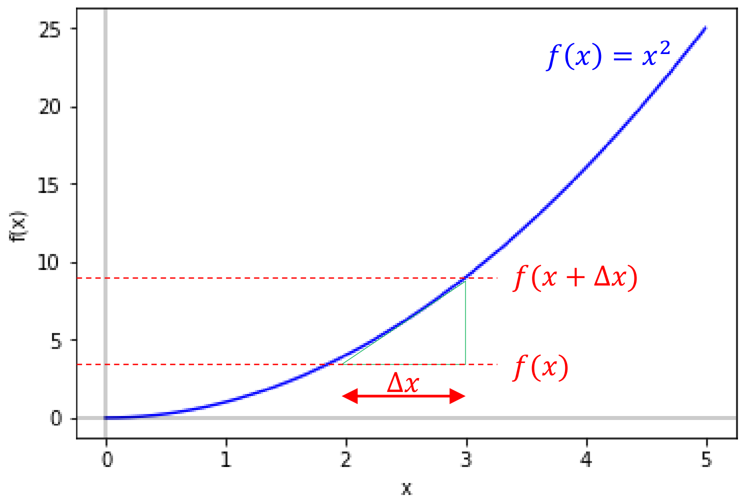
$$A \odot B = \begin{bmatrix} 1 \times 2 & 2 \times 2 & 3 \times 2 \\ 4 \times 2 & 5 \times 2 & 6 \times 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix}$$

- Transpose

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Linear Algebra

Calculus (Derivative)



- Derivative

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

$\Delta x = 1$

if $x = 2$ then $f(x) = 4$

if $x = 3$ then $f(x) = 9$

$\Delta x = 0.001$

if $x = 2$ then $f(x) = 4$

if $x = 2.001$ then $f(x) \approx 4.004$

if $x = 2$ then slope $= \dfrac{\Delta f(x)}{\Delta x} = \dfrac{0.004}{0.001} = 4$

Mathematically, $\dfrac{d}{dx} f(x) = 2x = 2(2) = 4$

Linear Algebra

Derivative of sigmoid

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

$$\begin{aligned}
\frac{d}{dz}g(z) &= \frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right] \\
&= \frac{d}{dz}(1+e^{-z})^{-1} \\
&= -(1+e^{-z})^{-2}\frac{d}{dz}(1+e^{-z}) \\
&= -(1+e^{-z})^{-2}\frac{d}{dz}(e^{-z})
\end{aligned}$$

$$\begin{aligned}
&= -(1+e^{-z})^{-2}(-e^{-z}) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{(1+e^{-z})-1}{1+e^{-z}} \\
&= \frac{1}{1+e^{-z}}\left[\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right] \\
&= \frac{1}{1+e^{-z}}\left[1 - \frac{1}{1+e^{-z}}\right]
\end{aligned}$$

$$\therefore \frac{d}{dz}g(z) = g(z)(1-g(z))$$

Linear Algebra

Derivative of tanh

$$g(z) = \tanh(z)$$

$$= \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\frac{d}{dz}g(z) = \frac{(e^z + e^{-z})d(e^z - e^{-z}) - (e^z - e^{-z})d(e^z + e^{-z})}{(e^z + e^{-z})^2}$$

$$= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2}$$

$$= 1 - \left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)^2$$

$$\therefore \frac{d}{dz}g(z) = 1 - \tanh^2(z)$$

Linear Algebra

Derivative of relu

$$g(z) = \max(0, z)$$

$$\frac{d}{dz} g(z) = \begin{cases} 0 & if\ z < 0 \\ 1 & if\ z > 0 \\ undefined & if\ z = 0 \end{cases}$$

In neural network practice, z can get close to zero but never be zero, e.g. 0.00000 ...

$$\therefore \frac{d}{dz} g(z) = \begin{cases} 0 & if\ z < 0 \\ 1 & if\ z > 0 \end{cases}$$

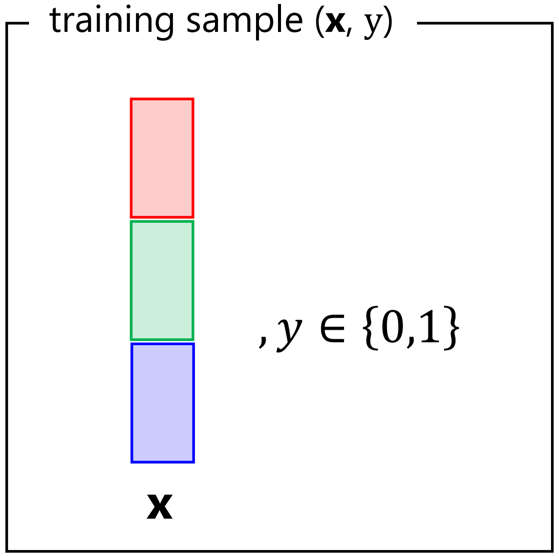
Perceptron

A smallest unit in neural network

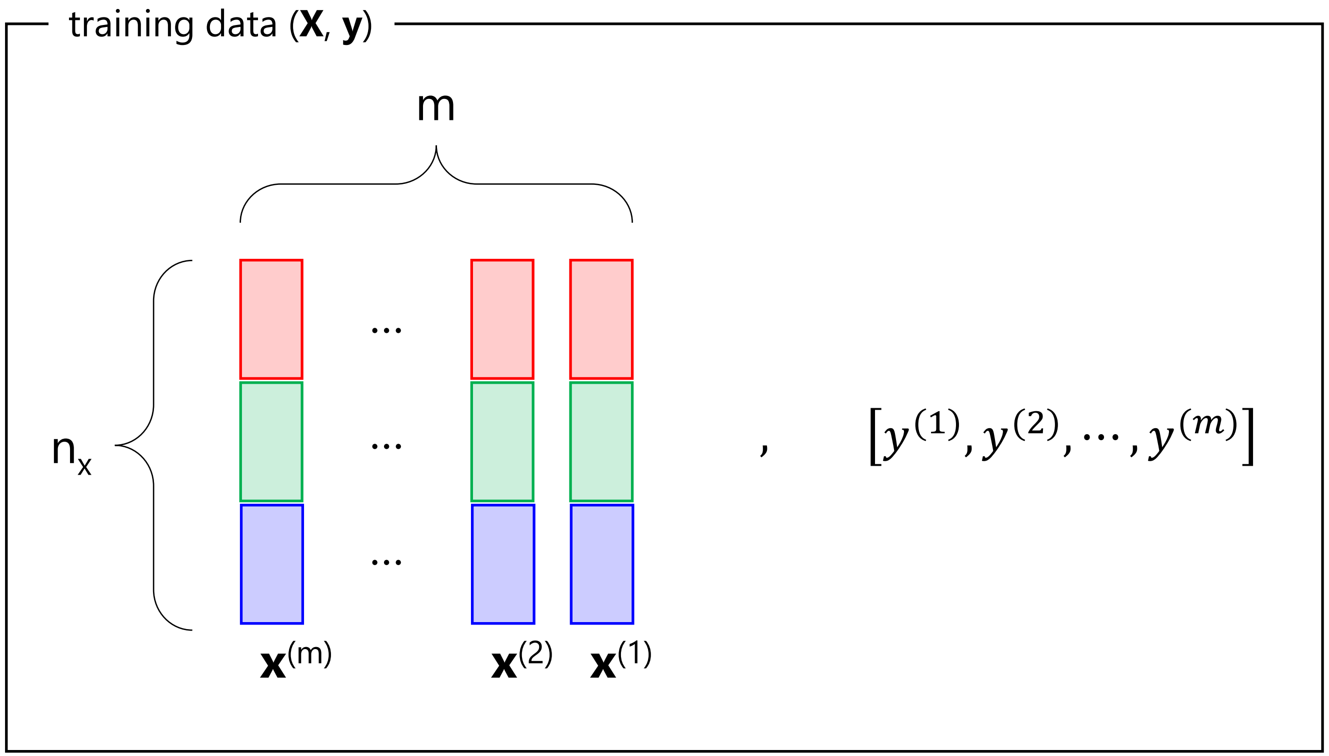
Training Sample VS Training Data

*Training sample is a pair of feature vector **x** and its class label*

$$(\mathbf{x}, y), \mathbf{x} \in \mathbb{R}^{n_x}, y \in \{0,1\}$$



*Training data is <u>m pair</u> of feature vector **x** and its class label*



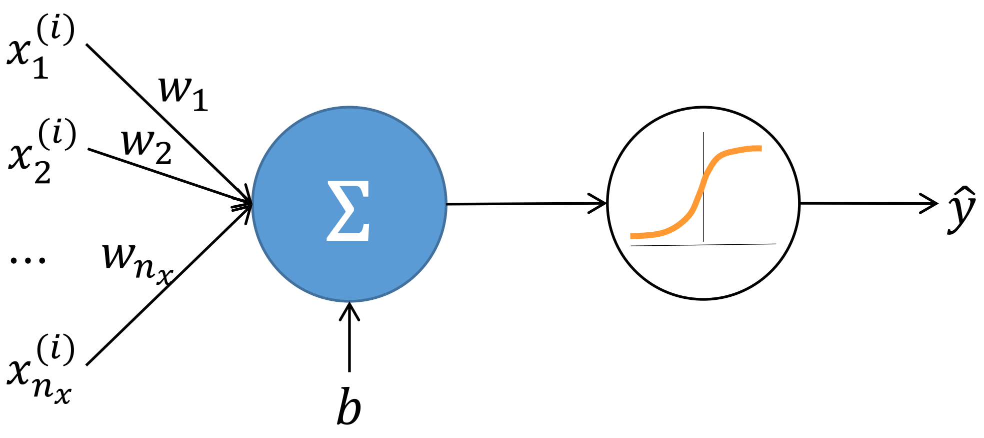
$\mathbf{x} \in \mathbb{R}^{n_x \times 1}$

$\mathbf{X} \in \mathbb{R}^{n_x \times m}$

$\mathbf{y} \in \mathbb{R}^{1 \times m}$

A Perceptron

The smallest unit of neural network

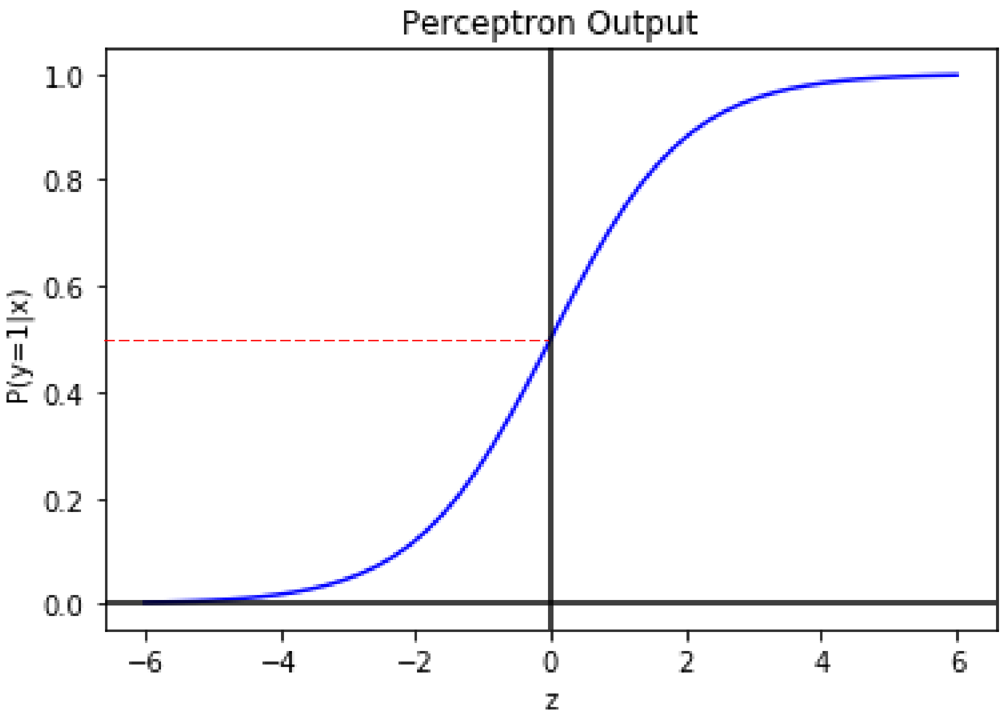


$$\hat{y} = P(y = 1|\mathbf{x}^{(i)}) = \frac{1}{1 + e^{-(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b)}}$$

- Given $\mathbf{x}^{(i)} \in \mathbb{R}^{n_x \times 1}$, determine
  - $\hat{y} = P(y = 1|\mathbf{x}^{(i)})$
- Parameters
  - $\mathbf{w} \in \mathbb{R}^{1 \times n_x}$
  - $b \in \mathbb{R}$
- Output
  - $\hat{y} = g(z) = g(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b)$

A Perceptron

The smallest unit of neural network



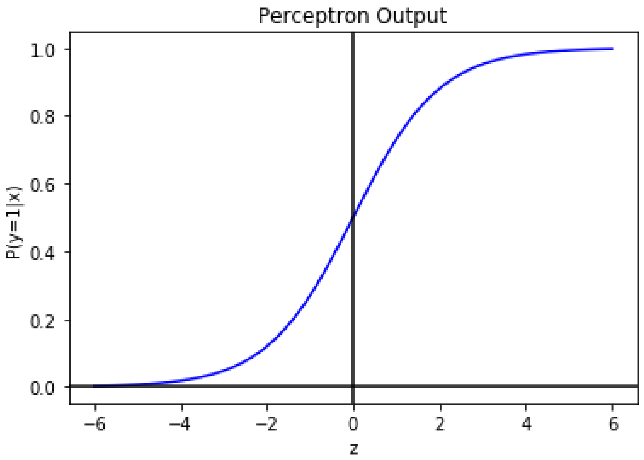
$$g(z) = \frac{1}{1 + e^{-z}}$$

- If $z \gg 0, g(z) \rightarrow 1$
- if $z \ll 0, g(z) \rightarrow 0$

Loss Function

Error of a training sample

The loss function $\mathcal{L}(\hat{y}, y)$ determines how close of $\hat{y}$ to the ground-truth y



$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} - (1 - y) \log(1 - \hat{y}))$$

Goal: $\mathcal{L}(\hat{y}, y) \rightarrow 0$

if $y = 1$,

$$\therefore z = w^T \cdot x + b, \quad \rightarrow \infty$$

$$\hat{y} = \frac{1}{1 + e^{-z}}, \quad \rightarrow 1$$

$$\mathcal{L}(\hat{y}, y) = -\log \hat{y}, \quad \rightarrow 0$$

if $y = 0$,

$$\therefore z = w^T \cdot x + b, \quad \rightarrow -\infty$$

$$\hat{y} = \frac{1}{1 + e^{-z}}, \quad \rightarrow 0$$

$$\mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y}), \quad \rightarrow 0$$

Cost Function

Error of training data

The cost function $\mathcal{J}(\mathbf{w}, b)$ indicates how well the model does in entire training samples

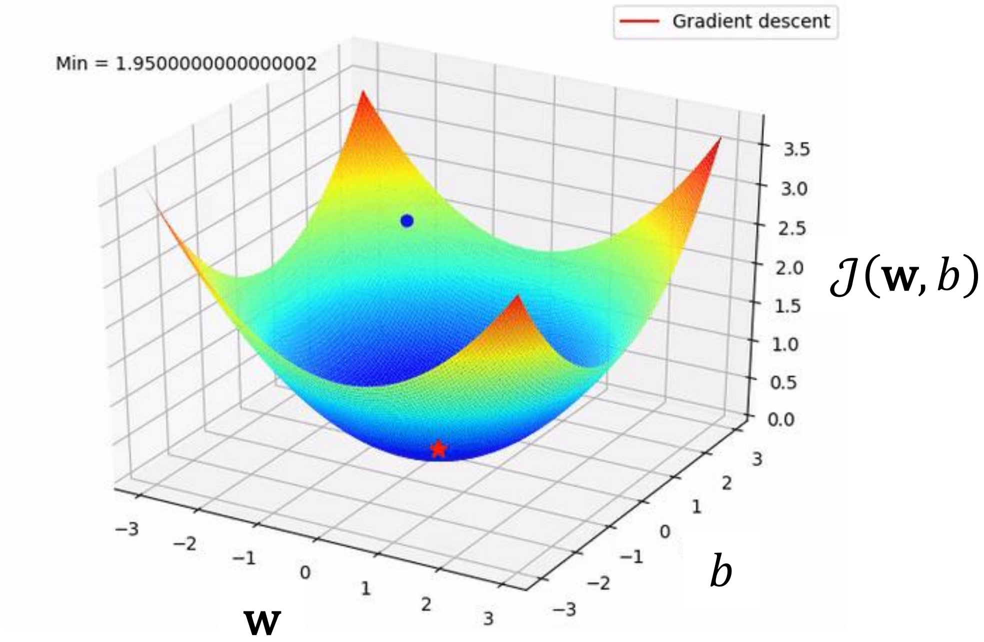
$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)\right]$$

- Training Goal
  - Find $\mathbf{w}$, b that minimize $\mathcal{J}(\mathbf{w}, b)$

Gradient Descent

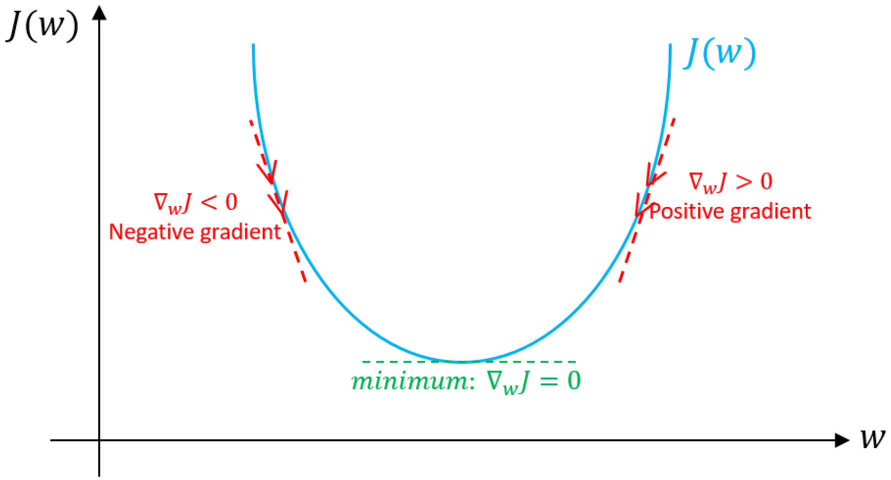
Searching for minimum point in hyperplane



- Goal
  - Determine $\mathbf{w}$, b that minimize $\mathcal{J}(\mathbf{w}, b)$

Gradient Descent

Initialize weight and adjust until the cost function approach to minimum



- Procedure

Repeat {

$$w := w - \alpha \frac{d}{dw} \mathcal{J}(w)$$

Until $\mathcal{J}(w) \rightarrow \min(\mathcal{J}(w))$

}

where α represents the learning rate (a small positive value)

On the right

$\frac{d}{dw} \mathcal{J}(w) > 0, \quad \rightarrow w$ is adjusted by decreasing $\frac{d}{dw} \mathcal{J}(w)$

On the left

$\frac{d}{dw} \mathcal{J}(w) < 0, \quad \rightarrow w$ is adjusted by increasing $\frac{d}{dw} \mathcal{J}(w)$

Gradient Descent

Summary

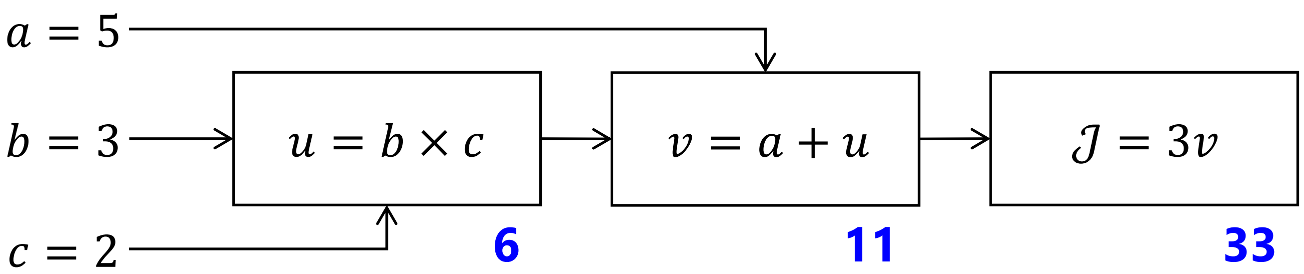
- Minimizing $\mathcal{J}(\mathbf{w}, b)$
  - $w := w - \alpha \frac{d}{dw}\mathcal{J}(w, b),$ where $w \in \mathbf{w}$
  - $b := b - \alpha \frac{d}{db}\mathcal{J}(w, b)$
- In each training iteration, we need to determine

$$\frac{d}{dw}\mathcal{J}(w, b) \quad \text{and} \quad \frac{d}{db}\mathcal{J}(w, b)$$

Computational Graph

Forward path

Given a function $\mathcal{J}(a, b, c) = 3(a + bc)$



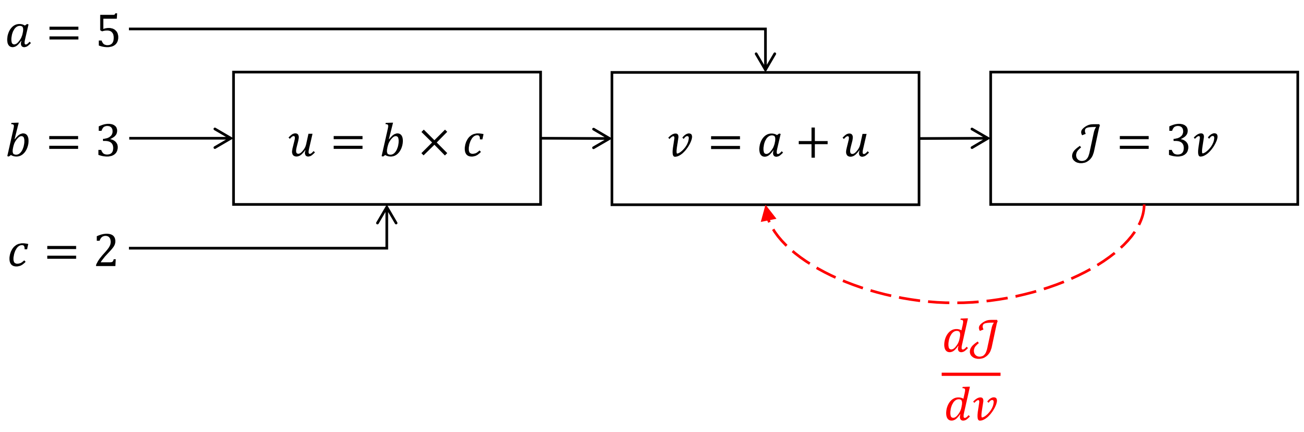
- Forward path
  - $\mathcal{J}(a, b, c)$ can be determined

- Backward path
  - Derivative of $\mathcal{J}(a, b, c)$ w.r.t. a or b or c can be determined

Computational Graph

Backward path

Given a function $\mathcal{J}(a, b, c) = 3(a + bc)$



- Finding $\frac{d\mathcal{J}}{dv}$
  - $v = 11 \xrightarrow{\text{Nudge its value}} v = 11.001$
  - $\mathcal{J} = 33 \xrightarrow{\text{Changed by } v} \mathcal{J} = 33.003$

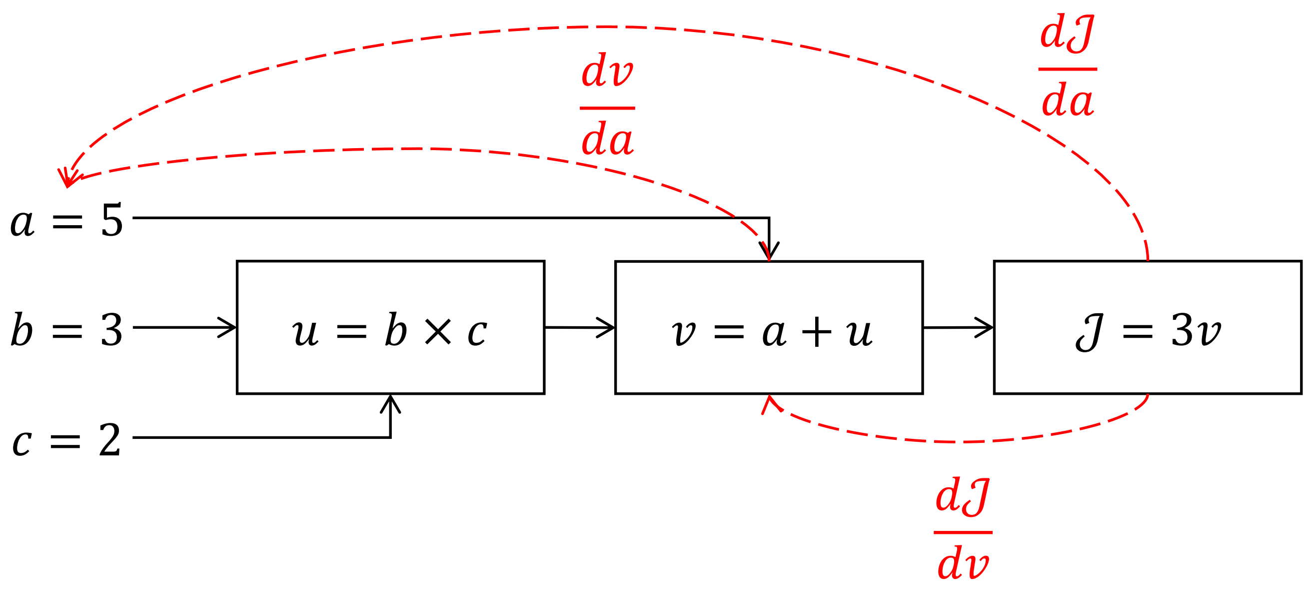
Therefore

$$\frac{d\mathcal{J}}{dv} = \frac{0.003}{0.001} = 3$$

Computational Graph

Backward path

Given a function $\mathcal{J}(a, b, c) = 3(a + bc)$



- Finding $\frac{d\mathcal{J}}{da}$
  - $a = 5 \xrightarrow{\text{Nudge its value}} a = 5.001$
  - $v = 11 \xrightarrow{\text{Changed by } a} v = 11.001$
  - $\mathcal{J} = 33 \xrightarrow{\text{Changed by } a, v} \mathcal{J} = 33.003$

Therefore

$$\frac{d\mathcal{J}}{da} = \underbrace{\frac{d\mathcal{J}}{dv} \cdot \frac{dv}{da}}_{\text{Chain rule}} = \frac{0.003}{0.001} \cdot \frac{0.001}{0.001} = 3$$

Perceptron with Gradient Descent

For a training sample

Given $\mathbf{w} = [w_1 \quad w_2]$ and b

$$
\begin{array}{l}
x_1^{(i)} \\
w_1 \\
x_2^{(i)} \\
w_2 \\
b
\end{array}
\rightarrow
\boxed{z = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b}
\rightarrow
\boxed{\hat{y} = a = g(z)}
\rightarrow
\boxed{\mathcal{L}(a, y)}
$$

- Recap

$$z = \mathbf{w}^T \cdot \mathbf{x}^{(i)} + b$$

$$\hat{y} = a = g(z)$$

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

$$\mathcal{L}(a, y) = -(y \log a + (1 - y)\log(1 - a))$$

- Goal
  - To adjust $\mathbf{w} = [w_1 \quad w_2]$ and b to minimize $\mathcal{L}(a, y)$

Perceptron with Gradient Descent

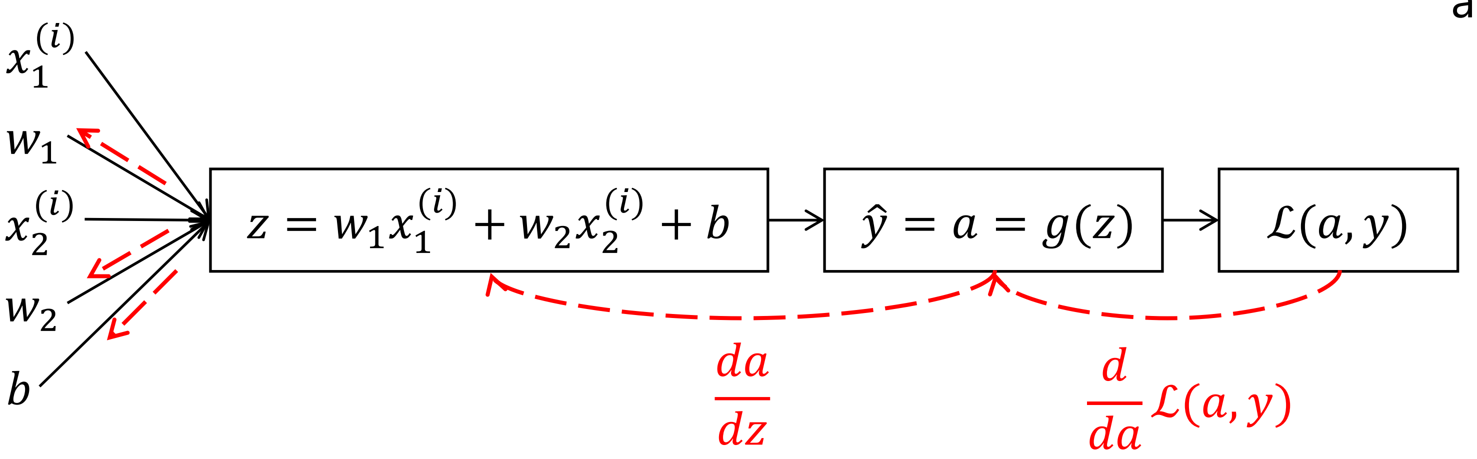
Backward Propagation

$$\Delta z = \frac{d}{dz}\mathcal{L}(a,y) = \frac{d}{da}\mathcal{L}(a,y)\cdot\frac{da}{dz}$$

since $\displaystyle \frac{d}{da}\mathcal{L}(a,y) = \frac{d}{da}\left[-y\log a-(1-y)\log(1-a)\right] = -\frac{y}{a}+\frac{1-y}{1-a}$

and $\displaystyle \frac{da}{dz} = a(1-a)$

$$\therefore \Delta z = \left(-\frac{y}{a}+\frac{1-y}{1-a}\right)\cdot a(1-a) = a-y$$



In other words, Δz is the difference between obtained output and desired output

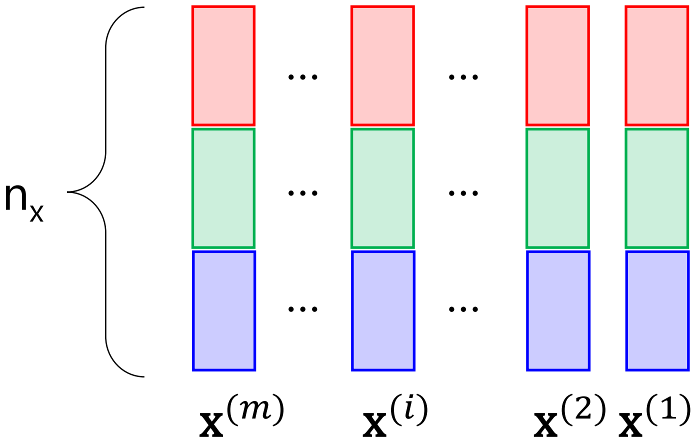
Implementation

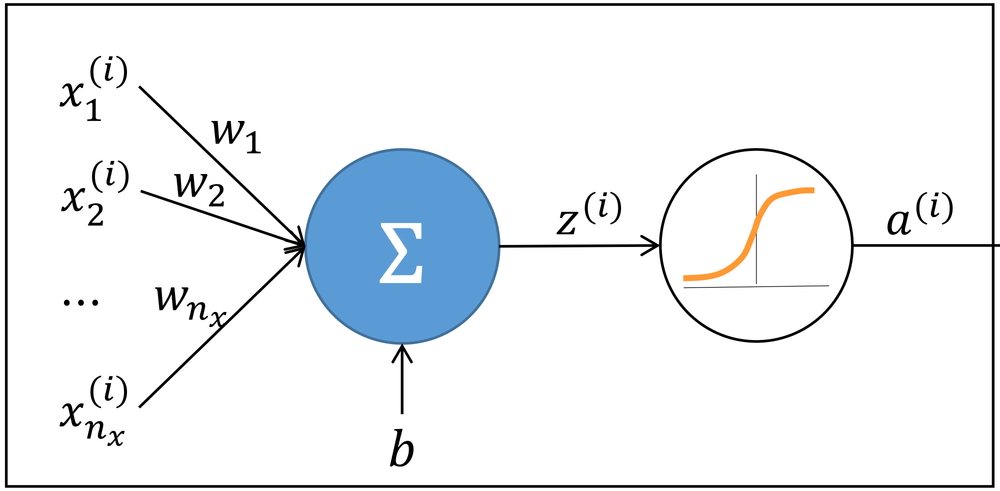
Vectorization

Applying a perceptron to m samples

Broadcasting property enable implementation without looping over training data

$$\mathbf{y} = g(\mathbf{z}) = g(\mathbf{w}^T \cdot \boldsymbol{X} + b)$$





$$[y^{(1)}, y^{(2)}, \cdots, y^{(m)}]$$

Naïve Implementation

Loop and Loop and Loop

This is NOT an efficient method

Note: Python programming does not have the Δ character. For simplicity, let's change

$$\Delta w_1 \rightarrow dw_1$$

$$\Delta w_2 \rightarrow dw_2$$

$$\Delta b \rightarrow db$$

Initialize

$$\mathcal{J} = 0, dw_1 = 0, dw_2 = 0, db = 0$$

for epoch = 1 to max_epoch:

for i = 1 to m:

compute $z^{(i)}, a^{(i)}$

$$\mathcal{J} \mathrel{+}= -\left[y^{(i)} \log a^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right]$$

compute $dz^{(i)}$

update db

for j = 1 to n_x:

update dw_j

$$\mathcal{J} /= \frac{\mathcal{J}}{m}$$

Implement with Vectorization

The efficient method

Initialize

$\mathcal{J} = 0, dw_1 = 0, dw_2 = 0, db = 0$

for epoch = 1 to max_epoch:

$$\mathbf{Z} = \mathbf{w}^T \cdot \mathbf{X} + b$$

$$\mathbf{A} = g(\mathbf{Z})$$

$$\mathcal{J} = -\frac{1}{m}\sum(\mathbf{y}\log\mathbf{A} - (1-\mathbf{y})\log(1-\mathbf{A}))$$

$$\mathbf{dz} = \mathbf{A} - \mathbf{Y}$$

$$\mathbf{dw} = \frac{1}{m}\mathbf{X} \cdot \mathbf{dz}^T$$

$$db = \frac{1}{m} \cdot np.sum(\mathbf{dz})$$

$$\mathbf{w} -= \alpha \cdot \mathbf{dw}$$

$$b -= \alpha \cdot db$$

Implement with Vectorization

The efficient method

Only loop over training iteration

for epoch = 1 to max_epoch:

$$\mathbf{Z}_{1\times m} = \underbrace{\left(\mathbf{w}_{1\times n_x}\right)^T \cdot \mathbf{X}_{n_x\times m}}_{1\times m} + b$$

$$\mathbf{A}_{1\times m} = g(\mathbf{Z}_{1\times m})$$

$$\mathcal{J} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}(\underbrace{\mathbf{y}_{1\times m}\log \mathbf{A}_{1\times m}}_{1\times m} - \underbrace{(1-\mathbf{y}_{1\times m})\log(1-\mathbf{A}_{1\times m})}_{1\times m})}_{1\times 1}$$

$$\mathbf{dz}_{1\times m} = \mathbf{A}_{1\times m} - \mathbf{y}_{1\times m}$$

$$\mathbf{dw}_{1\times n_x} = \frac{1}{m}\mathbf{X}_{n_x\times m} \cdot (\mathbf{dz}_{1\times m})^T$$

$$db = \frac{1}{m}\sum_{i=1}^{m}\mathbf{dz}_{1\times m}$$

$$\mathbf{w}_{1\times n_x} -= \alpha \cdot \mathbf{dw}_{1\times n_x}$$

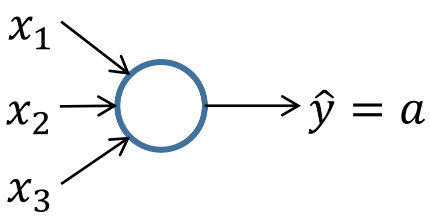
$$b -= \alpha \cdot db$$

Neural Network

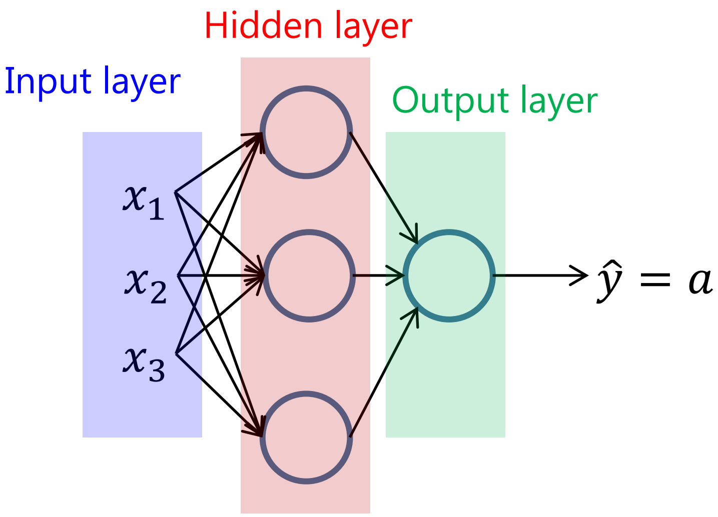
A collection of perceptrons

Shallow Neural Network

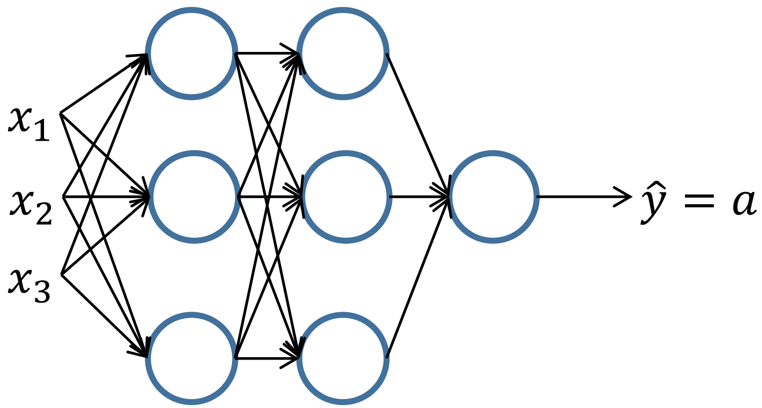
None or a few hidden layers



1-Layer Network
(0 Hidden Layer)
(perceptron or logistic regression)



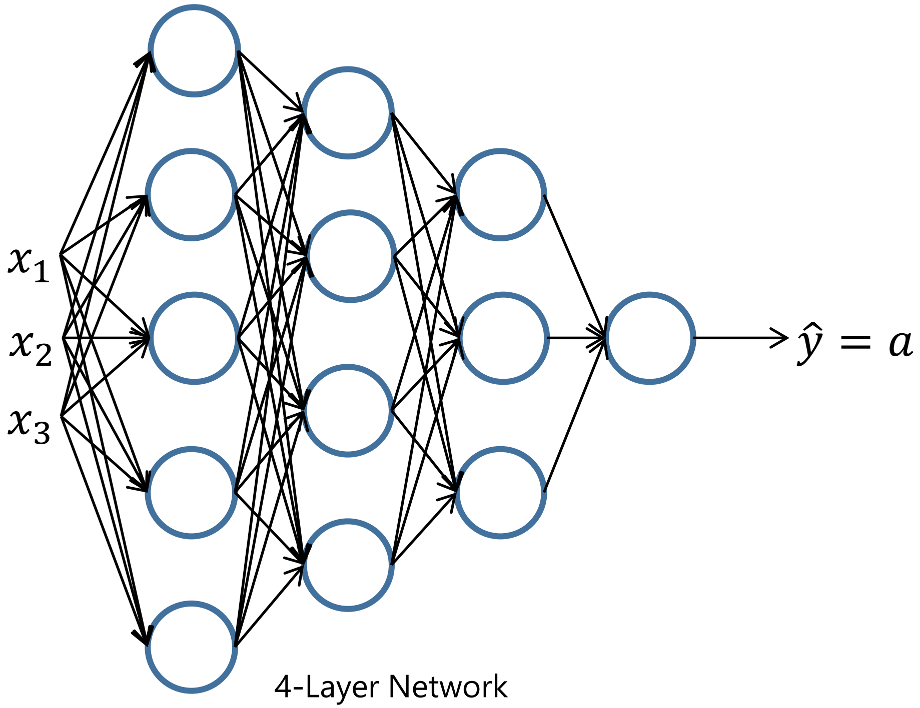
2-Layer Network
(1 Hidden Layer)



3-Layer Network
(2 Hidden Layer)

Deep Neural Network

More hidden layers



4-Layer Network
(3 Hidden Layer)

- Notations

L represents the number of layers (exclude the input layer)

$n^{[l]}$ represents the number of units in the l^{th} layer

Examples:

$n^{[1]} = 5, \quad n^{[2]} = 4, \quad n^{[3]} = 3, \quad n^{[4]} = 1$

$g^{[l]}(\cdot)$ represents the activation function in the l^{th} layer

$\boldsymbol{z}^{[l]}$ represents the linear combination in the l^{th} layer

$\boldsymbol{a}^{[l]}$ represents the activated values in the l^{th} layer

$\mathbf{W}^{[l]}$ represents the weights matrix in the l^{th} layer

$\mathbf{b}^{[l]}$ represents the bias in the l^{th} layer

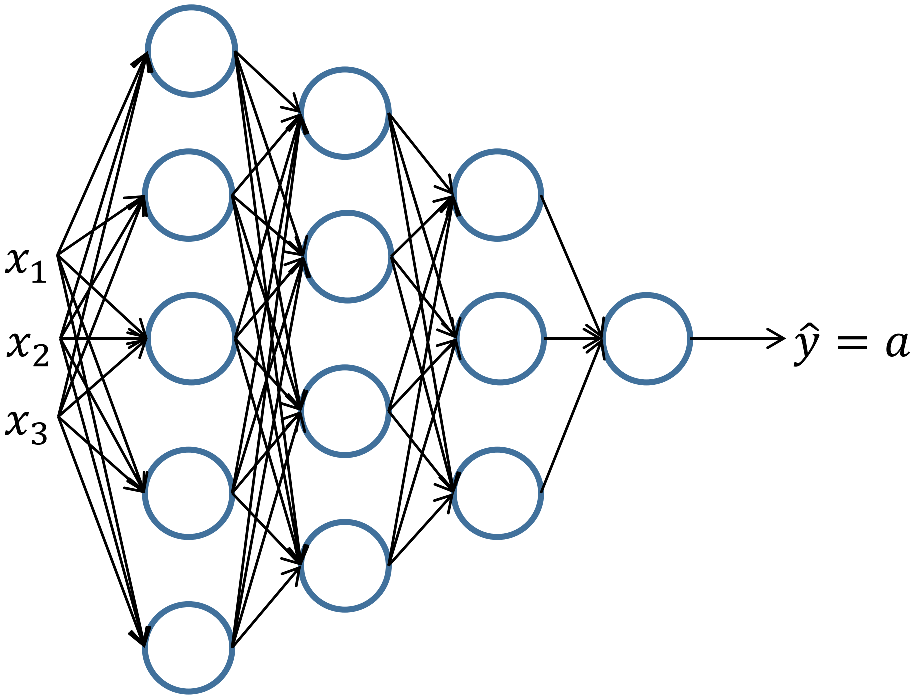
$\mathbb{W}$ represents the collection of weights matrix of all layers

$w_{i,j}$ represents the weights connecting between the i^{th} unit in the l^{th} layer to the j^{th} unit in the $(l-1)^{th}$ layer

$\mathbb{b}$ represents the collection of bias vectors of all layers

Deep Neural Network

Weights



- Weights $\mathbb{W} = \begin{bmatrix} \mathbf{W}^{[1]} \\ \mathbf{W}^{[2]} \\ \mathbf{W}^{[3]} \\ \mathbf{W}^{[4]} \end{bmatrix}$

$$\mathbf{W}^{[l]} = \begin{bmatrix} \mathbf{w}_1^{[l]} \\ \cdots \\ \mathbf{w}_{n^{[l]}}^{[l]} \end{bmatrix}_{n^{[l]},n^{[l-1]}}$$

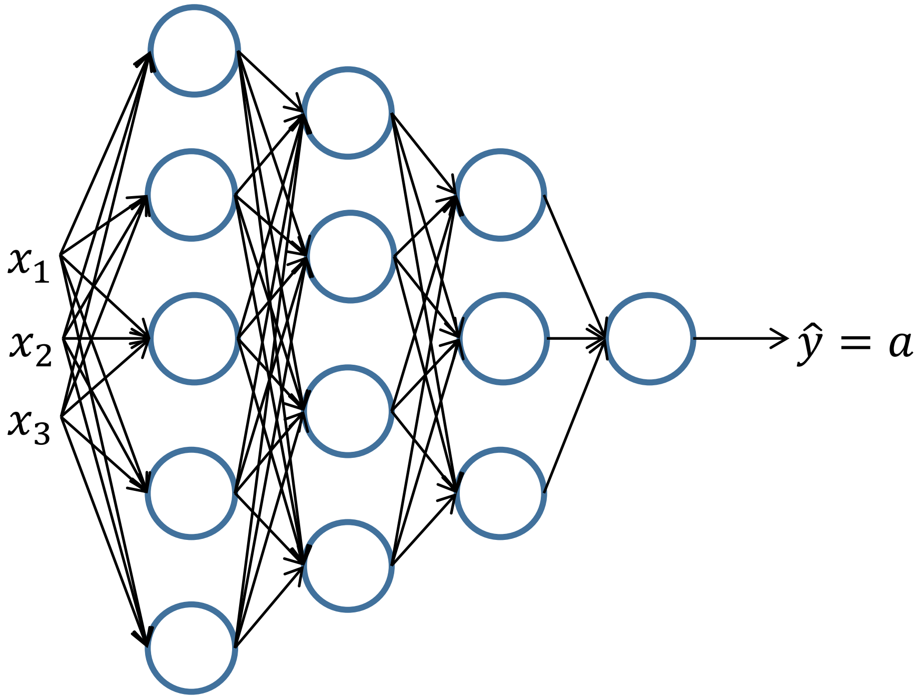
- Example

$$\mathbf{W}^{[3]} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,4} \\ \vdots & \ddots & \vdots \\ w_{3,1} & \cdots & w_{3,4} \end{bmatrix}$$

Dimension of $\mathbf{W}^{[l]}$ is $\left(n^{[l]}, n^{[l-1]}\right)$

Deep Neural Network

Biases



- Biases

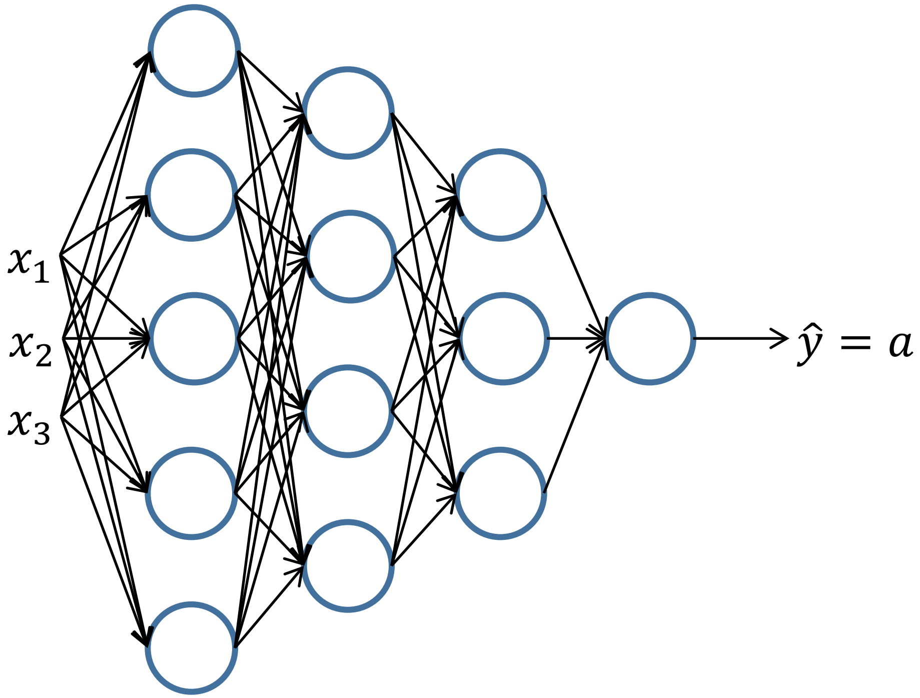
$$\mathbb{b} = [\mathbf{b}^{[1]} \quad \dots \quad \mathbf{b}^{[4]}] = \begin{bmatrix} b_1^{[1]} & \cdots & b_1^{[4]} \\ \vdots & \cdots & \vdots \\ b_5^{[1]} & \cdots & b_3^{[4]} \end{bmatrix}$$

$$\therefore \mathbf{b}^{[l]} = \begin{bmatrix} b_1^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}$$

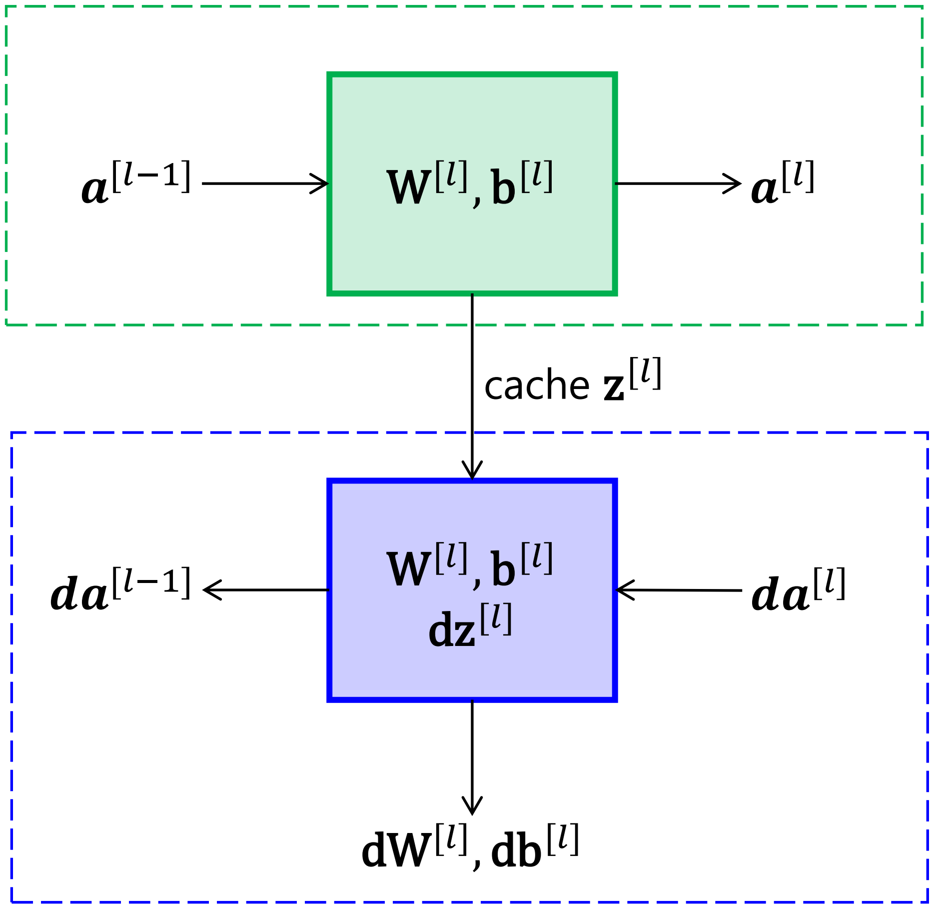
Dimension of $\mathbf{b}^{[l]}$ is $\left(n^{[l]}, 1\right)$

Implementation with Vectorization

The L layers neural network

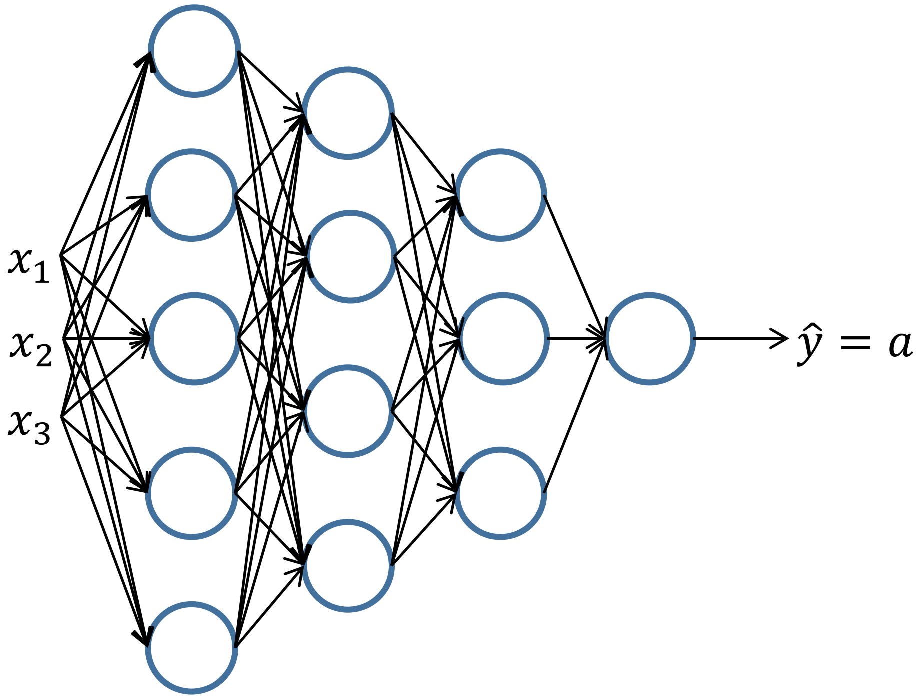


- At the layer l: $\mathbf{W}^{[l]}, \mathbf{b}^{[l]}$



Implementation with Vectorization

The L layers neural network



- At the layer l: $\mathbf{W}^{[l]}, \mathbf{b}^{[l]}$
  - Forward Propagation

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \boldsymbol{a}^{[l-1]} + \mathbf{b}^{[l]}$$

$$\boldsymbol{a}^{[l]} = g^{[l]}(\mathbf{z}^{[l]})$$

  - Backward Propagation

$$\mathbf{dz}^{[l]} = \mathbf{da}^{[l]} * g'^{[l]}(\mathbf{z}^{[l]})$$

$$\mathbf{dW}^{[l]} = \frac{1}{m}\mathbf{dz}^{[l]} \cdot \boldsymbol{a}^{[l-1]^T}$$

$$\mathbf{db}^{[l]} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{dz}^{[l]}$$

$$np.sum(\mathbf{dz}^{[l]}, axis = 1, keepdims = True)$$

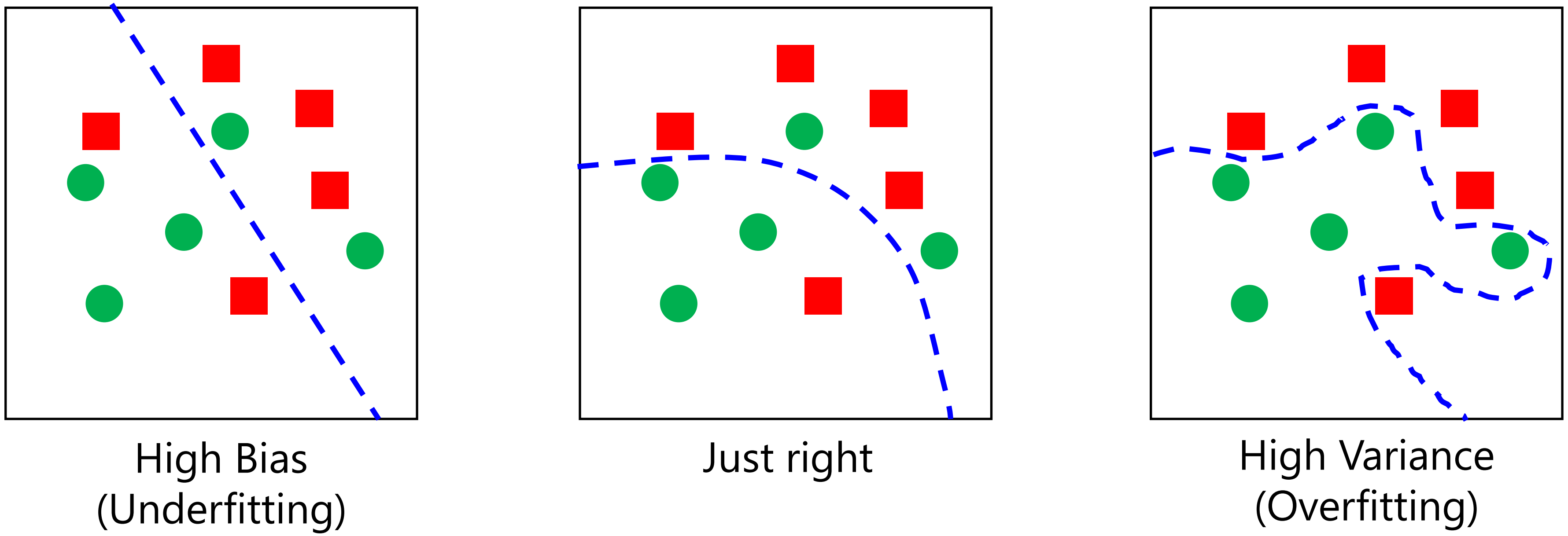
$$\mathbf{da}^{[l-1]} = \mathbf{W}^{[l]^T} \cdot \mathbf{dz}^{[l]}$$

Model Tuning

Bias and Variance Problem

Easy to learn but difficult to master

- Unable to visualize in high dimensional data



Regularization

L1 and L2

L1 and L2 Regularization

Recall the cost function of perceptron

Goal:

$\min_{\mathbf{w},b} \mathcal{J}(\mathbf{w}, b), \qquad \mathbf{w} \in \mathbb{R}^{n_x}, b \in \mathbb{R}$

- L1 regularization (Lasso)

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \boxed{\frac{\lambda}{2m}\|\mathbf{w}\|_1}$$

Regularization term

where $\|\mathbf{w}\|_1 = \sum_{j=1}^{n_x}|w_j|$

λ represents the regularization parameter

- L1 makes many weights become zeros
- Good for compacting the model
- L1 is not often used

L1 and L2 Regularization

Recall the cost function of perceptron

Goal:

$$\min_{\mathbf{w},b} \mathcal{J}(\mathbf{w}, b), \qquad \mathbf{w} \in \mathbb{R}^{n_x}, b \in \mathbb{R}$$

- L2 regularization (Ridge)

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \boxed{\frac{\lambda}{2m}\|\mathbf{w}\|_2^2}$$

Regularization term

where $\|\mathbf{w}\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = \mathbf{w}^T \cdot \mathbf{w}$

λ represents the regularization parameter

- L2 is much more often used compared to L1

L2 Regularization

L2 of Deep Neural Network

The cost function

$$\mathcal{J}\left(\mathbf{w}^{[1]}, \mathbf{b}^{[1]}, \dots, \mathbf{w}^{[L]}, \mathbf{b}^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|\mathbf{w}\|_F^2$$

where

Frobenious norm $\|\mathbf{w}\|_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}\left(w_{ij}^{[l]}\right)^2$

$\mathbf{w}^{[l]}: (n^{[l]}, n^{[l-1]})$

- Modifying back propagation

$$\mathbf{dw}^{[l]} = (\cdot) + \frac{\lambda}{m}\mathbf{w}^{[l]}$$

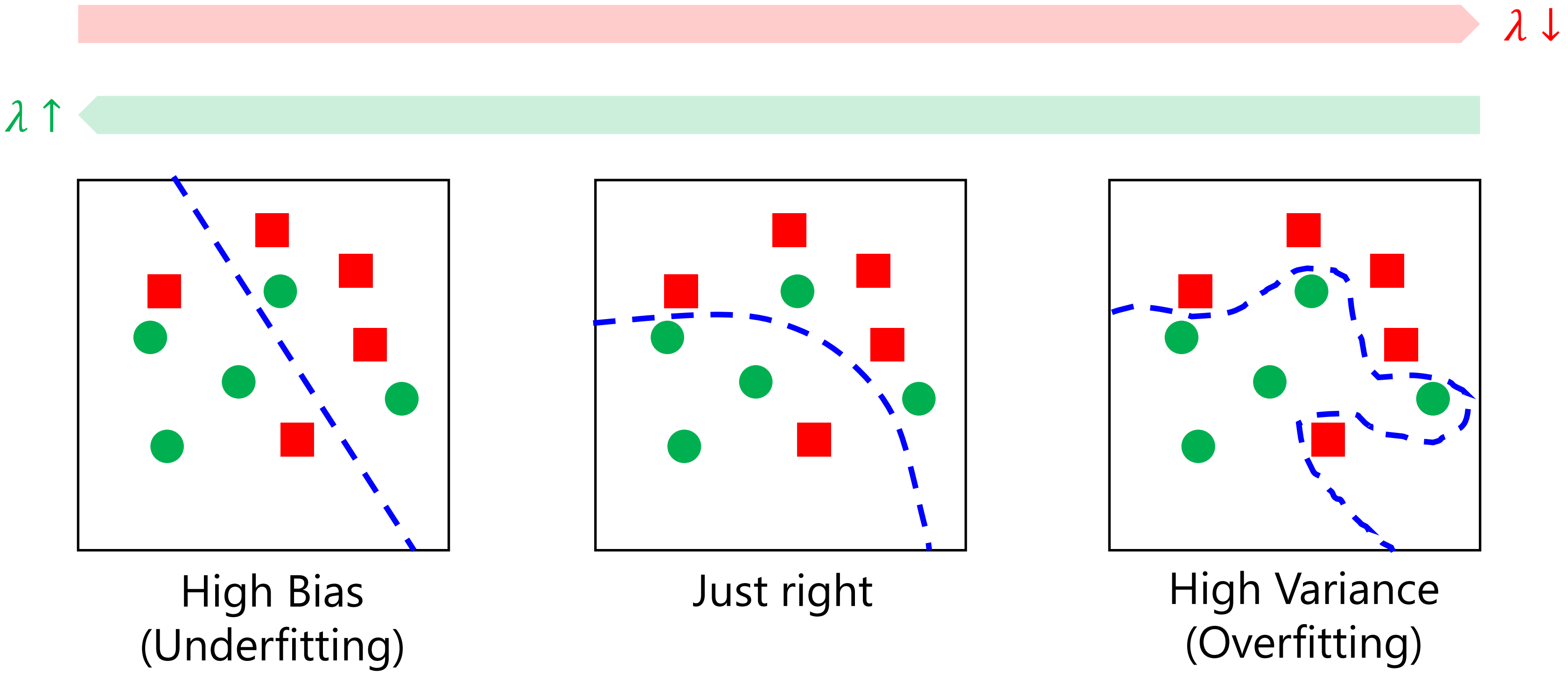
$$\begin{aligned}\mathbf{w}^{[l]} &= \mathbf{w}^{[l]} - \alpha\mathbf{dw}^{[l]} \\ &= \mathbf{w}^{[l]} - \alpha\left[(\cdot) + \frac{\lambda}{m}\mathbf{w}^{[l]}\right] \\ &= \mathbf{w}^{[l]} - \frac{\alpha\lambda}{m}\mathbf{w}^{[l]} - \alpha(\cdot) \\ &= \left(1 - \frac{\alpha\lambda}{m}\right)\mathbf{w}^{[l]} - \alpha(\cdot)\end{aligned}$$

Weight Decay

where $(\cdot)$ represents the term obtained from original back propagation

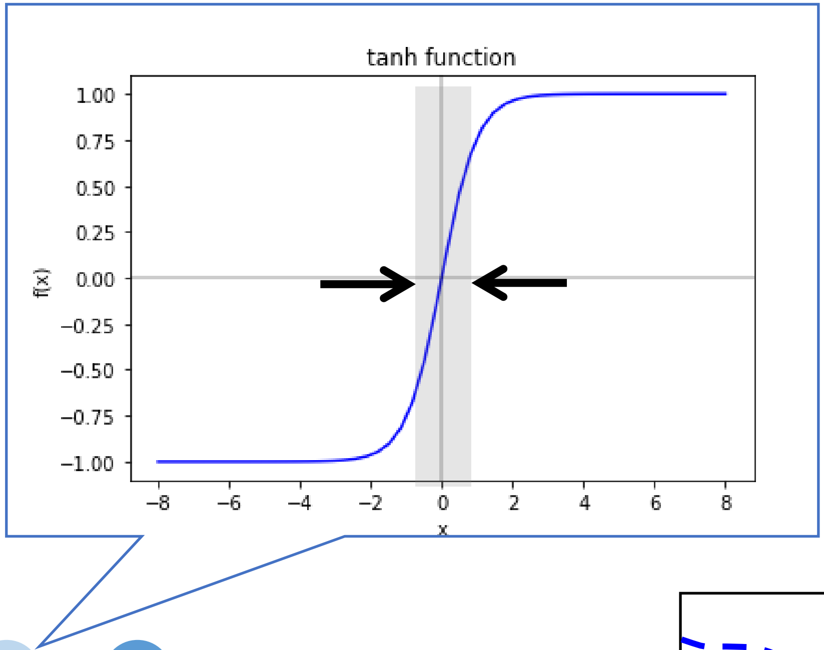
L2 Regularization

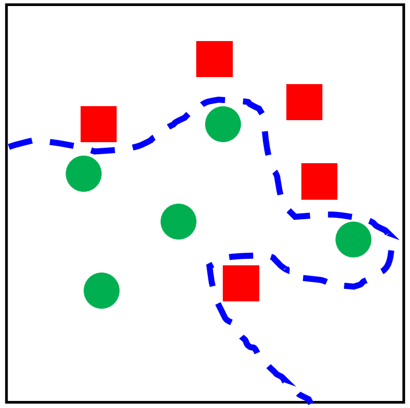
Why does L2 prevent overfitting?



L2 Regularization

Why does L2 prevent overfitting?



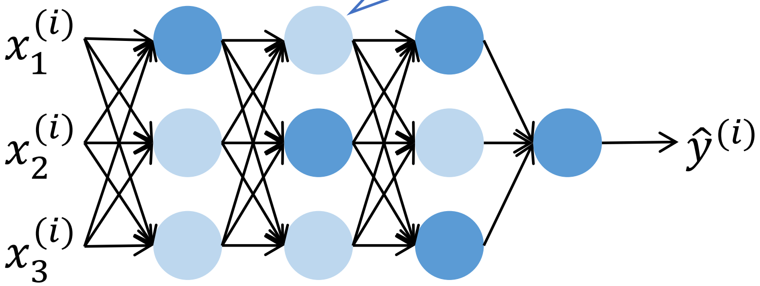


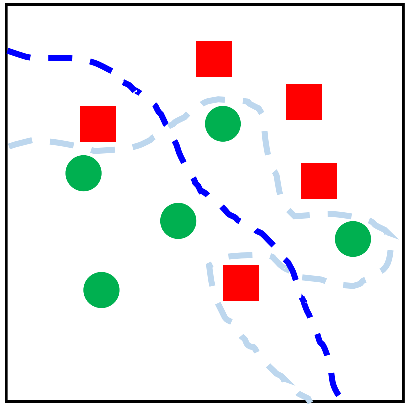
$\lambda \uparrow$

$\mathbf{w}^{[l]} \downarrow$

$\mathbf{z}^{[l]} \downarrow$

$\because \mathbf{z}^{[l]} = \mathbf{w}^{[l]} a^{[l-1]} + \mathbf{b}^{[l]}$





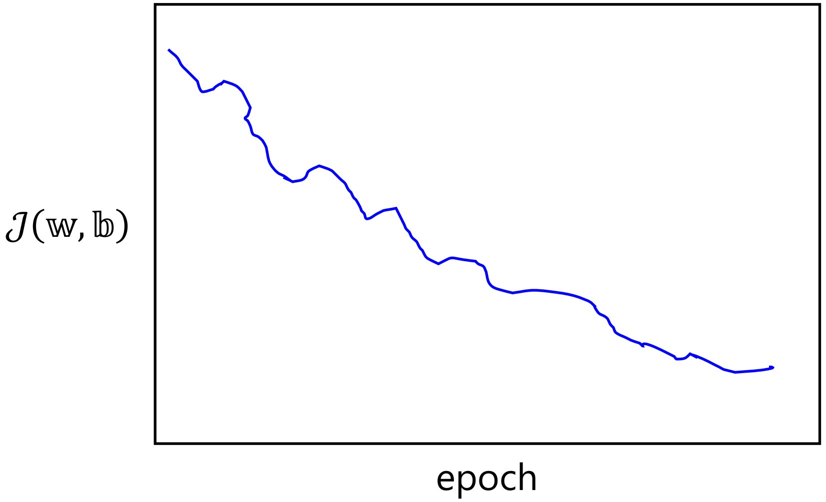
Decision boundary is stretched out

L2 Regularization

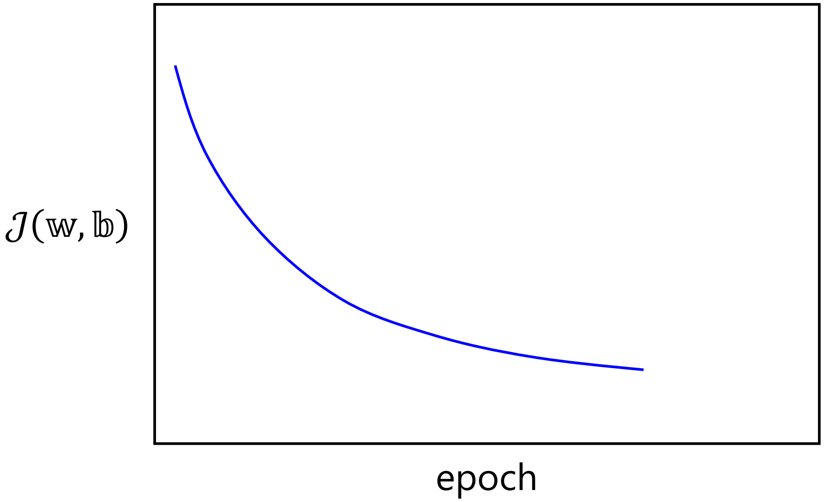
Smoother cost function

$$\mathcal{J}\left(\mathbf{w}^{[1]}, \mathbf{b}^{[1]}, \dots, \mathbf{w}^{[L]}, \mathbf{b}^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|\mathbf{w}\|_F^2$$

Without regularization



With regularization



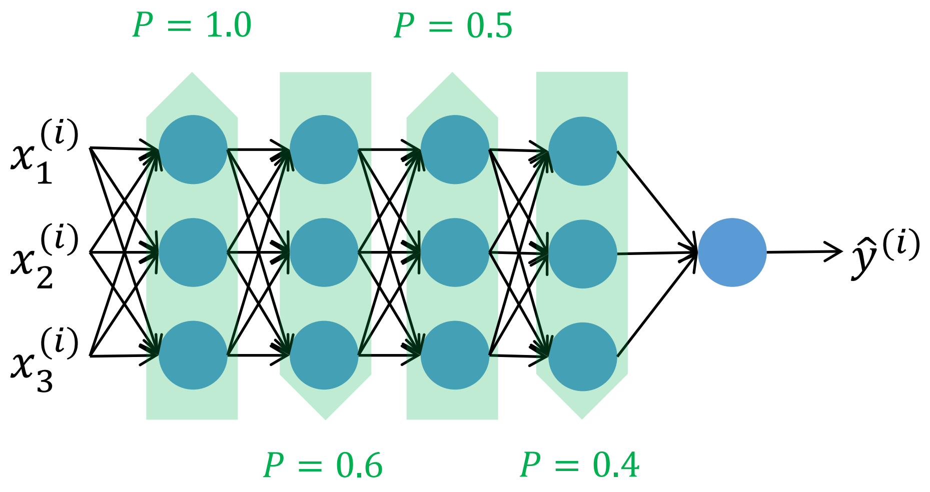
Regularization

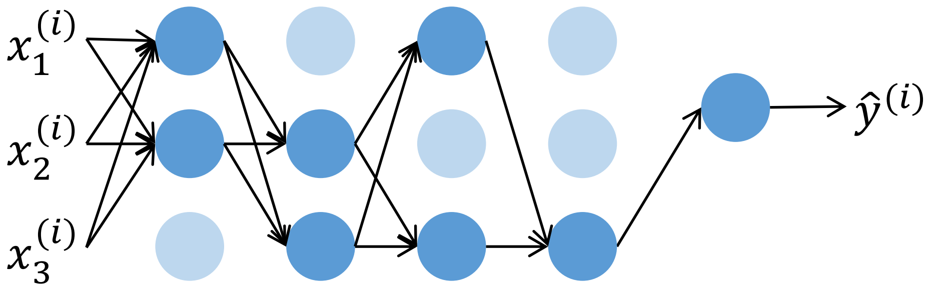
DropOut & DropConnect

DropOut

Ensemble Neural Network

- Set the probability of enable hidden units in each hidden layer
- The edges connected to any disable hidden unit will be removed
- DropOut is applied ONLY training but NOT testing
- DropOut produces many possible combination of neuron network
- DropOut is very popular in computer vision

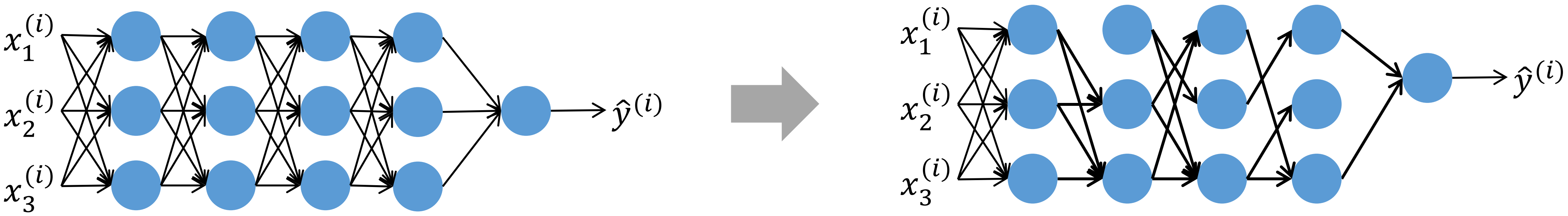




DropConnect

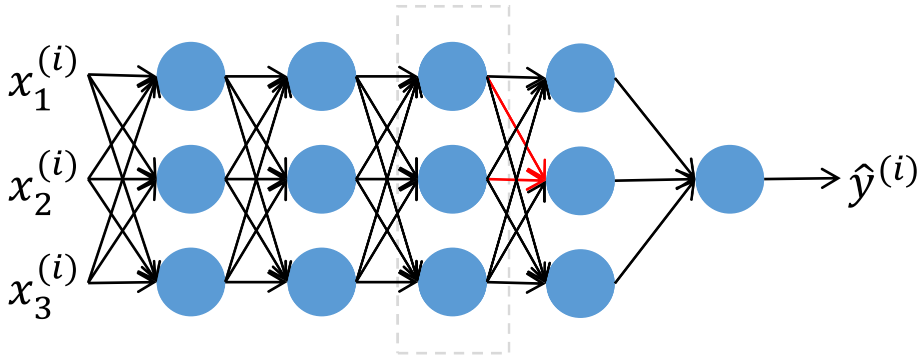
Generalization of DropOut

- Set the Boolean mask to randomly disable connections
- Rescale output on active connections
- DropConnect is applied ONLY for training but NOT testing
- DropConnect makes even more possible combination than dropout
- DropConnect is very popular in computer vision



Implementation

Illustration of DropConnect at the layer 3



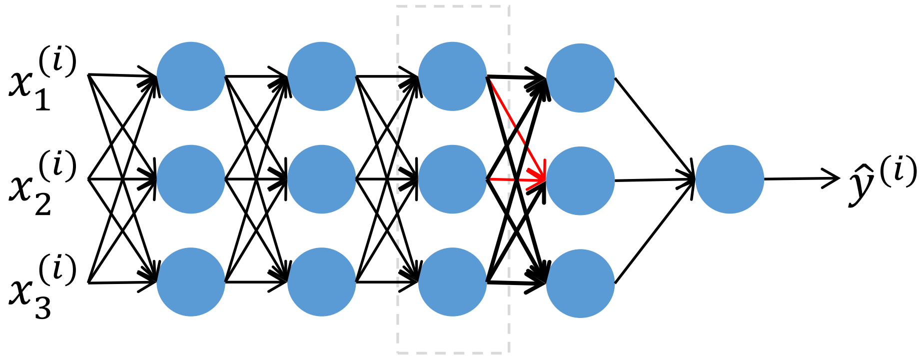
- Define theoretical variable to Python variable
  - P as keepprob
  - Create a Boolean mask to randomly disable weights

```
keepprob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keepprob
```

```
In [43]: d3
Out[43]:
array([[ True,  True,  True],
       [False,  True,  True],
       [ True, False,  True]])
```

Implementation

Illustration of DropConnect at the layer 3



- Rescale the $\boldsymbol{a}^{[3]}$

```
a3 /= keepprob
```

- Example: If $\boldsymbol{a}^{[3]}$ is

```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

- Rescale $\boldsymbol{a}^{[3]}$ after drop connect will be

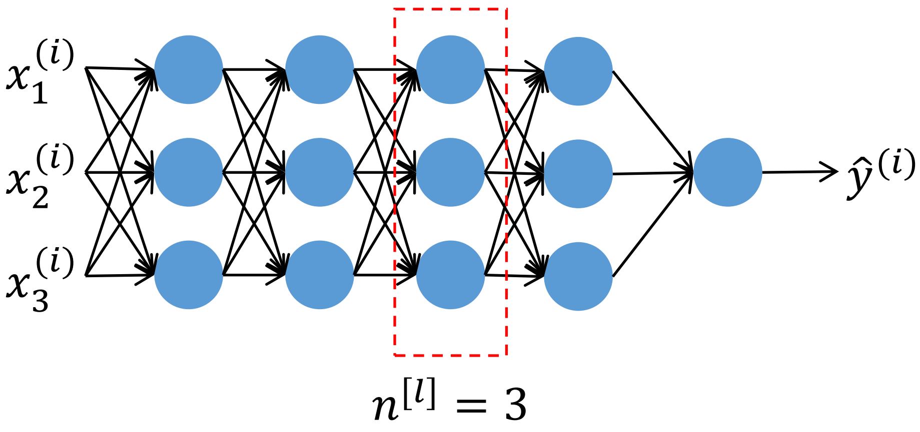
```
array([[ 1.25,  2.5 ,  3.75],
       [ 0.  ,  6.25,  7.5 ],
       [ 8.75,  0.  , 11.25]])
```

- In practice a3 is obtained from

$$\boldsymbol{a}^{[3]} = g^{[3]}\left(\mathbf{z}^{[3]}\right) = g^{[3]}(\mathbf{W}^{[3]} \cdot \boldsymbol{a}^{[2]} + \mathbf{b}^{[3]})$$

Recommended Settings

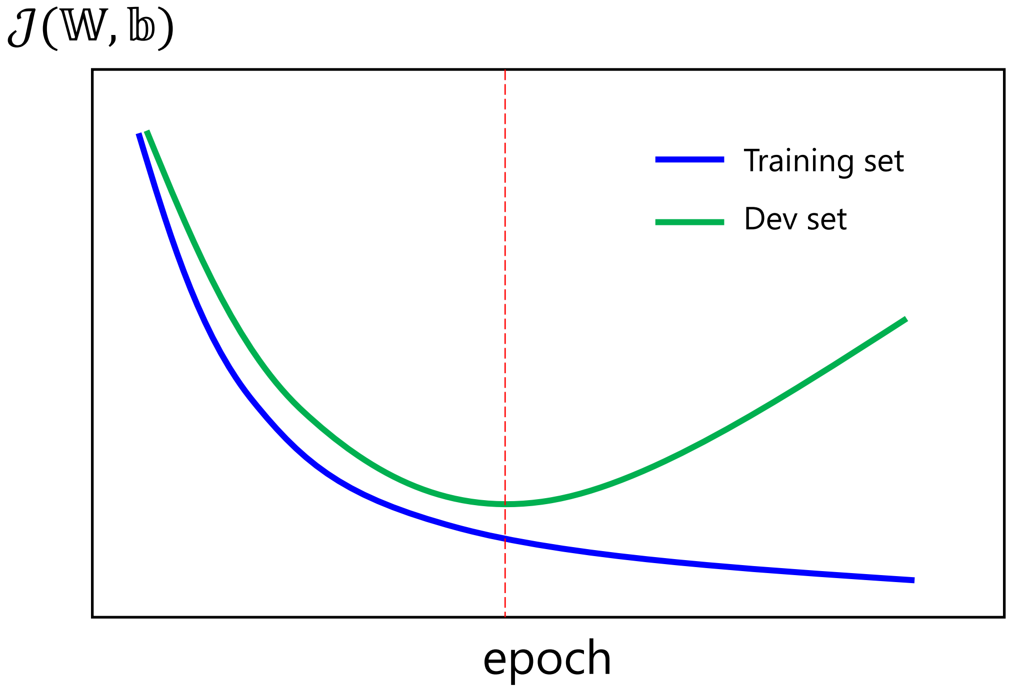
Some idea to set your keepprob



- The keepprob can be different value in different layer
- The lower $n^{[l]}$ (number of hidden units in the layer l) the higher keepprob
- Input layer has keepprob equal to 1

Other Regularizations

Early stopping



- Stop training based on the cost convergence

Gradient

Normalized Input

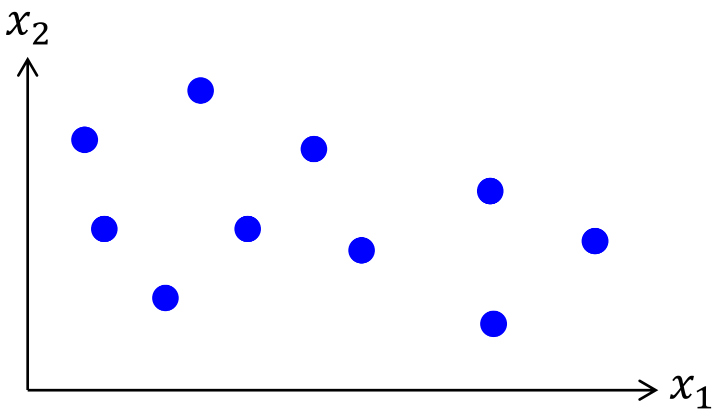
Speed up your training

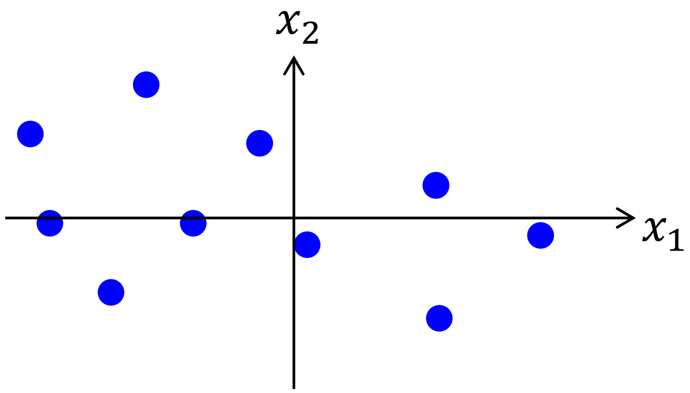
- Subtract mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}^{(i)}$$

$$\mathbf{X} \coloneqq \mathbf{X} - \mu$$

- Data will have zero mean

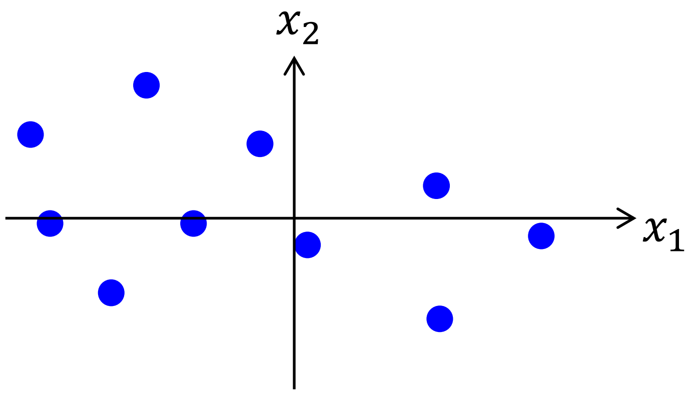




x_1 has higher variance compared to x_2

Normalized Input

Speed up your training

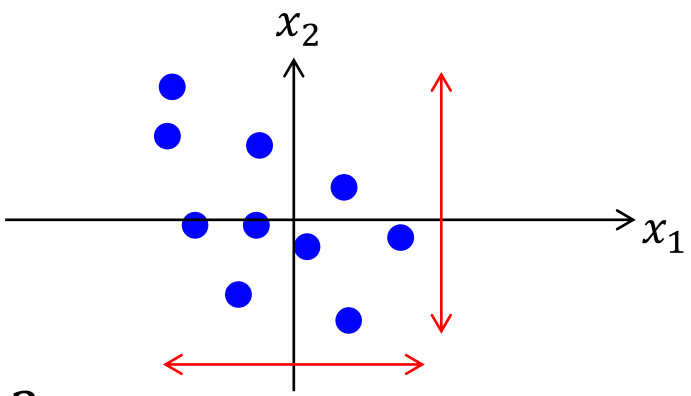


- Normalized variance

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)}\right)^2$$

$$\mathbf{X} = \frac{\mathbf{X}}{\sigma^2}$$

- Data will have zero mean and normalized variance



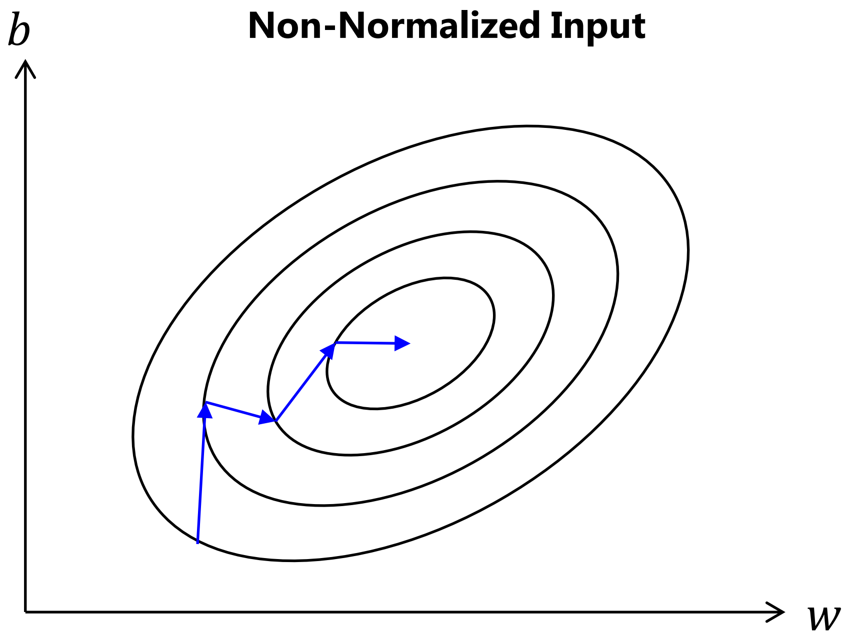
- μ and σ^2 from training set will be used in test set

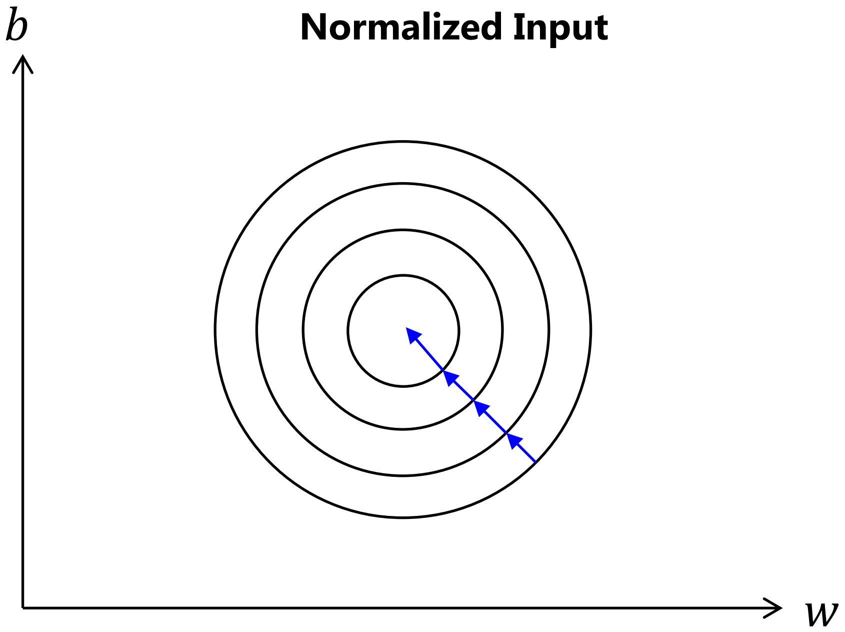
Normalized Input

Speed up your training

- Given a contour of the cost function

Gradient is oscillated and take longer time to the minimum value





Gradient moves more directly to the minimum value

Minibatch Gradient Descent

Batch VS Mini-batch

Dividing training data into minibatch

Notations

$\mathbf{X}^{\{t\}}$ and $\mathbf{y}^{\{t\}}$ for training samples of mini-batch t

- Vectorization enables effective computation over m samples

$$\mathbf{X}_{n_x \times m} = \left[\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}\right]$$

$$\mathbf{y}_{1 \times m} = \left[y^{(1)}, y^{(2)}, \dots, y^{(m)}\right]$$

- What if m is extremely large, e.g. 5,000,000
  - Use mini-batch

$$\mathbf{X}_{n_x \times m} = \Big[\underbrace{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(1000)}}_{\mathbf{X}^{\{1\}}} \Big| \underbrace{\mathbf{x}^{(1001)}, \mathbf{x}^{(1002)}, \dots, \mathbf{x}^{(2000)}}_{\mathbf{X}^{\{2\}}} \Big| \cdots\cdots \Big| \underbrace{\dots\dots \mathbf{x}^{(5,000,000)}}_{\mathbf{X}^{\{5000\}}}\Big]$$

$$\mathbf{y}_{1 \times m} = \Big[\underbrace{y^{(1)}, y^{(2)}, \dots, y^{(1000)}}_{\mathbf{y}^{\{1\}}} \Big| \underbrace{y^{(1001)}, y^{(1002)}, \dots, y^{(2000)}}_{\mathbf{y}^{\{2\}}} \Big| \cdots\cdots \Big| \underbrace{\dots\dots y^{(5,000,000)}}_{\mathbf{y}^{\{5000\}}}\Big]$$

Mini-Batch Implementation

From the previous example

while {

for t in range(0, 5000):

 # Forward propagation on $X^{\{t\}}$

$$\mathbf{z}^{[1]} = \mathbf{w}^{[1]} \cdot \mathbf{X}^{\{t\}} + \mathbf{b}^{[1]}$$

$$\boldsymbol{a}^{[1]} = g(\mathbf{z}^{[1]})$$

$$\vdots$$

$$\mathbf{z}^{[L]} = \mathbf{w}^{[L]} \cdot \boldsymbol{a}^{[L-1]} + \mathbf{b}^{[L]}$$

$$\boldsymbol{a}^{[L]} = g(\mathbf{z}^{[L]})$$

Use vectorization implementation

from $\mathbf{X}^{\{t\}}, \mathbf{y}^{\{t\}}$

$$\mathcal{J}^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \times 1000} \sum_{l} \left\| \mathbf{w}^{[l]} \right\|_F^2$$

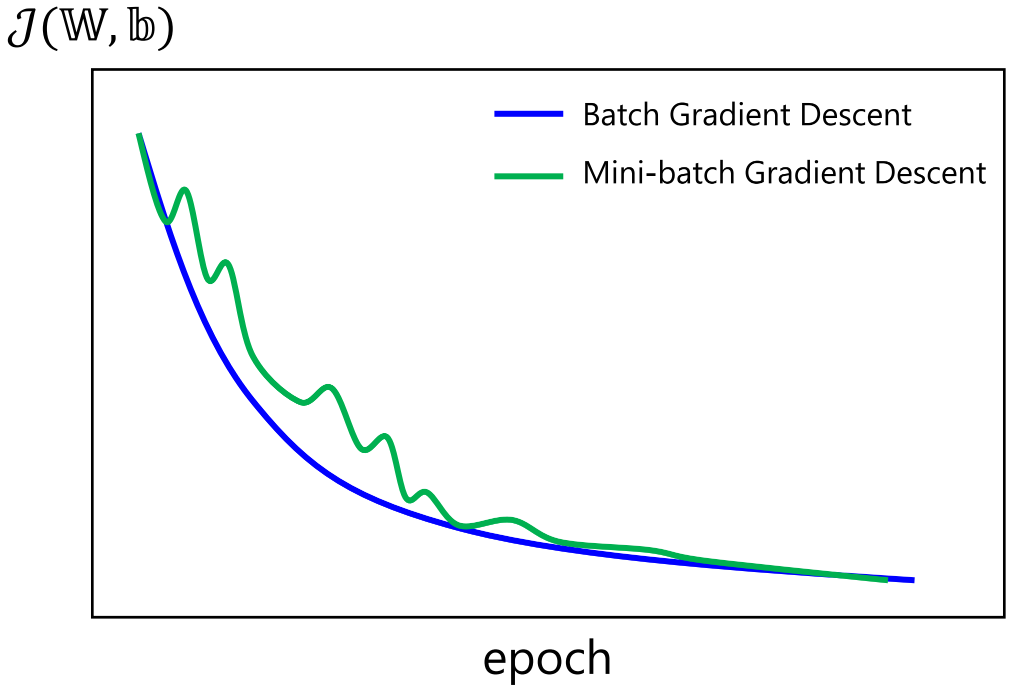
Backward propagation w.r.t. $\mathcal{J}^{\{t\}}$ using $\mathbf{X}^{\{t\}}$ and $\mathbf{y}^{\{t\}}$

$$\mathbf{w}^{[l]} = \mathbf{w}^{[l]} - \alpha \mathbf{dw}^{[l]}$$
$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \mathbf{db}^{[l]}$$

} until converge

Mini-batch gradient descent

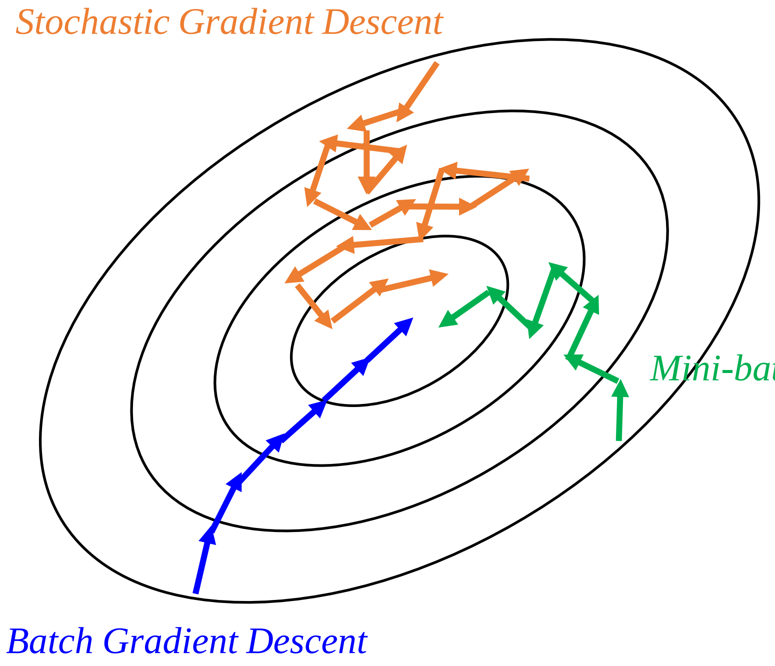
Cost function comparison



- The cost function of mini-batch gradient descent could be oscillated
  - $\mathbf{X}^{\{1\}}$, $\mathbf{y}^{\{1\}}$ might be an easy gradient
  - $\mathbf{X}^{\{2\}}$, $\mathbf{y}^{\{2\}}$ might be a harder gradient
  - Overall should be downward

Choosing Mini-batch size

The extreme cases



- Mini-batch size = m
  - Batch gradient descent

$$\left(\mathbf{X}^{\{1\}}, \mathbf{y}^{\{1\}}\right) = (\mathbf{X}, \mathbf{y})$$

- Mini-batch size = 1
  - Stochastic gradient descent

$$\left(\mathbf{X}^{\{1\}}, \mathbf{y}^{\{1\}}\right) = \left(\mathbf{x}^{(1)}, y^{(1)}\right)$$

Characteristics	Batch	Mini-batch	SGD
Speed	Very Slow	Fast	Slow *Does not leverage vectorization*
Convergence to minimum	Guarantee	Not Guarantee	Not Guarantee

Choosing Mini-batch size

Conclusion

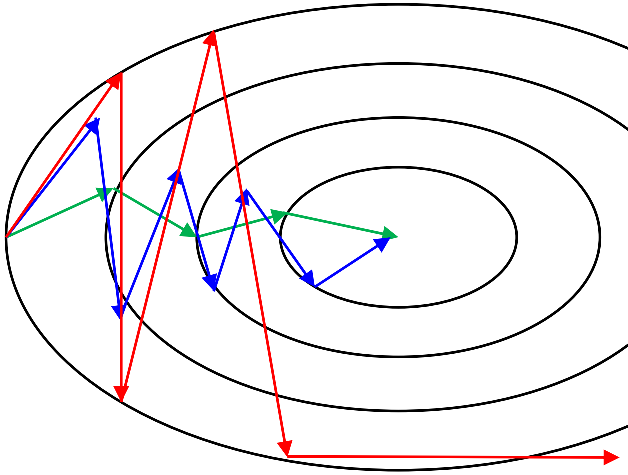
- Small Training Set ($m \leq 2000$)
  - Use Batch gradient descent
- Otherwise
  - Use Mini-batch
- Suggested mini-batch size
  - 2^k to perfectly fit in CPU/GPU memory, e.g. 64, 128, 256, 512, etc

Gradient Optimization

Momentum Gradient Descent, RMSprop, and ADAM

Momentum Gradient Descent

Idea



Gradient Descent

Momentum Gradient Descent

Overshoot (too much learning rate)

- Desired movement directions
  - Fast to the minimum point

    From our example, movement on this direction (→) should be fast

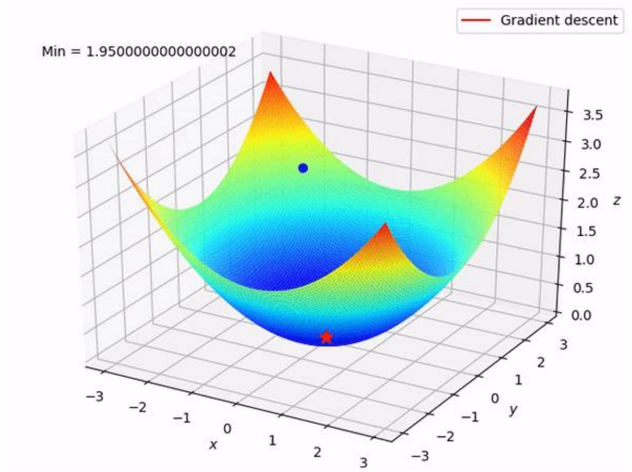
  - Slow to others

    From our example, movement on this direction (↕) should be slow

Momentum Gradient Descent

Velocity and Acceleration

Similar to a ball rolling in a bowl



- On iteration t:

  Compute **dW**, **db** on a set of mini-batch

$$\mathbf{v_{dW}} = \beta\mathbf{v_{dW}} + (1-\beta)\mathbf{dW}$$

friction velocity acceleration

$$\mathbf{v_{db}} = \beta\mathbf{v_{db}} + (1-\beta)\mathbf{db}$$

$$\mathbf{W} = \mathbf{W} - \alpha\mathbf{v_{dW}}$$

$$\mathbf{b} = \mathbf{b} - \alpha\mathbf{v_{db}}$$

Momentum Gradient Descent

Implementation

Initialize $\mathbf{v_{dW}} = 0, \mathbf{v_{db}} = 0$

Note: dimension of $\mathbf{v_{dW}} = \mathbf{dW}$

dimension of $\mathbf{v_{db}} = \mathbf{db}$

Hyper-parameters are α, β

- On iteration t:

  Compute **dW**, **db** on a set of mini-batch

  $$\mathbf{v_{dW}} = \beta \mathbf{v_{dW}} + (1-\beta)\mathbf{dW}$$

  $$\mathbf{v_{db}} = \beta \mathbf{v_{db}} + (1-\beta)\mathbf{db}$$

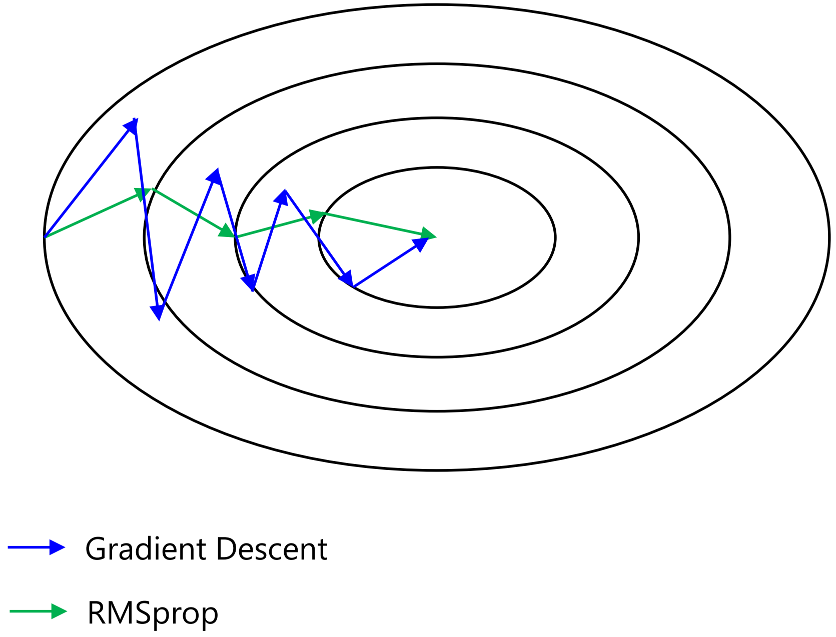
  $$\mathbf{W} = \mathbf{W} - \alpha \mathbf{v_{dW}}$$

  $$\mathbf{b} = \mathbf{b} - \alpha \mathbf{v_{db}}$$

- In practice, $\beta = 0.9$
- Some literature omitted $1 - \beta$

RMSprop

Speed up gradient descent



- On iteration t:

  Compute **dW**, **db** on a set of mini-batch

$$s_{\mathbf{dW}} = \beta s_{\mathbf{dW}} + (1-\beta)\mathbf{dW}^2$$

$$s_{\mathbf{db}} = \beta s_{\mathbf{db}} + (1-\beta)\mathbf{db}^2$$

$$\mathbf{W} = \mathbf{W} - \frac{\alpha \mathbf{dW}}{\sqrt{s_{\mathbf{dW}}} + \varepsilon}$$

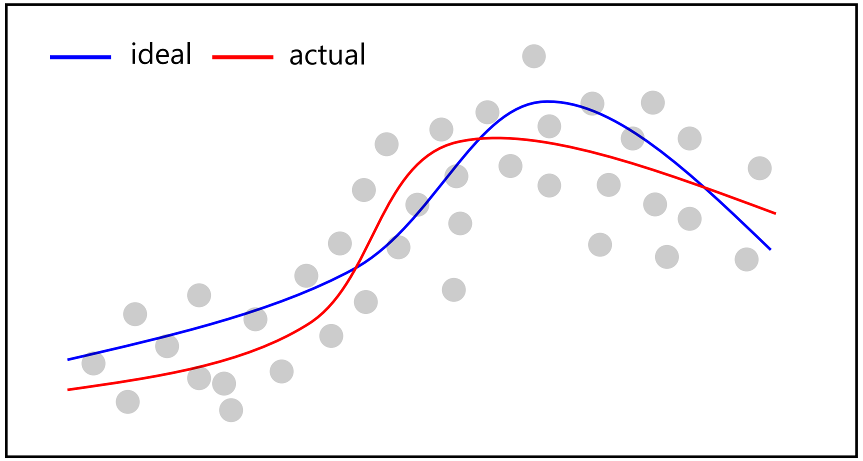
$$\mathbf{b} = \mathbf{b} - \frac{\alpha \mathbf{db}}{\sqrt{s_{\mathbf{db}}} + \varepsilon}$$

a small positive number for numerical stabilization (preventing divided by zero)

- RMSprop enables to use the larger learning rate without overshoot movement

Bias Correction

Idea



- Bias correction of data point t is defined as

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

data point t

$$\text{set } \beta = 0.98, \qquad v_0 = 0$$

$$v_1 = 0.98v_0 + 0.02\theta_1$$

$$v_2 = 0.98v_1 + 0.02\theta_2 = 0.0196\theta_1 + 0.02\theta_2$$

- Use $\dfrac{v_t}{1-\beta^t}$ to estimate the ideal data point t

- Not accurate at the beginning of t but later will be much more accurate

Adaptive Moment Estimation (Adam)

Combining momentum and RMSprop

Initialize $\mathbf{v}_{\mathbf{dW}} = 0, s_{\mathbf{dW}} = 0, \mathbf{v}_{\mathbf{db}} = 0, s_{\mathbf{db}} = 0$

Note: Adam works well in wide range across deep learning applications

- On iteration t:

  Compute **dw**, **db** on a set of mini-batch

$$\mathbf{v}_{\mathbf{dW}} = \beta_1 \mathbf{v}_{\mathbf{dW}} + (1-\beta_1)\mathbf{dW}$$

$$\mathbf{v}_{\mathbf{db}} = \beta_1 \mathbf{v}_{\mathbf{db}} + (1-\beta_1)\mathbf{db}$$

momentum

$$s_{\mathbf{dW}} = \beta_2 s_{\mathbf{dW}} + (1-\beta_2)\mathbf{dW}^2$$

$$s_{\mathbf{db}} = \beta_2 s_{\mathbf{db}} + (1-\beta_2)\mathbf{db}^2$$

RMSprop

$$\mathbf{v}_{\mathbf{dW}}^{\text{correct}} = \frac{\mathbf{v}_{\mathbf{dW}}}{1-\beta_1^t}, \quad \mathbf{v}_{\mathbf{db}}^{\text{correct}} = \frac{\mathbf{v}_{\mathbf{db}}^{\text{correct}}}{1-\beta_1^t}$$

$$\mathbf{s}_{\mathbf{dW}}^{\text{correct}} = \frac{\mathbf{s}_{\mathbf{dW}}}{1-\beta_2^t}, \quad \mathbf{s}_{\mathbf{db}}^{\text{correct}} = \frac{\mathbf{s}_{\mathbf{db}}^{\text{correct}}}{1-\beta_2^t}$$

bias correction

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\mathbf{v}_{\mathbf{dW}}^{\text{correct}}}{\sqrt{\mathbf{s}_{\mathbf{dW}}^{\text{correct}}} + \varepsilon}$$

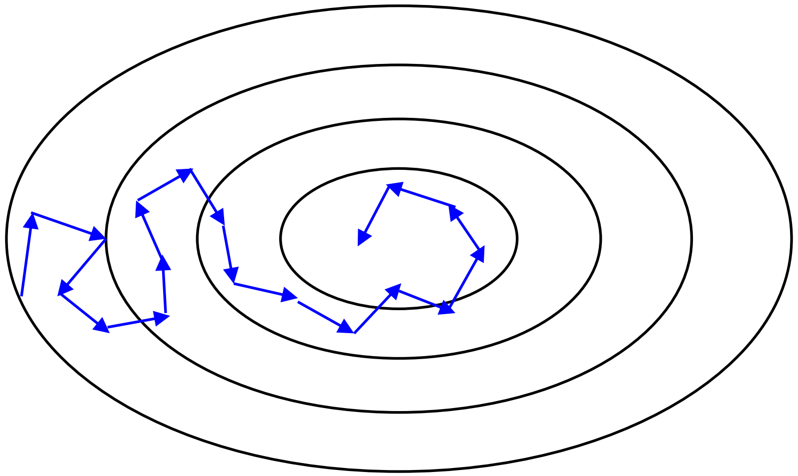
$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\mathbf{v}_{\mathbf{db}}^{\text{correct}}}{\sqrt{\mathbf{s}_{\mathbf{db}}^{\text{correct}}} + \varepsilon}$$

updating

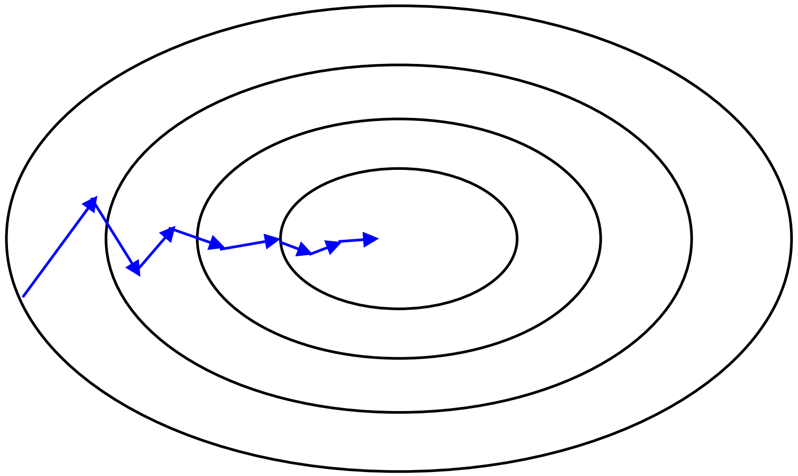
Suggested Hyper-parameters

Fixed learning rate VS Decayed Learning rate

- α: Need to be tuned
- β_1: 0.9
- β_2: 0.999
- ε: 10^{-8}



Fixed learning rate



Decayed Learning rate

Hyperparameter Tuning

Strategy to choose hyper-parameters

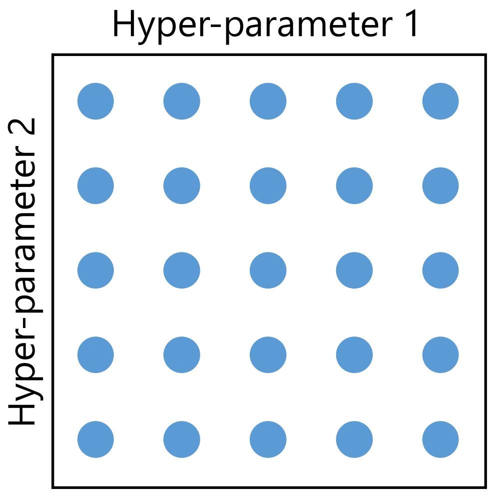
Deep learning hyperparameters

Ranked by importance

- 1st priority
  - α: Learning rate
- 2nd priority
  - β: Adam's parameters only $(\beta_1, \beta_2, \varepsilon)$ (Use default values)
  - Mini-batch size
  - $n^{[l]}$: Number of hidden units in the layer l
- 3rd priority
  - L: Number of layers
  - β: Momentum and RMSprop

Tuning Strategies

Grid approach



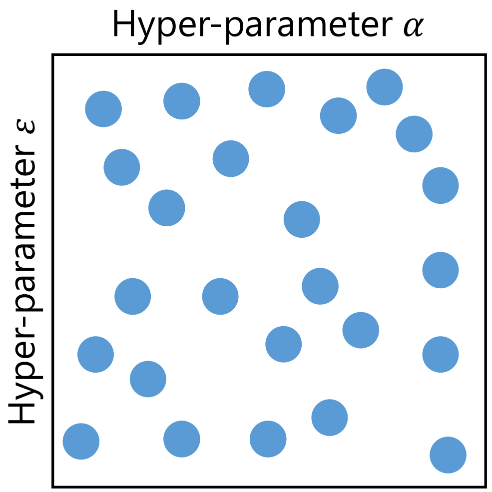
- Grid approach is acceptable for traditional ML

- Limitations
  - Important of some hyperparameters might not be fully address

  - Ex. Grid allows trial on 5 values of α and ε from 25 experiments

Tuning Strategies

For deep learning random grid is better



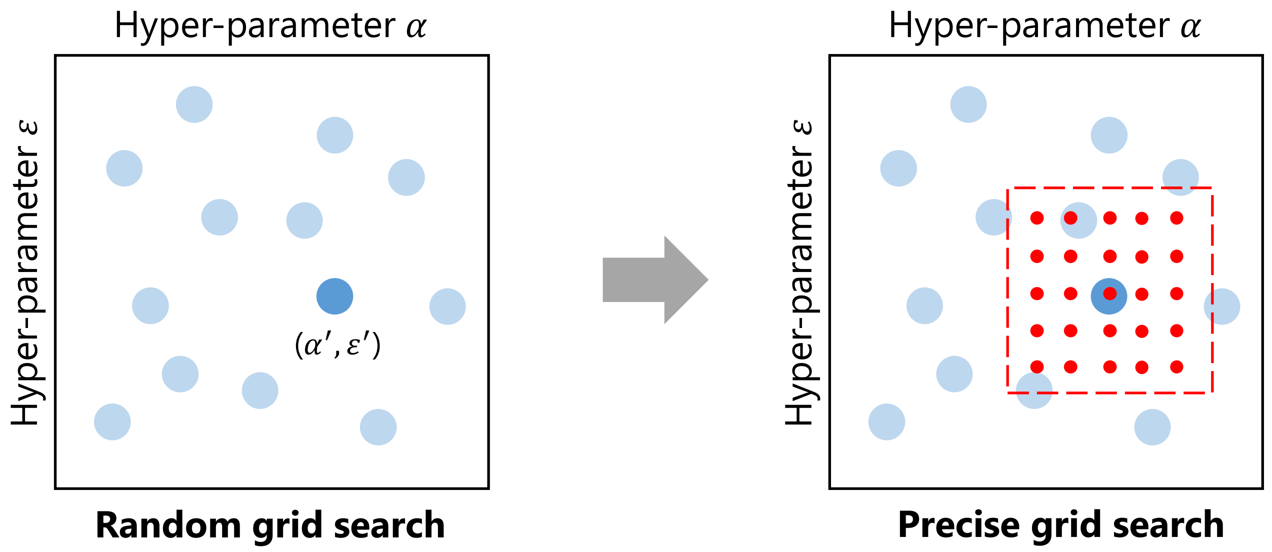
- Random grid allows trial on 25 values of α from 25 experiments

- Random grid gives us more richly to explore sets of possible hyperparameters

Tuning Strategies

Coarse to fine

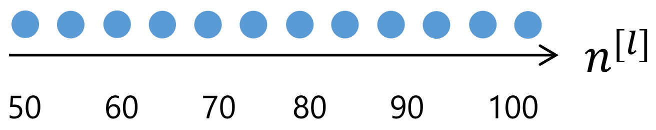
- Start from randomly coarse
- Randomly fine later



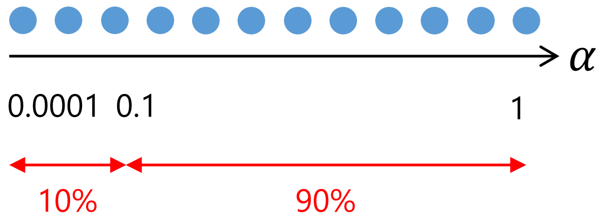
Tuning Strategies

Use the appropriate scale to pitch hyperparameters

- Normal scale works for some hyperparameters
  - $n^{[l]}$
  - L



- Log-scale is good for learning rate



Spending 10% of resource to search between 0.0001 to 0.1 whereas 90% of resources to search between 0.1 to 1