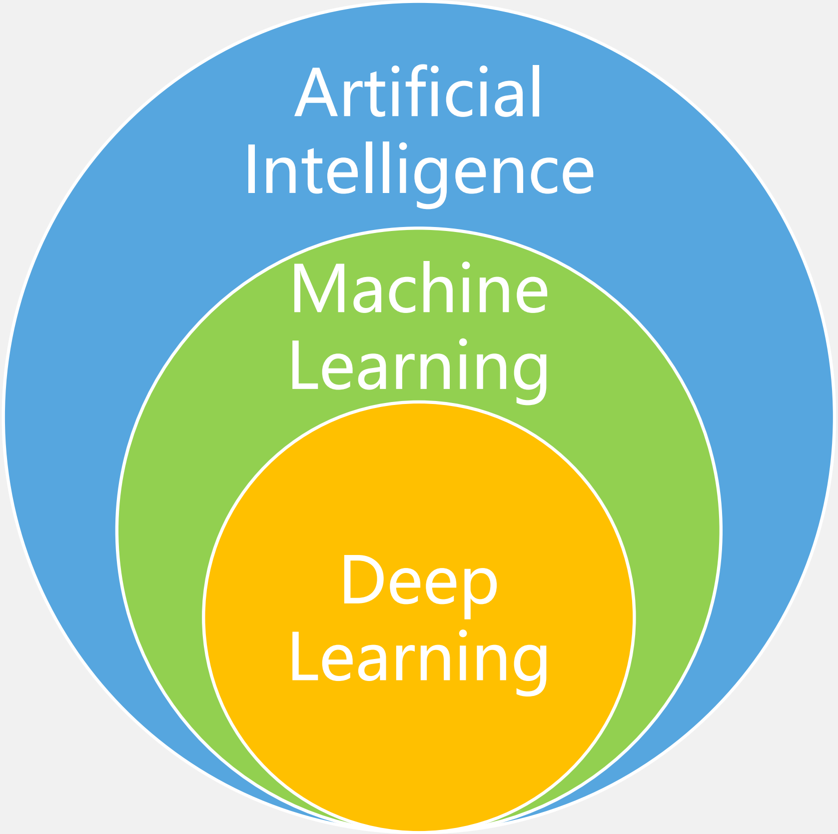
Deep Learning

01

What is AI?

What is AI?

Artificial Intelligence (AI) is using computer to solve the intelligence-related problem



AI≈Find function

Speech recognition

$f($  $)$ = "Morning"

Image recognition

$f($  $)$ = "Cat"

Game prediction

$f($  $)$ = "Move left and shot"

02

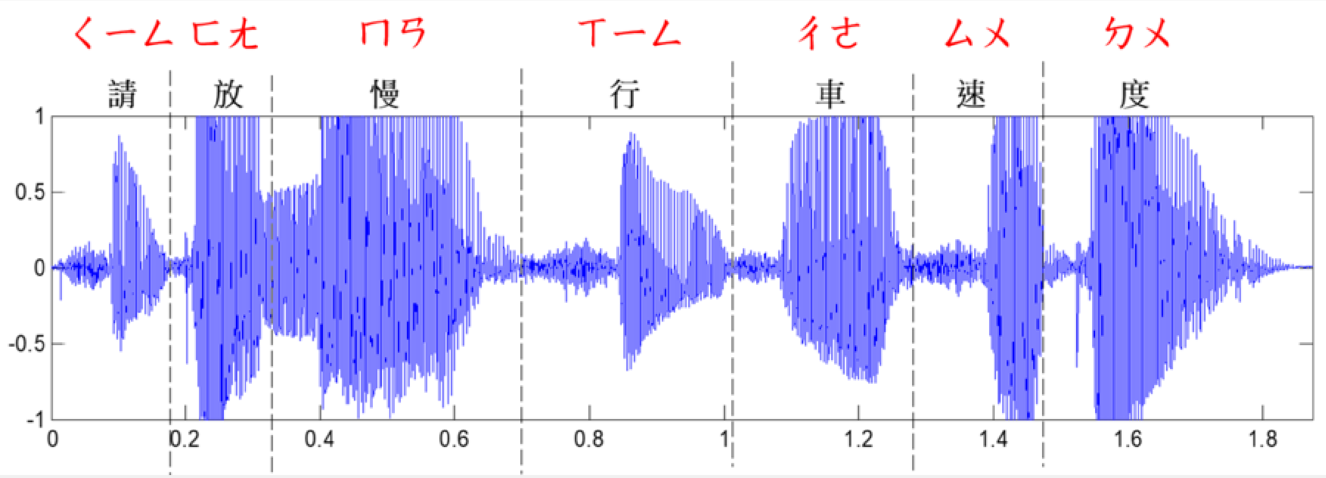
Deep Learning

Deep Learning

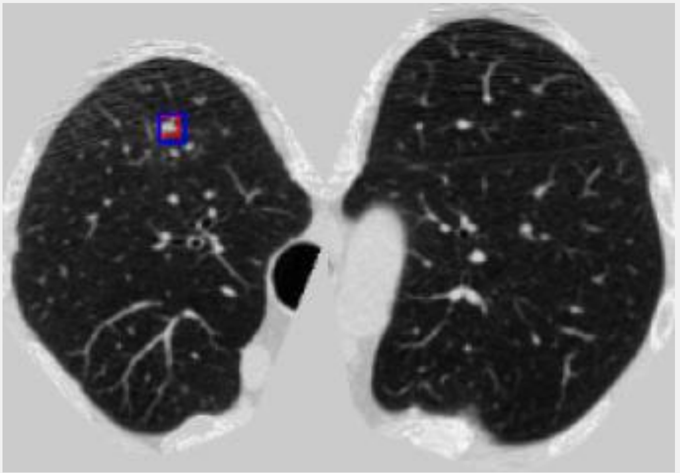
Examples

- Self-Driving
- Face
- Speech
- Medical Image

Link

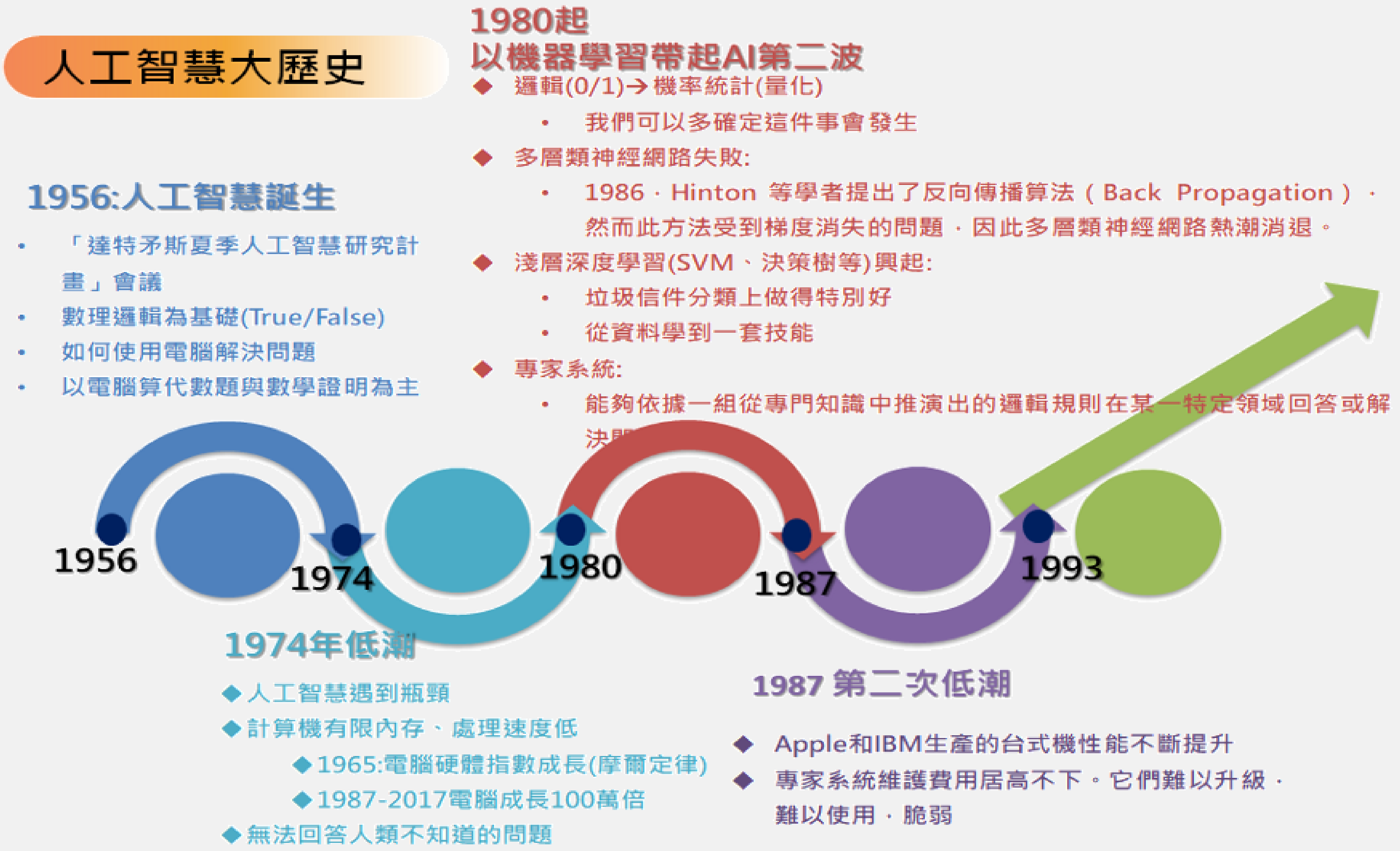


Link



Deep Learning

History



人工智慧大歷史

1956:人工智慧誕生

- 「達特矛斯夏季人工智慧研究計畫」會議
- 數理邏輯為基礎(True/False)
- 如何使用電腦解決問題
- 以電腦算代數題與數學證明為主

1980起
以機器學習帶起AI第二波

- ◆ 邏輯(0/1)→機率統計(量化)
  - 我們可以多確定這件事會發生
- ◆ 多層類神經網路失敗:
  - 1986，Hinton 等學者提出了反向傳播算法（Back Propagation），然而此方法受到梯度消失的問題，因此多層類神經網路熱潮消退。
- ◆ 淺層深度學習(SVM、決策樹等)興起:
  - 垃圾信件分類上做得特別好
  - 從資料學到一套技能
- ◆ 專家系統:
  - 能夠依據一組從專門知識中推演出的邏輯規則在某一特定領域回答或解決

1974年低潮

- ◆人工智慧遇到瓶頸
- ◆計算機有限內存、處理速度低
  - ◆1965:電腦硬體指數成長(摩爾定律)
  - ◆1987-2017電腦成長100萬倍
- ◆無法回答人類不知道的問題

1987 第二次低潮

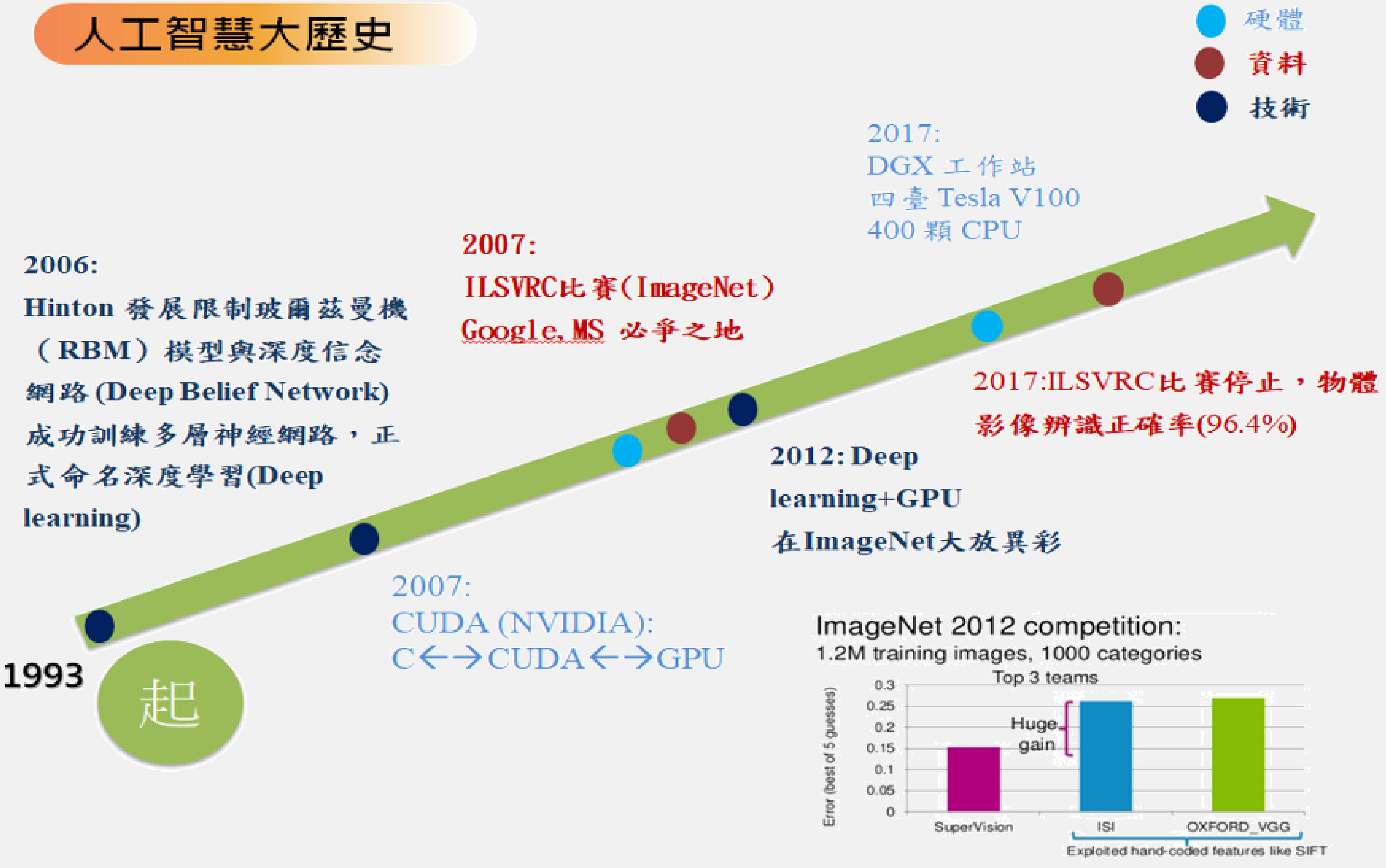
- ◆ Apple和IBM生產的台式機性能不斷提升
- ◆ 專家系統維護費用居高不下。它們難以升級，難以使用，脆弱

Reference

Deep Learning

History

人工智慧大歷史



● 硬體
● 資料
● 技術

1993
起

2006:
Hinton 發展限制玻爾茲曼機（RBM）模型與深度信念網路 (Deep Belief Network) 成功訓練多層神經網路，正式命名深度學習(Deep learning)

2007:
CUDA (NVIDIA):
C←→CUDA←→GPU

2007:
ILSVRC比賽(ImageNet)
Google, MS 必爭之地

2012: Deep learning+GPU 在ImageNet大放異彩

2017:
DGX 工作站
四臺 Tesla V100
400 顆 CPU

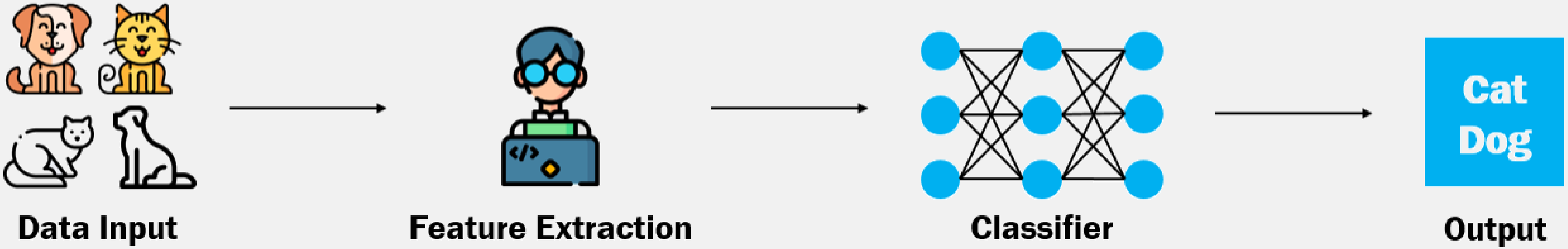
2017:ILSVRC比賽停止，物體影像辨識正確率(96.4%)

ImageNet 2012 competition:
1.2M training images, 1000 categories

Deep Learning

DL vs ML

Machine Learning

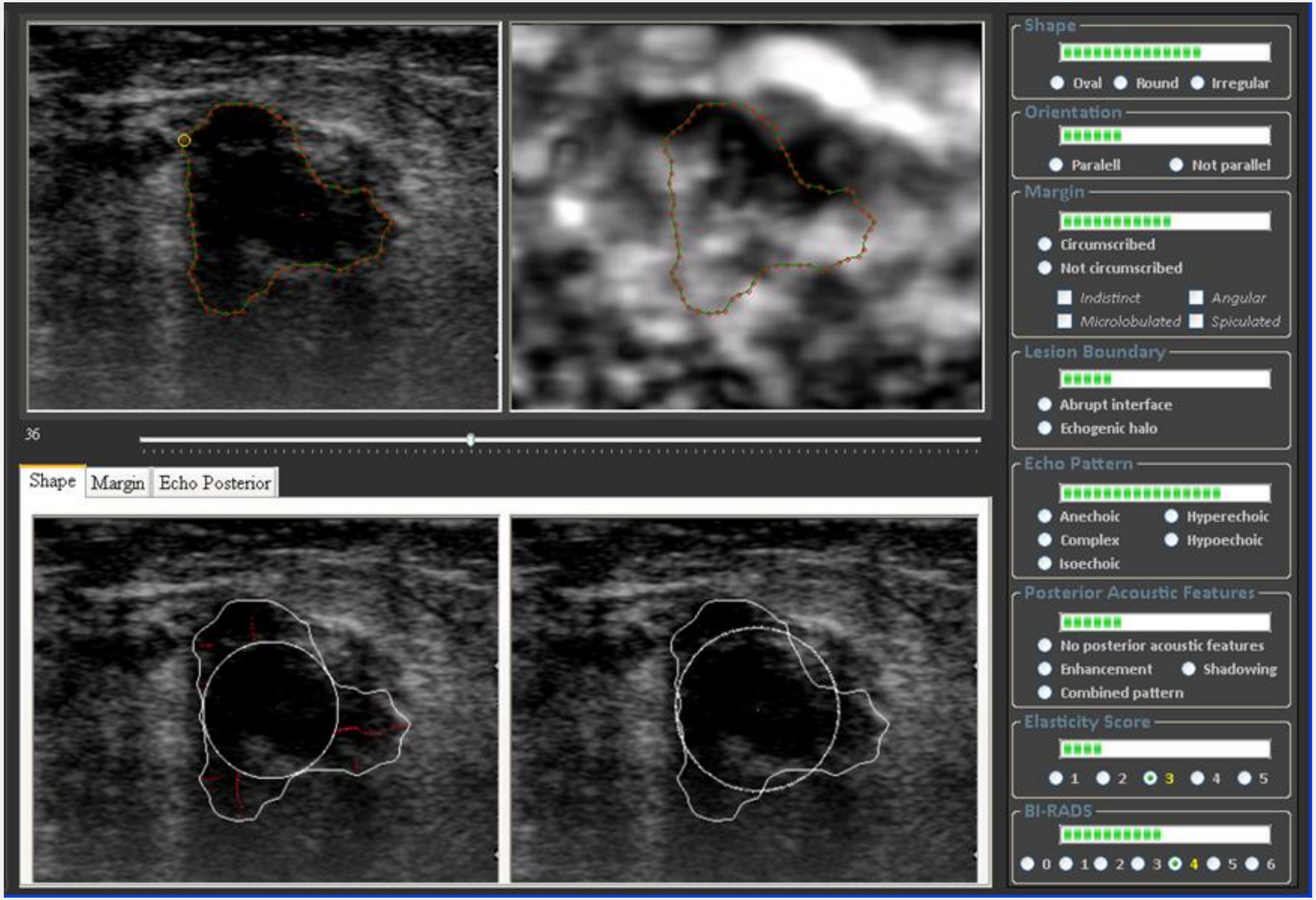


Data Input → Feature Extraction → Classifier → Output

Cat
Dog

[Reference](#)

Deep Learning



Shape
○ Oval ○ Round ○ Irregular

Orientation
○ Paralell ○ Not parallel

Margin
○ Circumscribed
○ Not circumscribed
☐ *Indistinct* ☐ *Angular*
☐ *Microlobulated* ☐ *Spiculated*

Lesion Boundary
○ Abrupt interface
○ Echogenic halo

Echo Pattern
○ Anechoic ○ Hyperechoic
○ Complex ○ Hypoechoic
○ Isoechoic

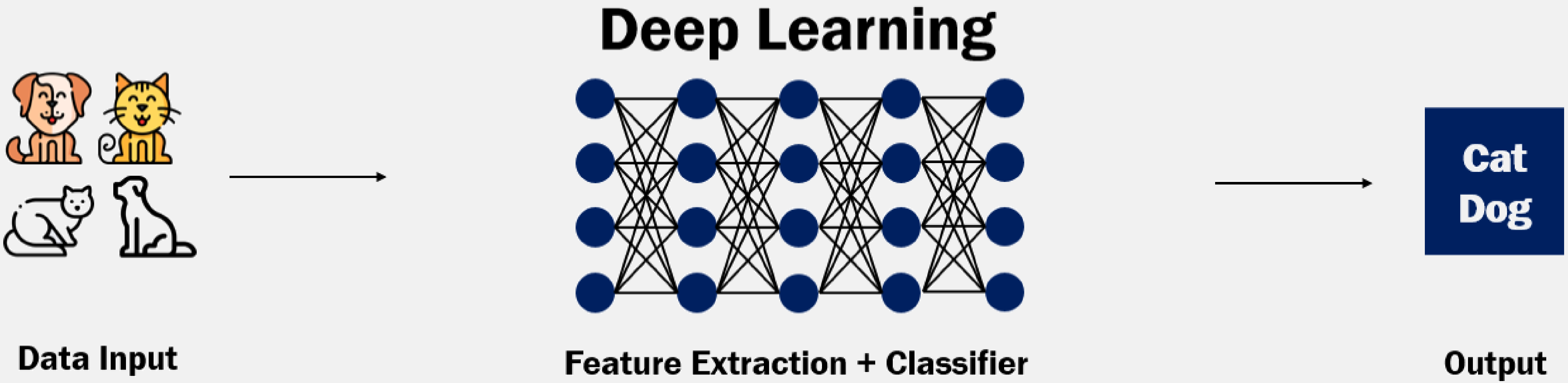
Posterior Acoustic Features
○ No posterior acoustic features
○ Enhancement ○ Shadowing
○ Combined pattern

Elasticity Score
○ 1 ○ 2 ● 3 ○ 4 ○ 5

BI-RADS
○ 0 ○ 1 ○ 2 ○ 3 ● 4 ○ 5 ○ 6

Deep Learning

DL vs ML



Deep Learning

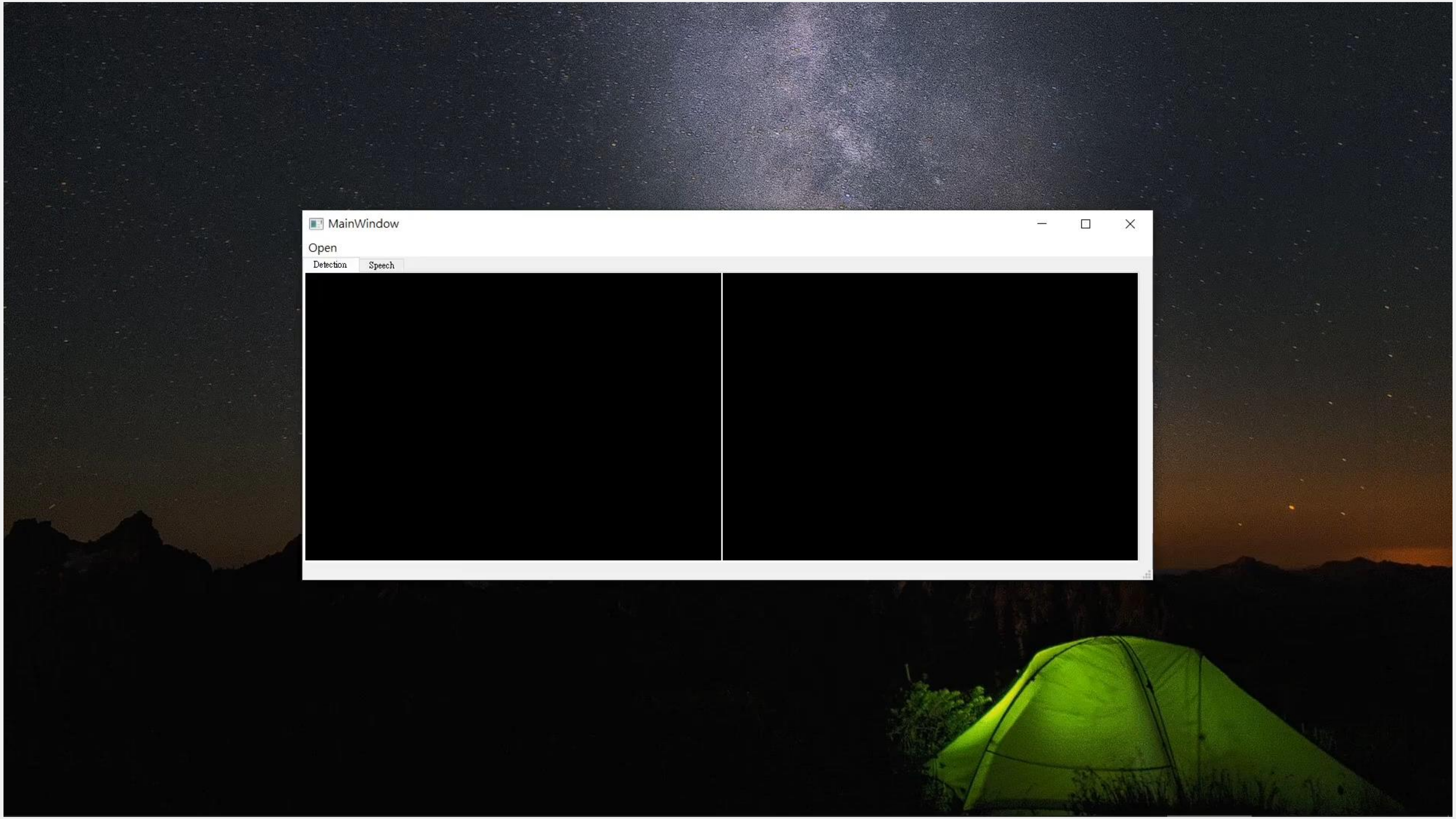
Data Input

Feature Extraction + Classifier

Output

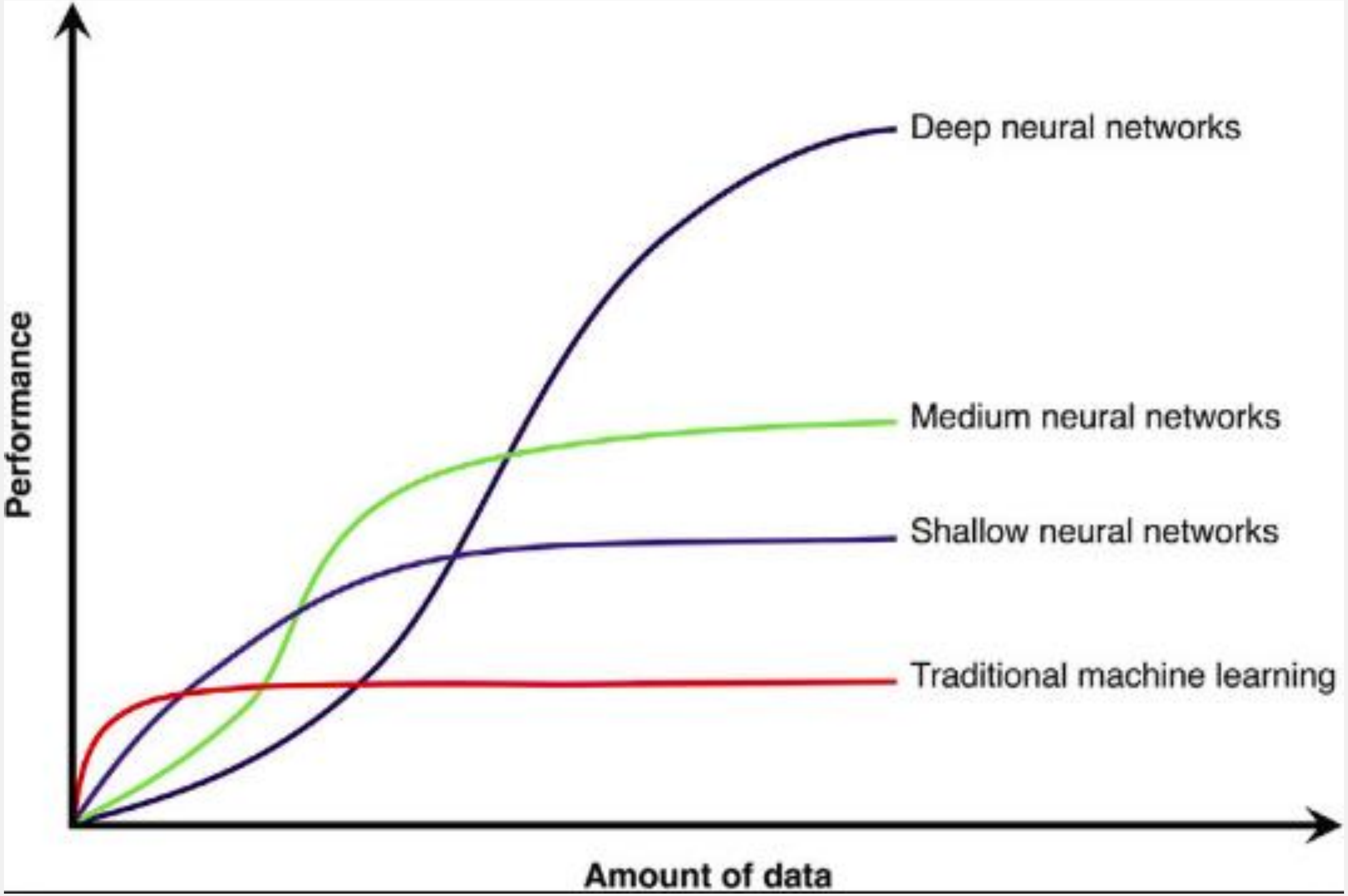
[Reference](#)

Deep Learning



Deep Learning

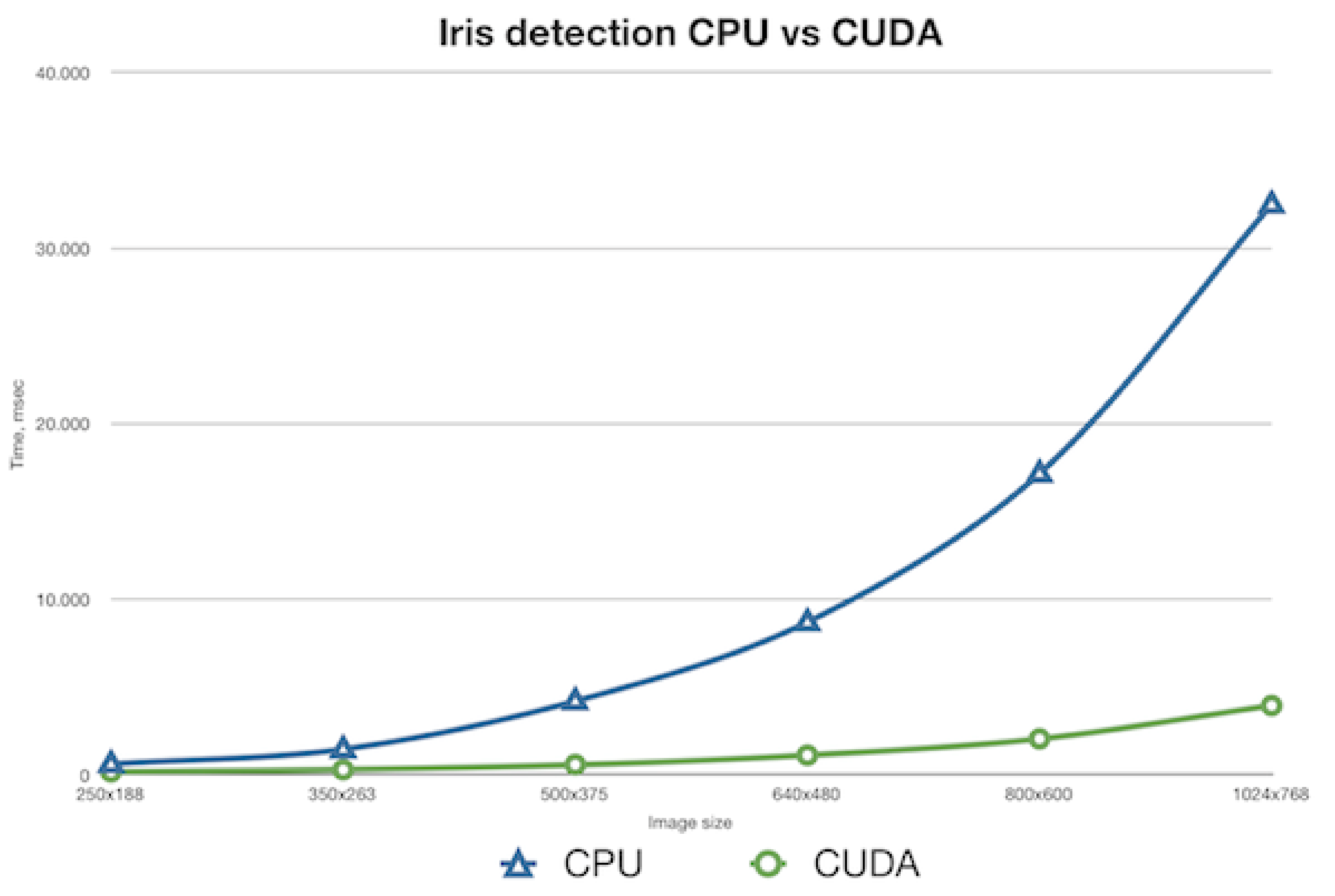
DL vs ML



[Reference]

Deep Learning

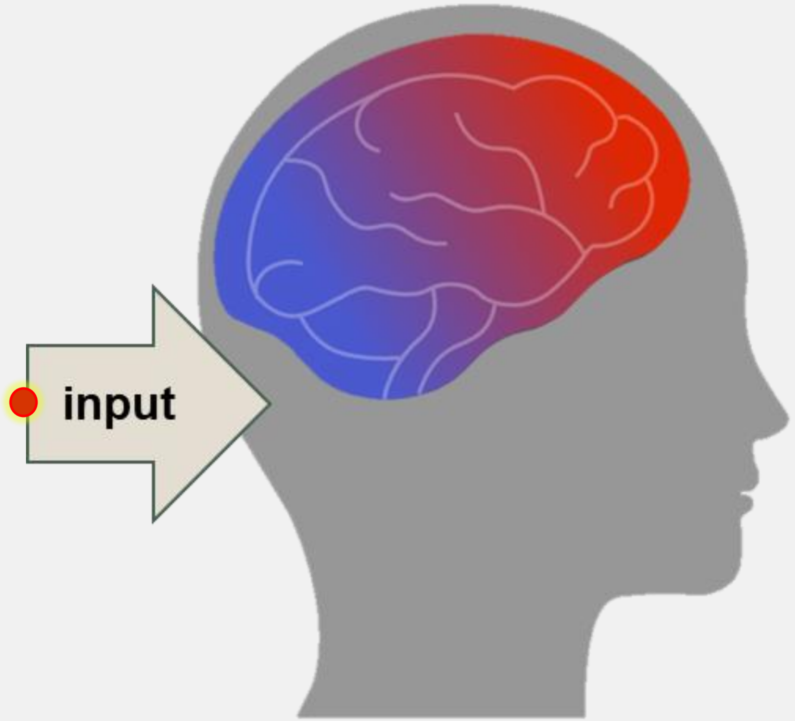
GPU

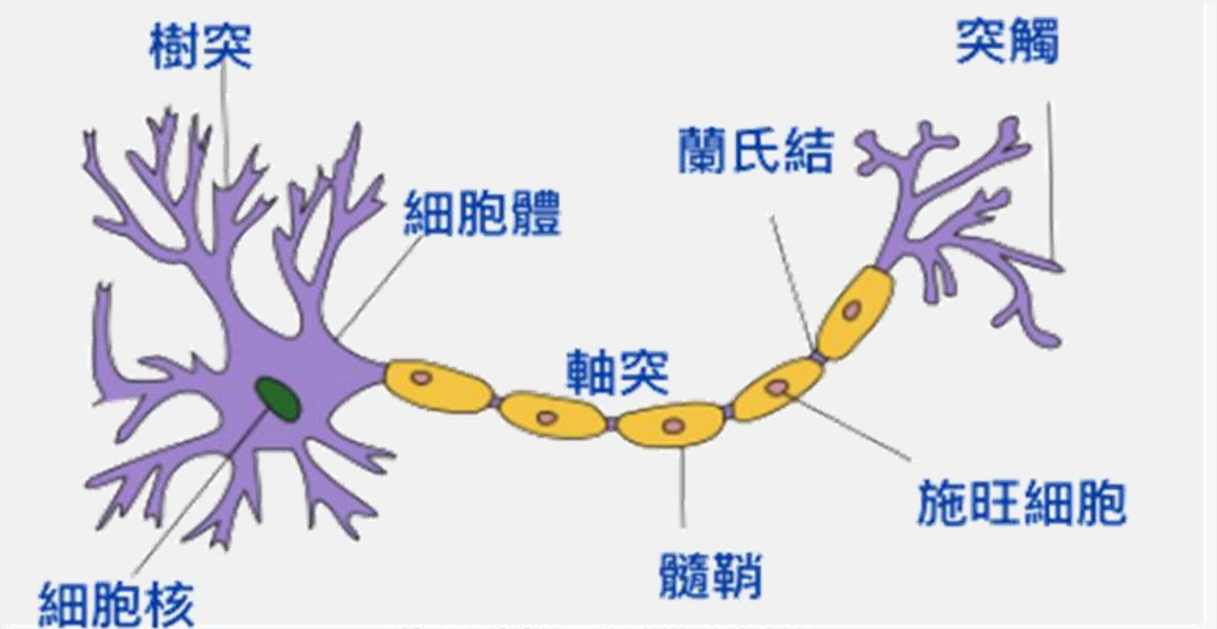


Deep Learning

Biological neural networks

- To understand how deep learning has progressed, we may first look at its inspiration, the neuron





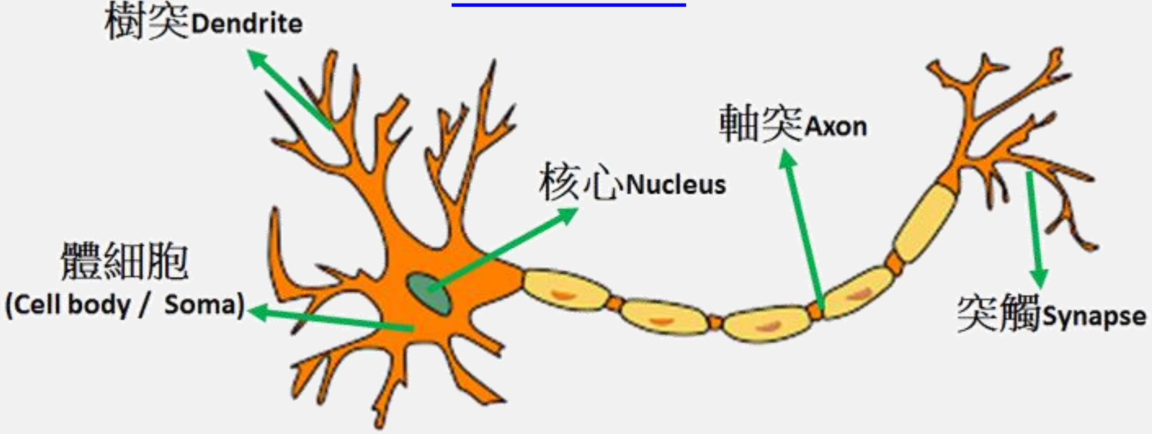
典型神經元的結構

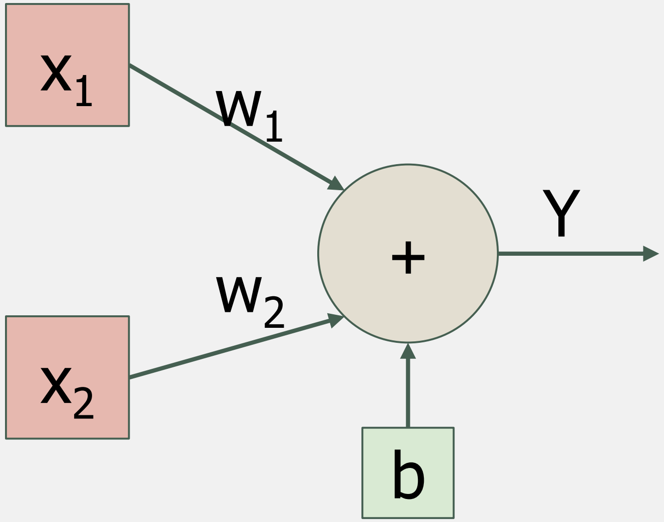
output

[Reference]

Deep Learning

Reference





Neurons

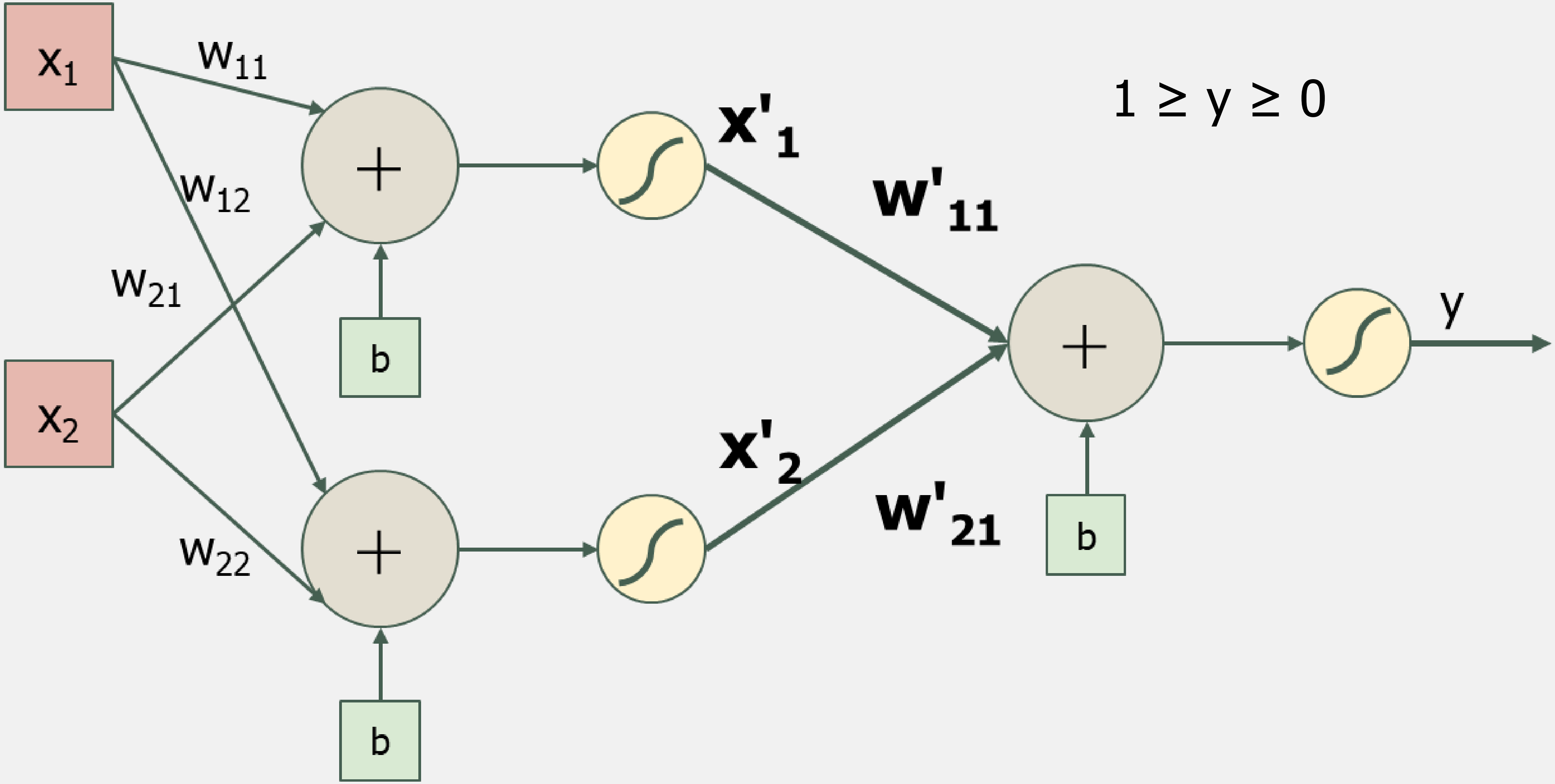
- Have K inputs (dendrites)
- Have 1 output (axon)
- If the sum of the input signals surpasses a threshold, sends an action potential to the axon

Synapses

- Transmit electrical signals between neurons

Deep Learning

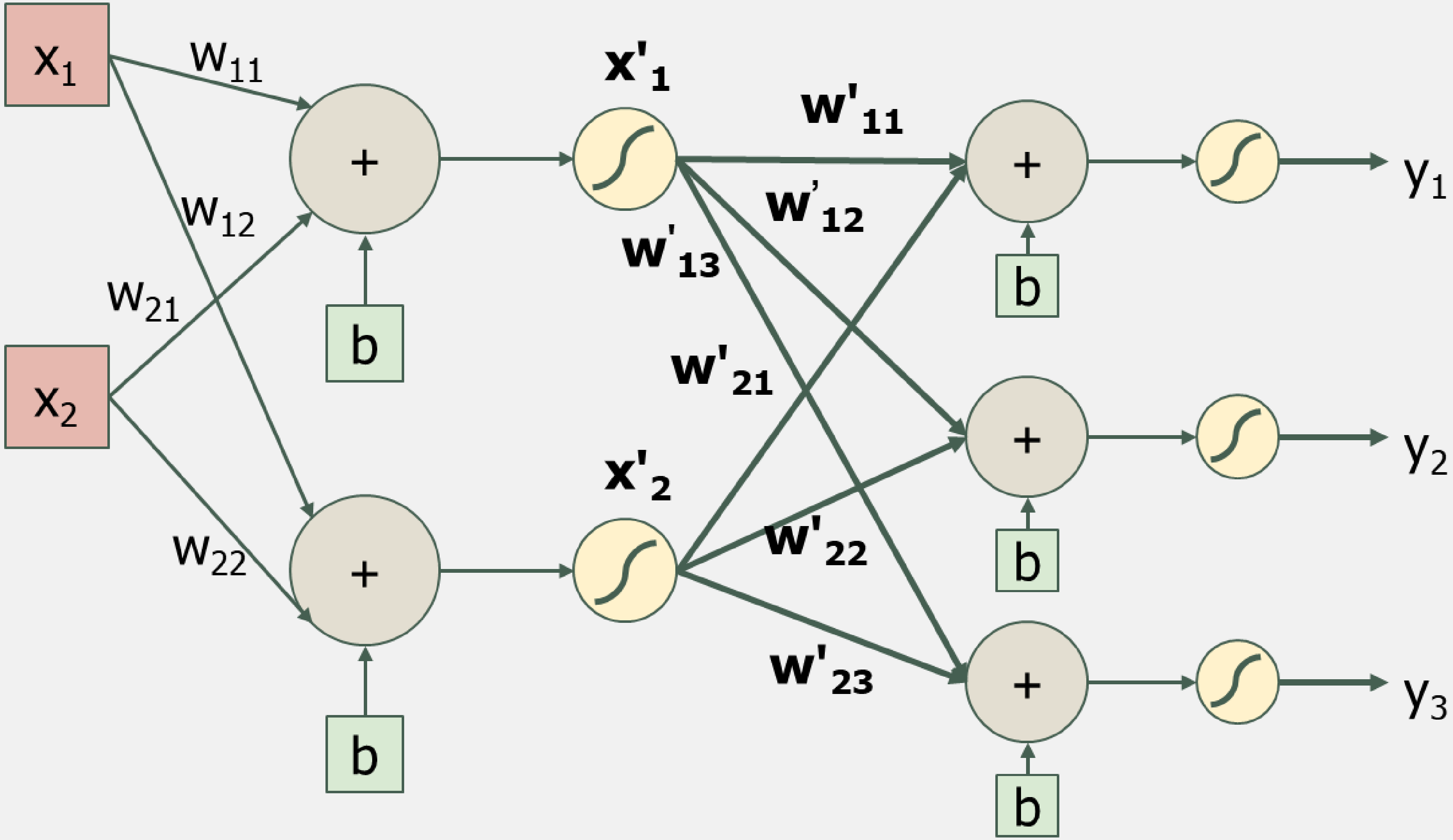
Neural Network



$1 \geq y \geq 0$

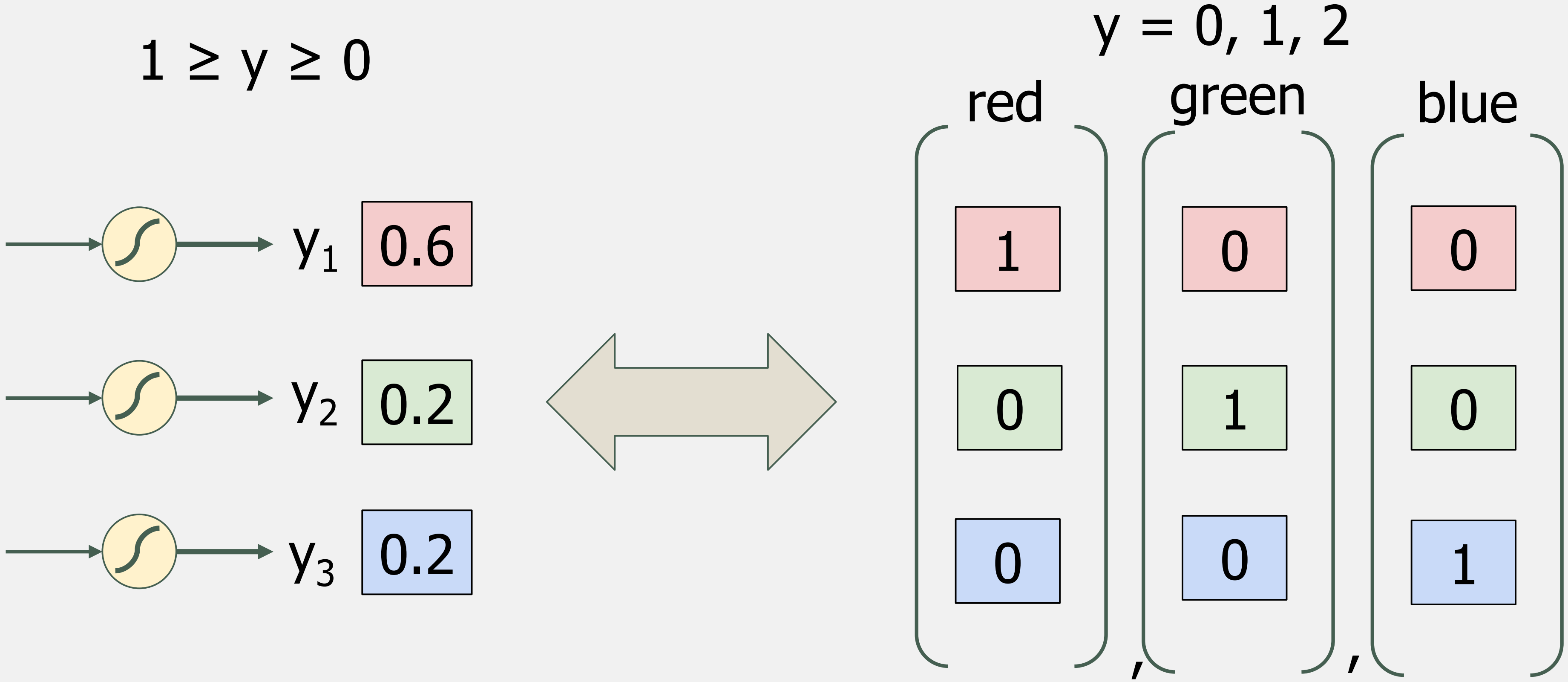
Deep Learning

Neural Network



Deep Learning

Neural Network



$1 \geq y \geq 0$

y_1 0.6

y_2 0.2

y_3 0.2

$y = 0, 1, 2$

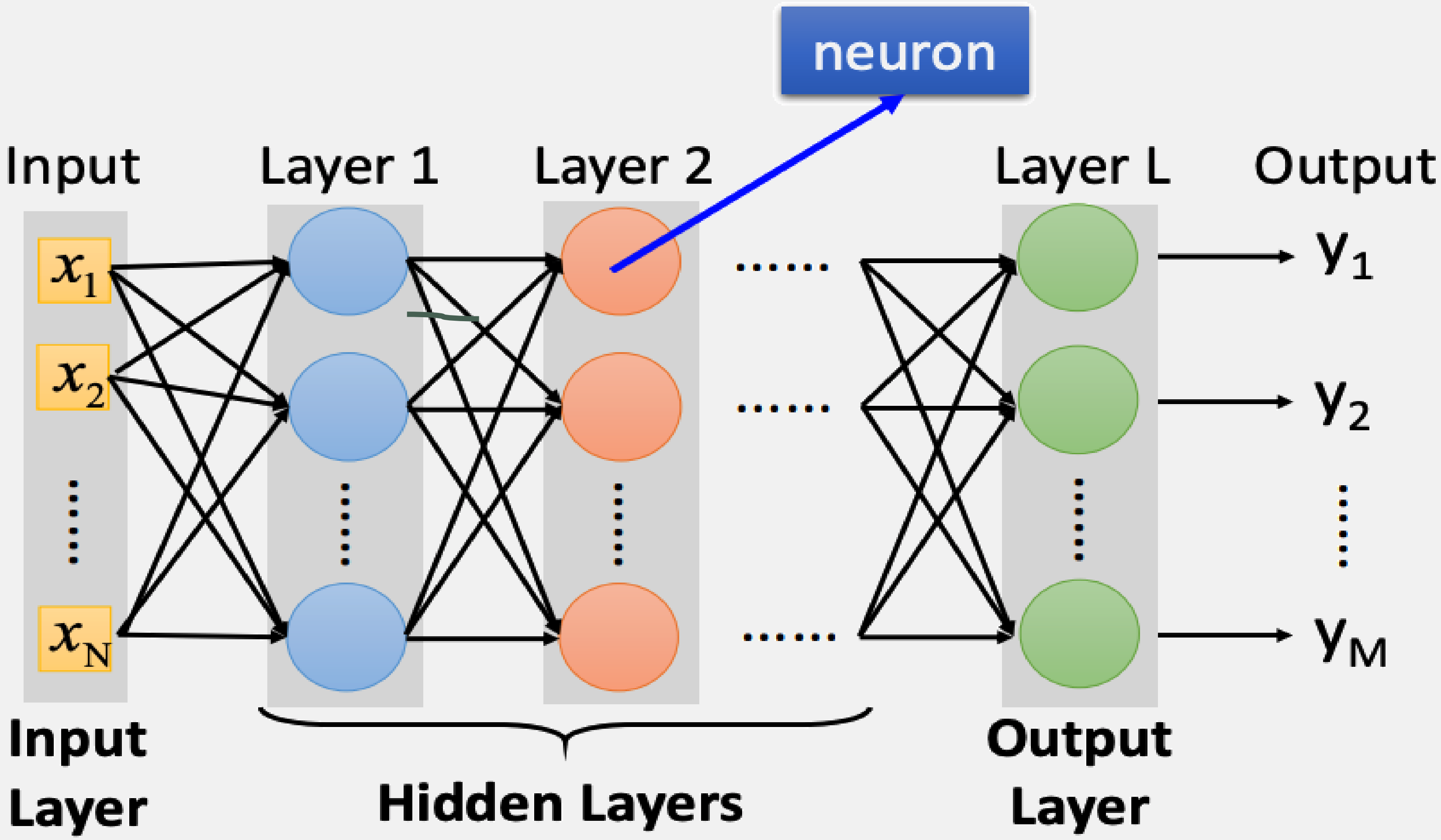
red $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, green $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, blue $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

prediction

label

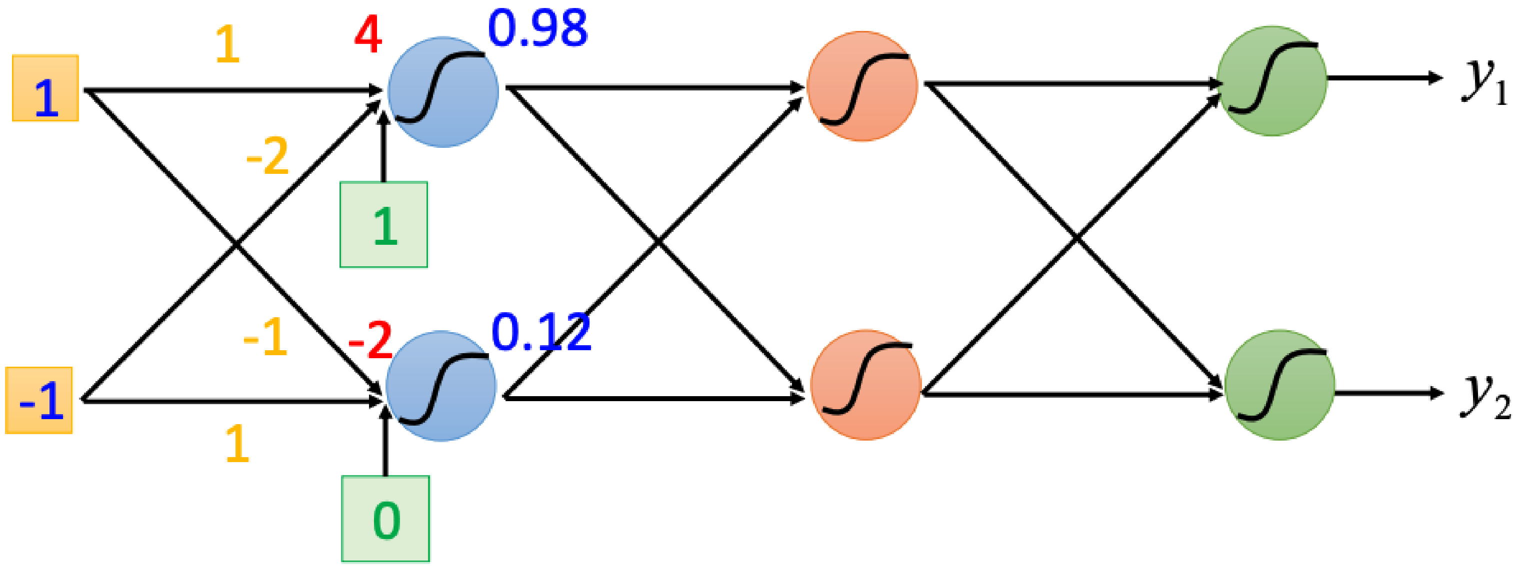
Deep Learning

Fully connected neural network



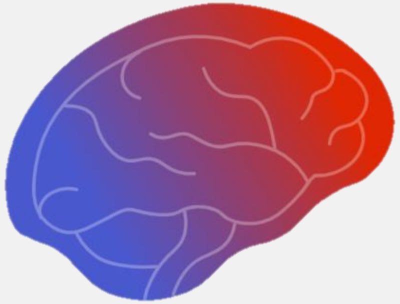
Deep Learning

Matrix computation

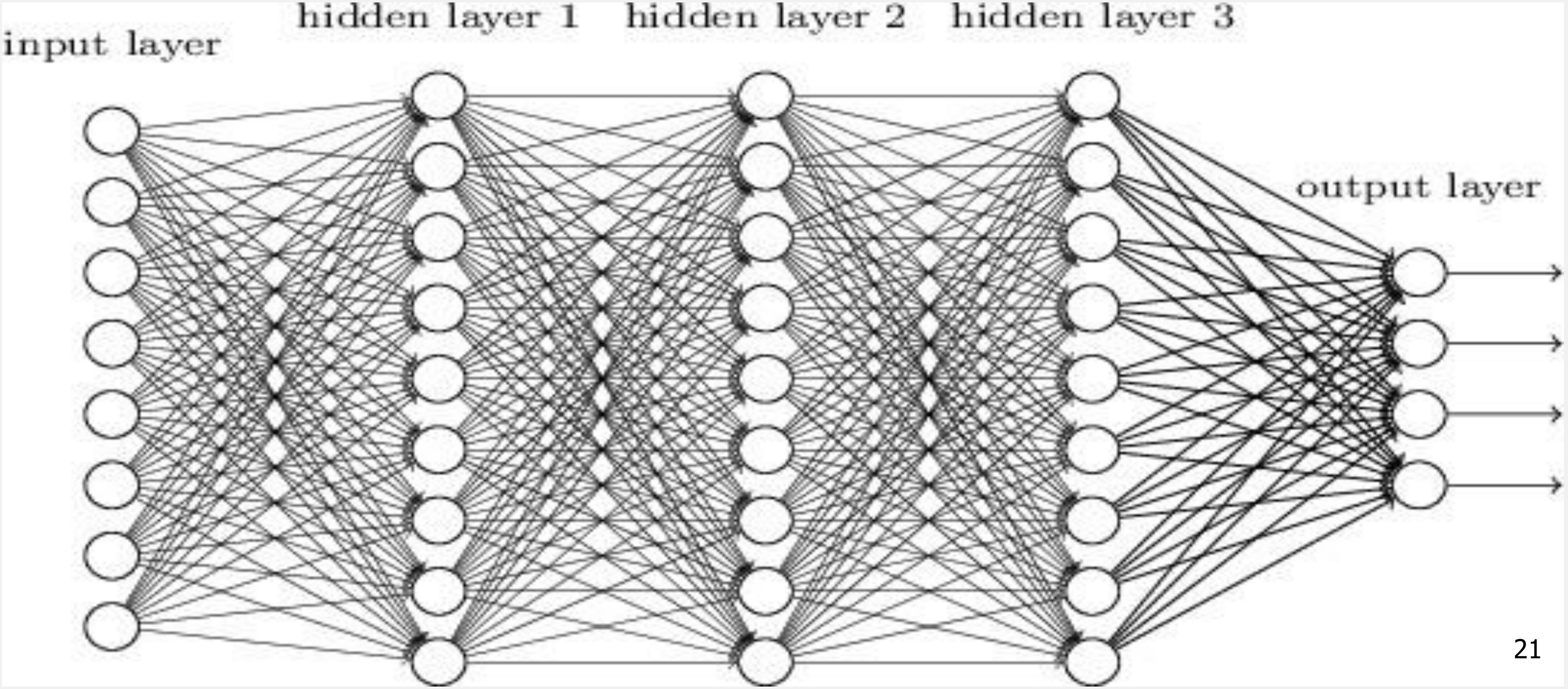


$$\sigma\left(\underbrace{\begin{bmatrix}1 & -2\\ -1 & 1\end{bmatrix}\begin{bmatrix}1\\ -1\end{bmatrix}+\begin{bmatrix}1\\ 0\end{bmatrix}}_{\begin{bmatrix}4\\ -2\end{bmatrix}}\right)=\begin{bmatrix}0.98\\ 0.12\end{bmatrix}$$

Deep Learning

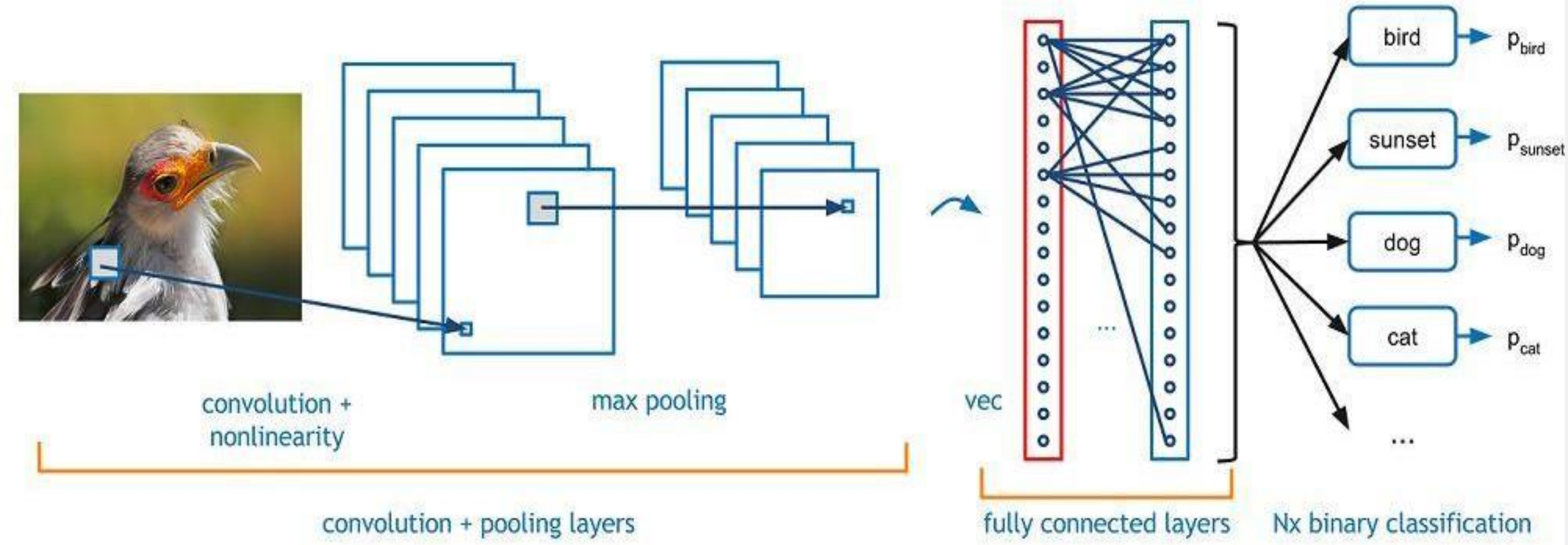


Feed image pixel by pixel to DNN is not efficiency
Can we have better solution?



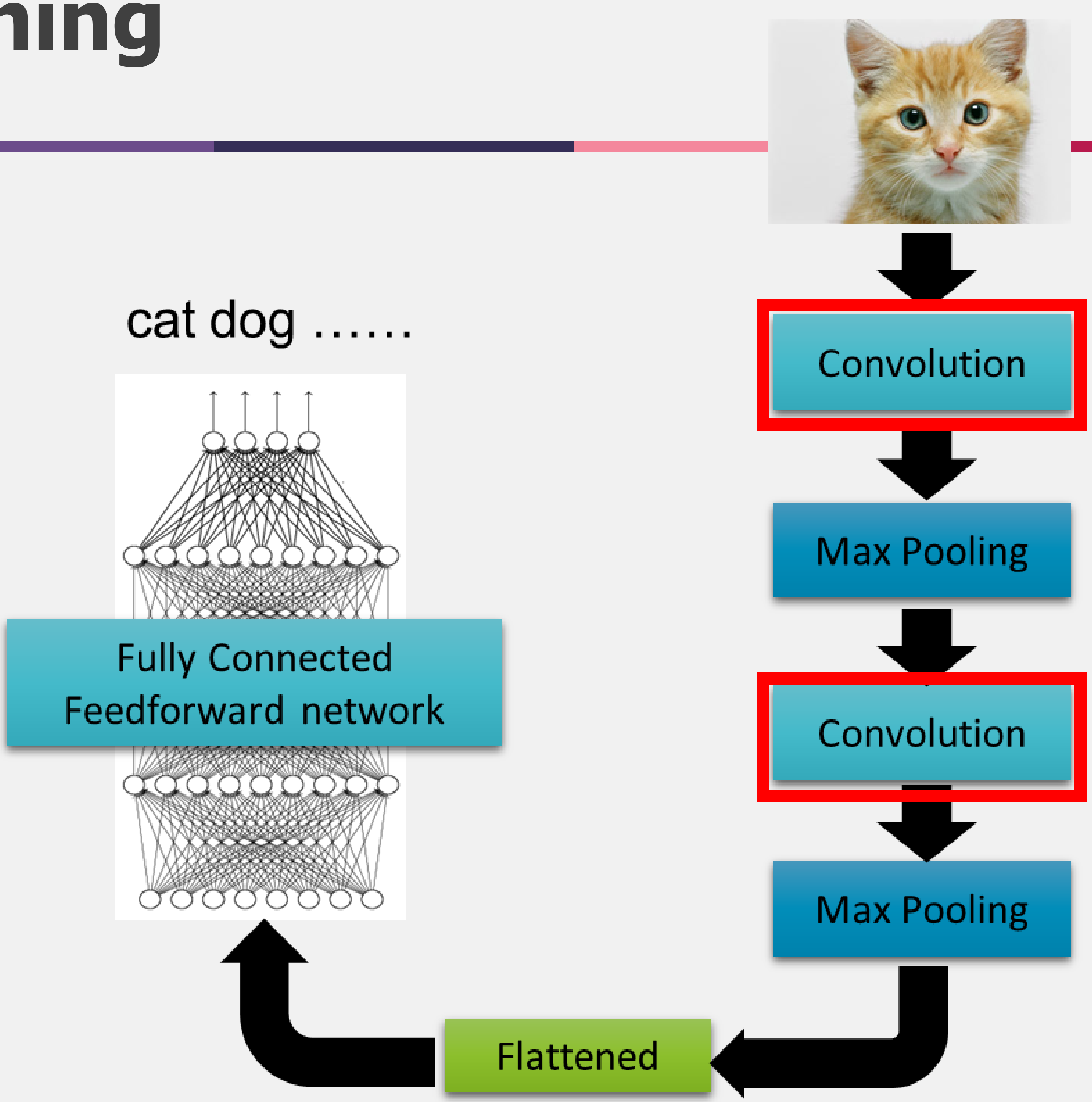
Deep Learning

CNN Concept



Deep Learning

The Whole of CNN



cat dog ……

Fully Connected Feedforward network

Convolution

Max Pooling

Convolution

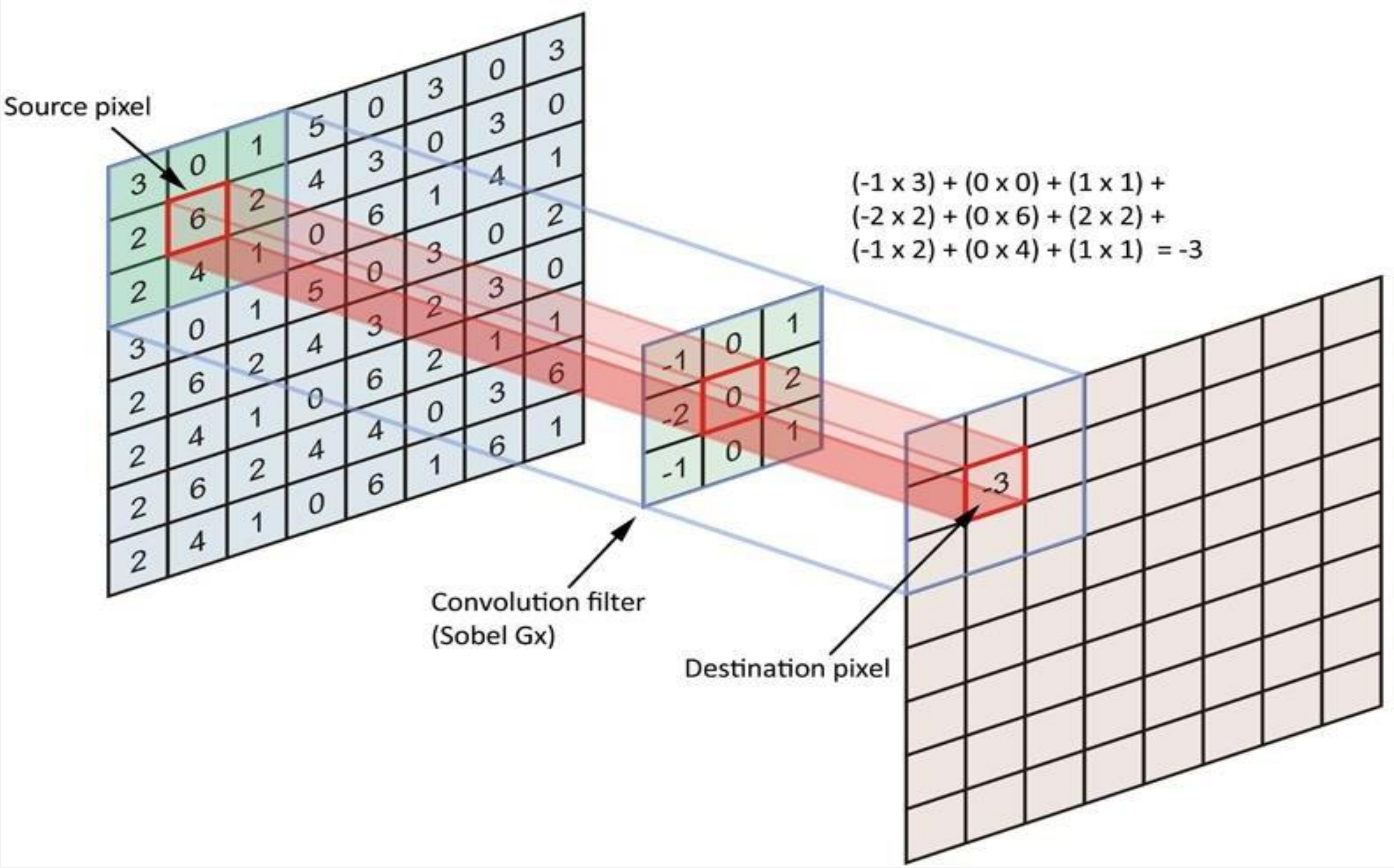
Max Pooling

Flattened

Deep Learning

Convolution

Extract some features on specific local area

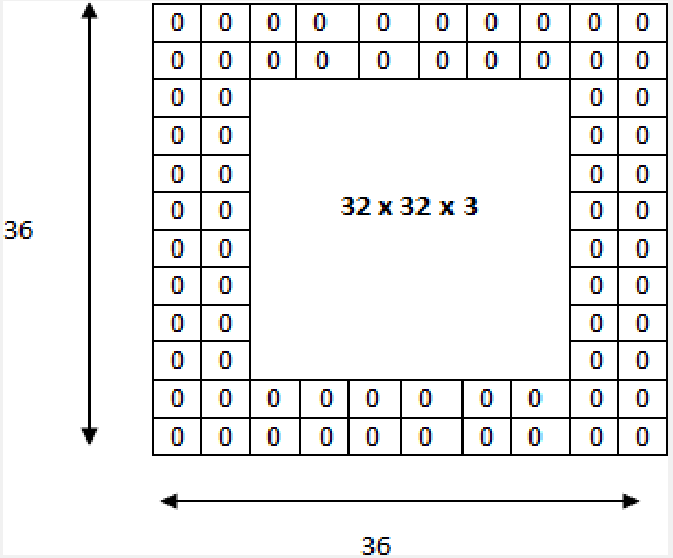


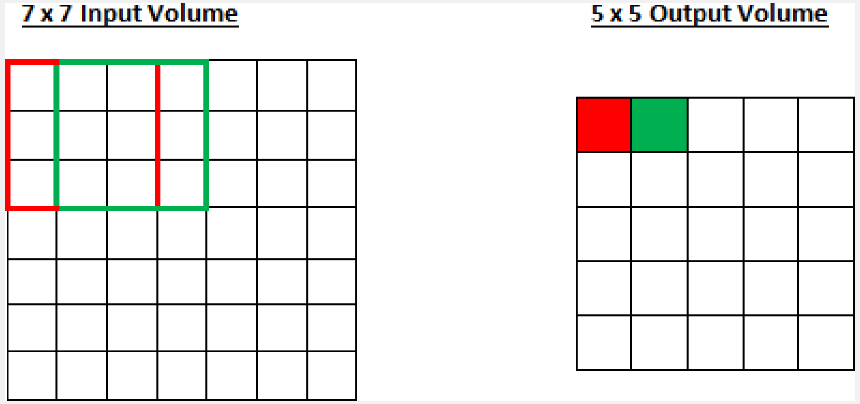
Deep Learning

Convolution

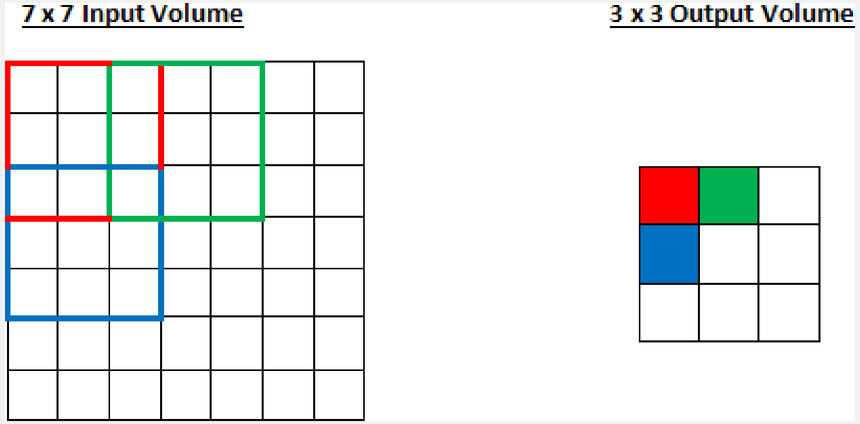
Convolution parameters

- ➢ Kernel size (Filter)
- ➢ Stride
  - Amount of filter shift
- ➢ Padding
  - Padding zero on image boundary
  - Remain the same size after convolution





Size = 3*3 stride = 1



Size = 3*3 stride = 2

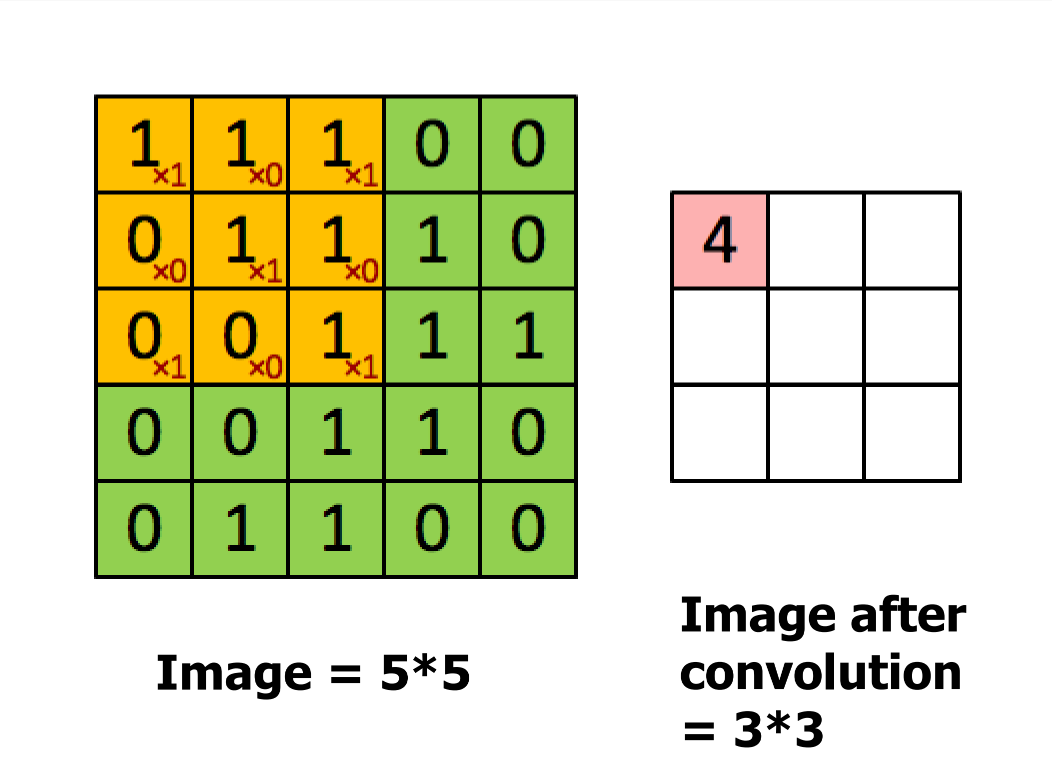
Deep Learning

Convolution

1	0	1
0	1	0
1	0	1

Kernel and stride

Kernel = 3*3 , Padding = No, **Stride = 1**



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

Image = 5*5

4		

Image after convolution = 3*3

Kernel = 3*3 , Padding = No, **Stride = 2**

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	0	1	1	0

Image = 5*5

4	4
2	4

Image after convolution = 2*2

Image = W*W, Kernel(Filter = F*F), Stride = S
new_height = new_width = **(W – F +1) /S** (if 1.5 = 2)

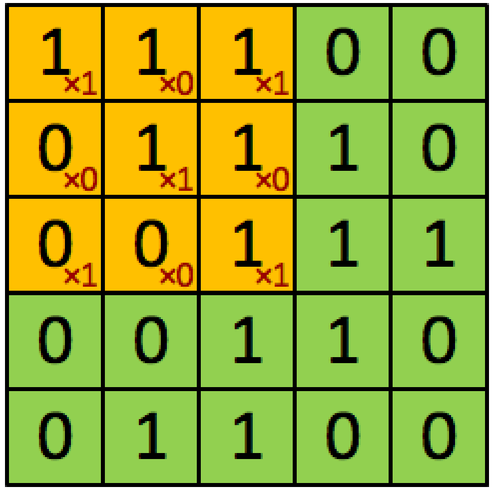
Reference

Deep Learning

Convolution

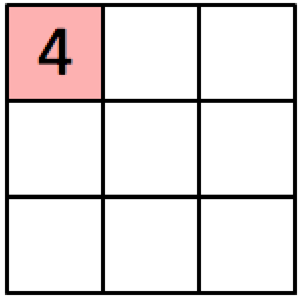
Kernel and padding

Kernel = 3*3 , Padding = No, **Stride = 1**



$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

Image = 5*5



4		

Image after convolution = 3*3

Kernel = 3*3 , **Padding = Yes**, Stride = 1

0	**0**	**0**	**0**	**0**	**0**	**0**
0	1	1	1	0	0	**0**
0	0	1	1	1	0	**0**
0	0	0	1	1	1	**0**
0	0	0	1	1	0	**0**
0	0	0	1	1	0	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**

Image = (5+2)*(5+2)

2	2	2	1	1
1	4	3	4	1
1	2	4	3	3
0	2	3	4	2
0	1	2	2	1

Image after convolution = 5*5

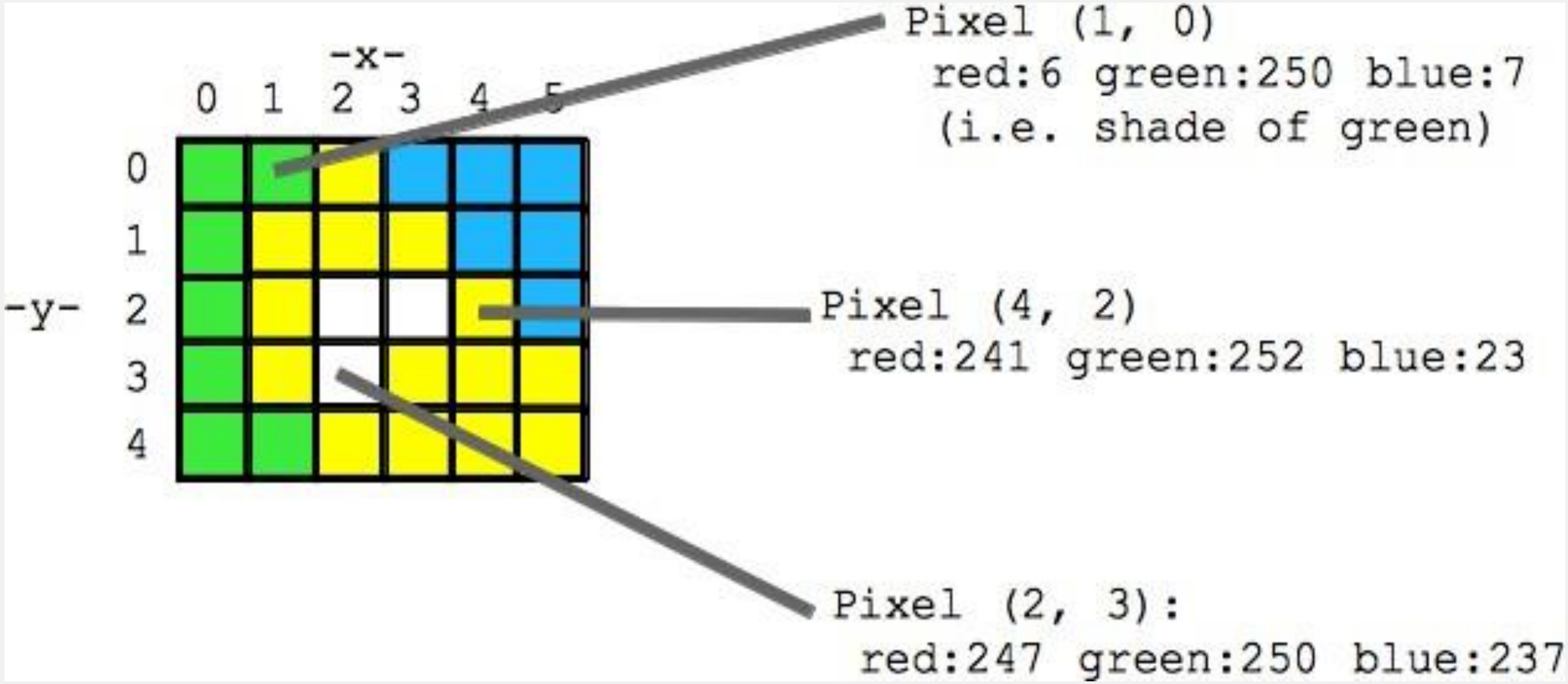
Deep Learning

Pixel in Image

Each image contain many pixels

- Each pixels compose red, green, blue(RGB)

Each channel have brightness levels between 0~255



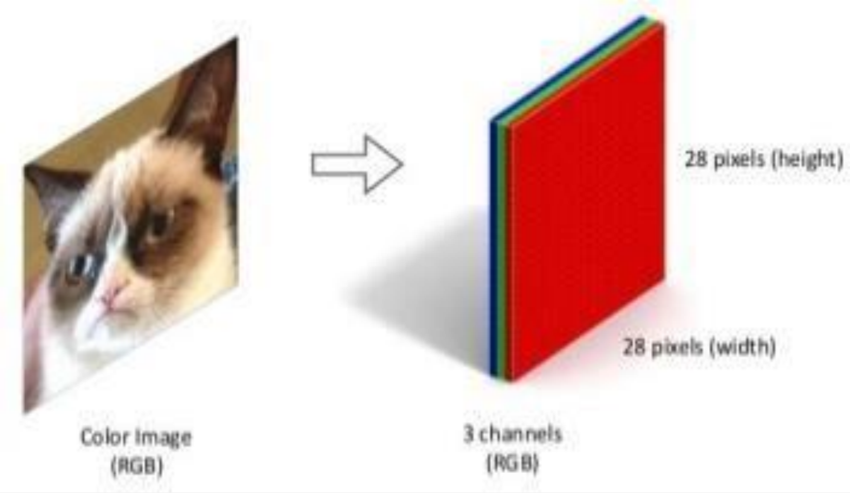
Pixel (1, 0)
red:6 green:250 blue:7
(i.e. shade of green)

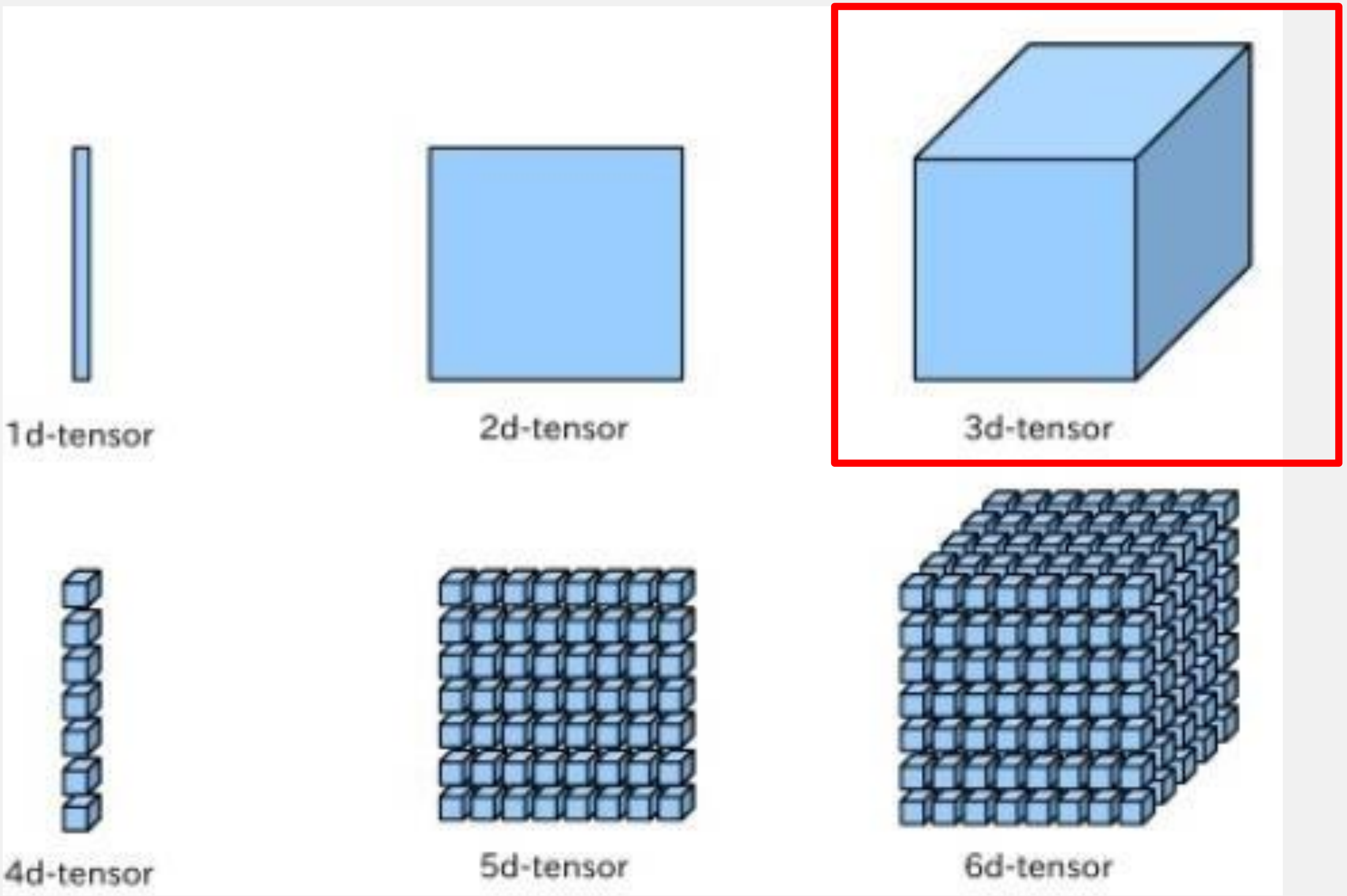
Pixel (4, 2)
red:241 green:252 blue:23

Pixel (2, 3):
red:247 green:250 blue:237

Deep Learning

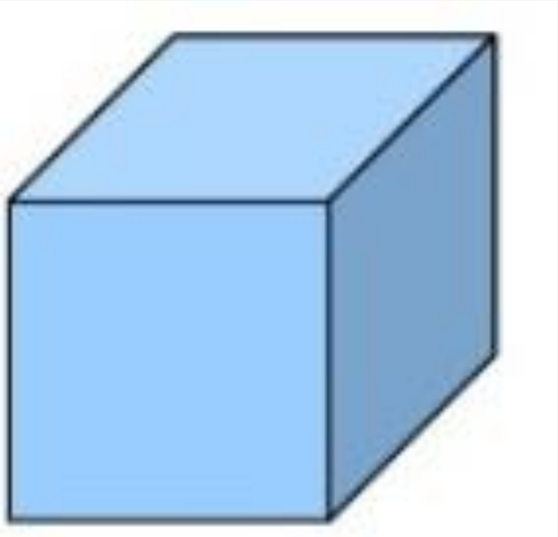
Pixel in Image





Deep Learning

Pixel in Image

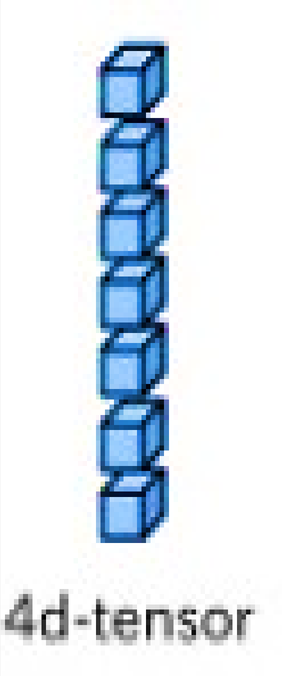


= [image width, image height, image channel(feature map)]

A image is a 3D-tensor

Deep Learning

Pixel in Image



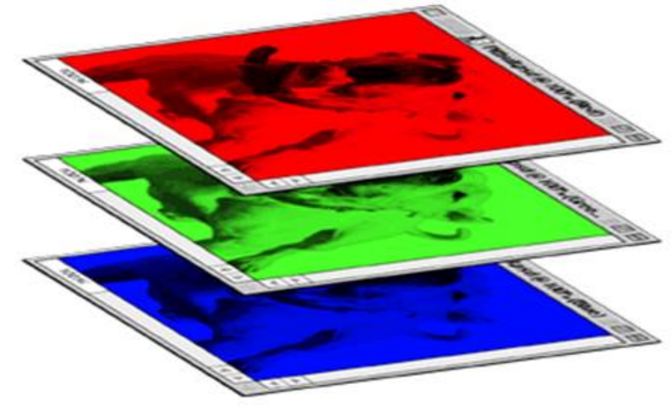
= **[batch size, image width, image height, image channel(feature map)]**

A Batch of images is a 4D-tensor

Deep Learning

Pixel in Image

Color image



1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

1	-1	-1
-1	1	-1
-1	-1	1

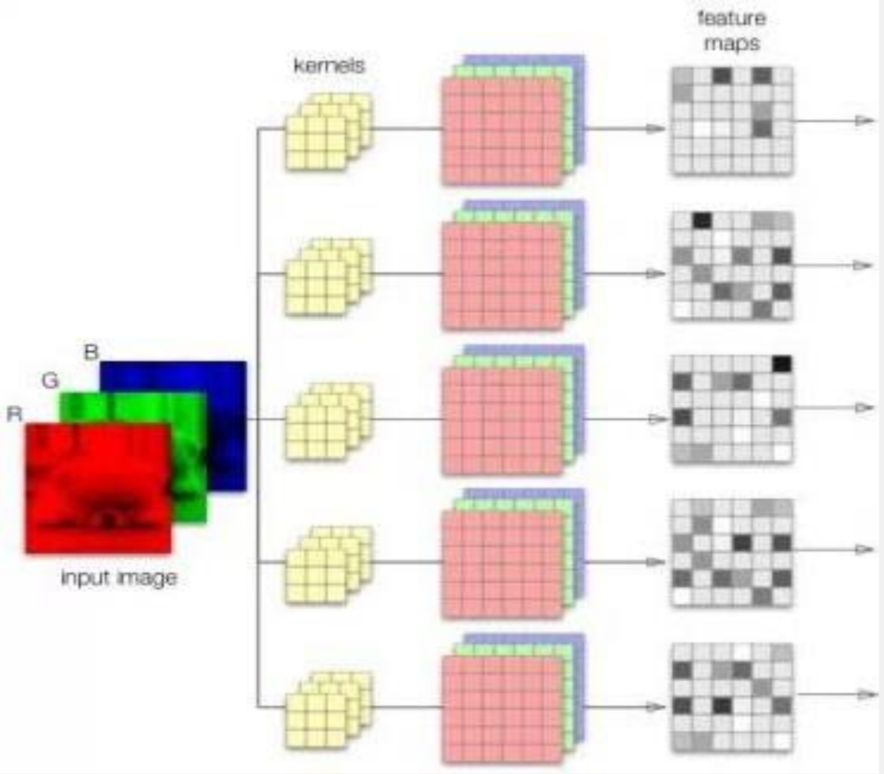
Filter 1

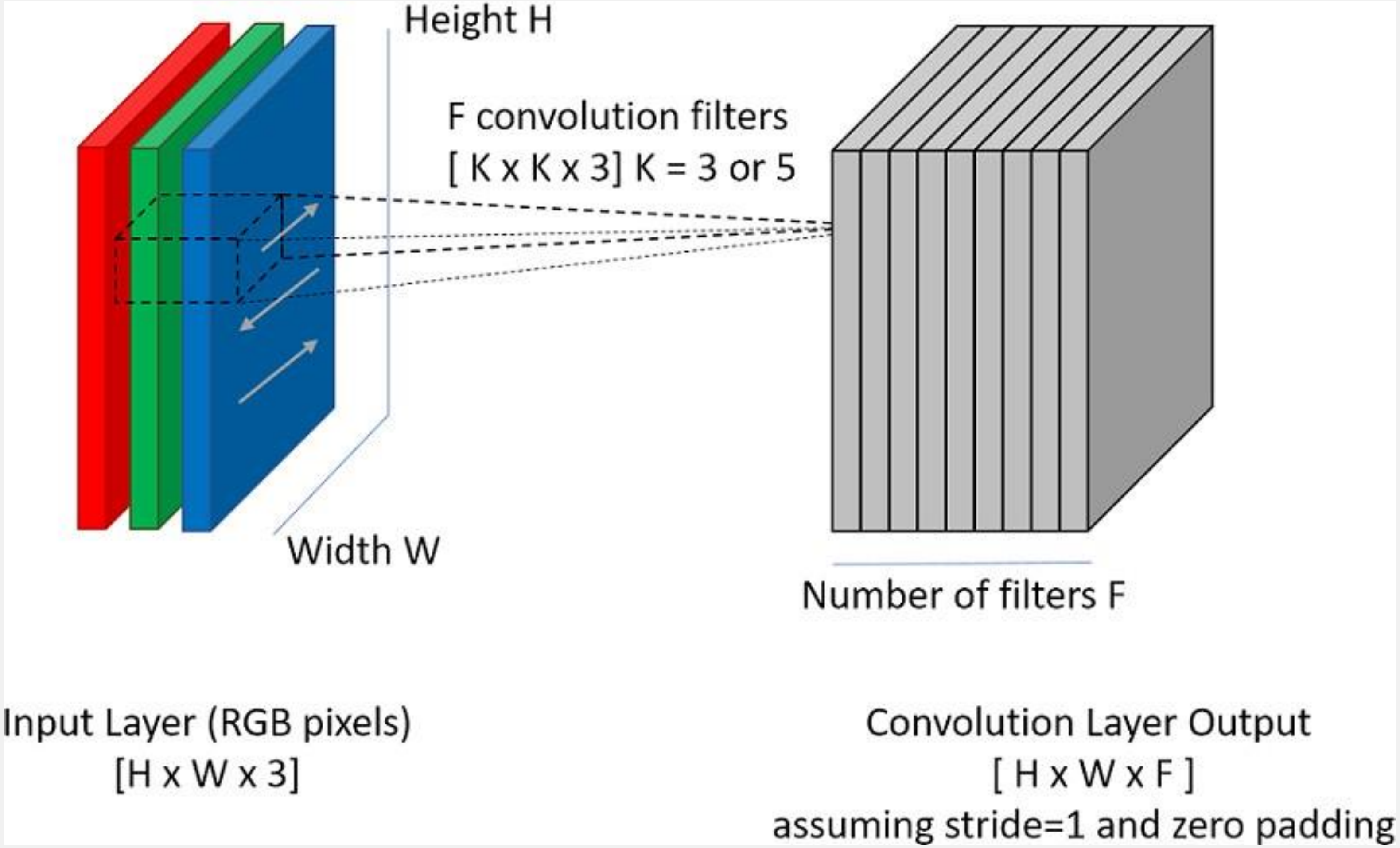
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

Deep Learning

Pixel in Image

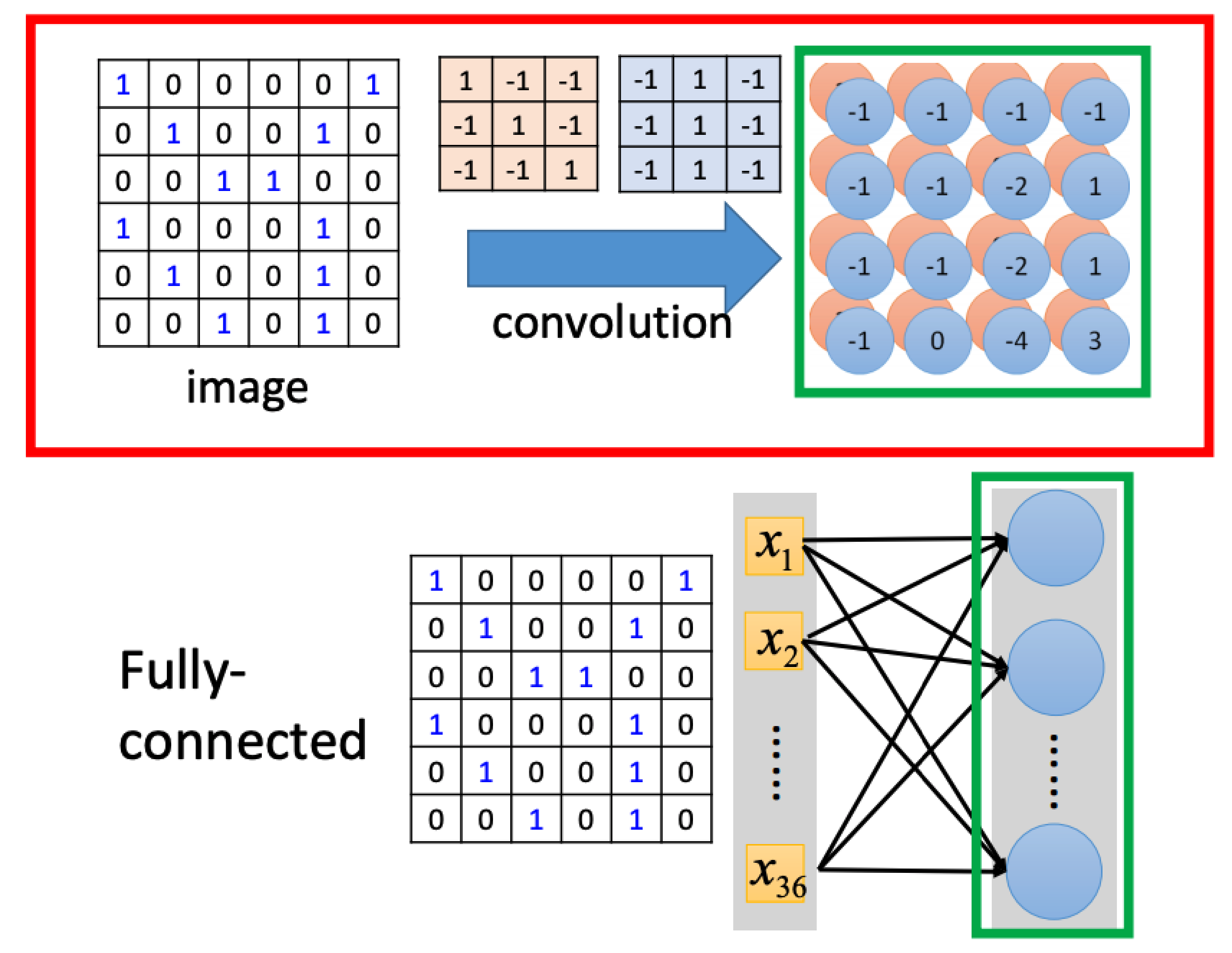




Deep Learning

Pixel in Image

Convolution vs Fully connected (# of parameters)



1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

1	-1	-1
-1	1	-1
-1	-1	1

-1	1	-1
-1	1	-1
-1	1	-1

-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

filter(neuron) count

$(\underline{3 \times 3 + 1}) \times 3 = 30$

filter
w x h

bias

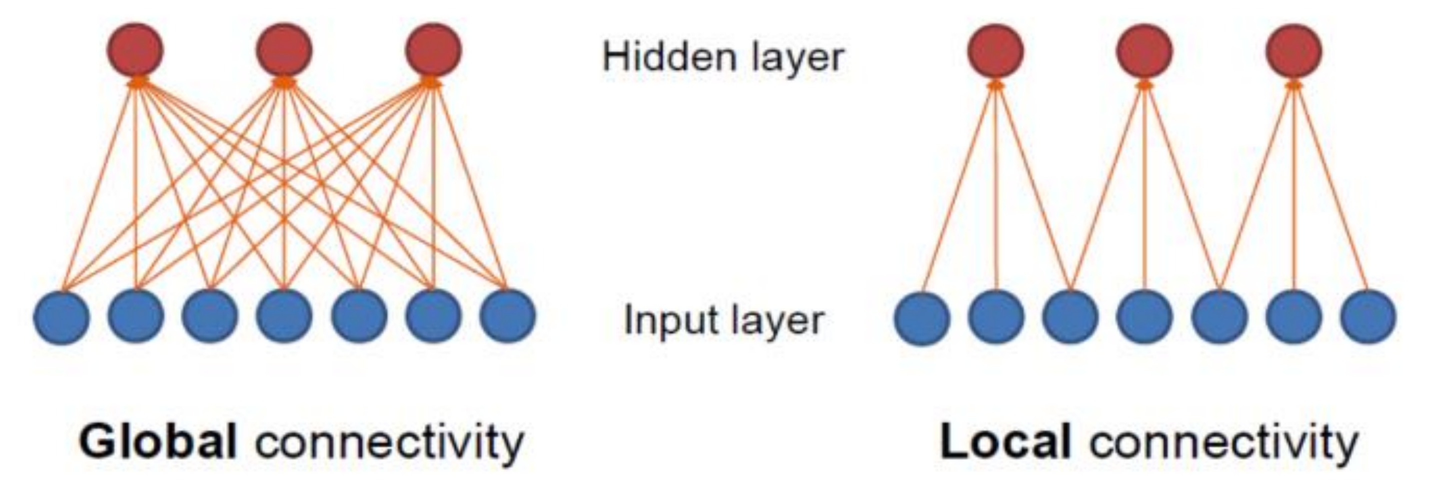
$(\underline{36 + 1}) \times 3 = 111$

previous pixel count + bias

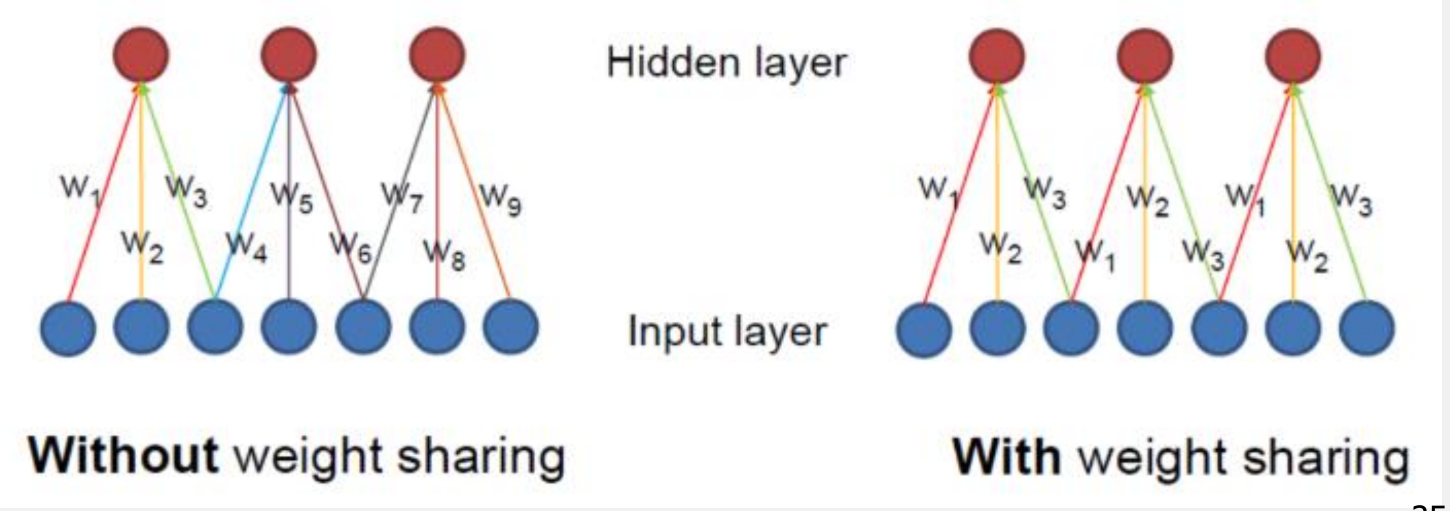
Deep Learning

Connection and Weight

Local connectivity

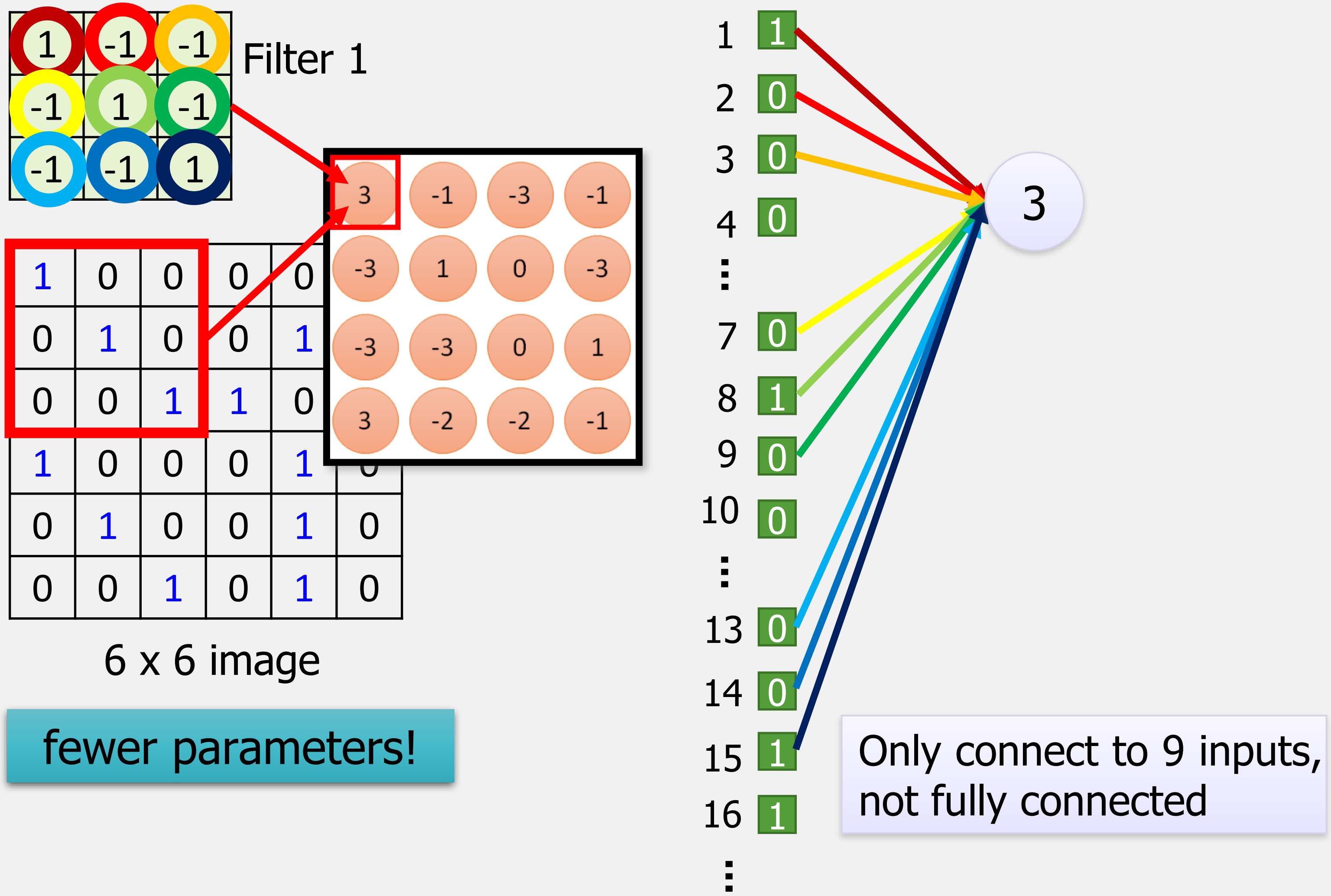


Weight sharing



Deep Learning

Convolution Processing



Filter 1

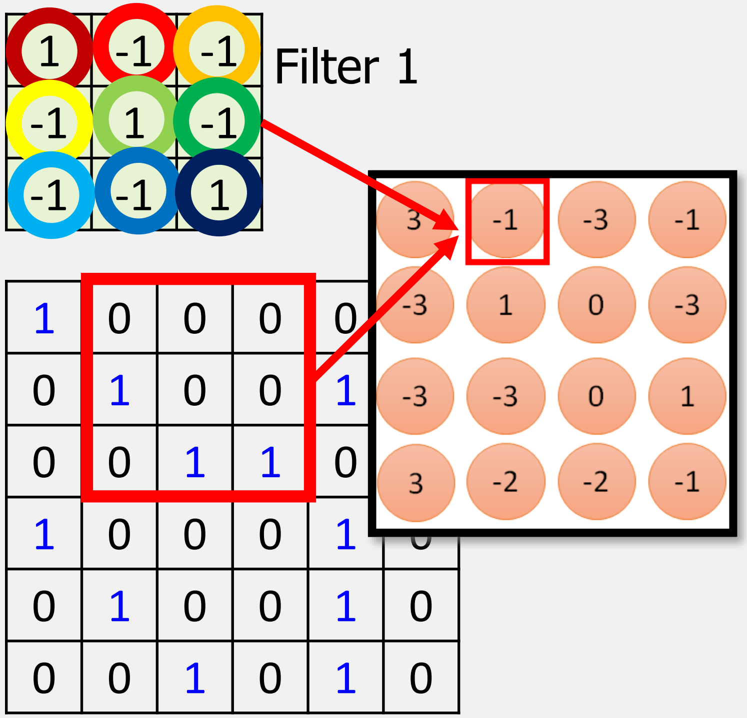
6 x 6 image

fewer parameters!

Only connect to 9 inputs, not fully connected

Deep Learning

Convolution Processing



Filter 1

1	-1	-1
-1	1	-1
-1	-1	1

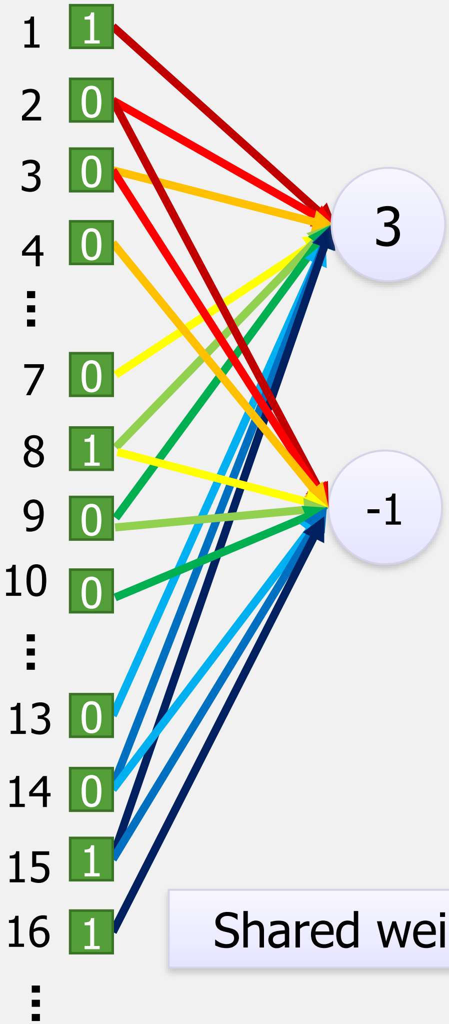
3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

1	0	0	0	0	
0	1	0	0	1	
0	0	1	1	0	
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

fewer parameters!

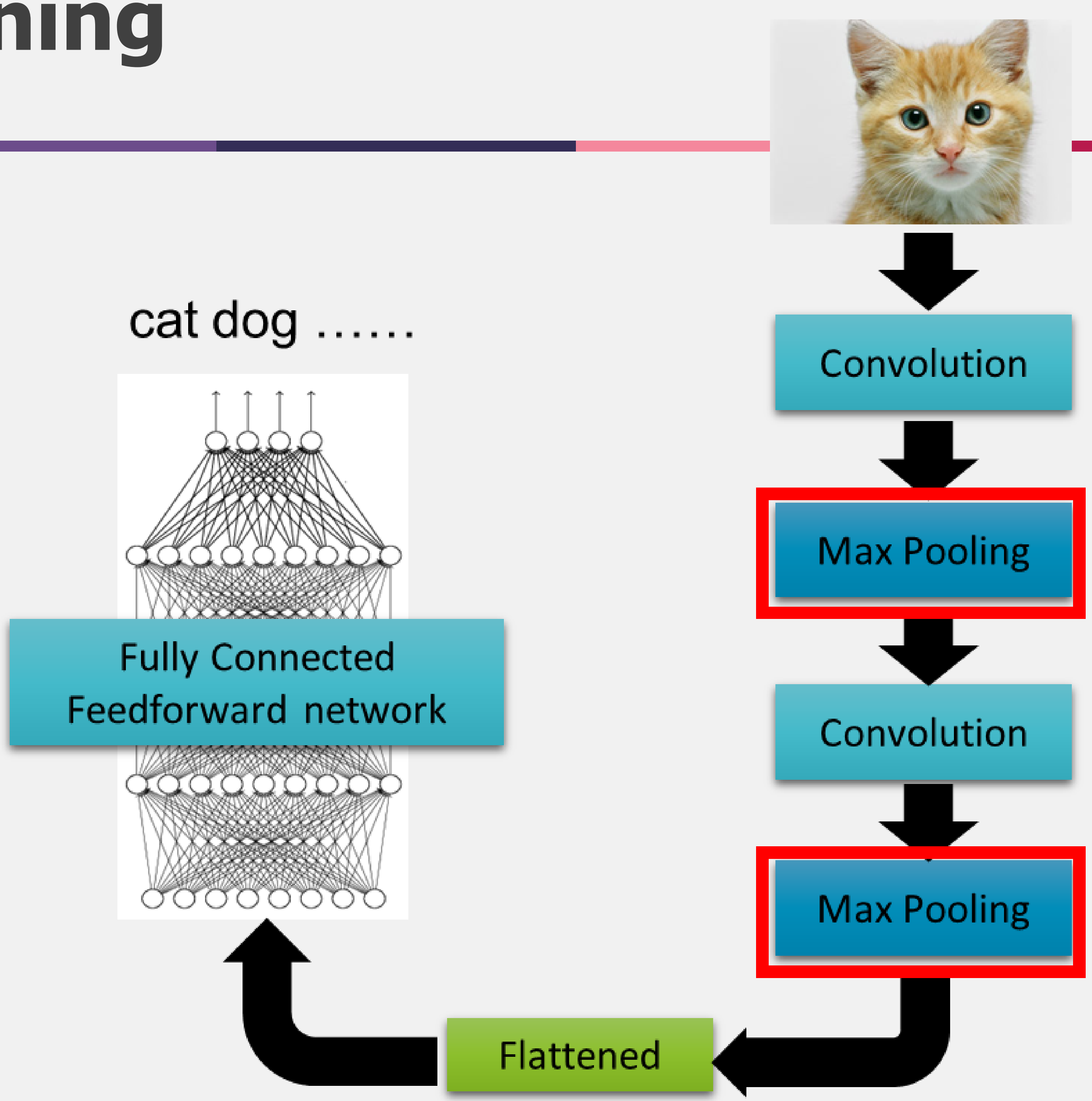
Even fewer parameters!



Shared weights

Deep Learning

The Whole of CNN



cat dog ……

Convolution

Max Pooling

Convolution

Max Pooling

Flattened

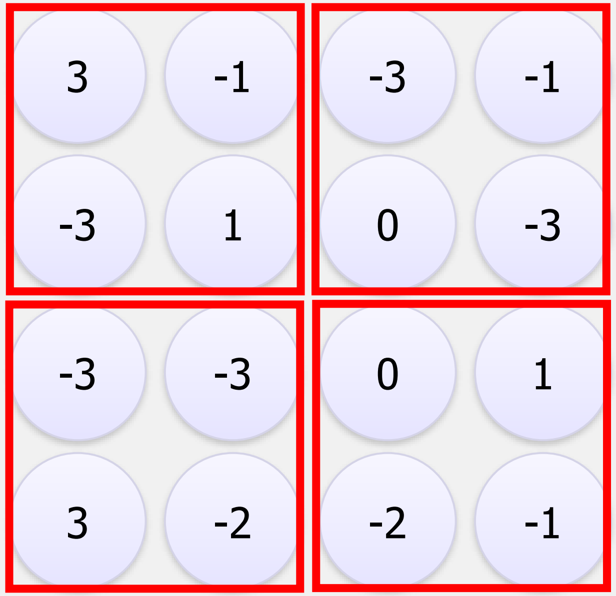
Fully Connected Feedforward network

Deep Learning

CNN - Max Pooling

1	-1	-1
-1	1	-1
-1	-1	1

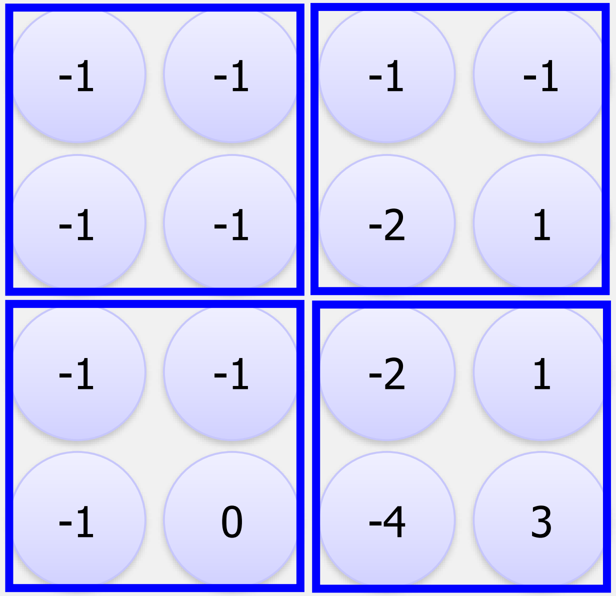
Filter 1



3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

-1	1	-1
-1	1	-1
-1	1	-1

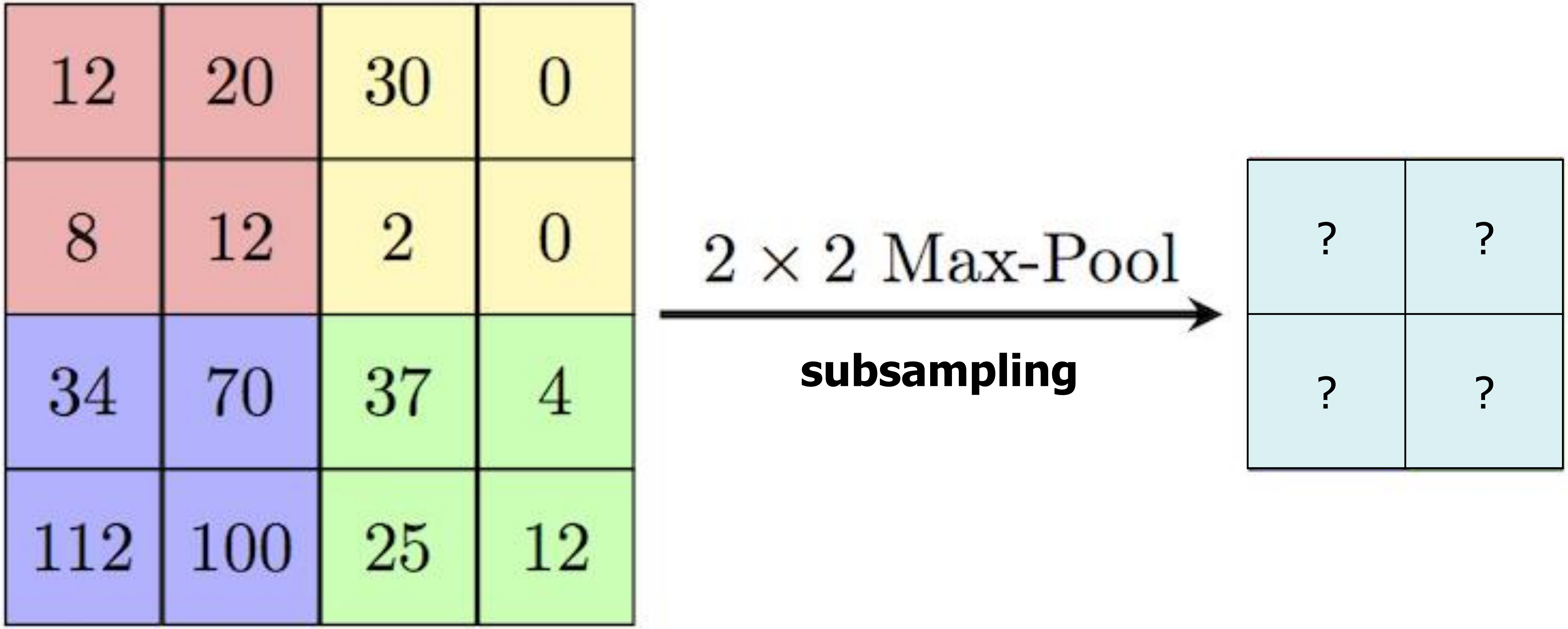
Filter 2



-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

Deep Learning

CNN - Max Pooling



12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

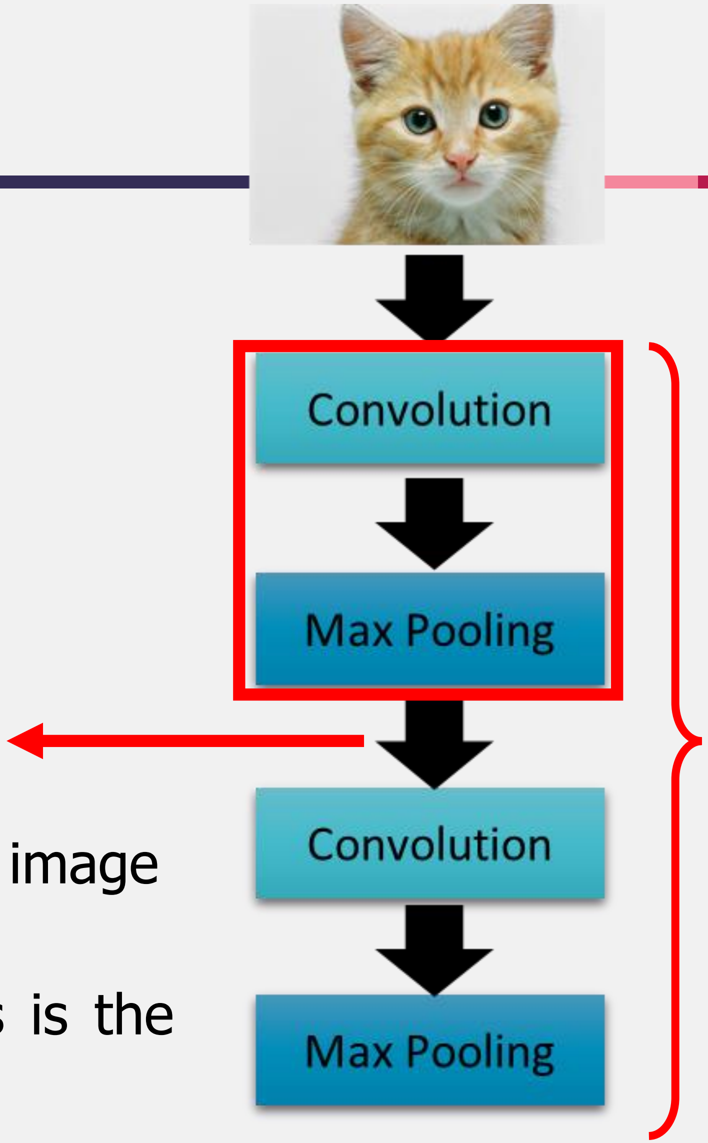
2×2 Max-Pool

subsampling

?	?
?	?

Deep Learning

The Whole of CNN



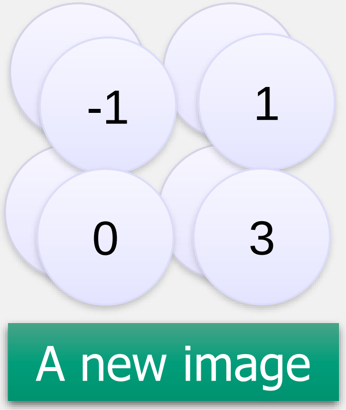
Convolution

Max Pooling

Convolution

Max Pooling

Can repeat many times



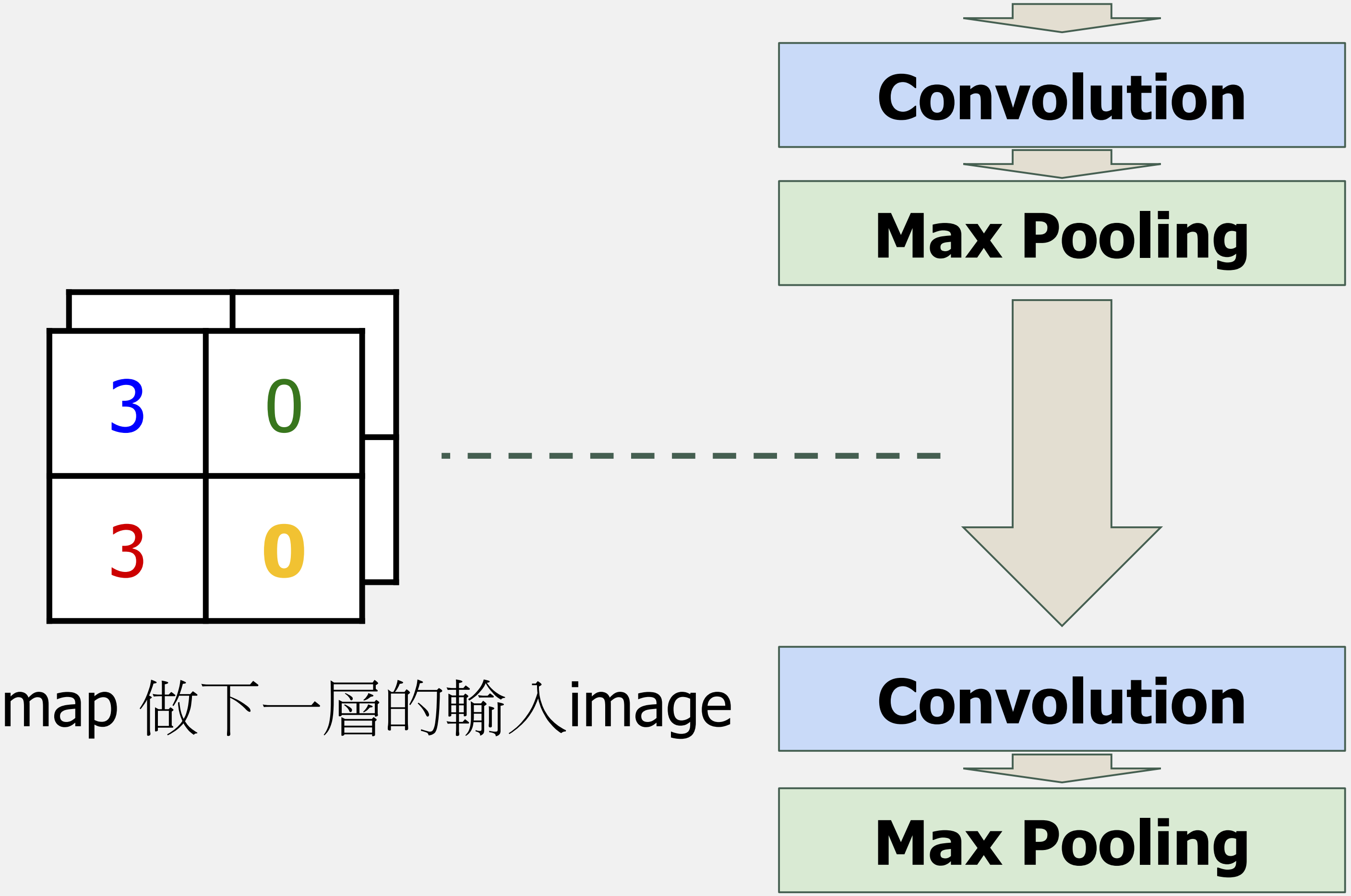
A new image

Smaller than the original image

The number of channels is the number of filters

Deep Learning

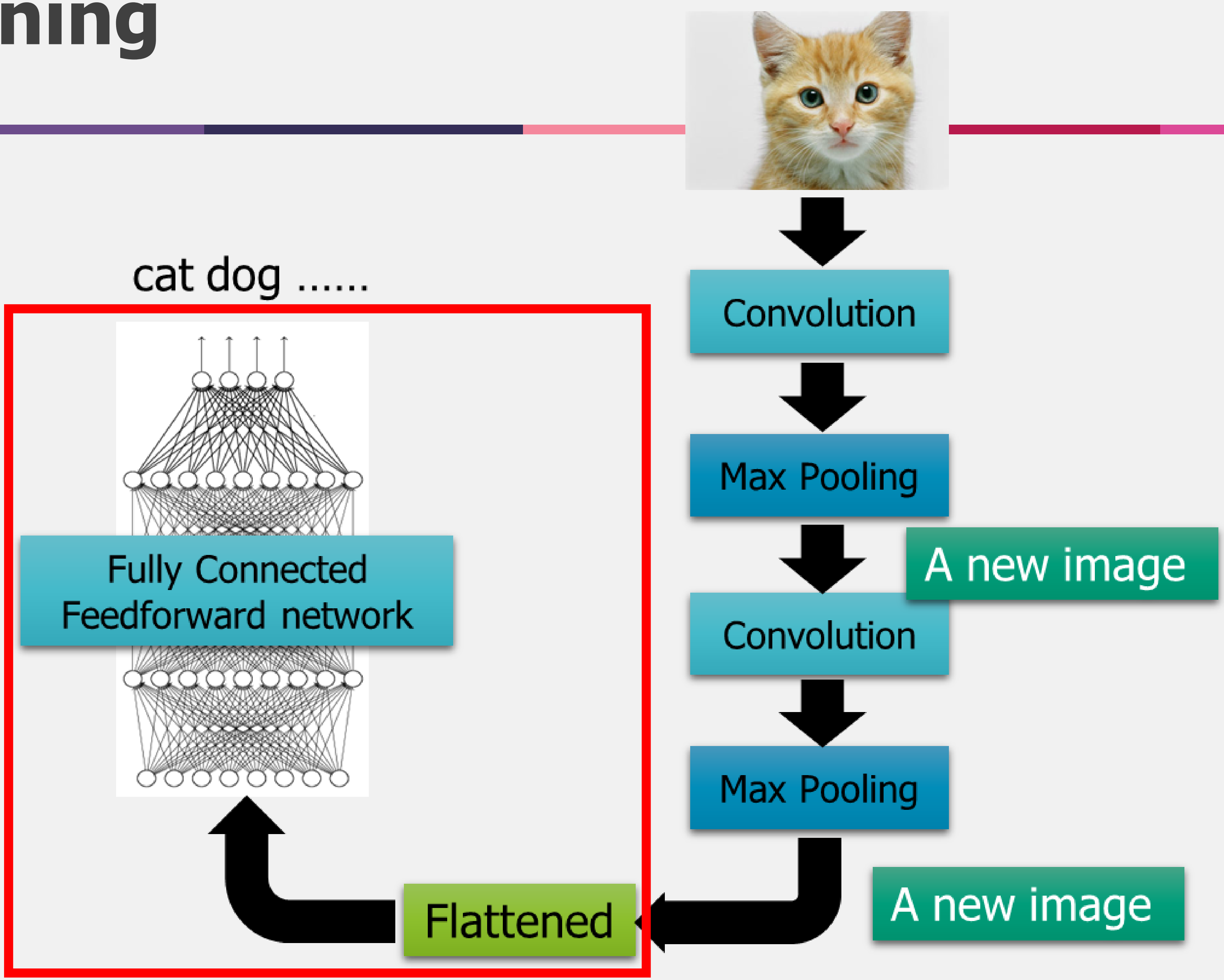
After Convolution + Max-pooling



feature map 做下一層的輸入image

Deep Learning

The Whole of CNN



cat dog ……

Fully Connected Feedforward network

Flattened

Convolution

Max Pooling

A new image

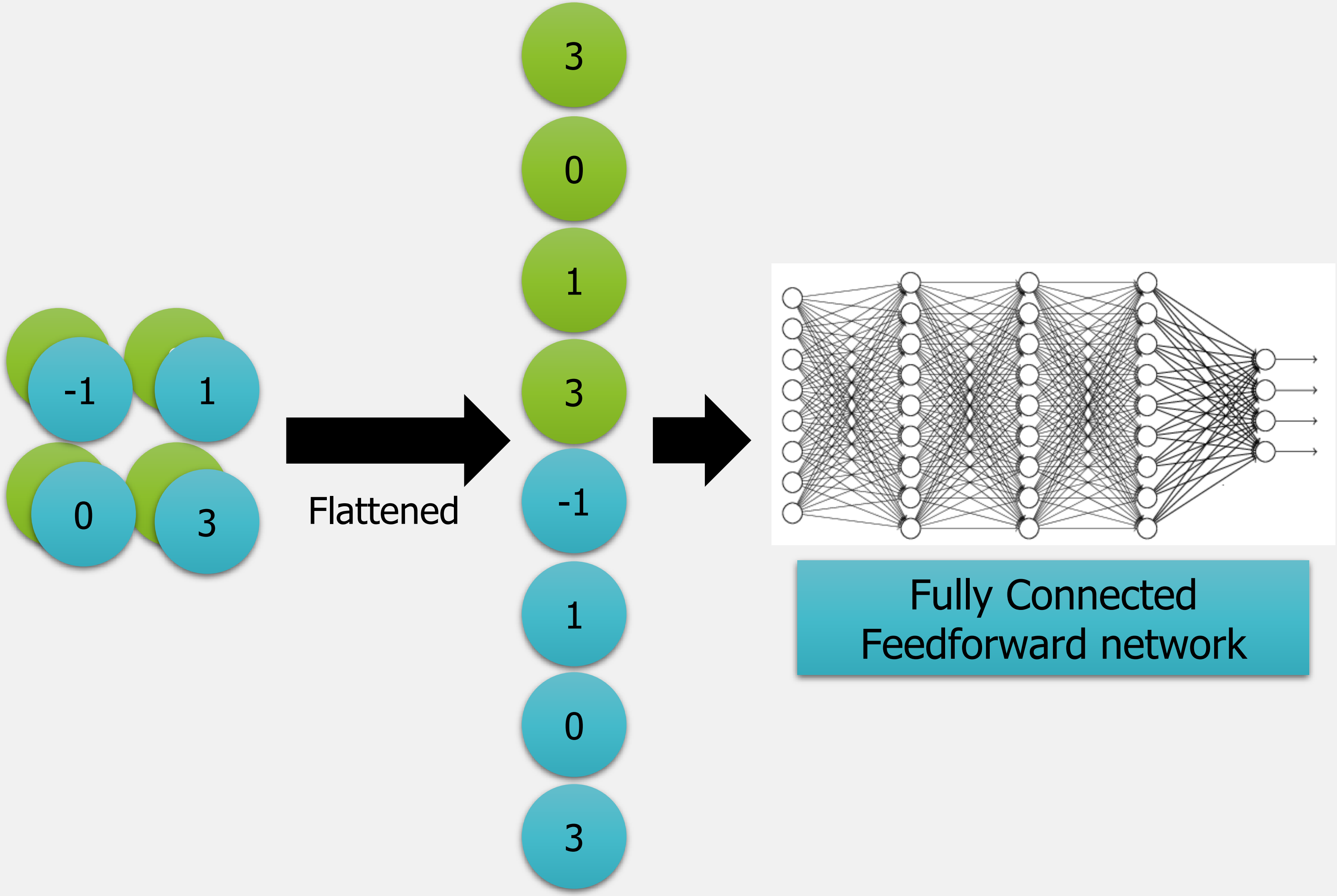
Convolution

Max Pooling

A new image

Deep Learning

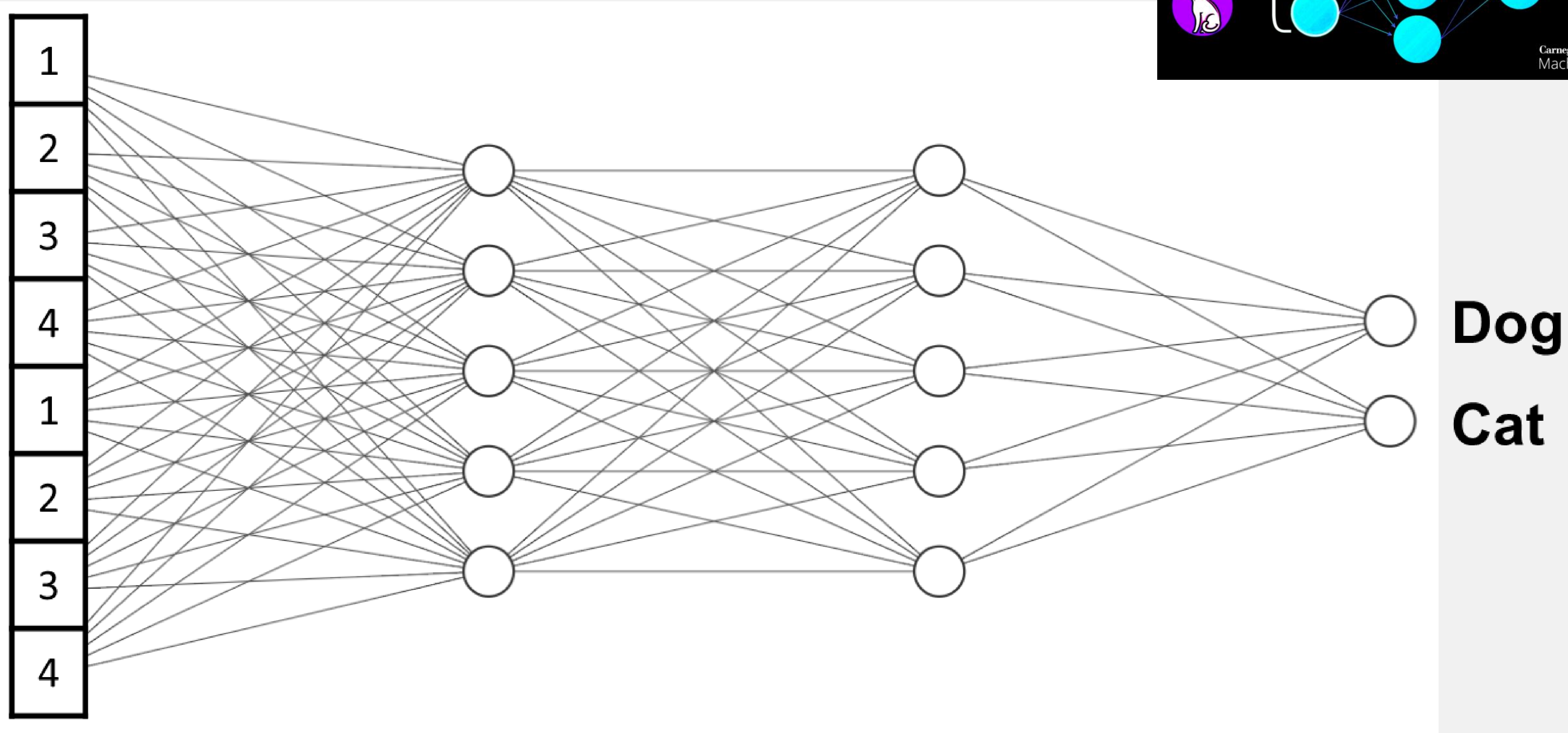
Flattened

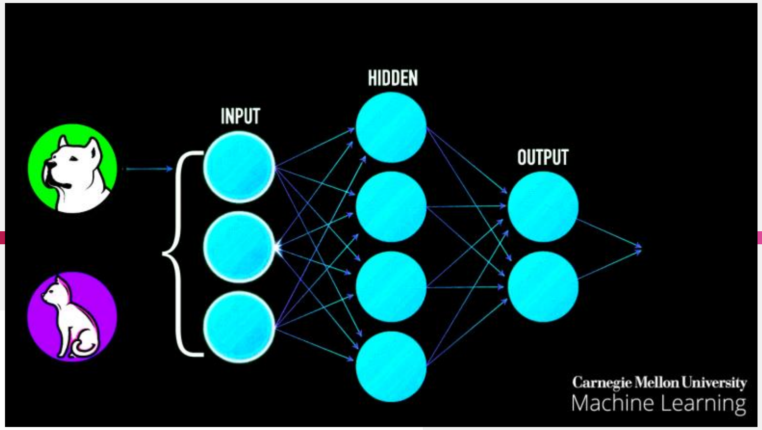


Fully Connected Feedforward network

Deep Learning

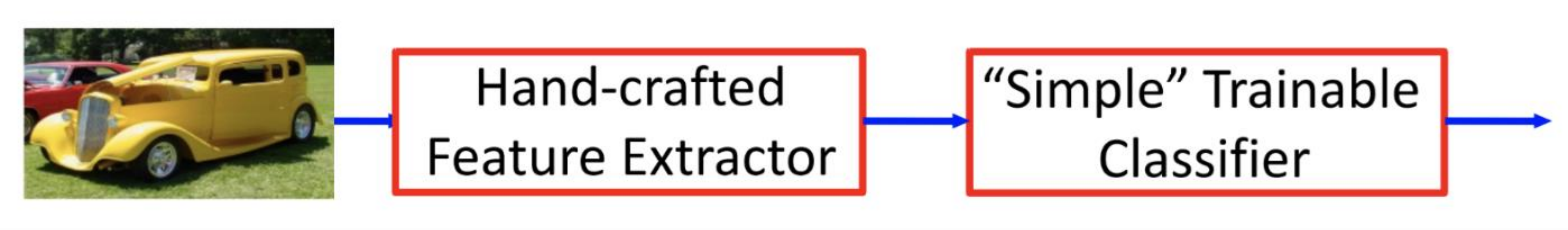
FC

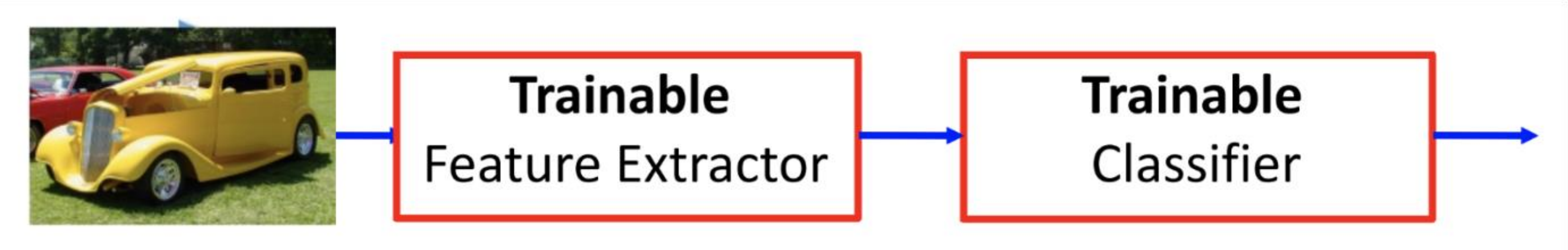




Deep Learning

Traditional vs DL

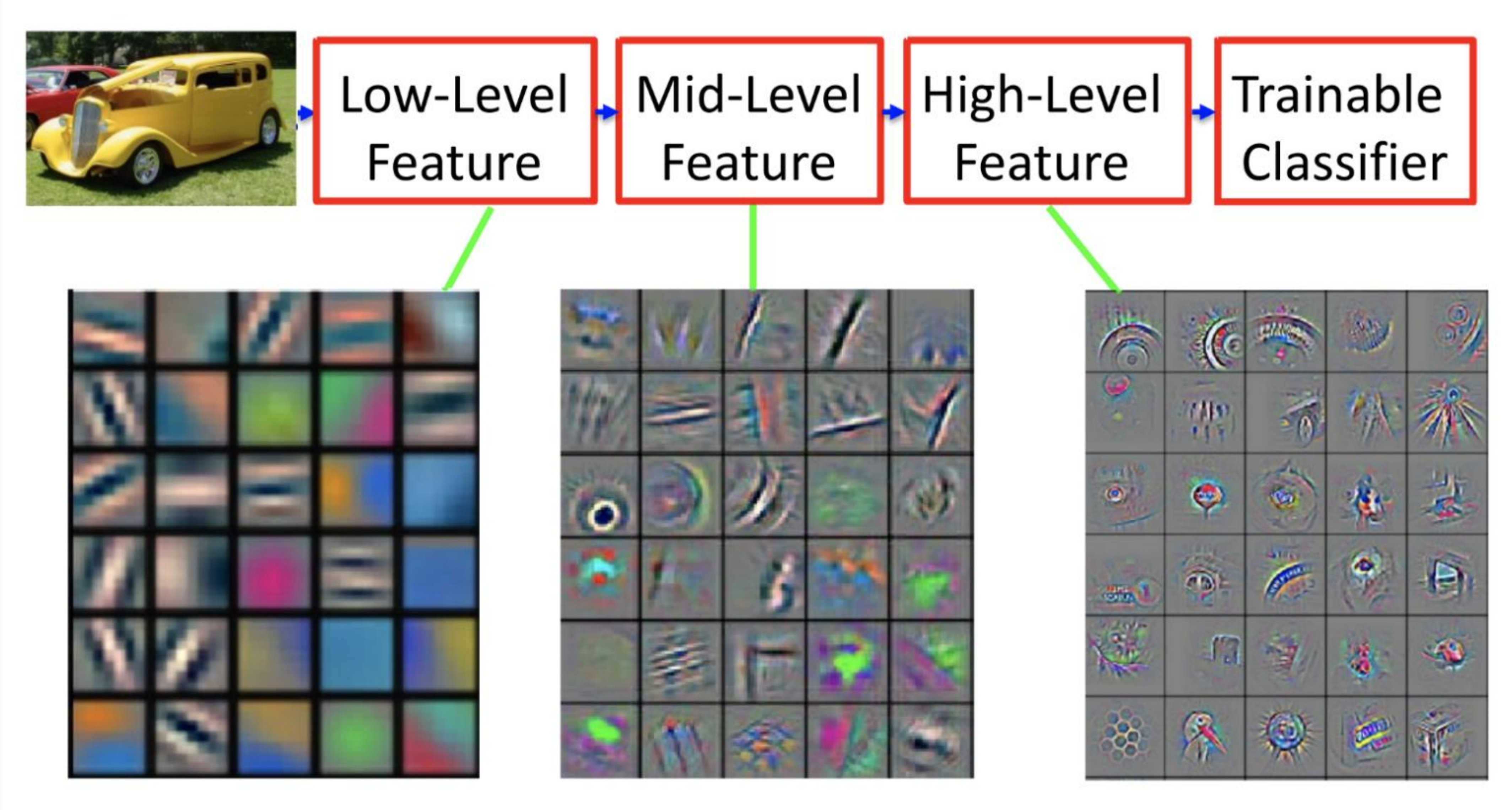




Deep Learning

What CNN Learn?

[Reference](#)



Deep Learning

Yann LeCun - BP on CNN

[Reference](#)

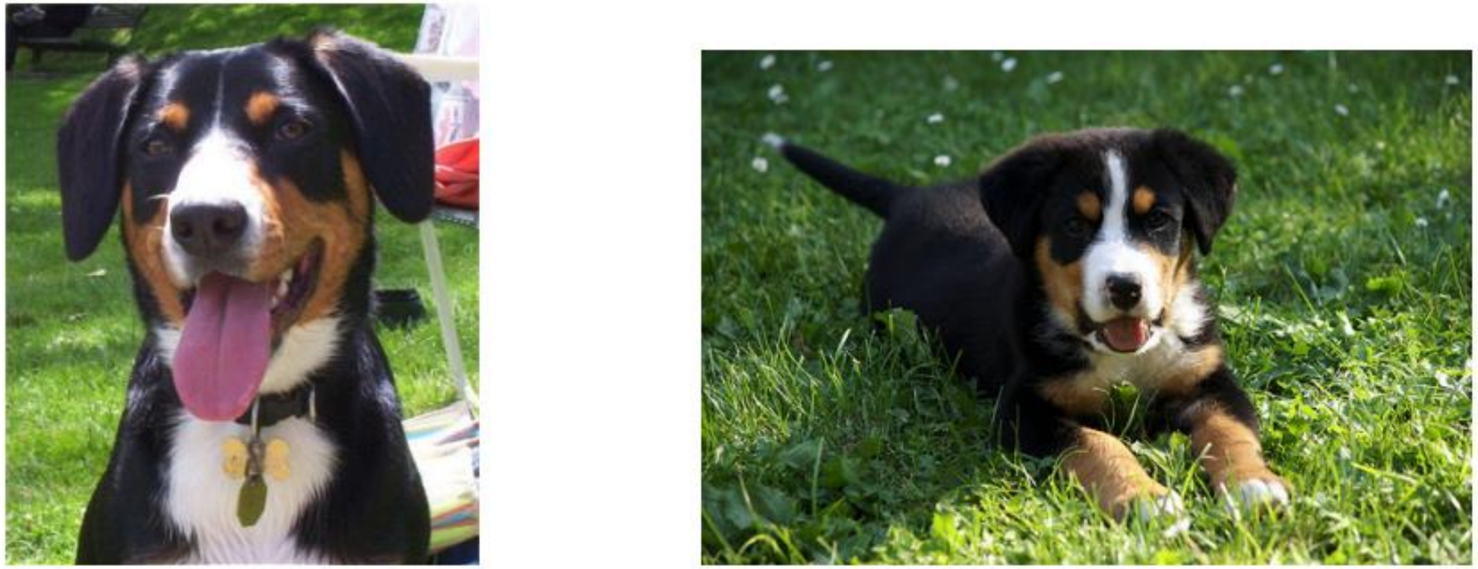
Deep Learning

ImageNet ILSVRC

1000 classes 1.2M images

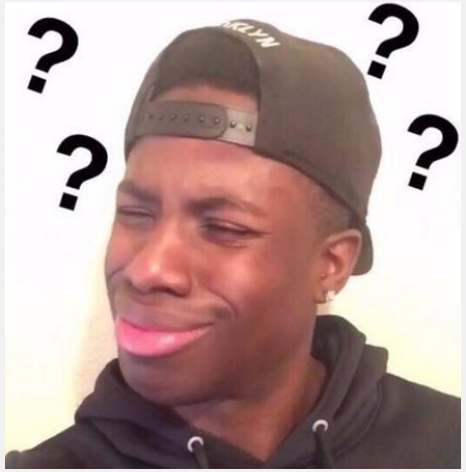
李飛飛等

~2017



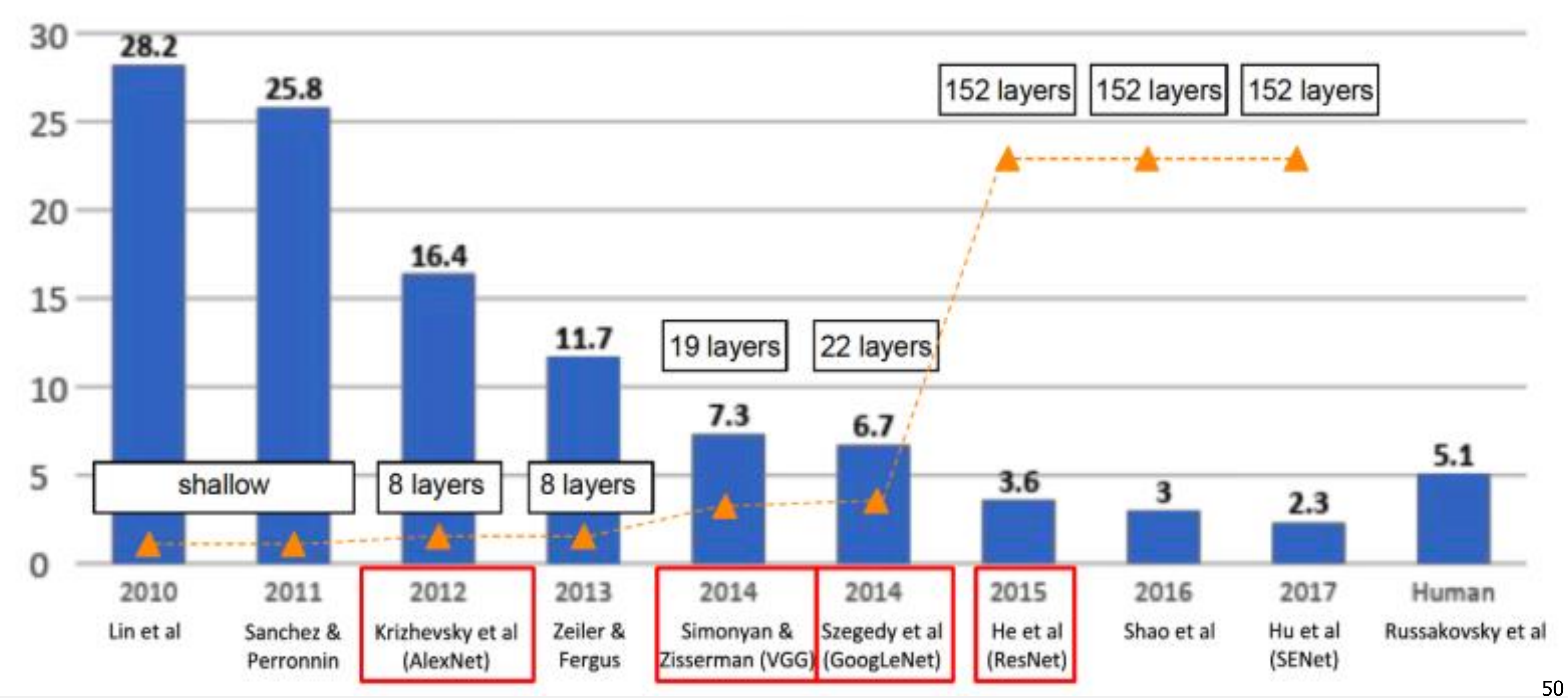
恩特布山犬

阿彭策爾山犬



Deep Learning

ImageNet ILSVRC



03

Forward and Backward

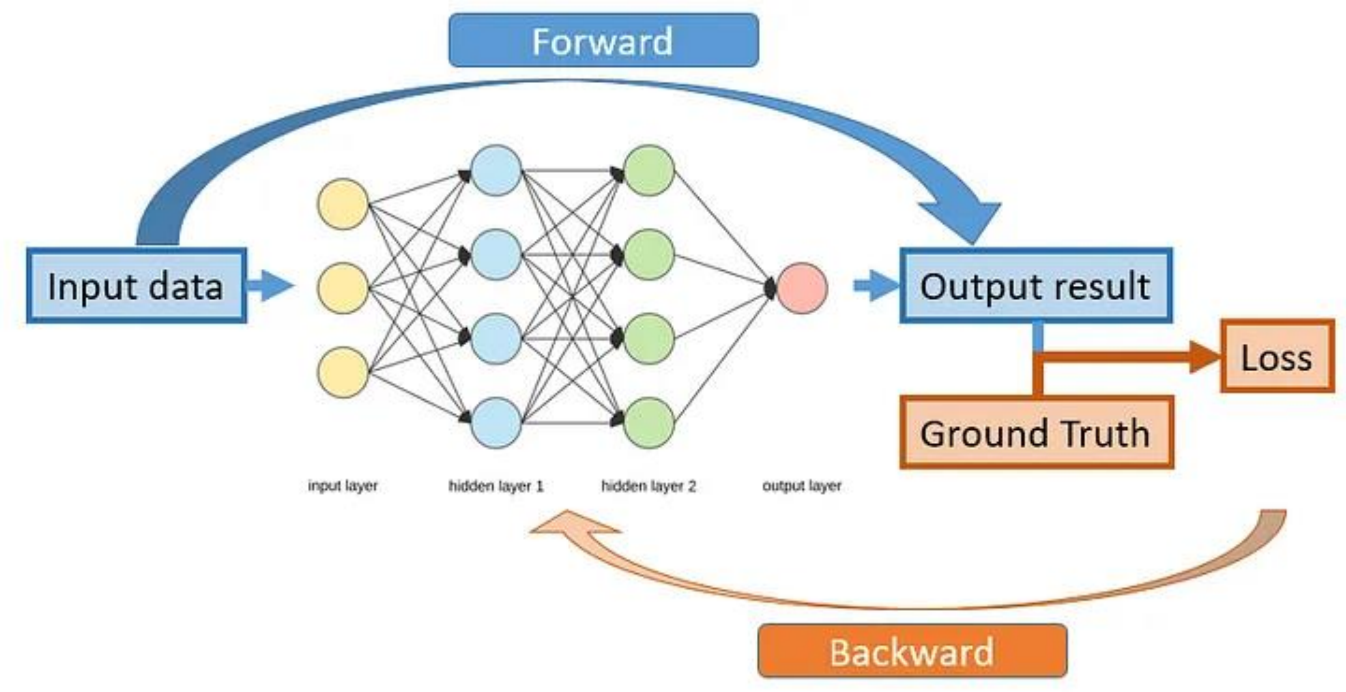
Forward and Backward

Forward

- Input -> Network -> Output
  - Linear operation and non-linear translation

Backward

- Compute loss -> Optimize weight
  - Differential equation



04

Loss Function

Loss Function

Object Function

- Most ML algorithms aim to maximize or minimize a function
  - K-means is to minimize the sum of squared errors between the data points and the centroid within each group
  - PCA is to maximize the variance of the projected data by finding the projection vectors

Loss Function

- The residual between the actual value and the predicted value

$$\text{loss (residual)} = y - \hat{y}$$

- Regression: MSE (Mean Square Error)

$$L = \frac{1}{m}\sum_{i=1}^{m}(y - \hat{y})^2$$

- Classification: CE (Cross-Entropy)

$$L = -\sum_{i=1}^{m} y_i \, (log\hat{y}_i)$$

05

Gradient Descent

Gradient Descent

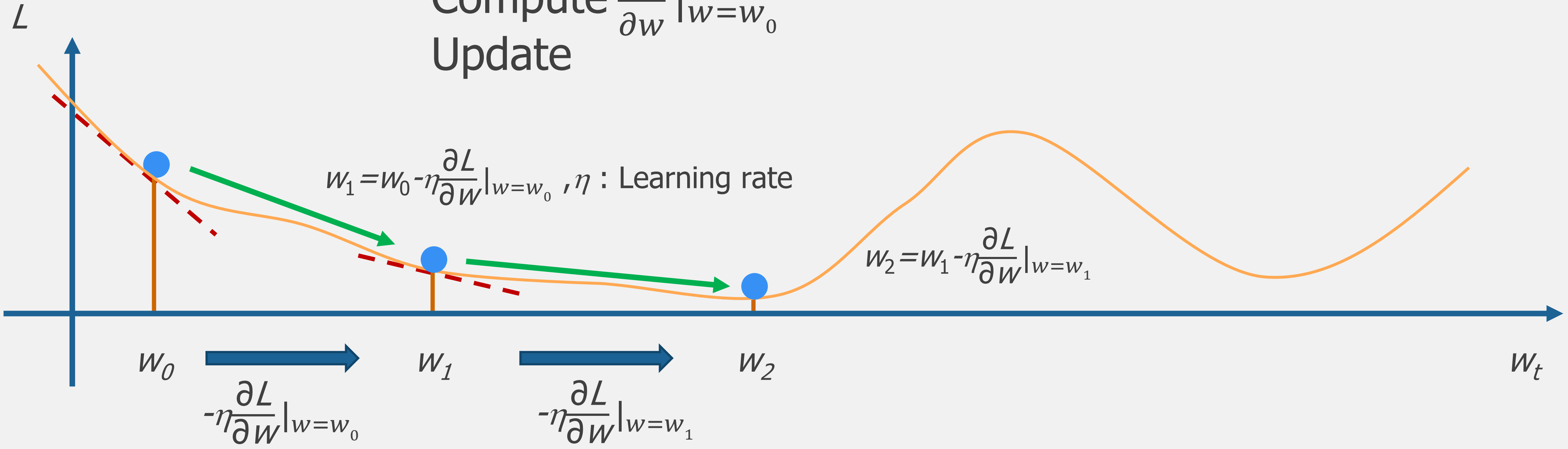
Gradient descent

Considering $L(w)$ only has a parameter w $\quad w^* = \arg\min_{w} L(w)$

Initializing w_0

Compute $\frac{\partial L}{\partial w}|_{w=w_0}$

Update



$w_1 = w_0 - \eta\frac{\partial L}{\partial w}|_{w=w_0}$, η : Learning rate

$w_2 = w_1 - \eta\frac{\partial L}{\partial w}|_{w=w_1}$

Gradient Descent

If loss function has two parameters, $L(w, b)$ $w^*, b^* = arg\min_{w,b} L(w, b)$

Random initialization for w_0 , b_0

Calculate $\frac{\partial L}{\partial w}|_{w=w_0,b=b_0}, \frac{\partial L}{\partial b}|_{w=w_0,b=b_0}$

$w_1=w_0-\eta\frac{\partial L}{\partial w}|_{w=w_0,b=b_0}$ $b_1=b_0-\eta\frac{\partial L}{\partial b}|_{w=w_0,b=b_0}$

$w_2=w_1-\eta\frac{\partial L}{\partial w}|_{w=w_1,b=b_1}$ $b_2=b_1-\eta\frac{\partial L}{\partial b}|_{w=w_1,b=b_1}$

⋮

Gradient Descent

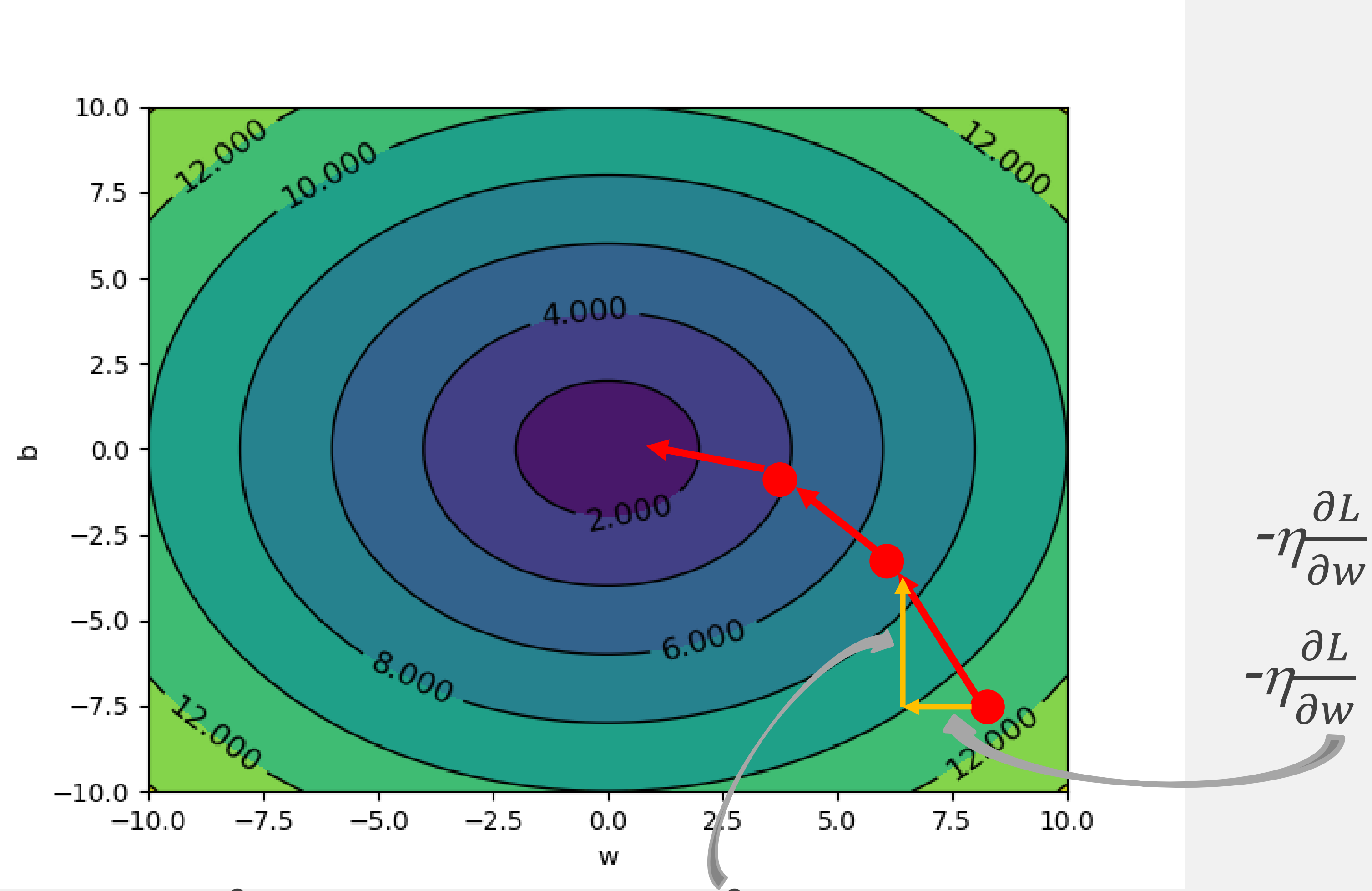
$y' = b + wx$ MSE (Mean Square Error) $L = \frac{1}{m}\sum_{i=1}^{m}(y - y')^2$

$$L = \frac{1}{m}\sum_{i=1}^{m}(y - y')^2 = \frac{1}{m}\sum_{i=1}^{m}(y - (b + wx))^2$$

for w ➡ $\frac{\partial L}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} 2 * \left(y - (b + wx)\right) * (-x)$

for b ➡ $\frac{\partial L}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} 2 * \left(y - (b + wx)\right) * (-1)$

Gradient Descent



$-\eta\frac{\partial L}{\partial w}|_{w=w_1,b=b_1}$

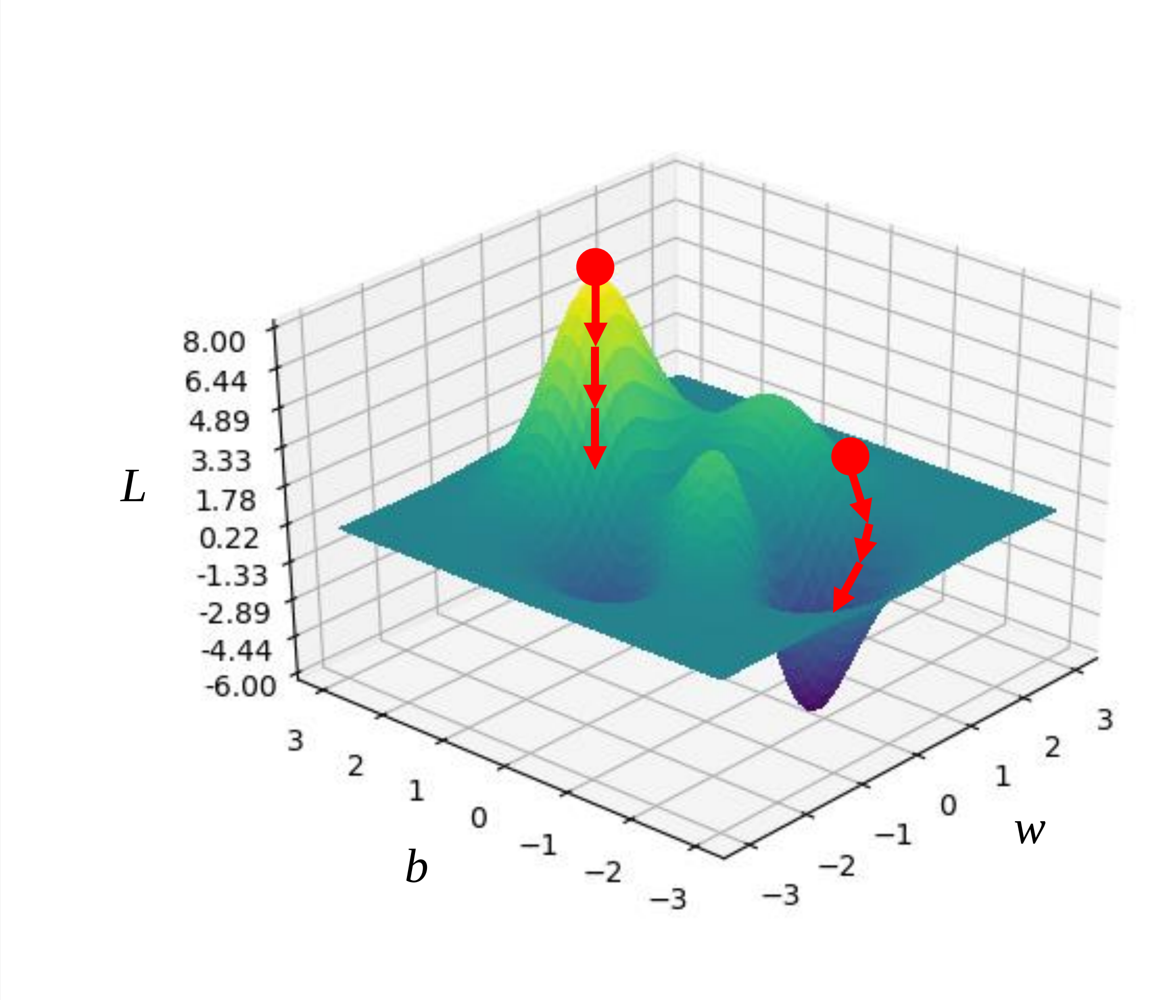
$-\eta\frac{\partial L}{\partial w}|_{w=w_0,b=b_0}$

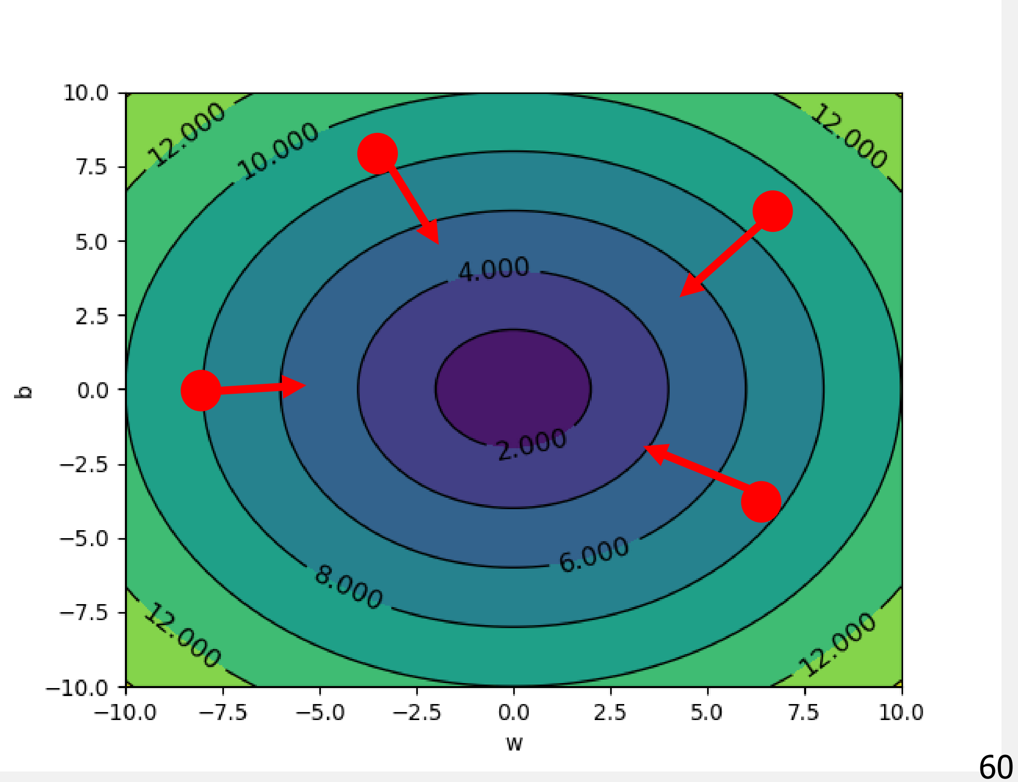
$-\eta\frac{\partial L}{\partial b}|_{w=w_1,b=b_1}$

$-\eta\frac{\partial L}{\partial b}|_{w=w_0,b=b_0}$

Gradient Descent

God bless you



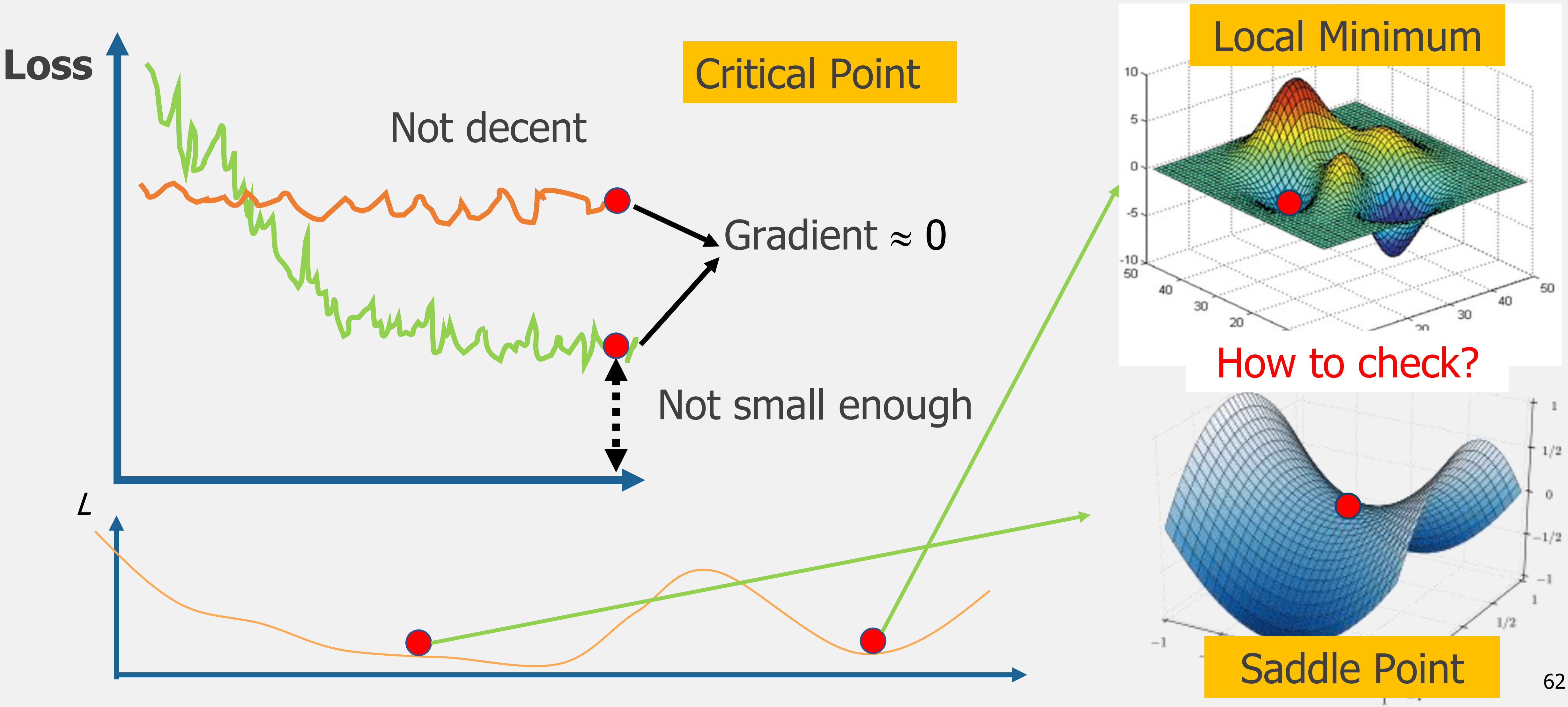


06

Why Gradient is Small?

Why Gradient is small?

Why does optimization fail?



Loss

Critical Point

Not decent

Gradient ≈ 0

Not small enough

L

Local Minimum

How to check?

Saddle Point

Why Gradient is small?

How to check?

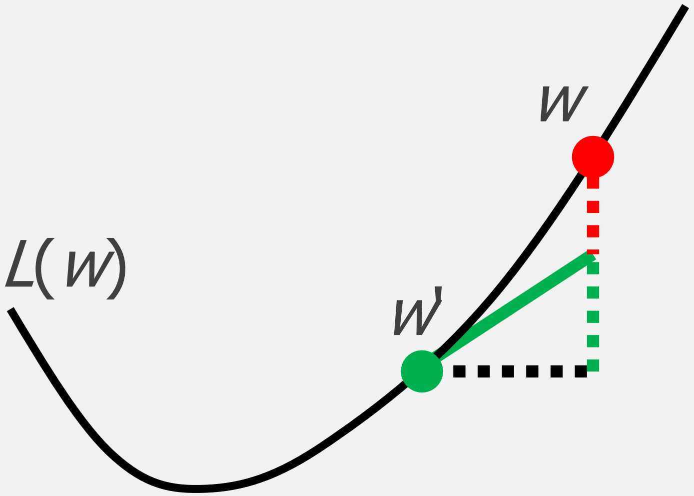
Taylor Series Approximation

$L(w)$ around $w = w'$ can be approximated as

$$L(w) \approx L(w') + (w - w')^T g + \frac{1}{2}(w - w')^T H (w - w')$$

g is the gradient vector H is the Hessian matrix

$$g = \nabla L(w') \qquad H = \frac{\partial^2}{(\partial w)^2} L(w')$$



Why Gradient is small?

How to check?

Taylor Series Approximation

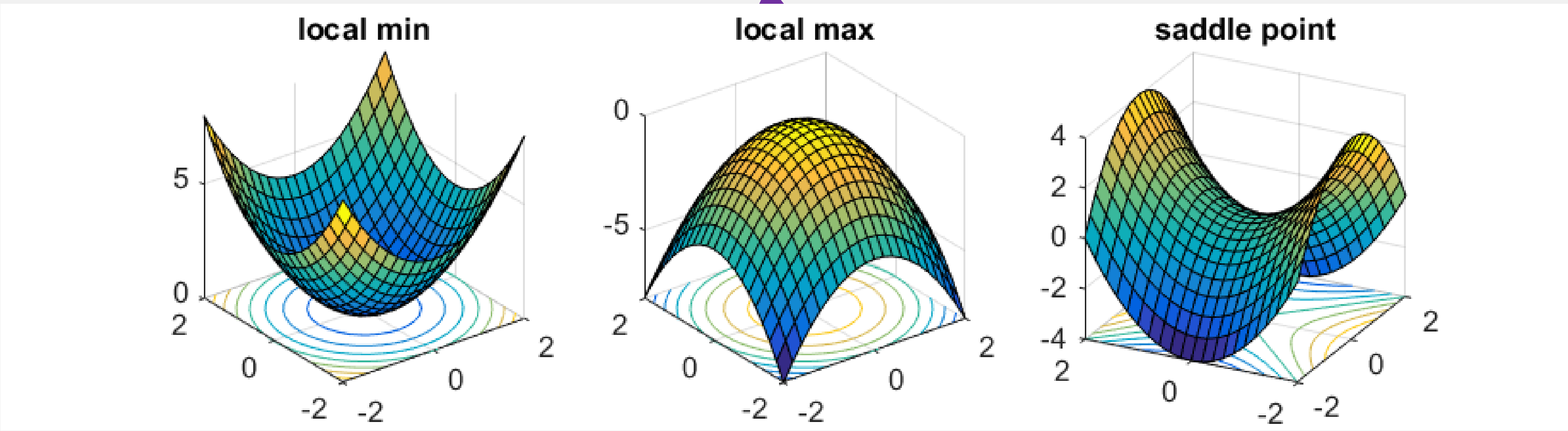
$L(w)$ around $w = w'$ can be approximated as

Critical Point

$$L(w) \approx L(w') + \cancel{(w - w')^T g} + \boxed{\frac{1}{2}(w - w')^T H (w - w')}$$

$$g = \nabla L(w') = 0$$

How to determine?



Source of image

Why Gradient is small?

How to check?

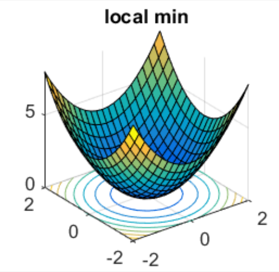
Hessian

Critical Point → $L(w) \approx L(w') + \frac{1}{2}(w - w')^{T} H (w - w') = L(w') + \frac{1}{2} v^{T} H v$

All v

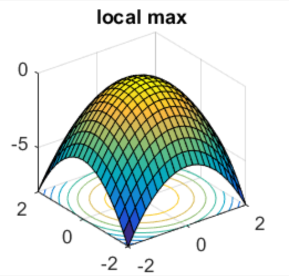
$v^{T} H v > 0$ → $L(w) > L(w')$

$= H > 0$ is defined as all eigenvalues are positive →


local min

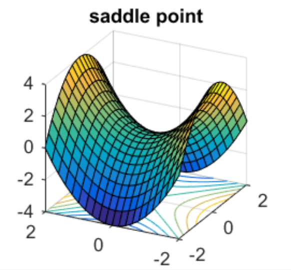
$v^{T} H v < 0$ → $L(w) < L(w')$

$= H < 0$ is defined as all eigenvalues are negative →


local max

All v $v^{T} H v < 0$ or $v^{T} H v > 0$

Some eigenvalues are negative, and some are positive →

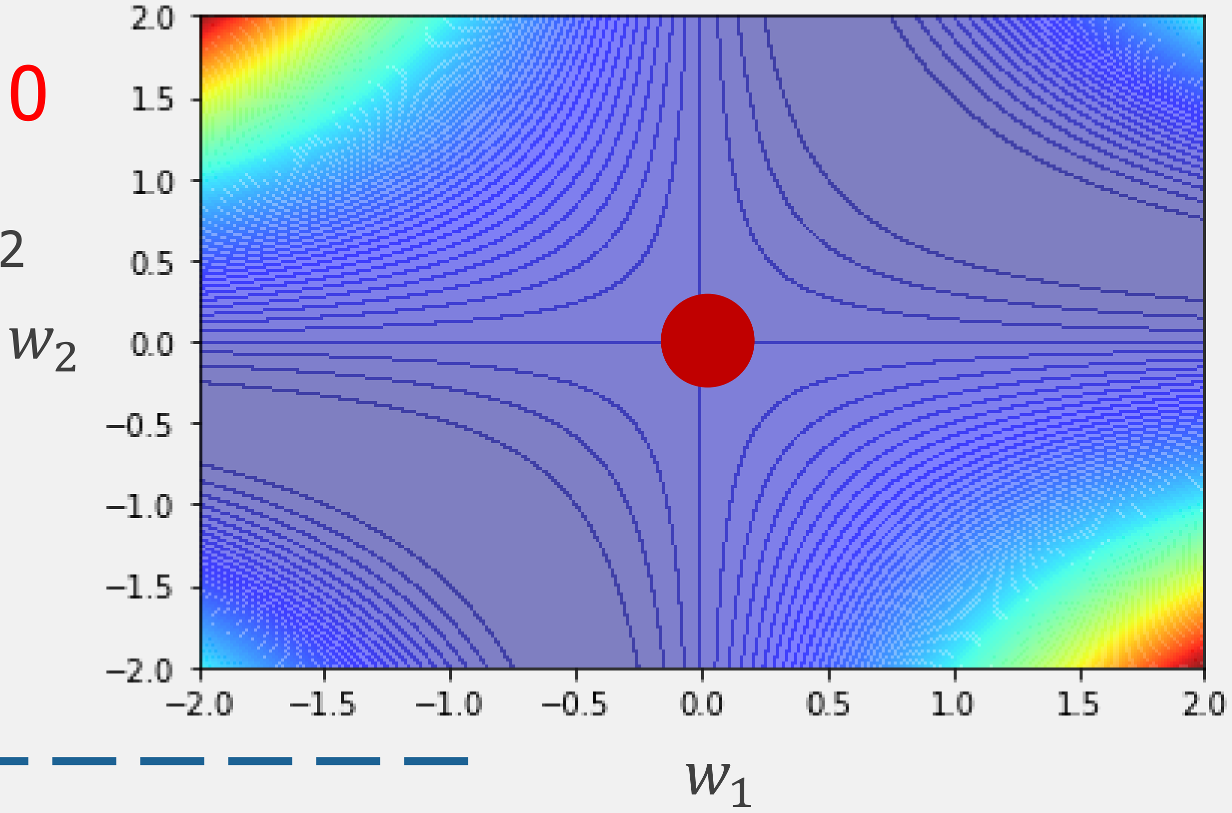

saddle point

Why Gradient is small?

How to check?

$$\hat{y} = f(x) = w_2 w_1 x \quad \begin{cases} x = 1 \\ y = 1 \end{cases} \quad w_1 = w_2 = 0$$

$$L(w_1, w_2) = (y - w_2 w_1 x)^2 = (1 - w_2 w_1)^2$$



$$\left.\begin{aligned} \frac{\partial L}{\partial w_1} &= 2(1 - w_2 w_1)(-w_2) = 0 \\ \frac{\partial L}{\partial w_2} &= 2(1 - w_2 w_1)(-w_1) = 0 \end{aligned}\right\} g$$

$$\left.\begin{aligned} \frac{\partial^2 L}{(\partial w_1)^2} &= 2(-w_2)(-w_2) = 0 \quad & \frac{\partial^2 L}{\partial w_1 \partial w_2} &= -2+4w_1 w_2 = -2 \\ \frac{\partial^2 L}{\partial w_2 \partial w_1} &= -2+4w_1 w_2 = -2 \quad & \frac{\partial^2 L}{(\partial w_2)^2} &= 2(-w_1)(-w_1) = 0 \end{aligned}\right\} H$$

Why Gradient is small?

How to check?

Eigenvalues

$H = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix}$ $\lambda_1 = 2$ $\lambda_2 = -2$

$Ax = \lambda x$

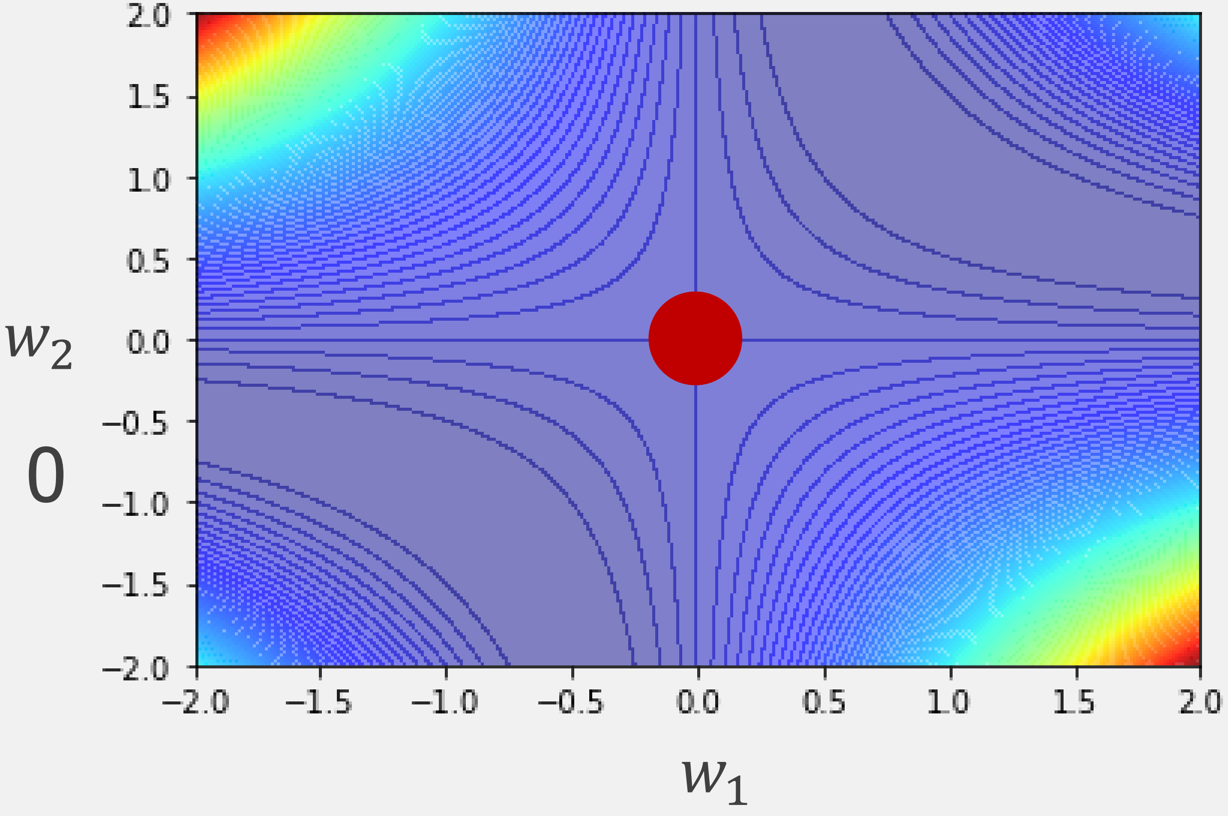
$Ax - \lambda x = 0$ $(A-\lambda I)x = 0$ $\det(A-\lambda I) = 0$

$\det\left(\begin{bmatrix} -\lambda & -2 \\ -2 & -\lambda \end{bmatrix}\right) = 0$

$\lambda^2 - 4 = 0$

$(\lambda+2)(\lambda-2) = 0$

Saddle point



Why Gradient is small?

How to deal with the saddle point?

$$L(w) \approx L(w') + \frac{1}{2}(w-w')^{T} H (w-w') = L(w') + \frac{1}{2} v^{T} H v$$ at critical point

H has positive and negative eigenvalues at saddle point

Determine update direction by H

λ, u is the eigenvalue and the corresponding eigenvector of H

$$u^{T} H u = u^{T} \lambda u = \lambda \|u\|^{2} \begin{cases} \boxed{< 0 \quad \lambda < 0} \\ > 0 \quad \lambda > 0 \end{cases}$$

$$L(w) \approx L(w') + \frac{1}{2}(\underset{u}{w-w'})^{T} H (w-w') \Rightarrow w = u + w'$$

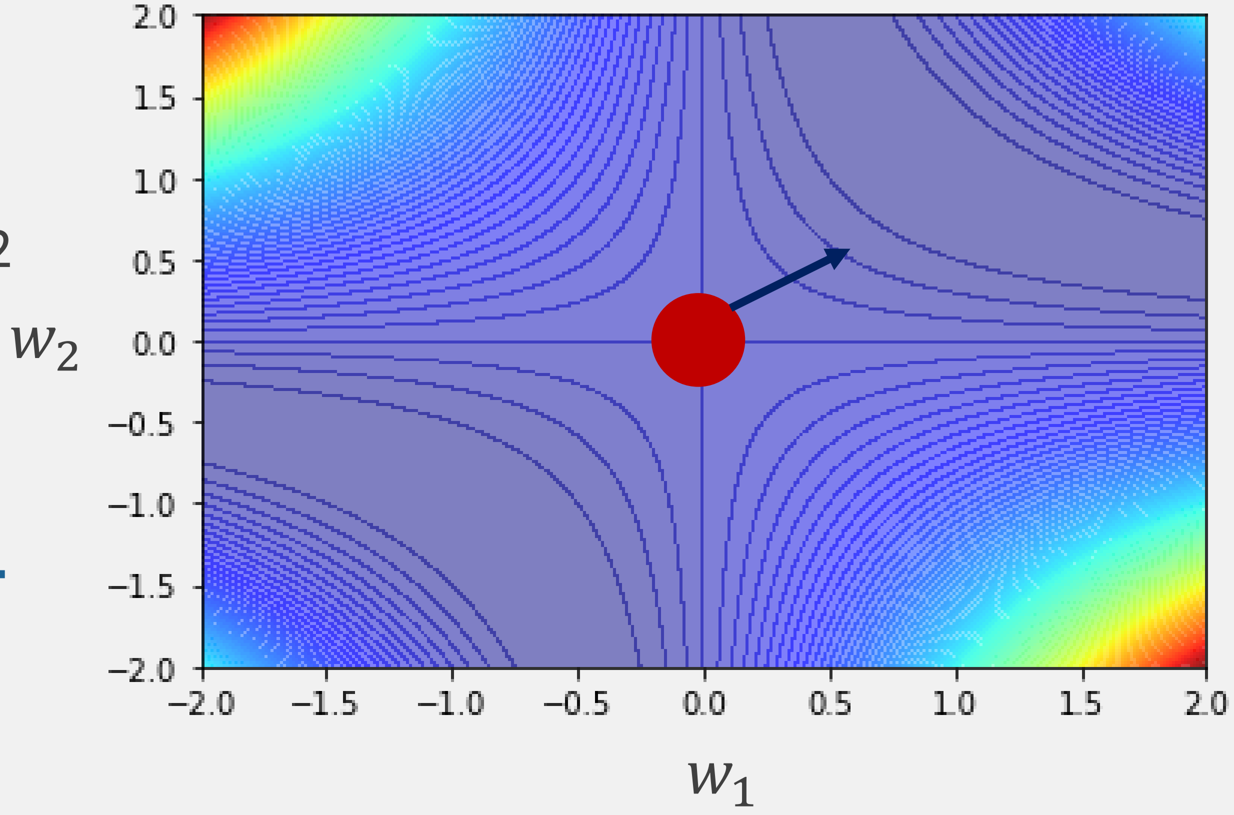
Why Gradient is small?

How to deal with the saddle point?

$$\hat{y} = f(x) = w_2 w_1 x$$

$$L(w_1, w_2) = (y - w_2 w_1 x)^2 = (1 - w_2 w_1)^2$$

$$H = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix} \quad \lambda_1 = 2 \quad \lambda_2 = -2$$



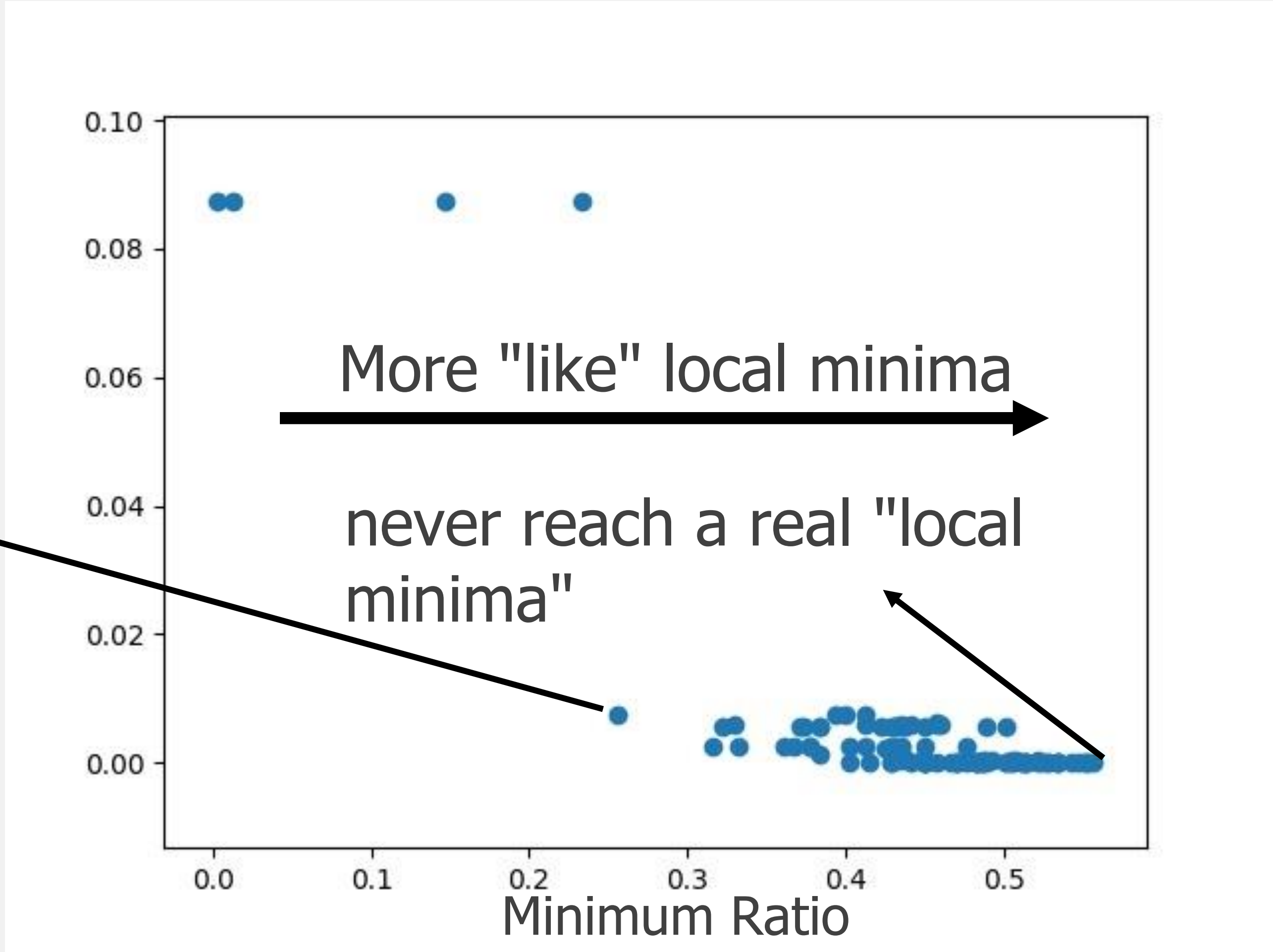
$$\lambda = -2 \qquad \text{eigenvector } u = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Update w along the direction of u

Why Gradient is small?

Saddle Point vs Local Minima

Training Loss



More "like" local minima

never reach a real "local minima"

Train a network once, until it converges to critical point.

$$\text{Minimum ratio} = \frac{\#\text{ of }\textbf{Positive}\text{ Eigen values}}{\#\text{ of Eigen values}}$$

Source

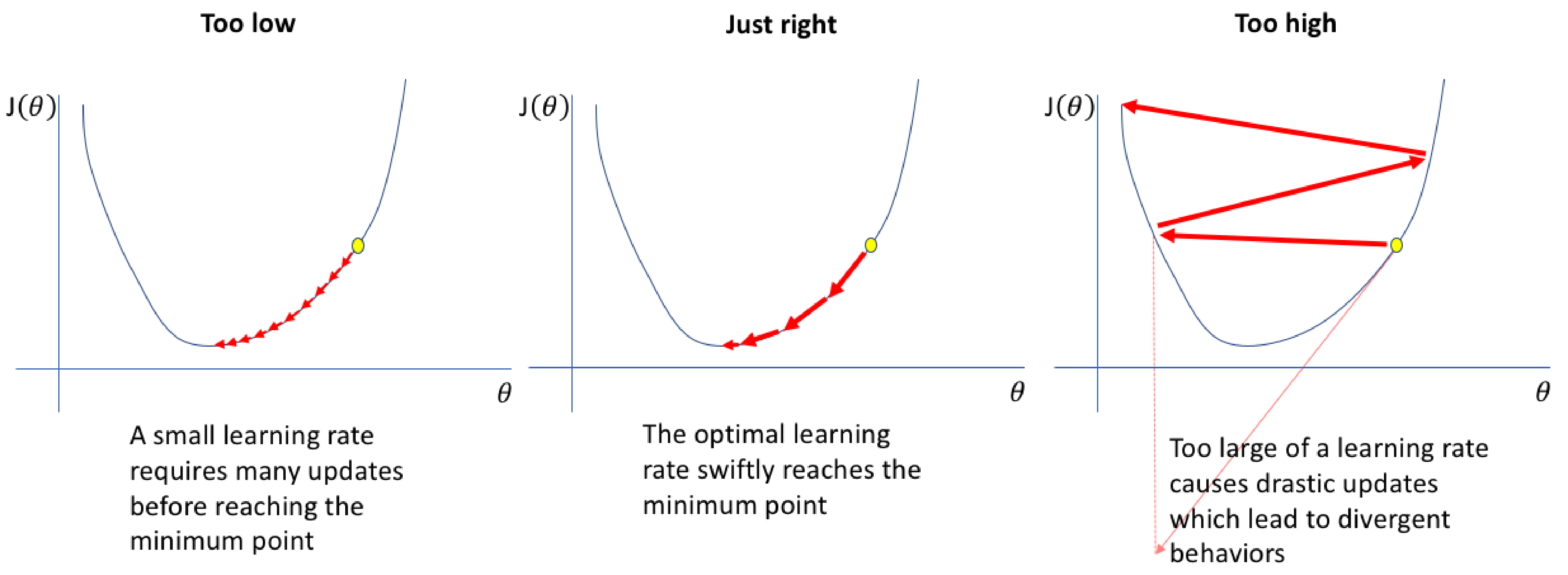
07

Adaptive Learning Rate

Adaptive Learning Rate

Training stuck is because the parameters are near to a critical point?

$$-\eta \frac{\partial L}{\partial w}\Big|_{w=w_0}$$



A small learning rate requires many updates before reaching the minimum point

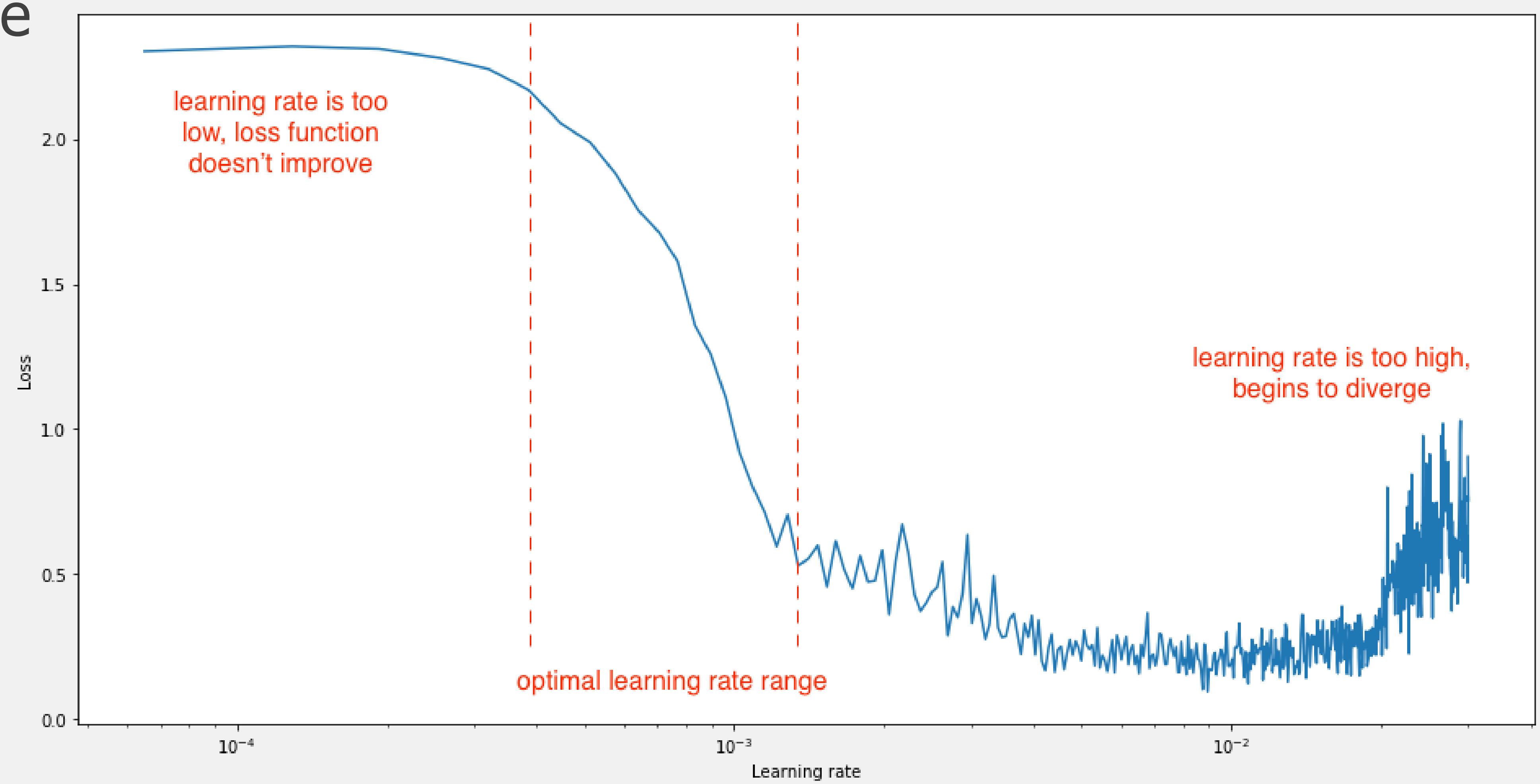
The optimal learning rate swiftly reaches the minimum point

Too large of a learning rate causes drastic updates which lead to divergent behaviors

Reference

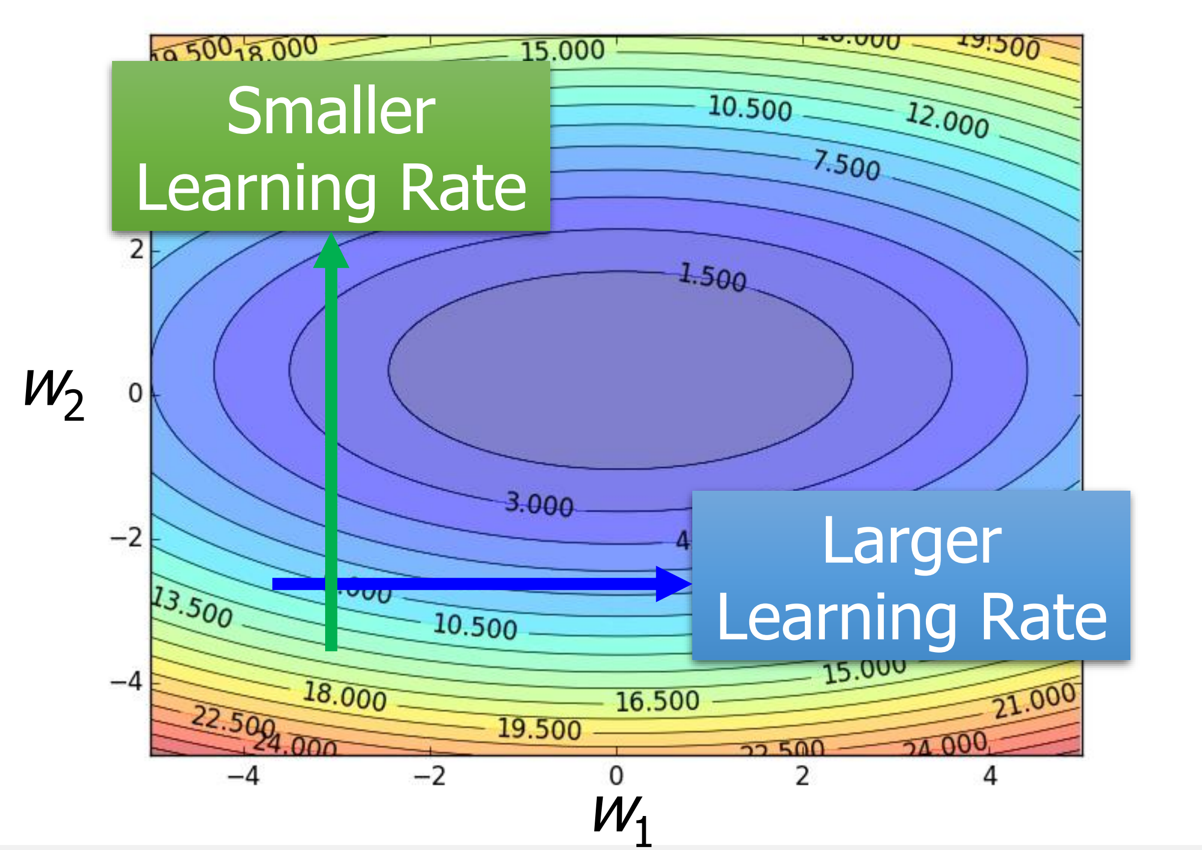
Adaptive Learning Rate

No, learning rate cannot be one-size-fits-all and you should adjust your learning rate



Adaptive Learning Rate

Difference parameters needs different learning rate



$$\theta_{t+1}^i = \theta_t^i - \eta \frac{\partial L}{\partial \theta}|_{\theta=\theta_t}$$

$$\theta_{t+1}^i = \theta_t^i - \eta g_t^i$$

$$\theta_{t+1}^i = \theta_t^i - \boxed{\frac{\eta}{\sigma_t^i}} g_t^i$$

Parameter dependent

Adaptive Learning Rate

$$\theta_{t+1}^{i} = \theta_{t}^{i} - \boxed{\frac{\eta}{\sigma_{t}^{i}}} g_{t}^{i}$$

Root Mean Square (Adagrad)

$$\boldsymbol{\theta}_{1}^{i} \leftarrow \boldsymbol{\theta}_{0}^{i} - \frac{\eta}{\sigma_{0}^{i}} \boldsymbol{g}_{0}^{i} \qquad \sigma_{0}^{i} = \sqrt{\left(\boldsymbol{g}_{0}^{i}\right)^{2}} = \left|\boldsymbol{g}_{0}^{i}\right|$$

$$\boldsymbol{\theta}_{2}^{i} \leftarrow \boldsymbol{\theta}_{1}^{i} - \frac{\eta}{\sigma_{1}^{i}} \boldsymbol{g}_{1}^{i} \qquad \sigma_{1}^{i} = \sqrt{\frac{1}{2}\left[\left(\boldsymbol{g}_{0}^{i}\right)^{2} + \left(\boldsymbol{g}_{1}^{i}\right)^{2}\right]}$$

$$\boldsymbol{\theta}_{3}^{i} \leftarrow \boldsymbol{\theta}_{2}^{i} - \frac{\eta}{\sigma_{2}^{i}} \boldsymbol{g}_{2}^{i} \qquad \sigma_{2}^{i} = \sqrt{\frac{1}{3}\left[\left(\boldsymbol{g}_{0}^{i}\right)^{2} + \left(\boldsymbol{g}_{1}^{i}\right)^{2} + \left(\boldsymbol{g}_{2}^{i}\right)^{2}\right]}$$

$$\vdots$$

$$\boldsymbol{\theta}_{t+1}^{i} \leftarrow \boldsymbol{\theta}_{t}^{i} - \frac{\eta}{\sigma_{t}^{i}} \boldsymbol{g}_{t}^{i} \qquad \sigma_{t}^{i} = \sqrt{\frac{1}{t+1}\sum_{i=0}^{t}\left(\boldsymbol{g}_{t}^{i}\right)^{2}}$$

Adaptive Learning Rate

RMSProp

$$\theta_{t+1}^{i} = \theta_t^{i} - \boxed{\frac{\eta}{\sigma_t^{i}}} g_t^{i}$$

$$\boldsymbol{\theta}_1^{i} \leftarrow \boldsymbol{\theta}_0^{i} - \frac{\eta}{\sigma_0^{i}} \boldsymbol{g}_0^{i} \qquad \boldsymbol{\sigma}_0^{i} = \sqrt{\left(\boldsymbol{g}_0^{i}\right)^2} = \left|\boldsymbol{g}_0^{i}\right|$$

$$\boldsymbol{\theta}_2^{i} \leftarrow \boldsymbol{\theta}_1^{i} - \frac{\eta}{\sigma_1^{i}} \boldsymbol{g}_1^{i} \qquad \sigma_1^{i} = \sqrt{\alpha(\sigma_0^{i})^2 + (1-\alpha)(g_1^{i})^2}$$

$$\boldsymbol{\theta}_3^{i} \leftarrow \boldsymbol{\theta}_2^{i} - \frac{\eta}{\sigma_2^{i}} \boldsymbol{g}_2^{i} \qquad \sigma_2^{i} = \sqrt{\alpha(\sigma_1^{i})^2 + (1-\alpha)(g_2^{i})^2}$$

$$\vdots$$

$$\boldsymbol{\theta}_{t+1}^{i} \leftarrow \boldsymbol{\theta}_t^{i} - \frac{\eta}{\sigma_t^{i}} \boldsymbol{g}_t^{i} \qquad \sigma_t^{i} = \sqrt{\alpha(\sigma_{t-1}^{i})^2 + (1-\alpha)(g_t^{i})^2}$$

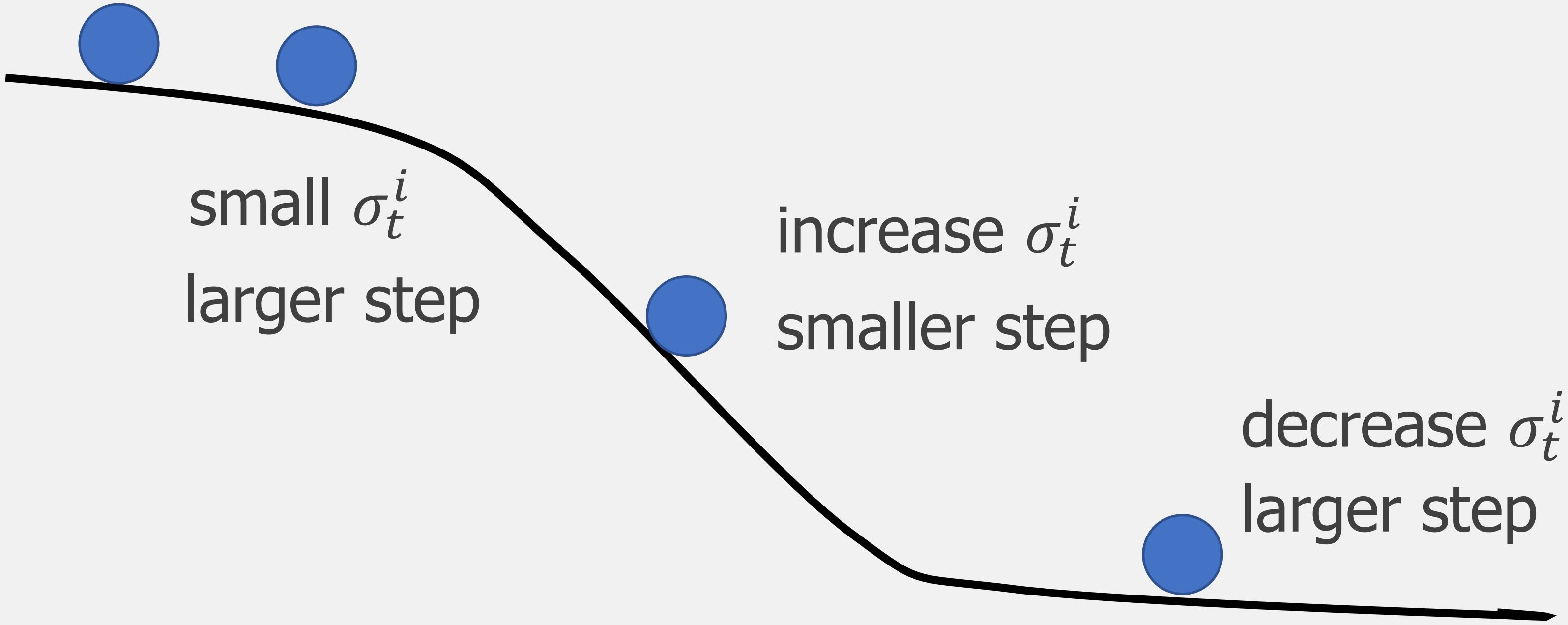
Adaptive Learning Rate

RMSProp

$$\theta_{t+1}^i = \theta_t^i - \boxed{\frac{\eta}{\sigma_t^i}} g_t^i$$

$$\sigma_t^i = \sqrt{\alpha(\sigma_{t-1}^i)^2 + (1-\alpha)(g_t^i)^2}$$

The recent gradient has larger influence, and the past gradients have less influence.



small σ_t^i

larger step

increase σ_t^i

smaller step

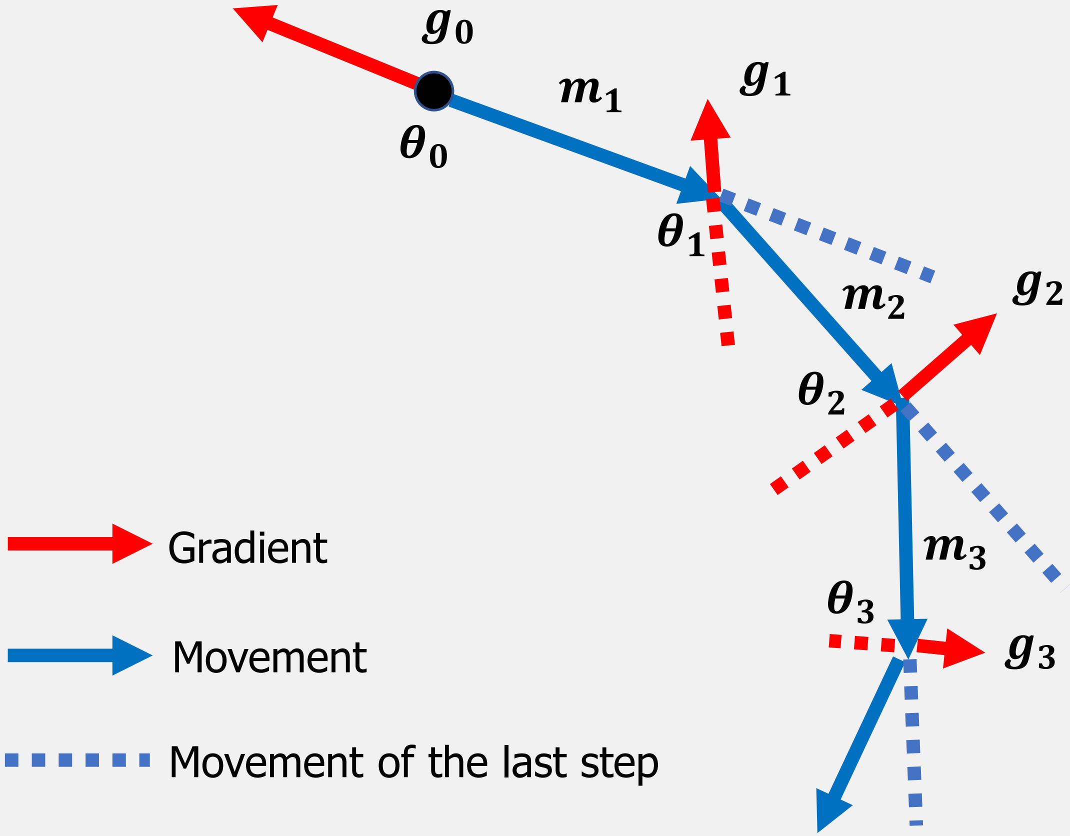
decrease σ_t^i

larger step

Adaptive Learning Rate

Momentum

Movement not just based on gradient, but previous movement.



Starting at $\boldsymbol{\theta}_0$
Movement $\boldsymbol{m}_0 = \boldsymbol{0}$
Compute gradient $\boldsymbol{g}_0$
Movement $\boldsymbol{m}_1 = \lambda \boldsymbol{m}_0 - \eta \boldsymbol{g}_0$
Move to $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0 + \boldsymbol{m}_1$
Compute gradient $\boldsymbol{g}_1$
Movement $\boldsymbol{m}_2 = \lambda \boldsymbol{m}_1 - \eta \boldsymbol{g}_1$
Move to $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1 + \boldsymbol{m}_2$

$$\theta_{t+1}^{i} = \theta_{t}^{i} - \eta \frac{\partial L}{\partial \theta} |_{\theta=\theta_t}$$

$$\Downarrow$$

$$\theta_{t+1}^{i} = \theta_{t}^{i} + v_{t}^{i}$$

$$v_0^i = 0$$

$$v_t^i = \lambda v_{t-1}^i - \eta g_t^i$$

Adaptive Learning Rate

Adam (RMSProp + Momentum + Bias-correction)

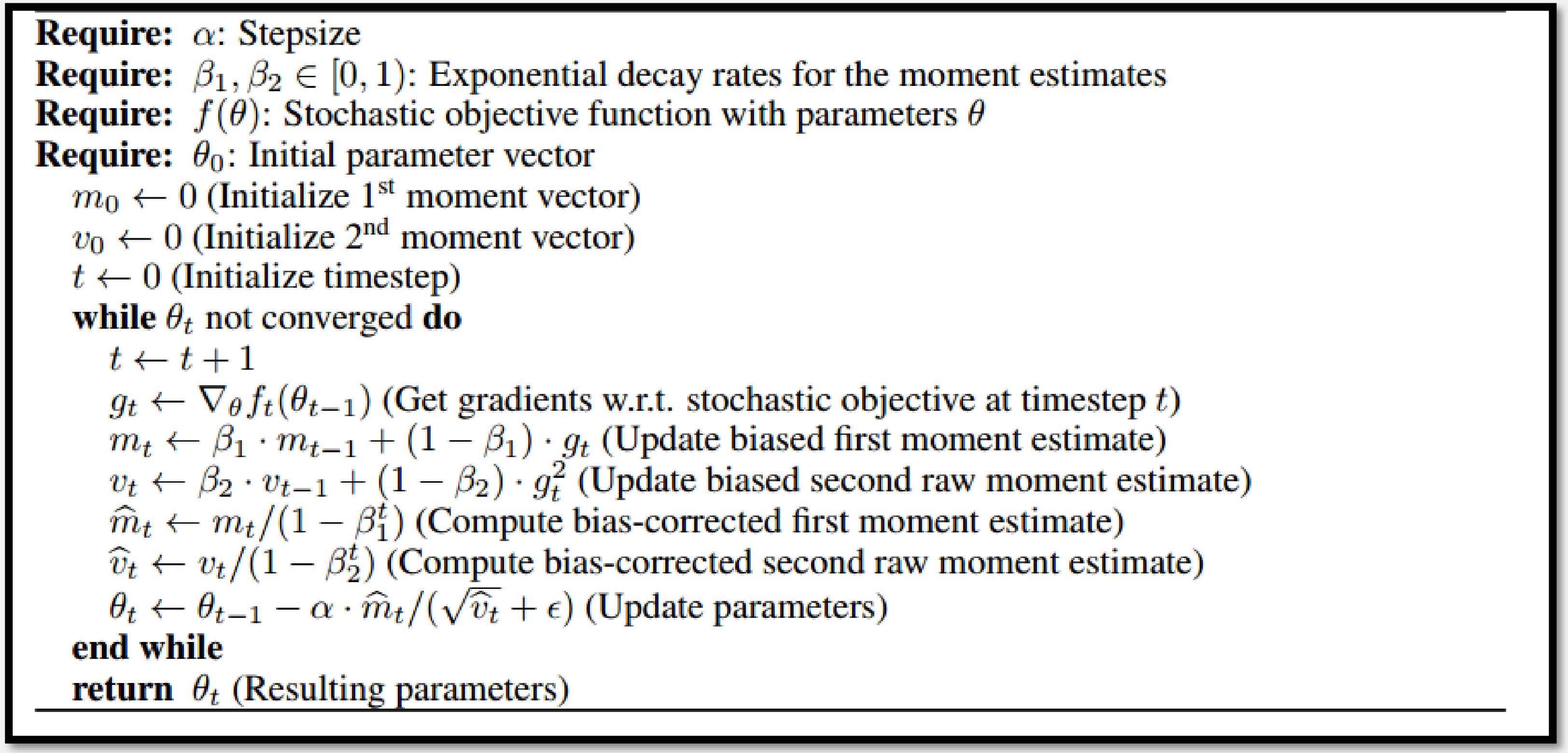
$$\boldsymbol{\theta}_{t+1}^{i} \leftarrow \boldsymbol{\theta}_{t}^{i} - \eta \frac{\hat{m}_t^i}{\sqrt{\hat{v}_t^i + \varepsilon}}$$

$$m_0^i = 0 \quad m_t^i = \beta_1 (m_{t-1}^i)^2 + (1-\beta_1)(g_t^i)^2$$

$$v_0^i = 0 \quad v_t^i = \beta_2 (v_{t-1}^i)^2 + (1-\beta_2)(g_t^i)^2$$

$$\hat{m}_t^i = \frac{m_t^i}{1-\beta_1}$$

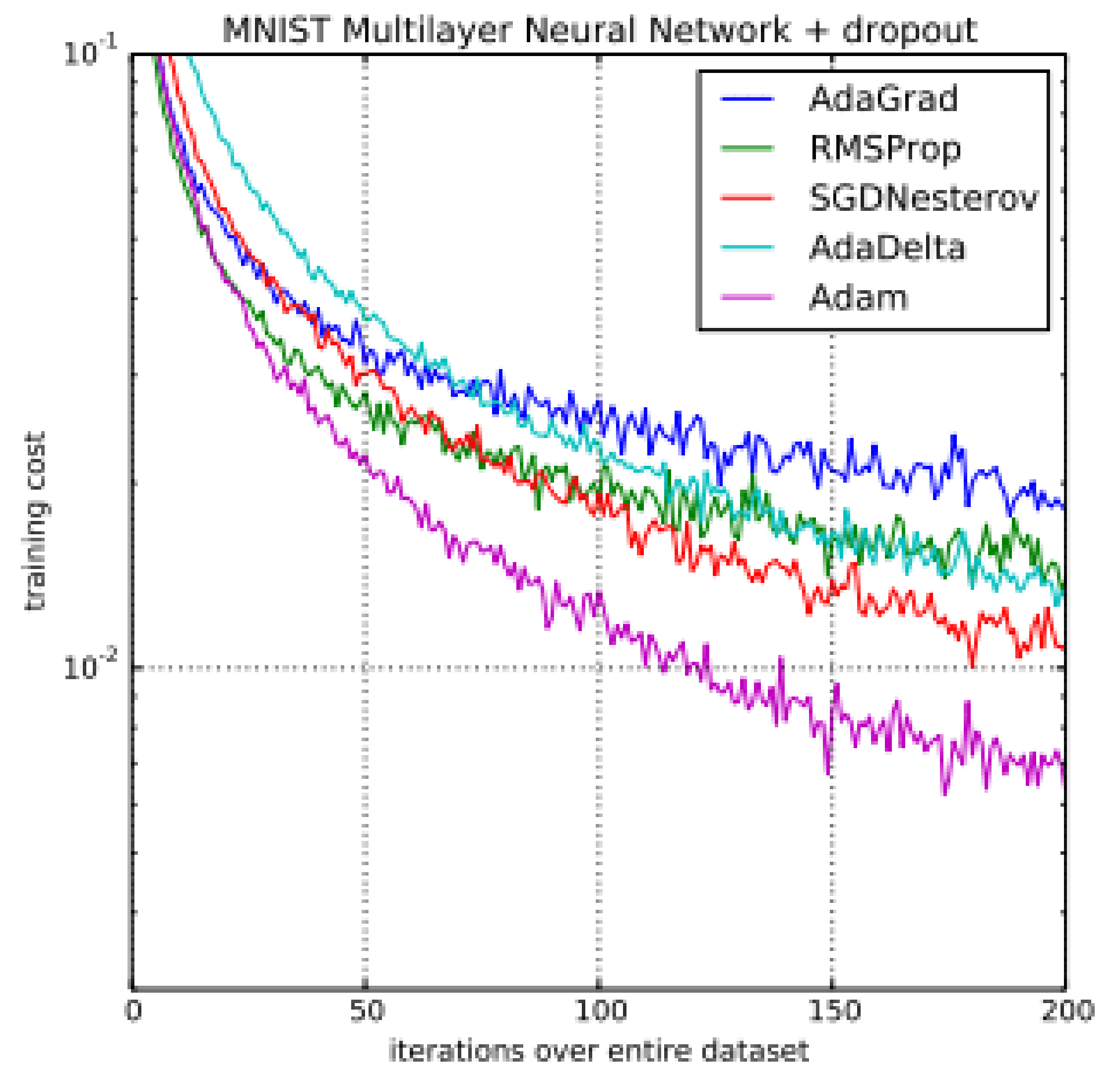
$$\hat{v}_t^i = \frac{v_t^i}{1-\beta_2}$$



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

Reference

Adaptive Learning Rate



Reference

08

Cross Validation

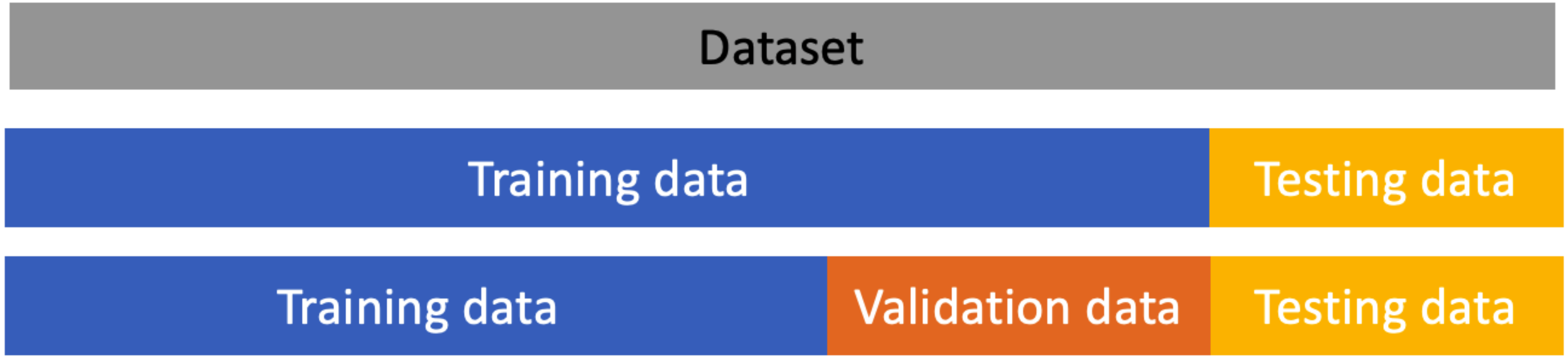
Cross Validation

Cross validation (All figures come from here)

- Holdout
- K-fold
  - K-fold
  - Nested K-fold
  - Repeated K-fold
  - Stratified K-fold
- Leave one out
- Random subsampling
- Bootstrap

Cross Validation

Holdout

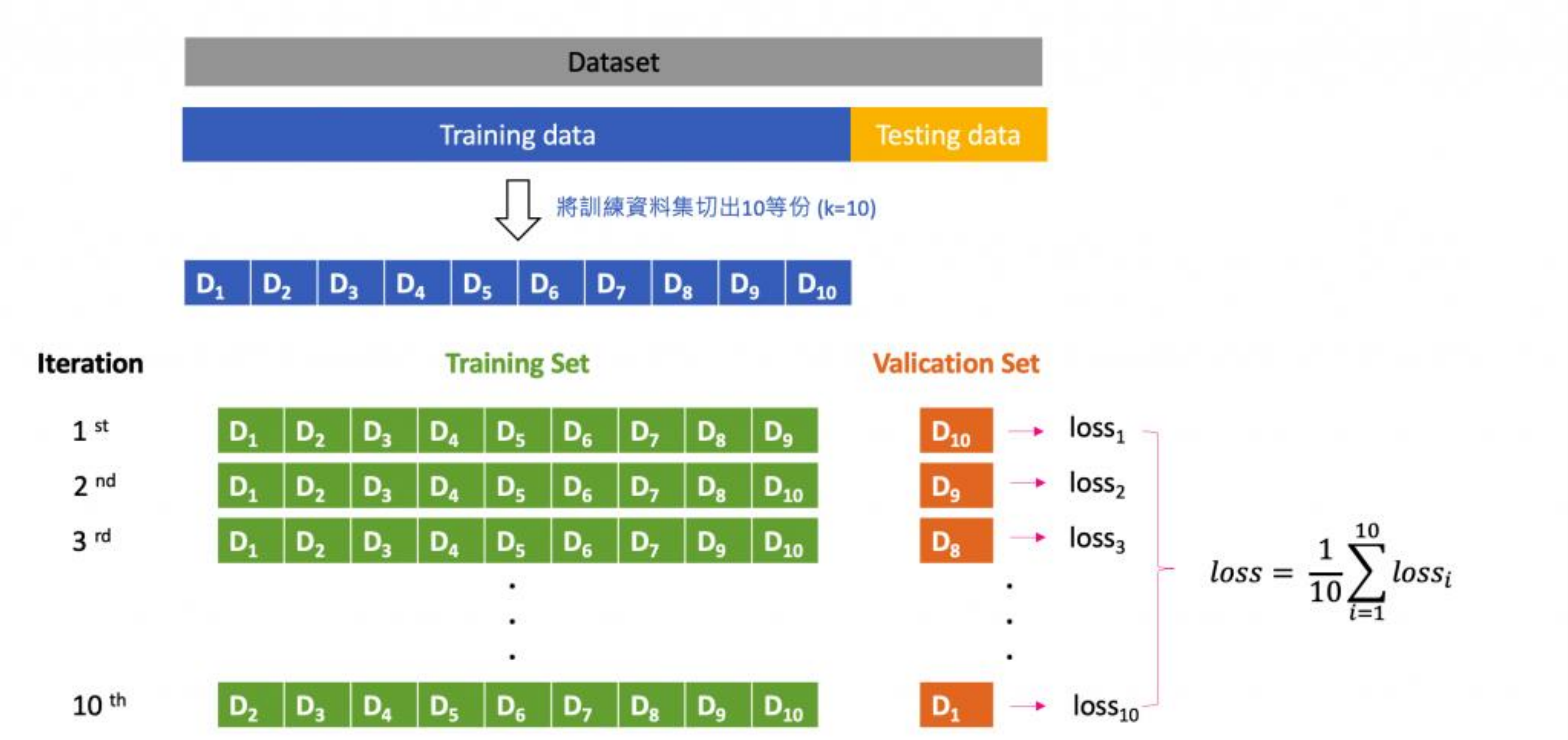

Dataset

Training data | Testing data

Training data | Validation data | Testing data

Cross Validation

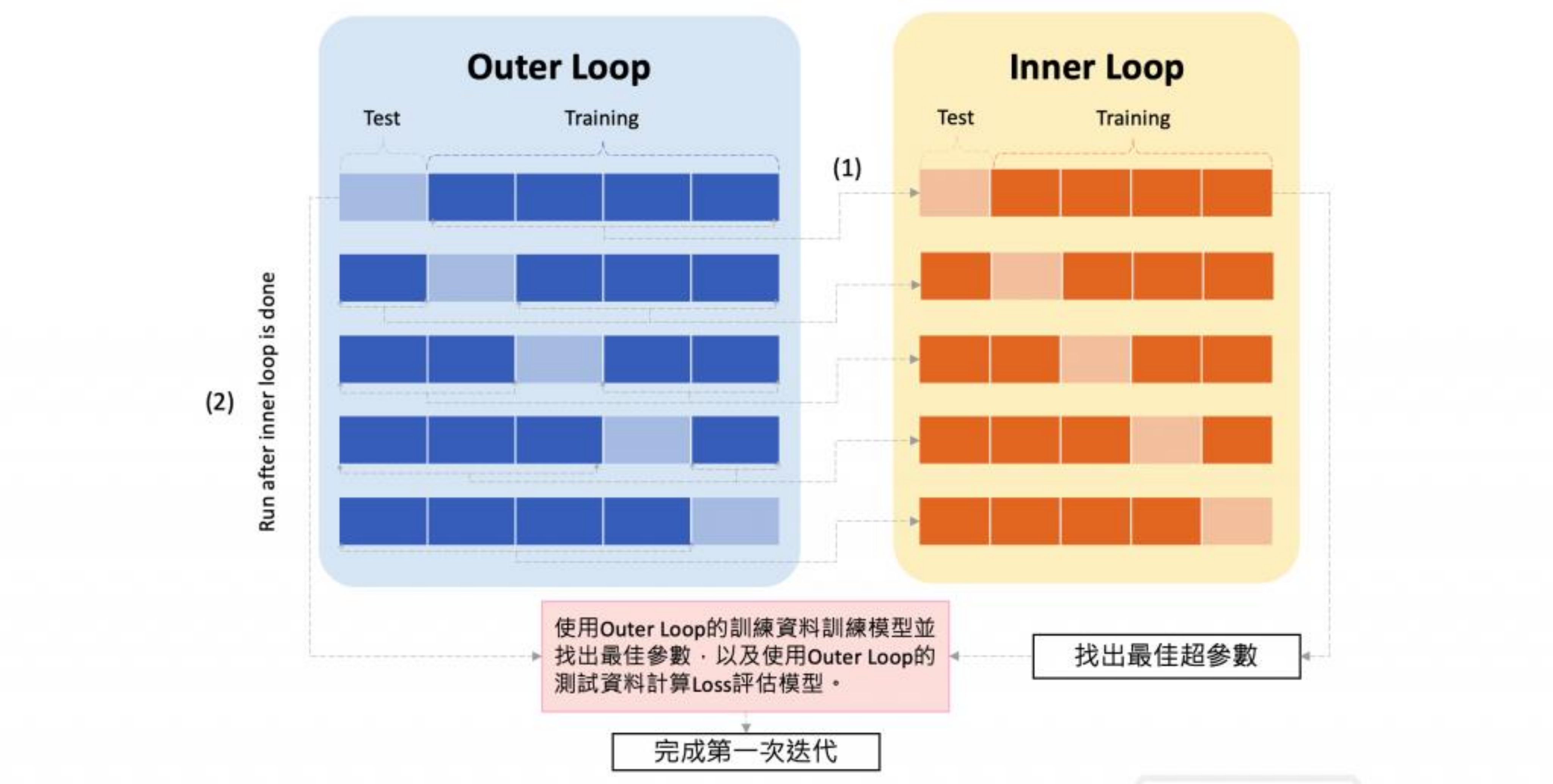
K-fold



$$loss = \frac{1}{10}\sum_{i=1}^{10} loss_i$$

Cross Validation

Nested K-fold



Cross Validation

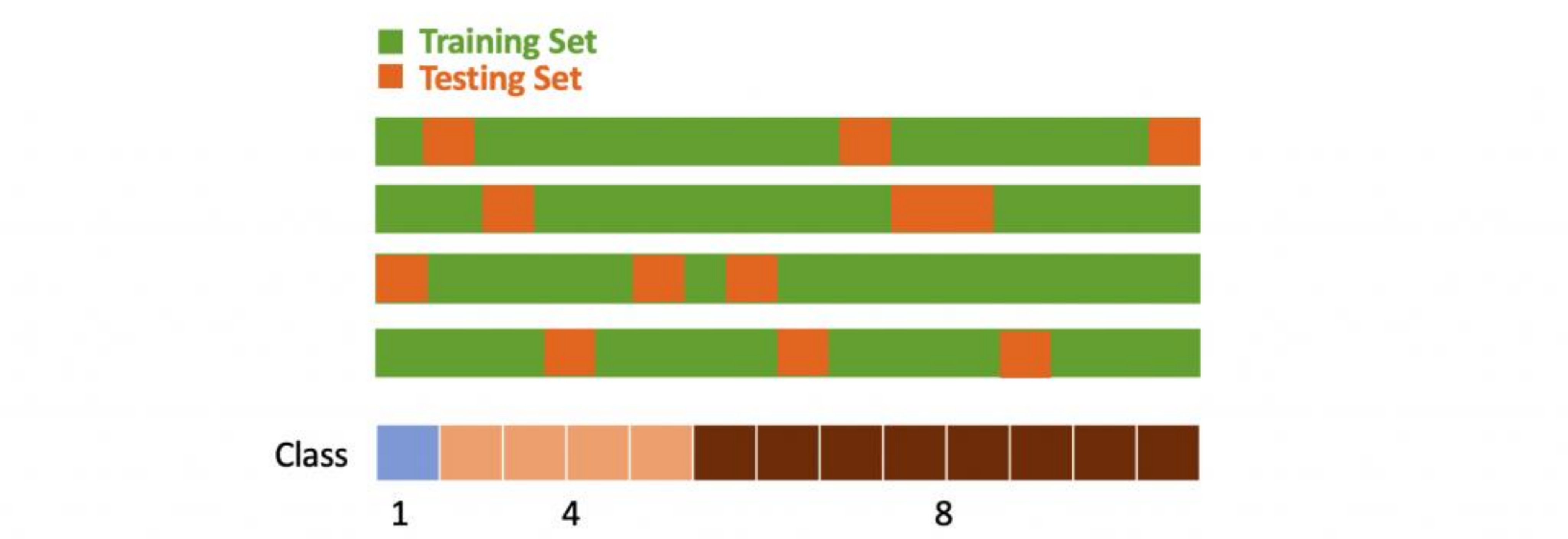
Repeated K-fold

🟩 Training Set
🟧 Testing Set

Iteration				
1st	🟧 D_1	🟧 D_2	🟩 D_3	🟩 D_4
2nd	🟩 D_1	🟩 D_2	🟧 D_3	🟧 D_4
3rd	🟧 D_1	🟩 D_2	🟩 D_3	🟧 D_4
4th	🟩 D_1	🟧 D_2	🟧 D_3	🟩 D_4

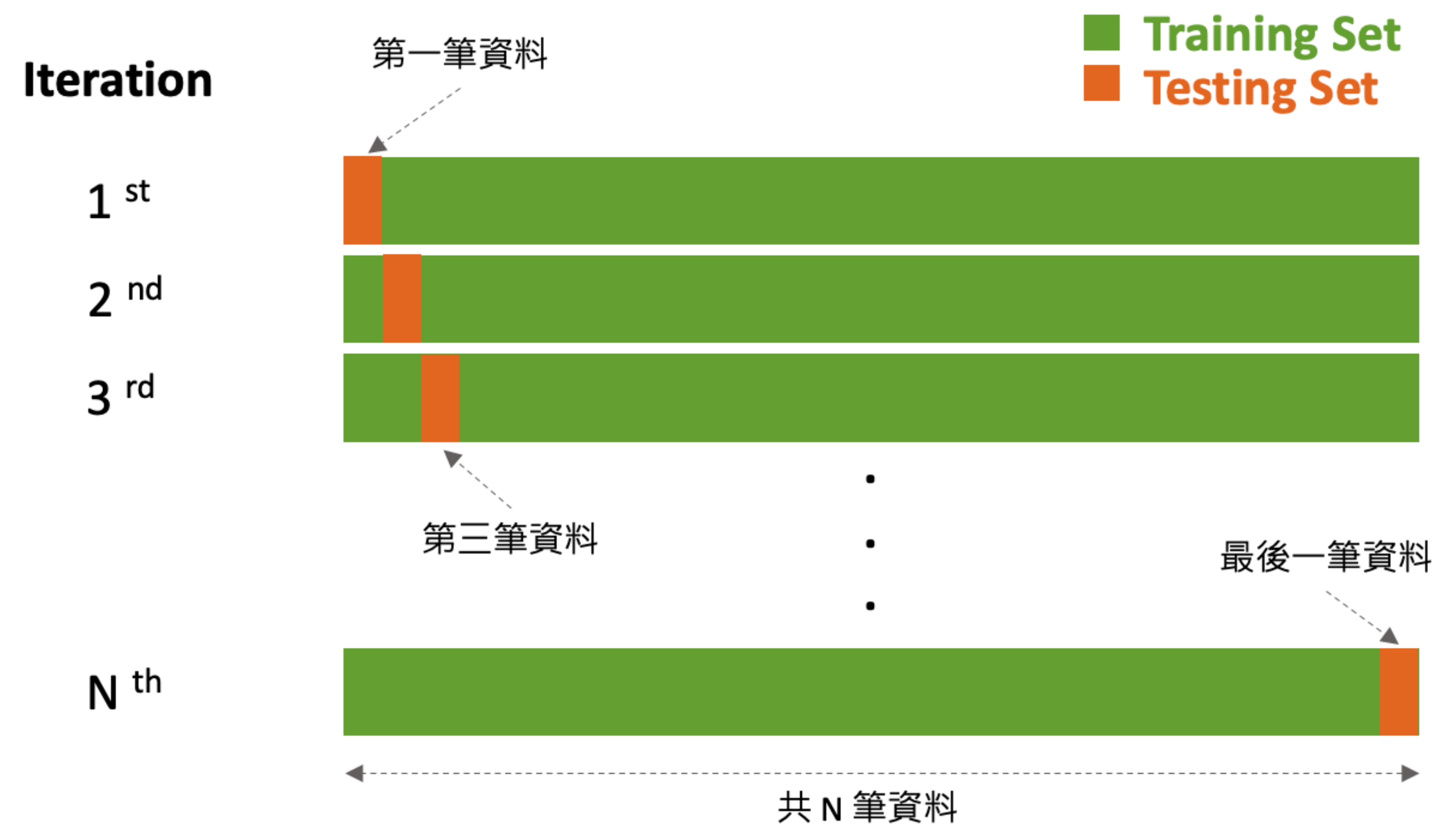
Cross Validation

Stratified K-fold



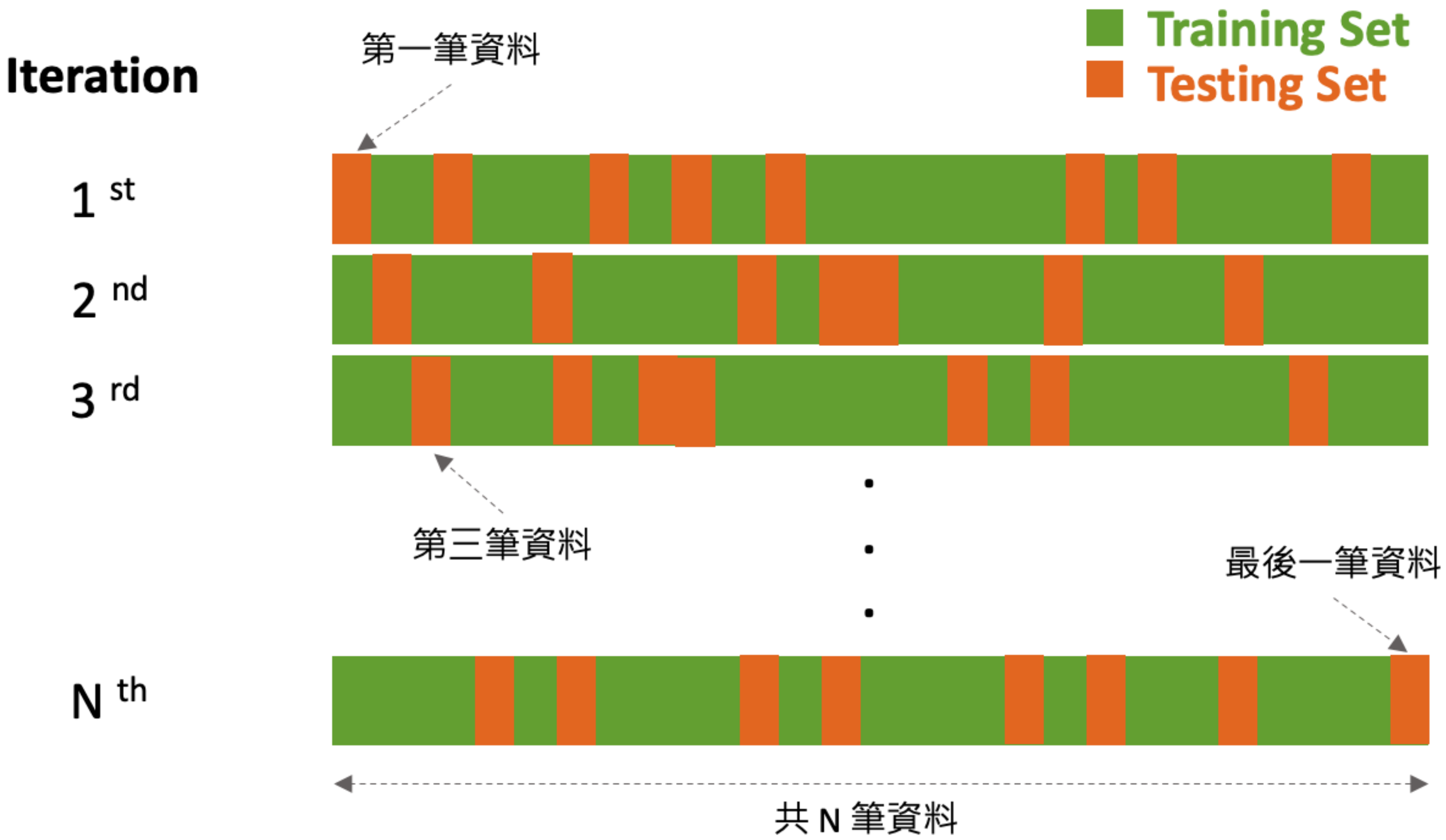
Cross Validation

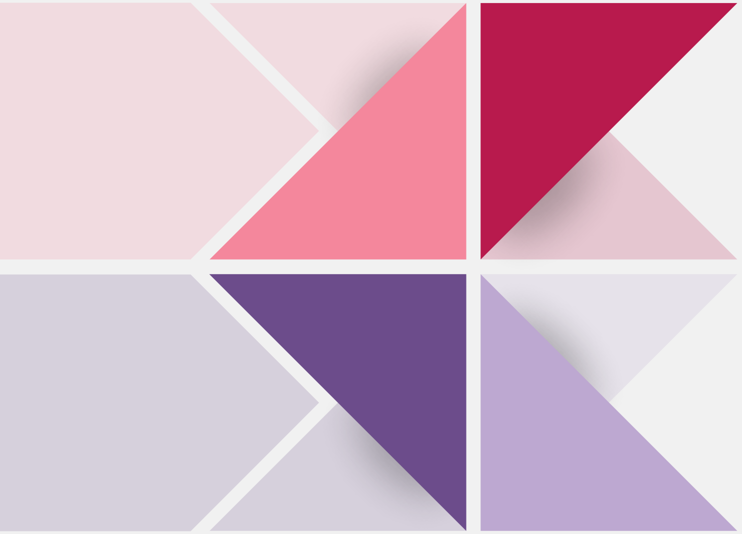
Leave one out



Cross Validation

Random subsampling





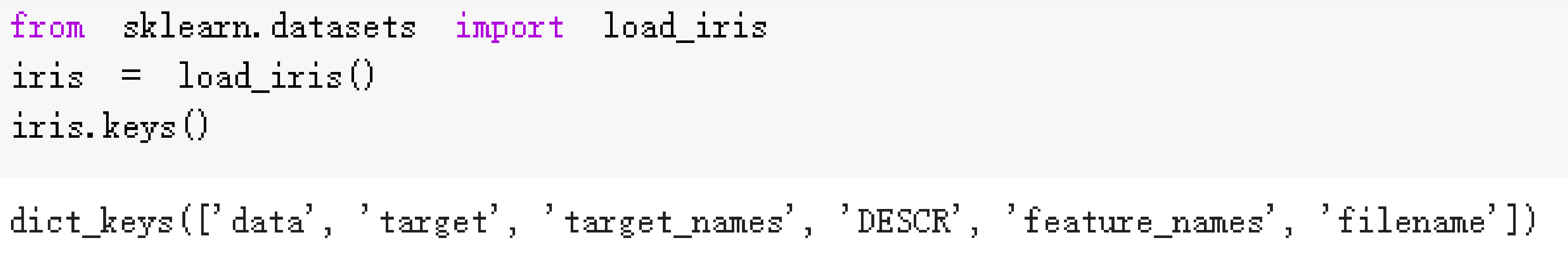
09

Example

Deep Learning

Iris Classification

Iris dataset



```
from  sklearn.datasets  import  load_iris
iris  =  load_iris()
iris.keys()
```

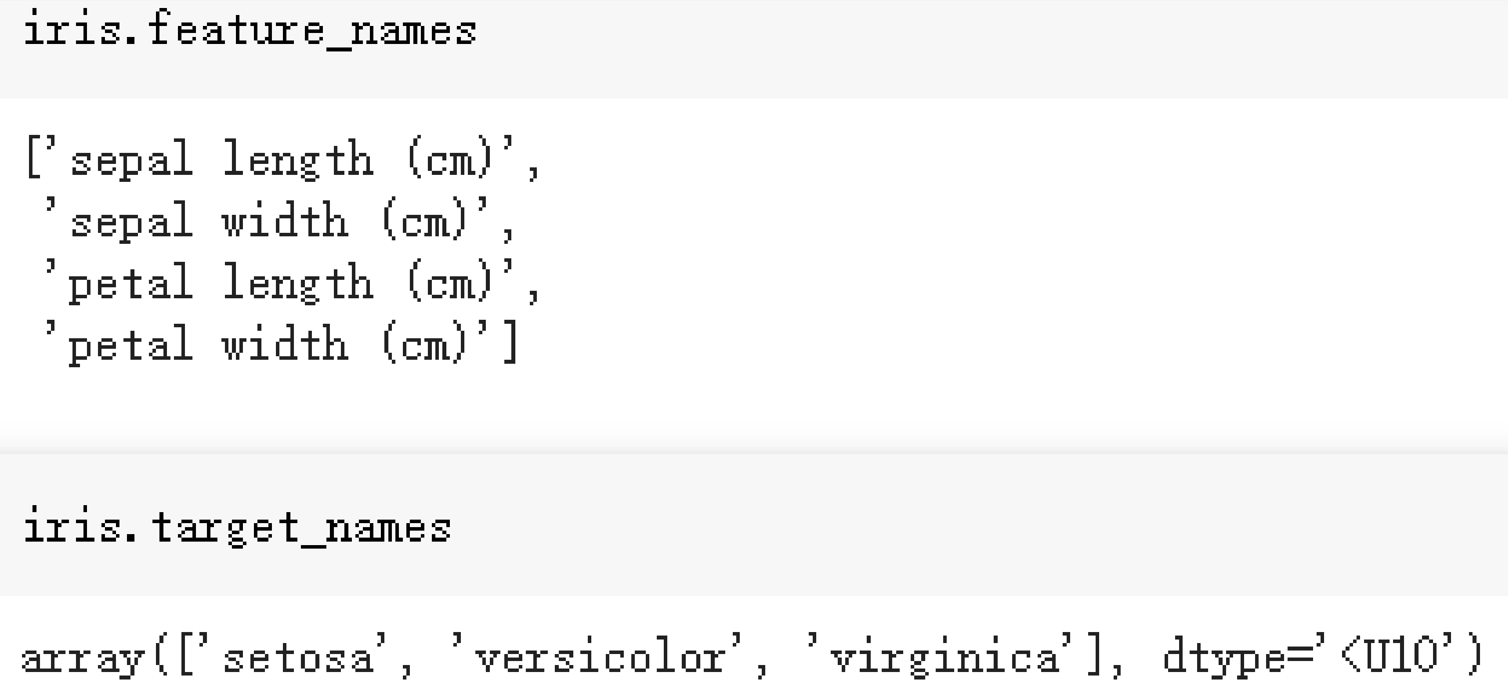
```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

- DESCR: The dataset description
- data: The content
- target: Label
- feature_names: The feature names
- target_names: The target names

Deep Learning

Iris Classification

Iris dataset



```
iris.feature_names
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

```
iris.target_names
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

Deep Learning

Iris Classification

Iris dataset

```
print(iris.DESCR)
```

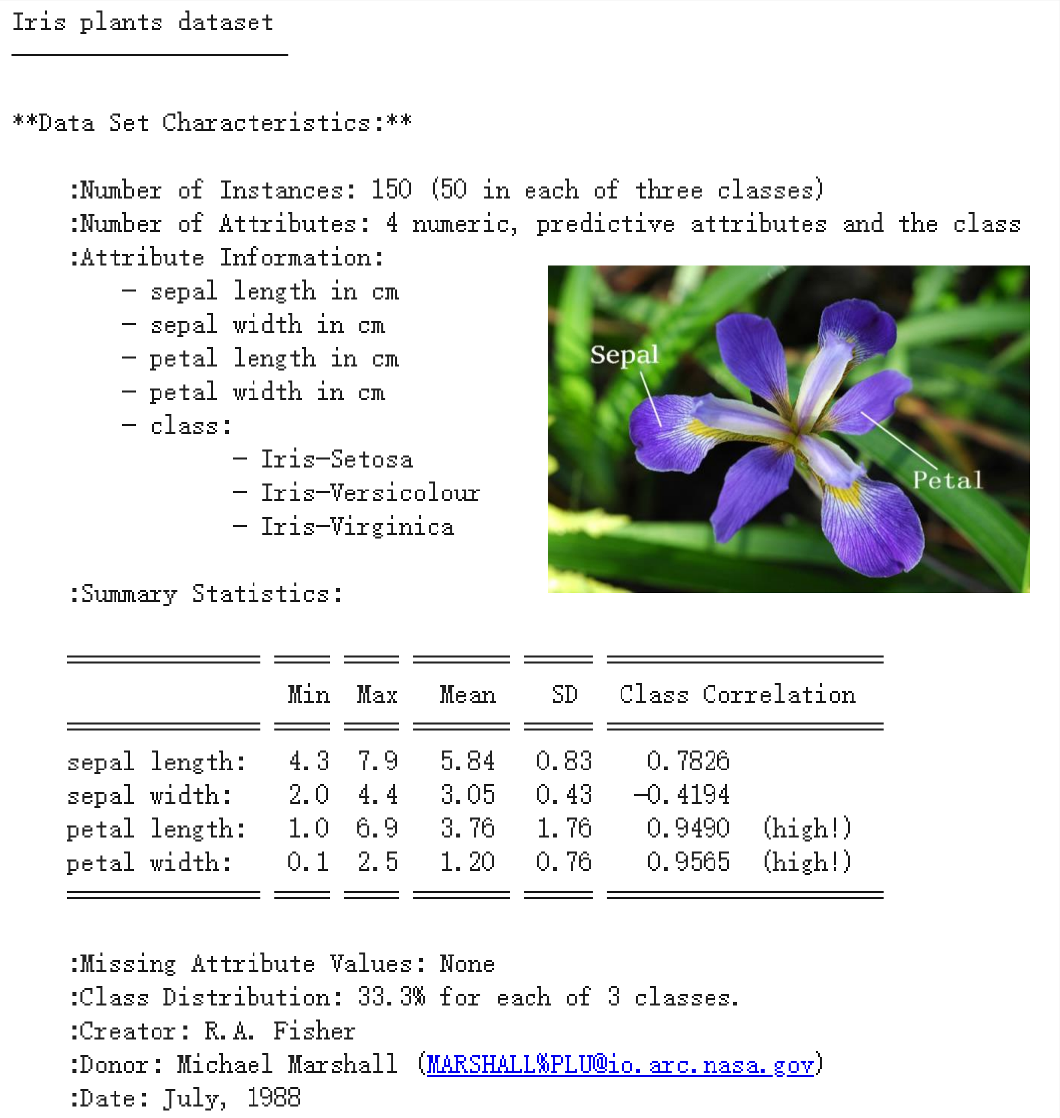
特徵

1. sepal length (花萼長)
2. sepal width (花萼寬)
3. petal length (花瓣長)
4. petal width (花瓣寬)

目標值

鳶尾花種類

(Setosa, Versicolour, Virginica)

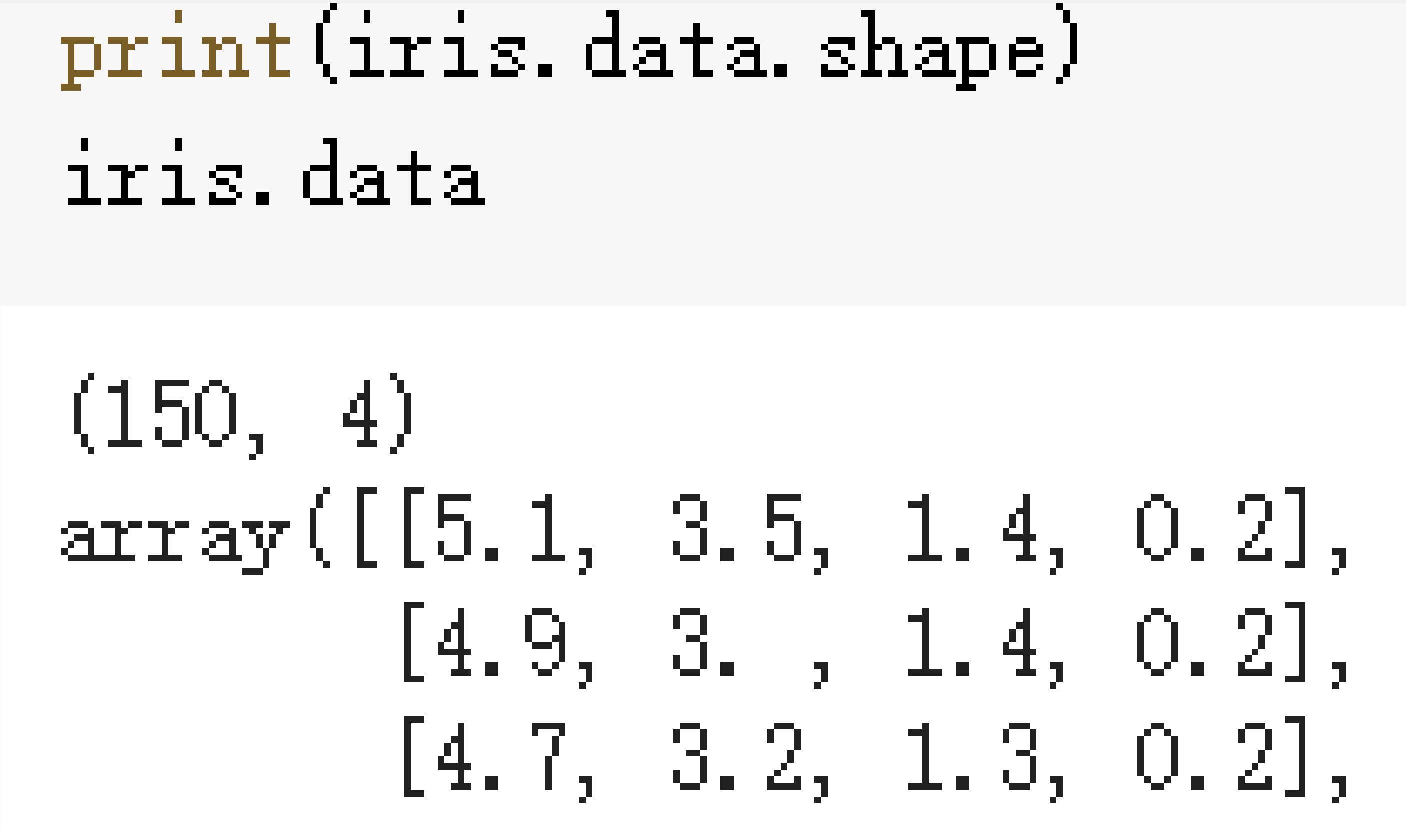


```
Iris plants dataset
____________________

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988
```

Deep Learning

Iris Classification

Iris dataset



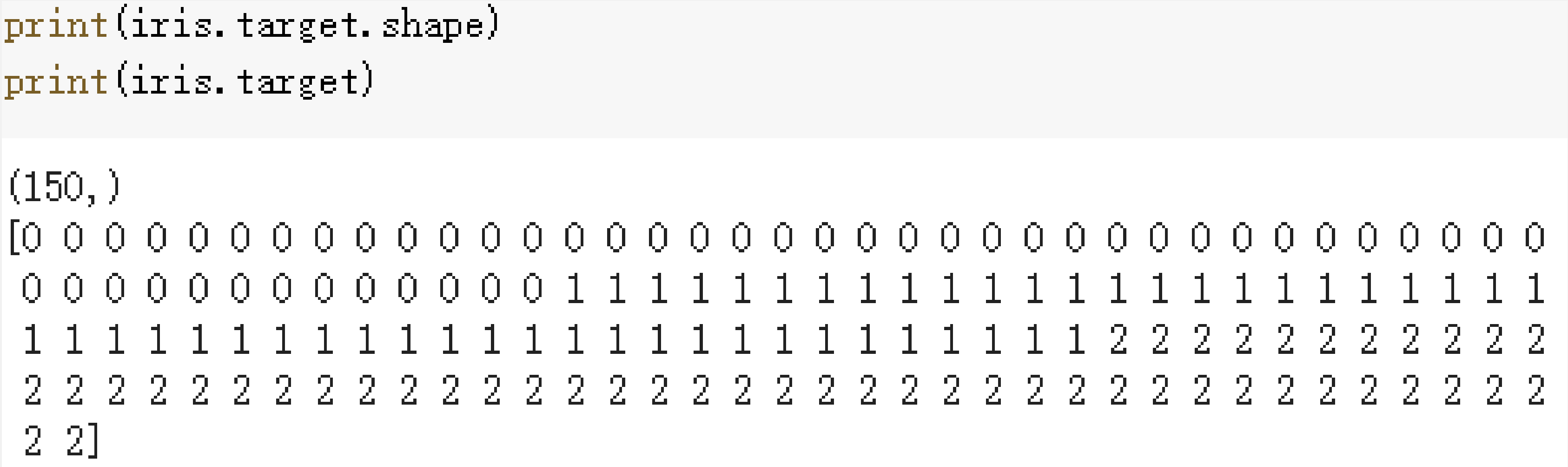
```python
print(iris.data.shape)
iris.data
```

```
(150, 4)
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
```

Deep Learning

Iris Classification

Iris dataset



```
print(iris.target.shape)
print(iris.target)
```

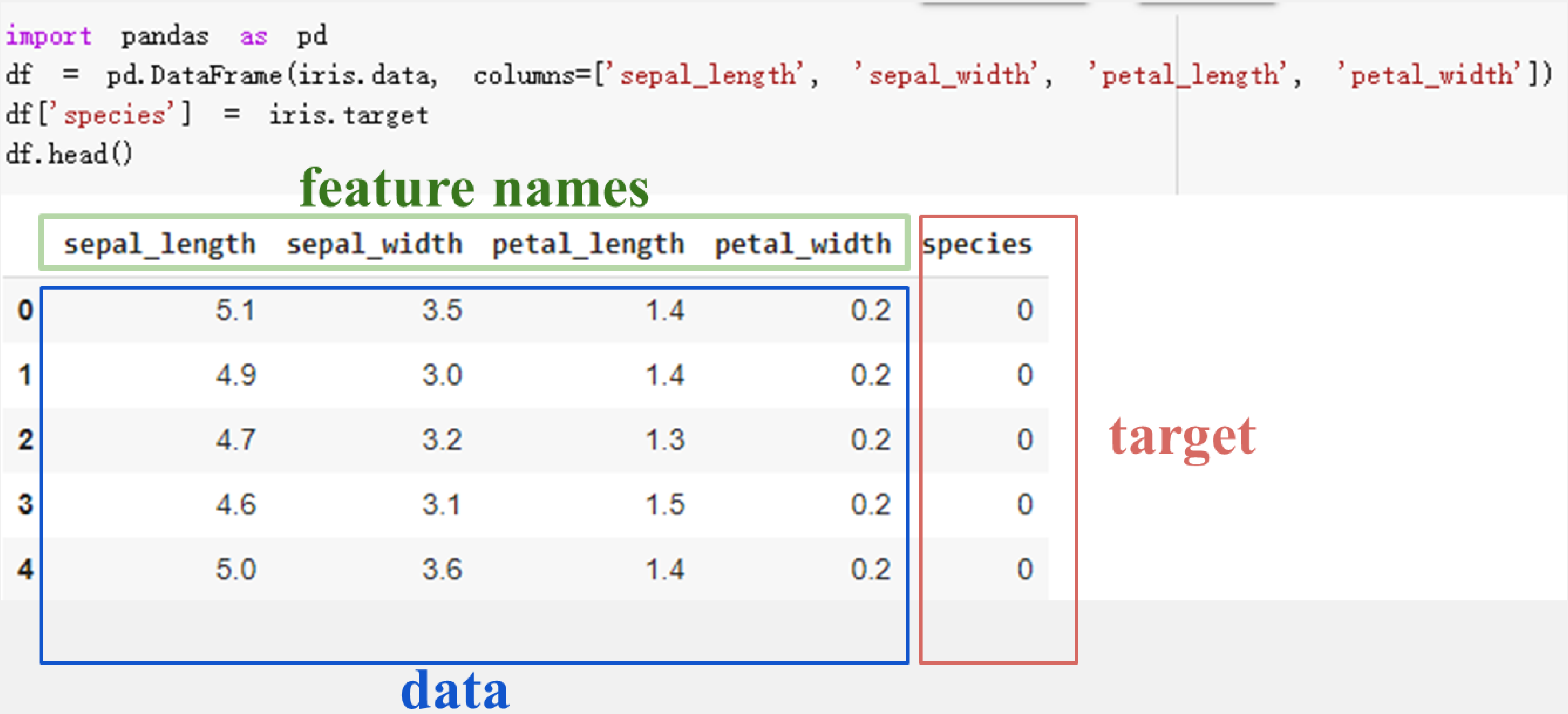
```
(150,)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

- target_names
  - 0: setosa
  - 1: versicolour
  - 2: verginica

Deep Learning

Iris Classification

Iris dataset



```python
import pandas as pd
df = pd.DataFrame(iris.data, columns=['sepal_length', 'sepal_width', 'petal_length', 'petal_width'])
df['species'] = iris.target
df.head()
```

feature names

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

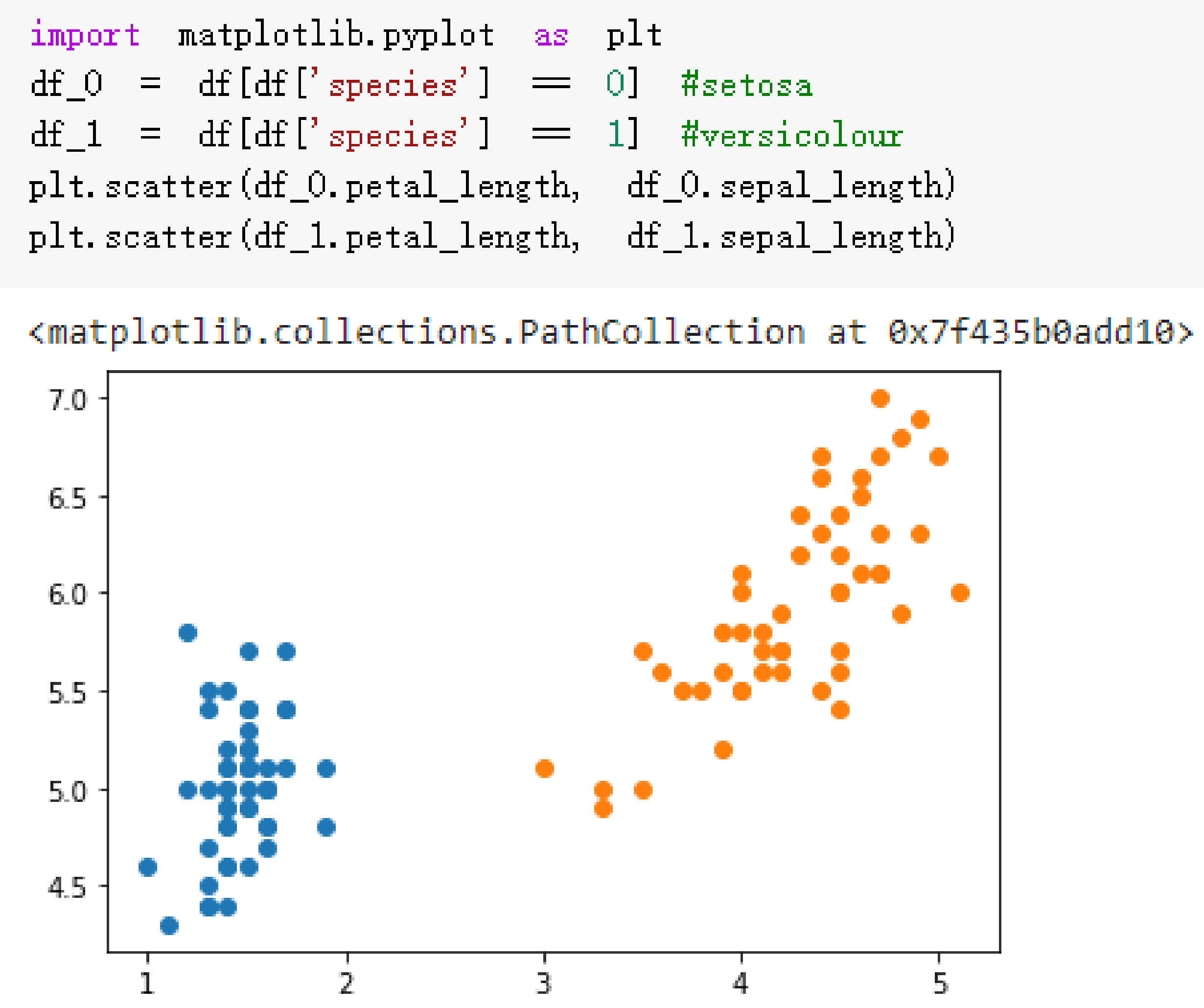
target

data

Deep Learning

Iris Classification

Iris dataset



```
import  matplotlib.pyplot  as  plt
df_0  =  df[df['species']  ==  0]  #setosa
df_1  =  df[df['species']  ==  1]  #versicolour
plt.scatter(df_0.petal_length,  df_0.sepal_length)
plt.scatter(df_1.petal_length,  df_1.sepal_length)
```

```
<matplotlib.collections.PathCollection at 0x7f435b0add10>
```

Deep Learning

Iris Classification

Build perception

$$z = w_1x_1 + w_2x_2 + b$$

$$\sigma(z) \begin{cases} +1 \text{ if } z \geq 0 \\ -1 \text{ otherwise} \end{cases}$$

$$y = \sigma(z)$$

```python
# Python
import numpy as np

class Perceptron():
    """
    learning_rate: float (0.0-1.0)
    n_iter: int
    """
    def __init__(self, learning_rate=1e-2, n_iters=10):
        self.learning_rate = learning_rate
        self.n_iters = n_iters
        self.activation_func = self._uint_step_func
        self.weights = None
        self.bias = None
        self.errors = []

    def _uint_step_func(self, x):
        return np.where(x >= 0.0, 1, -1)


    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        y_prediction = self.activation_func(linear_output)
        return y_prediction
```

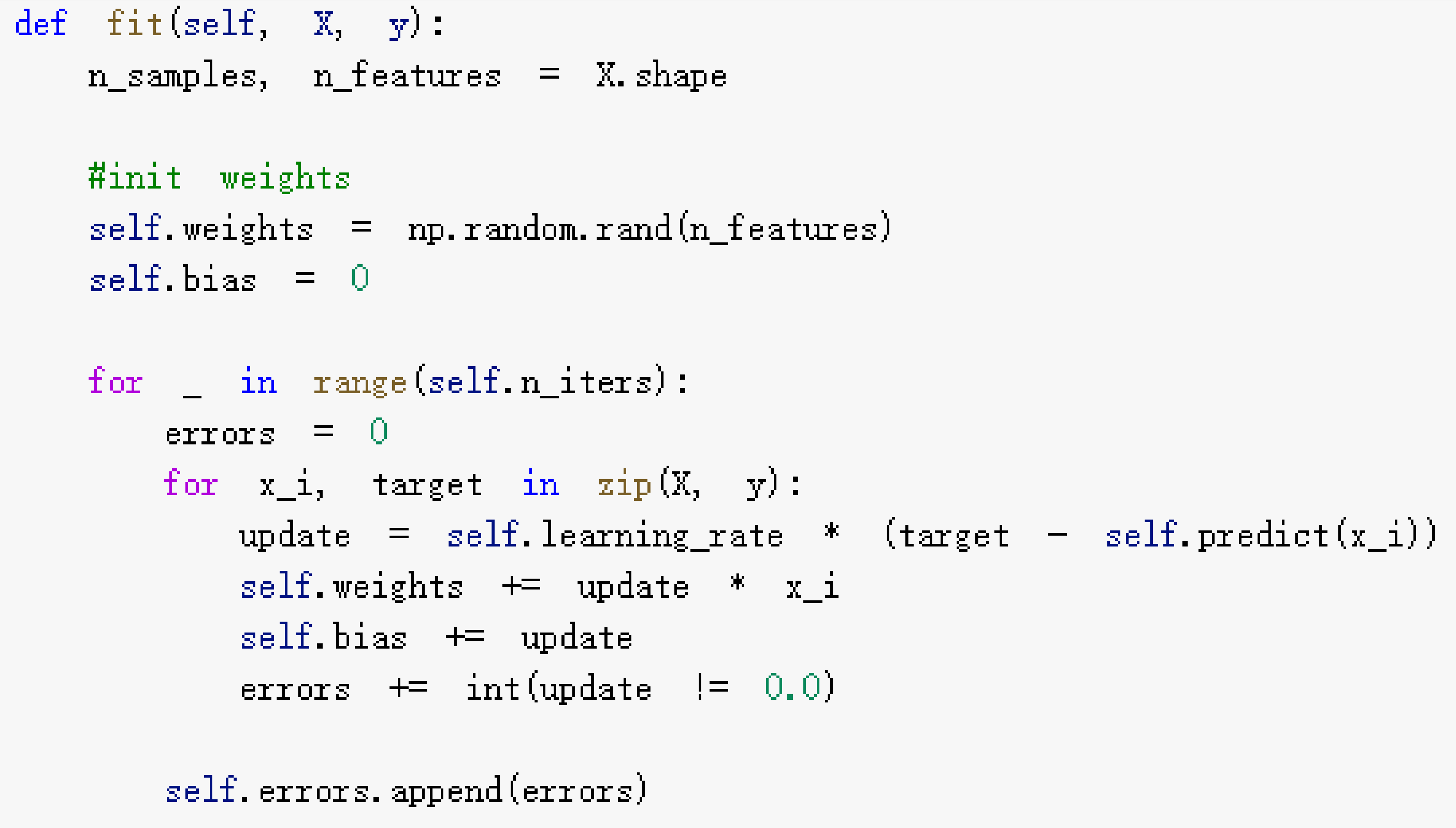
Deep Learning

Iris Classification

Build perception

$$w = w + \Delta w$$
$$\Delta w = \boldsymbol{\eta}(y - \hat{y}) * x$$

$$\Delta w_0 = \boldsymbol{\eta}(y - \text{output}) * x_0$$
$$\Delta w_1 = \boldsymbol{\eta}(y - \text{output}) * x_1$$
$$\Delta w_2 = \boldsymbol{\eta}(y - \text{output}) * x_2$$



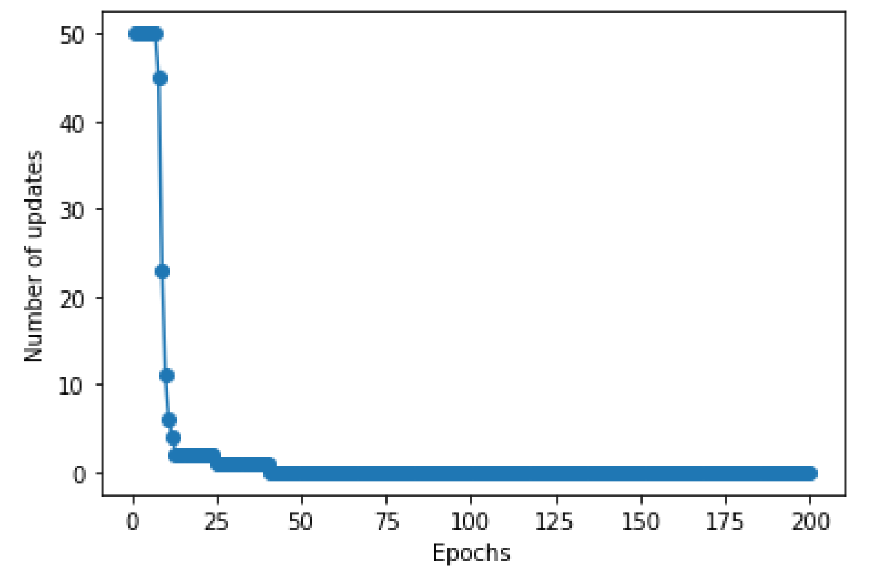
```python
def fit(self, X, y):
    n_samples, n_features = X.shape

    #init weights
    self.weights = np.random.rand(n_features)
    self.bias = 0

    for _ in range(self.n_iters):
        errors = 0
        for x_i, target in zip(X, y):
            update = self.learning_rate * (target - self.predict(x_i))
            self.weights += update * x_i
            self.bias += update
            errors += int(update != 0.0)

        self.errors.append(errors)
```

Deep Learning

Iris Classification

Training

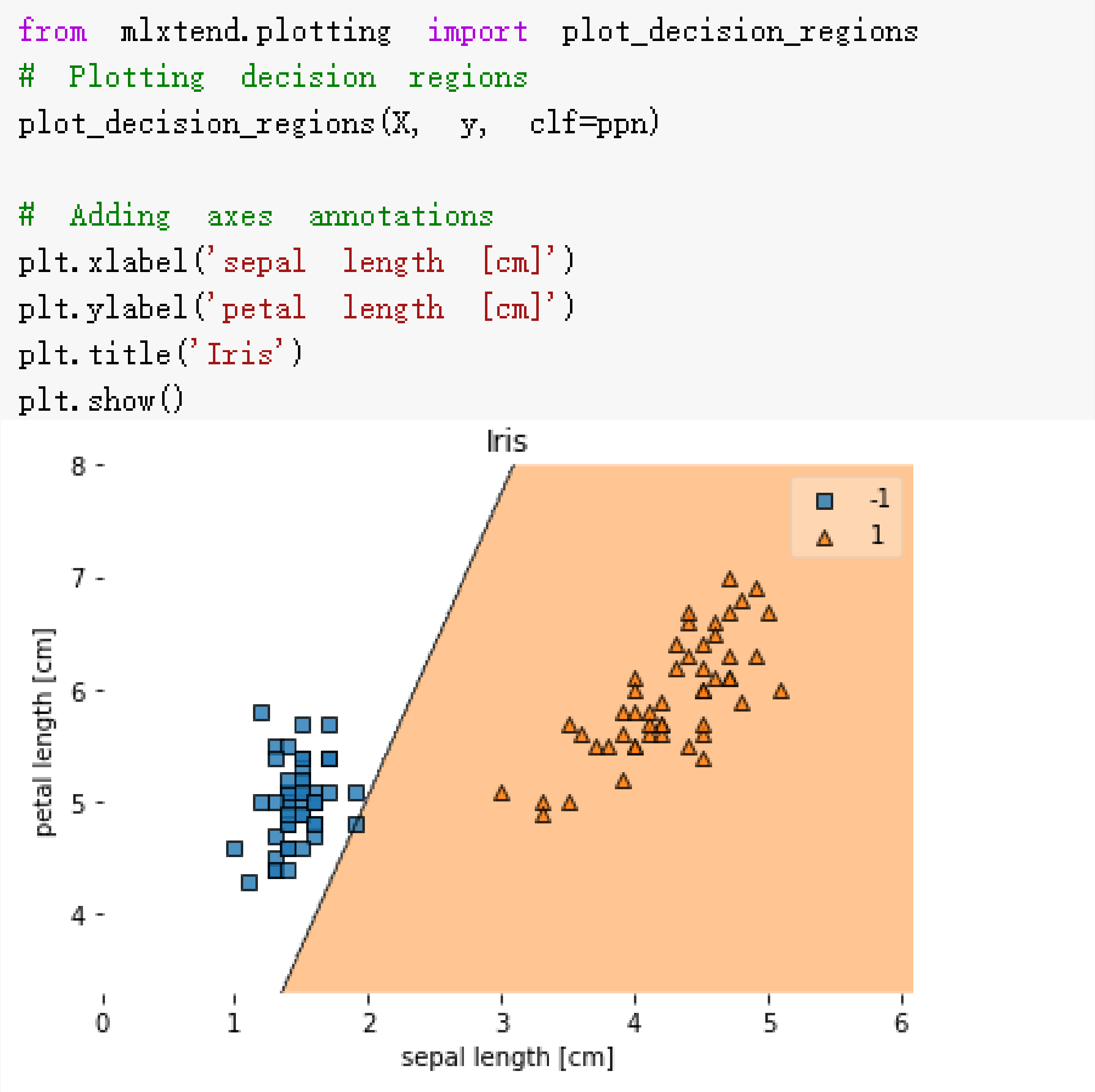
```
# prepare data
df_data = df[(df.species == 0)|(df.species == 1)].sample(frac=1)
X = df_data[['petal_length', 'sepal_length']].values
y = np.where(df_data.species.values == 0, -1, 1)

#train model
ppn = Perceptron(learning_rate=1e-4, n_iters=200)
ppn.fit(X, y)

#draw error log
plt.plot(range(1, len(ppn.errors)+1), ppn.errors, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.show()
```



Deep Learning

Iris Classification

Analysis



```
from mlxtend.plotting import plot_decision_regions
# Plotting decision regions
plot_decision_regions(X, y, clf=ppn)

# Adding axes annotations
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.title('Iris')
plt.show()
```

Deep Learning

Iris Classification

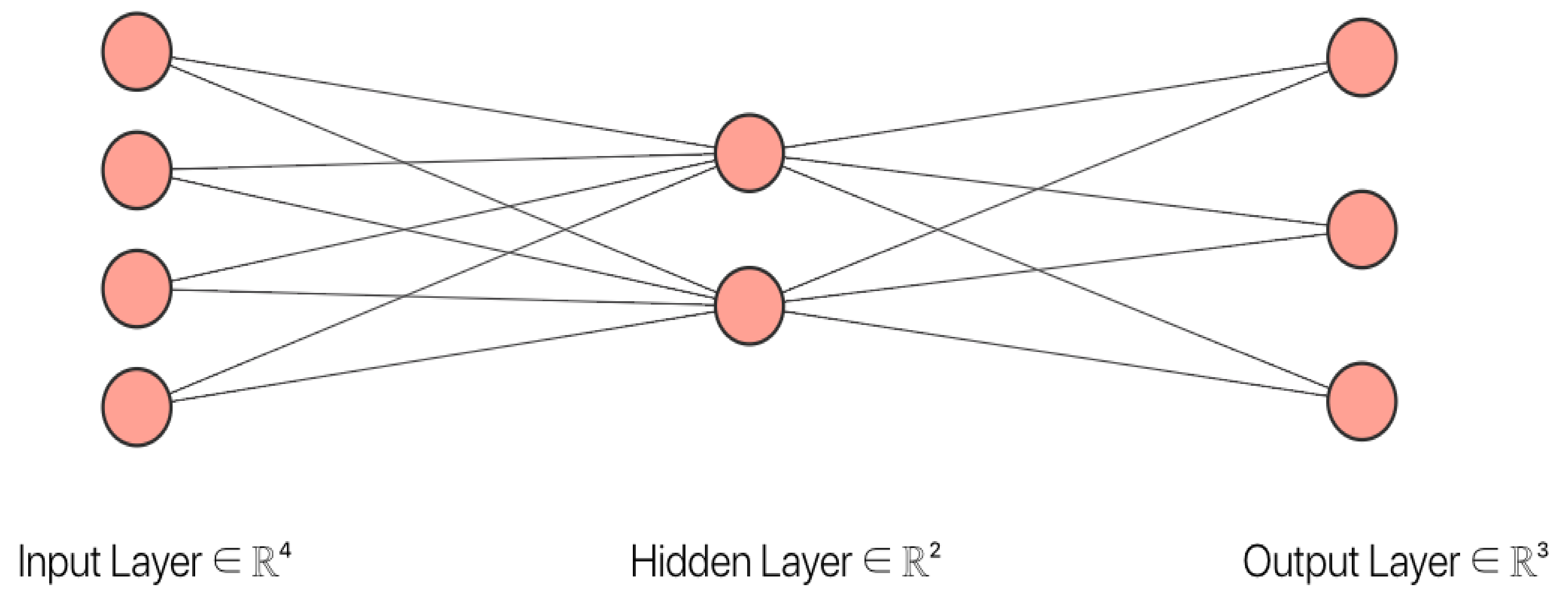
Practice - using different training data and parameters

- Different assortment
- Different features
- Different learning rate

Deep Learning

Iris Classification

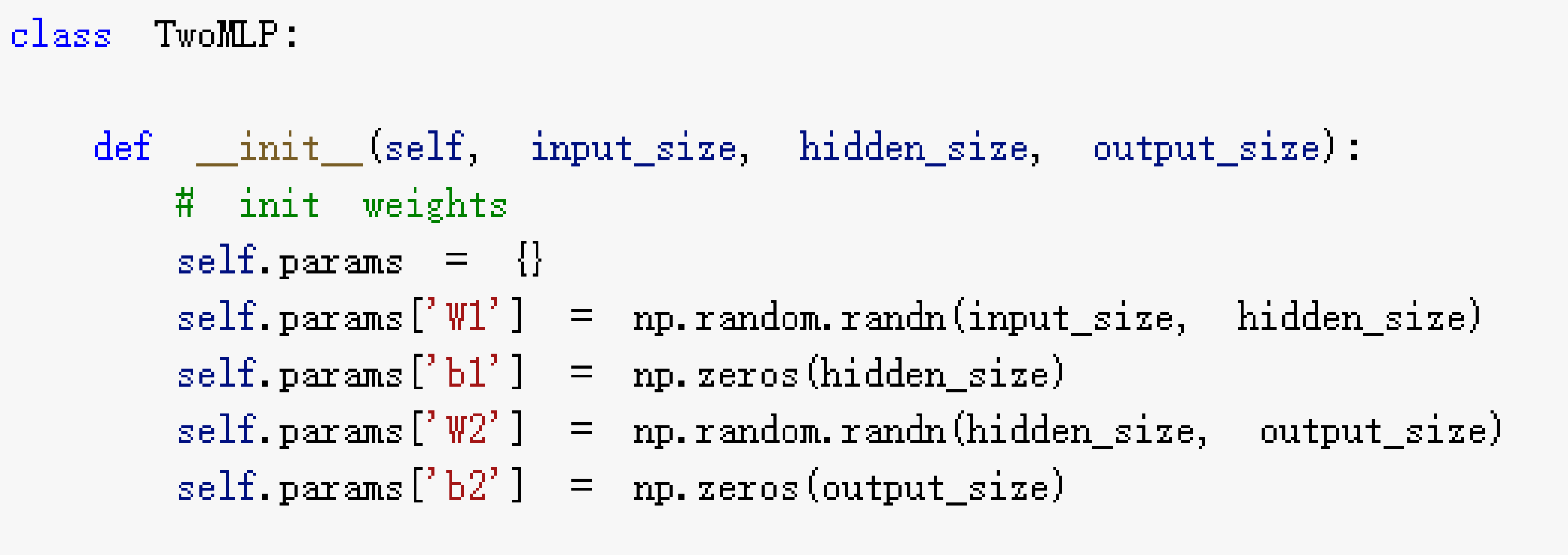
Build neural network manually



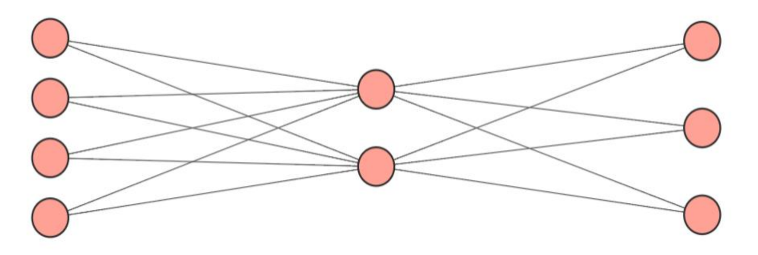
Deep Learning

Iris Classification

Build neural network manually



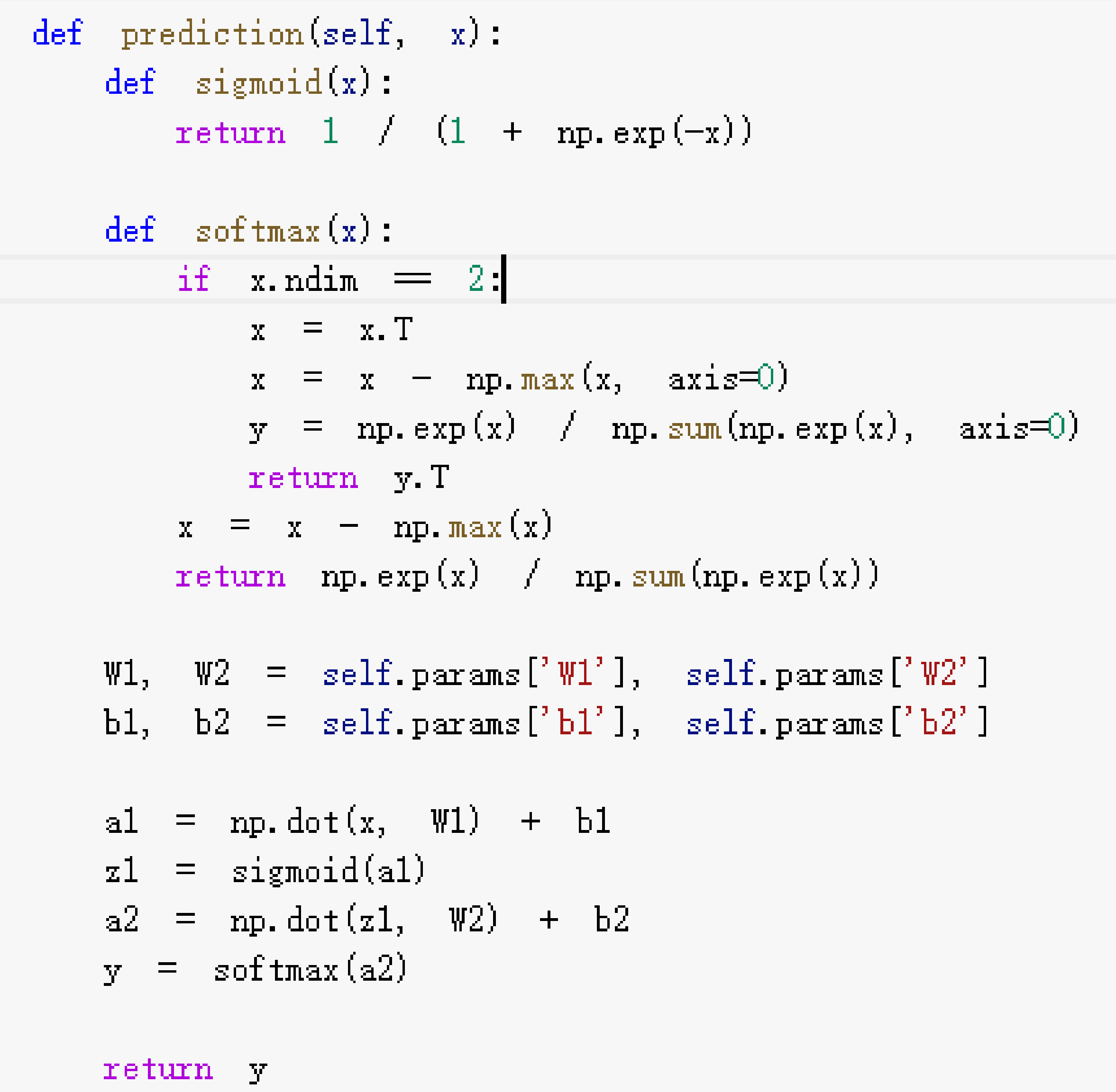
```
class  TwoMLP:

    def  __init__(self,  input_size,  hidden_size,  output_size):
        #  init  weights
        self.params  =  {}
        self.params[’W1’]  =  np.random.randn(input_size,  hidden_size)
        self.params[’b1’]  =  np.zeros(hidden_size)
        self.params[’W2’]  =  np.random.randn(hidden_size,  output_size)
        self.params[’b2’]  =  np.zeros(output_size)
```



4 x 2   **2 x 3**

Deep Learning

Iris Classification



```
def prediction(self, x):
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    def softmax(x):
        if x.ndim == 2:
            x = x.T
            x = x - np.max(x, axis=0)
            y = np.exp(x) / np.sum(np.exp(x), axis=0)
            return y.T
        x = x - np.max(x)
        return np.exp(x) / np.sum(np.exp(x))

    W1, W2 = self.params['W1'], self.params['W2']
    b1, b2 = self.params['b1'], self.params['b2']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    y = softmax(a2)

    return y
```

Deep Learning

Iris Classification

```python
def loss(self, x, t):
    def cross_entropy_error(y, t):
        if y.ndim == 1:
            t = t.reshape(1, t.size)
            y = y.reshape(1, y.size)
        if t.size == y.size:
            t = t.argmax(axis=1)
        batch_size = y.shape[0]
        return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size

    return cross_entropy_error(self.prediction(x), t)
```

Deep Learning

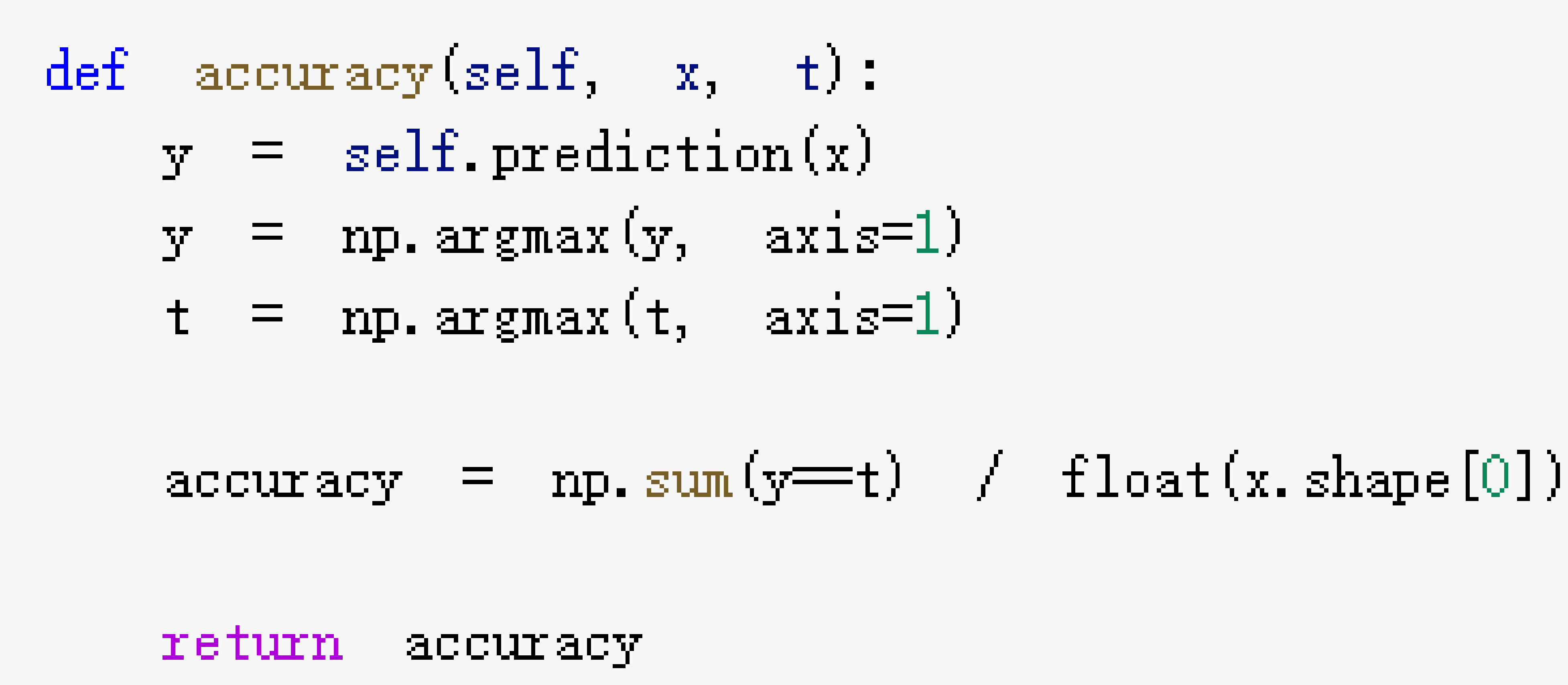
Iris Classification

```
def numerical_gradient(f, x):
        h = 1e-2
        grad = np.zeros_like(x)

        it = np.nditer(x, flags=[’multi_index’], op_flags=[’readwrite’])
        while not it.finished:
            idx = it.multi_index
            tmp_val = x[idx]
            x[idx] = float(tmp_val) + h
            fxh1 = f(x)

            x[idx] = float(tmp_val) - h
            fxh2 = f(x)

            grad[idx] = (fxh1 - fxh2) / (2*h)
            x[idx] = tmp_val
            it.iternext()
```

```
def compute_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)
    grads = {}
    grads[’W1’] = numerical_gradient(loss_W, self.params[’W1’])
    grads[’b1’] = numerical_gradient(loss_W, self.params[’b1’])
    grads[’W2’] = numerical_gradient(loss_W, self.params[’W2’])
    grads[’b2’] = numerical_gradient(loss_W, self.params[’b2’])

    return grads
```

Deep Learning

Iris Classification



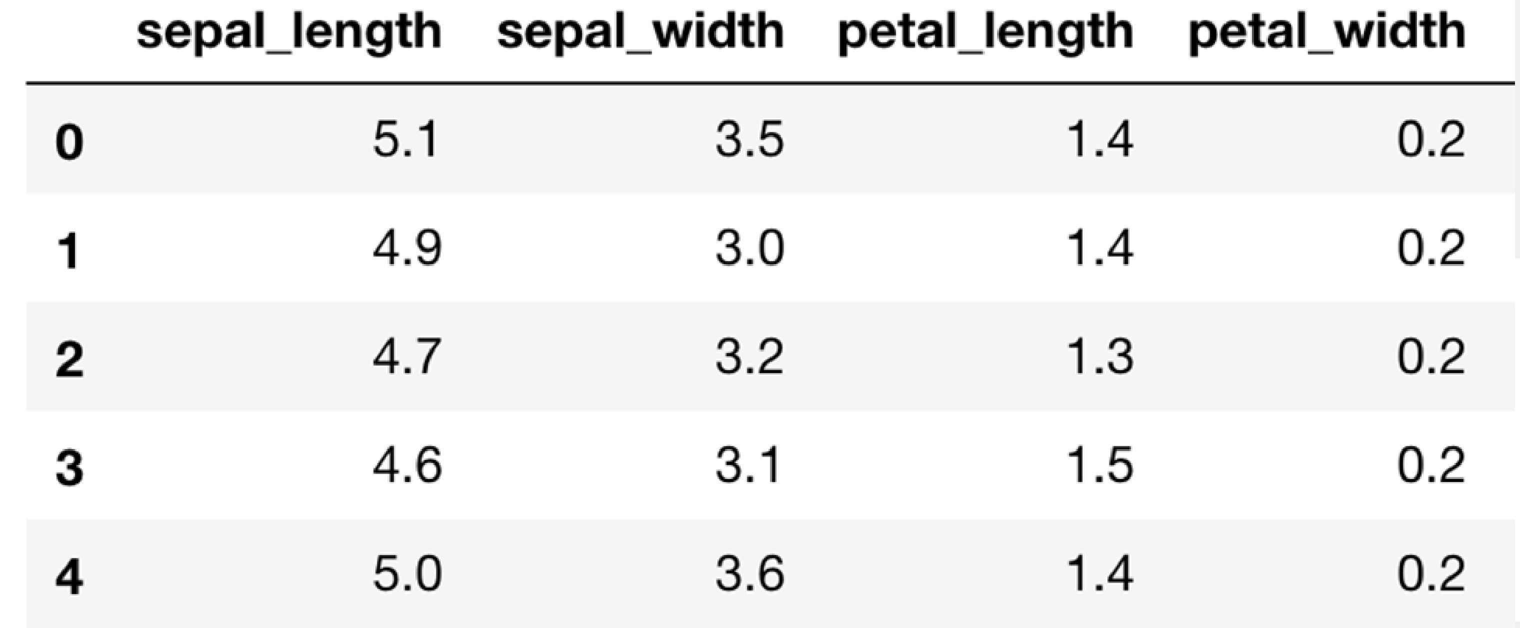
```
def accuracy(self, x, t):
    y = self.prediction(x)
    y = np.argmax(y, axis=1)
    t = np.argmax(t, axis=1)

    accuracy = np.sum(y==t) / float(x.shape[0])

    return accuracy
```

Deep Learning

Iris Classification

```
twomlp = TwoMLP(input_size=4, hidden_size=2, output_size=3)
```

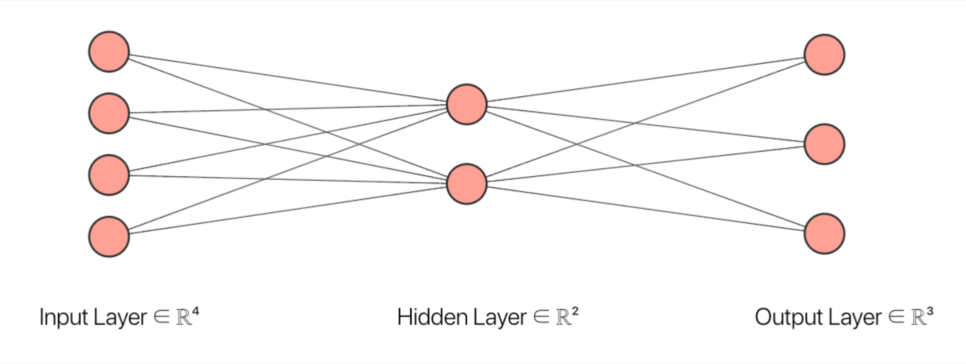
4



	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Target

0: [1, 0, 0]
1: [0, 1, 0]
2: [0, 0, 1]



Deep Learning

Iris Classification

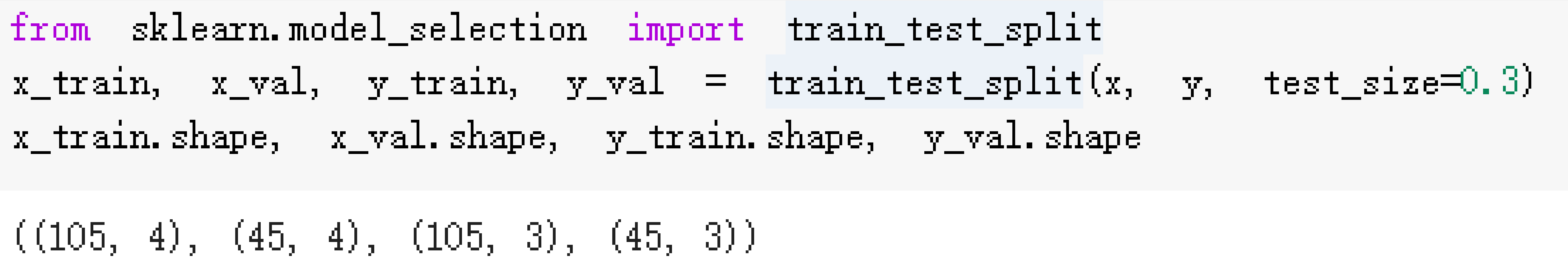
```python
def one_hot_encoding(x):
    from sklearn.preprocessing import OneHotEncoder
    enc = OneHotEncoder()
    enc.fit(x)
    return enc.transform(x).toarray()

iris = load_iris()
x = iris.data
y = iris.target.reshape(-1,1)
y = one_hot_encoding(y)
```

Target

0-> [1, 0, 0]
1-> [0, 1, 0]
2-> [0, 0, 1]

Deep Learning

Iris Classification



```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x, y, test_size=0.3)
x_train.shape, x_val.shape, y_train.shape, y_val.shape
```

```
((105, 4), (45, 4), (105, 3), (45, 3))
```

參考書

70%

模擬考

30%

Deep Learning

Iris Classification

```
train_loss = []
train_acc = []
val_acc = []

iterations = 10000
learning_rate = 1e-4
train_size = x_train.shape[0]
batch_size = 8
iter_per_epoch = max(train_size/batch_size, 1)
```

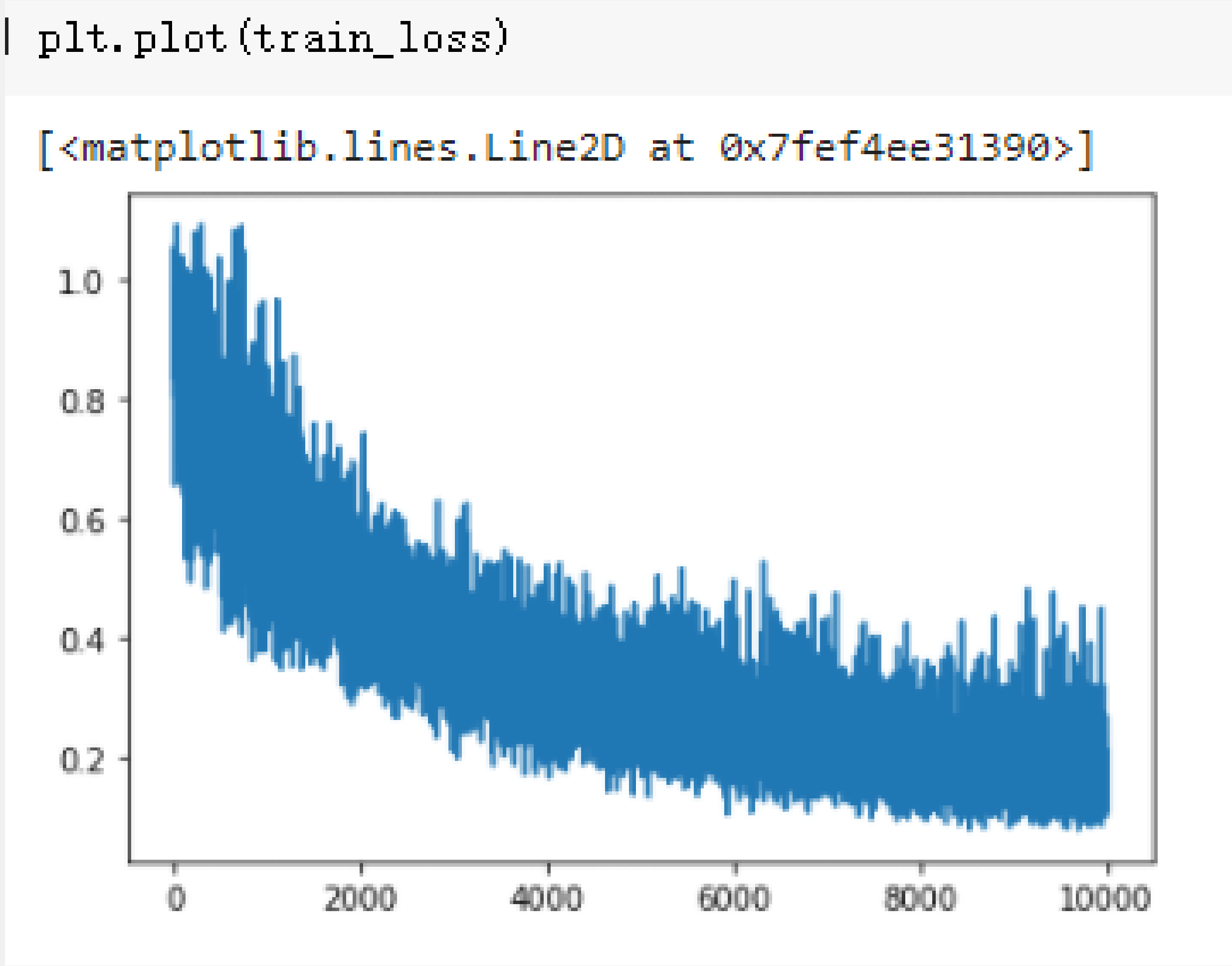
Deep Learning

Iris Classification

```
for i in range(iterations):
    # Batch sample
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    y_batch = y_train[batch_mask]

    # Compute gradient
    grad = twomlp.compute_gradient(x_batch, y_batch)

    # Update weights
    for key in ('W1', 'b1', 'W2', 'b2'):
        twomlp.params[key] -= learning_rate * grad[key]

    # Record training loss
    loss = twomlp.loss(x_batch, y_batch)
    # print("loss: %f" %(loss))
    train_loss.append(loss)

    # Record learning and validation accuracy in each iteration
    if i % iter_per_epoch == 0:
        train_acc.append(twomlp.accuracy(x_train, y_train))
        val_acc.append(twomlp.accuracy(x_val, y_val))
        print(f'iter: {i}: train_acc: {train_acc[-1]}, val_acc: {val_acc[-1]}\n')
```

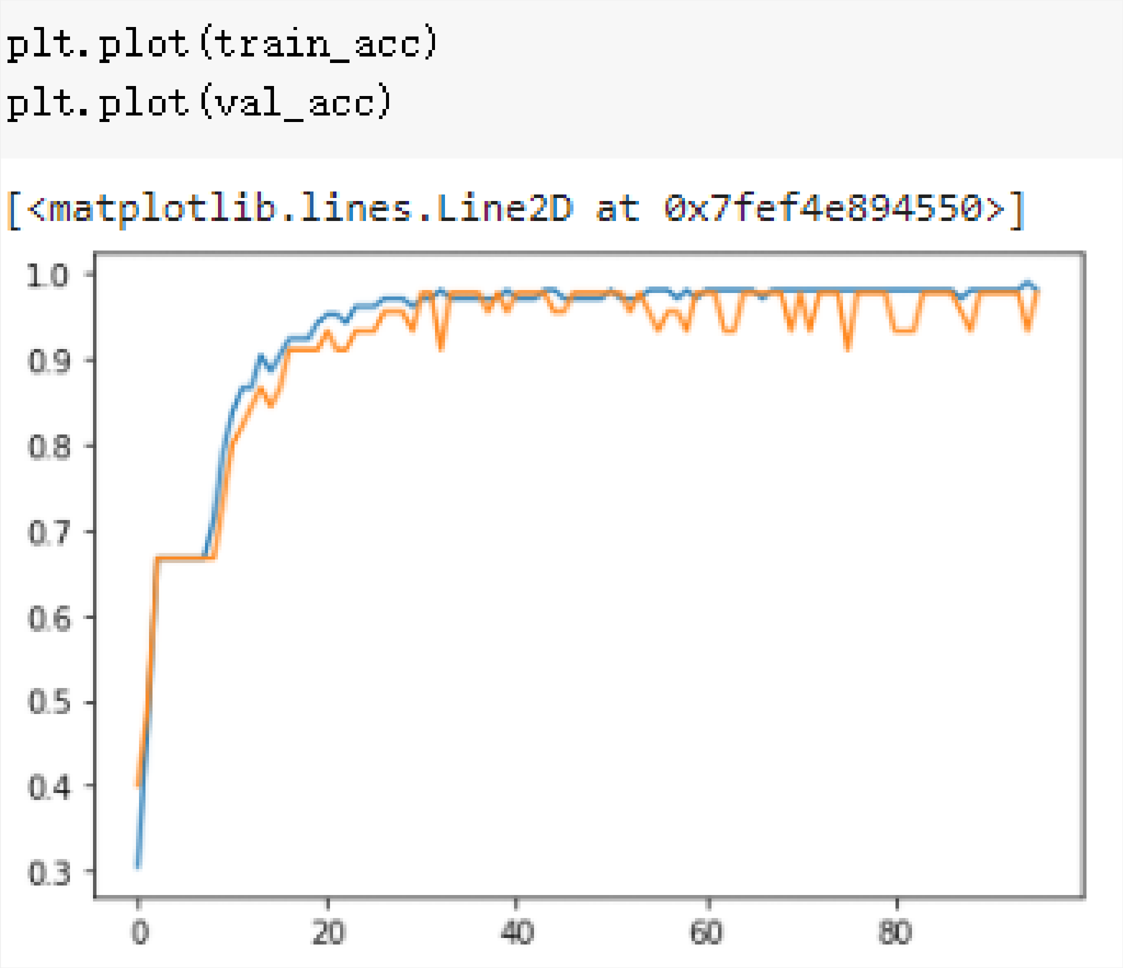
Deep Learning

Iris Classification



```
plt.plot(train_loss)
```

```
[<matplotlib.lines.Line2D at 0x7fef4ee31390>]
```



```
plt.plot(train_acc)
plt.plot(val_acc)
```

```
[<matplotlib.lines.Line2D at 0x7fef4e894550>]
```

Deep Learning

Iris Classification

Practice - using different training parameters

- Different learning rate
- Different batch_size
- Different hidden_size

Backpropagation

Chain Rule

$z = t^2$

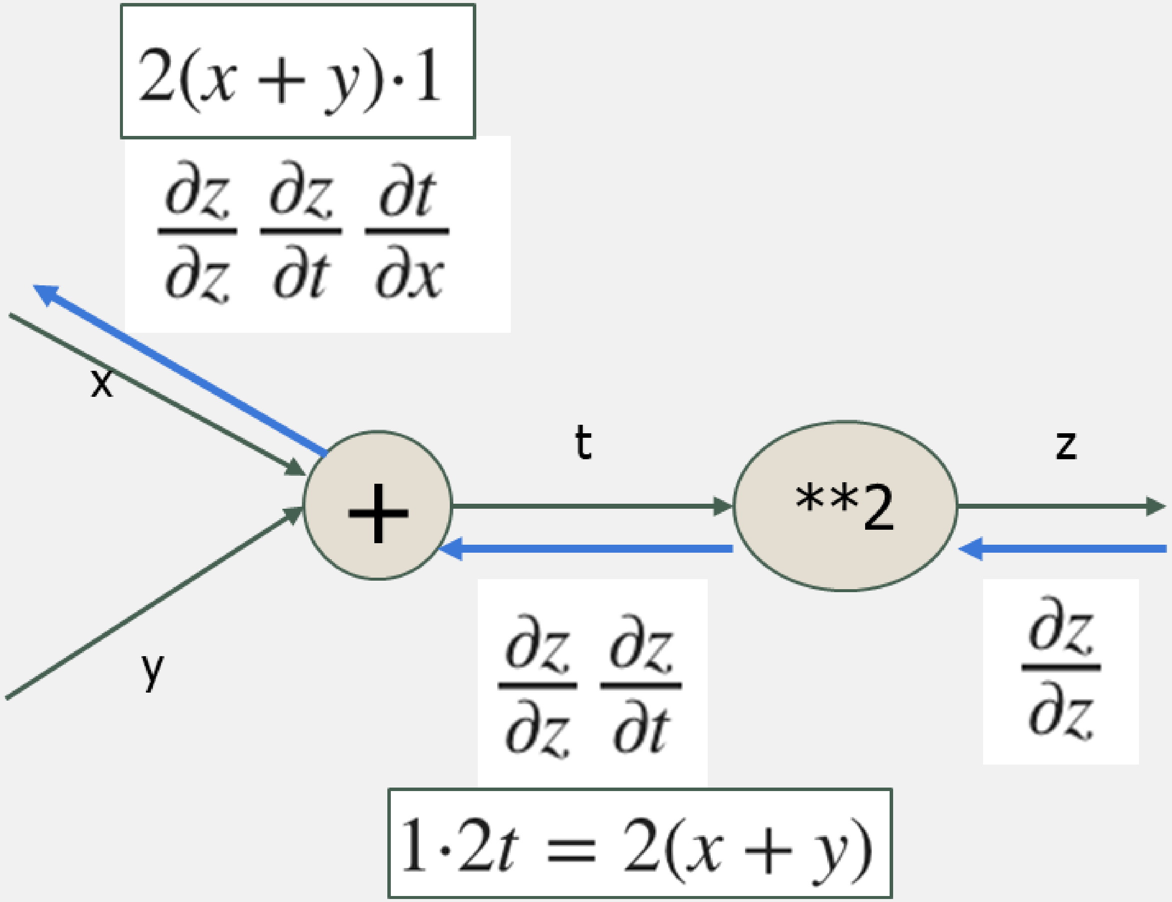
$t = x + y$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$$

$$\frac{\partial z}{\partial t} = 2t$$

$$\frac{\partial t}{\partial x} = 1$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x+y)$$



$$\boxed{2(x+y)\cdot 1}$$

$$\frac{\partial z}{\partial z}\frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$$

$$\frac{\partial z}{\partial z}\frac{\partial z}{\partial t}$$

$$\boxed{1\cdot 2t = 2(x+y)}$$

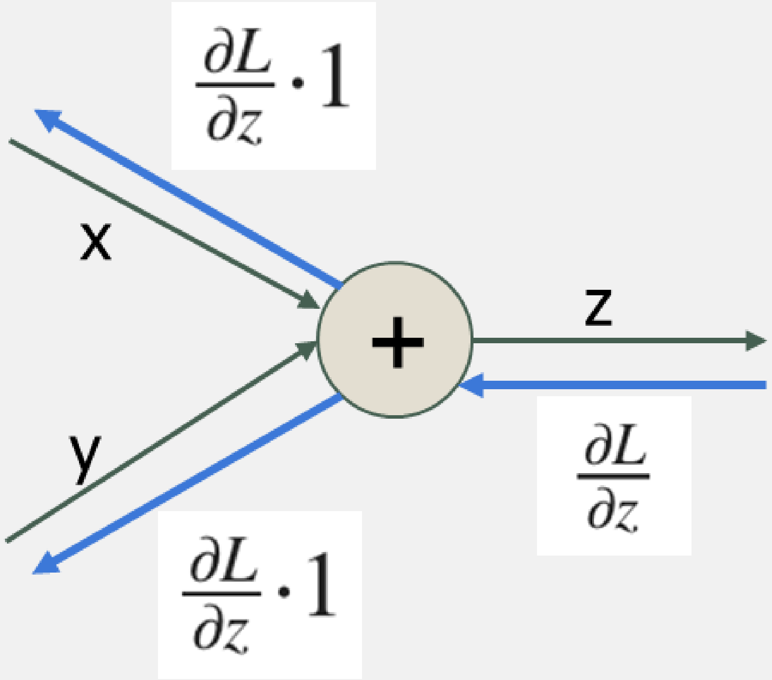
$$\frac{\partial z}{\partial z}$$

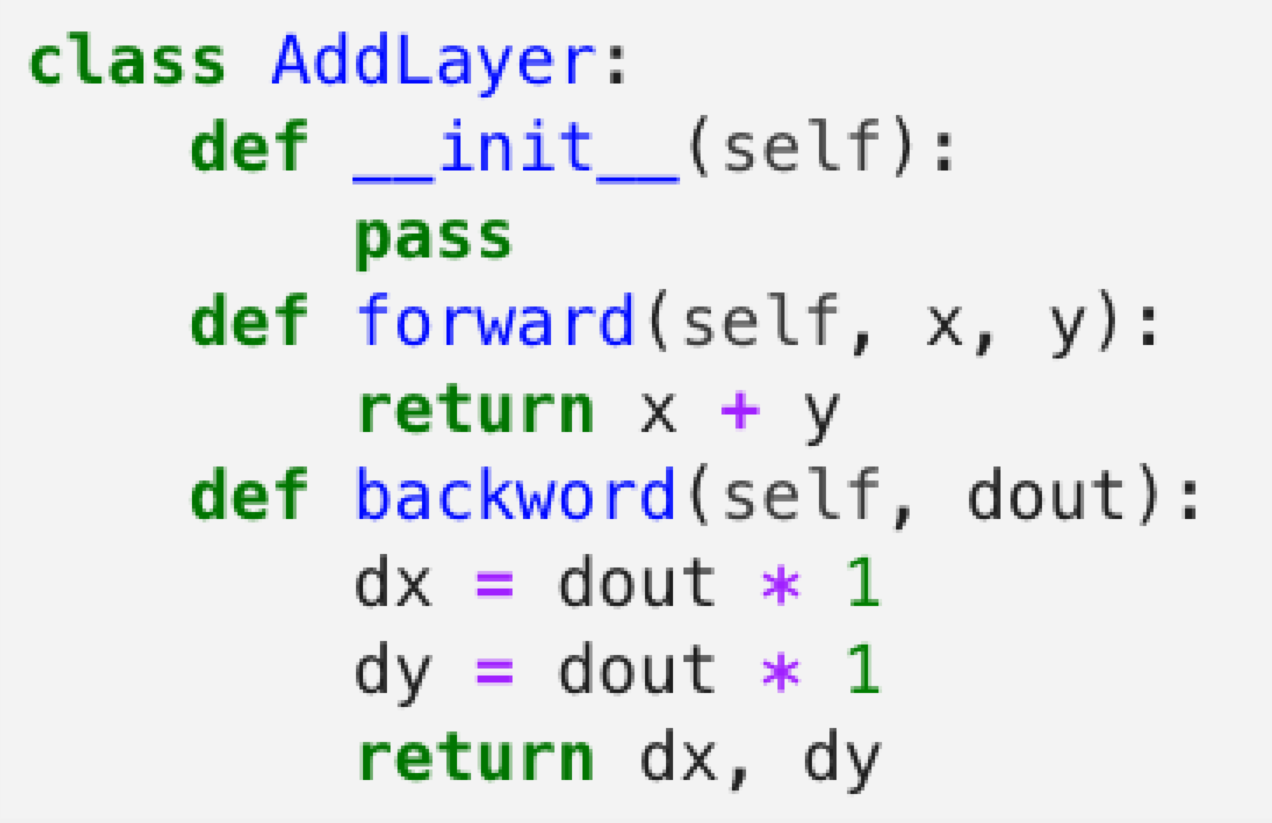
Backpropagation

Add

$$\frac{\partial z}{\partial x} = 1$$

$$\frac{\partial z}{\partial y} = 1$$





```
class AddLayer:
    def __init__(self):
        pass
    def forward(self, x, y):
        return x + y
    def backword(self, dout):
        dx = dout * 1
        dy = dout * 1
        return dx, dy
```

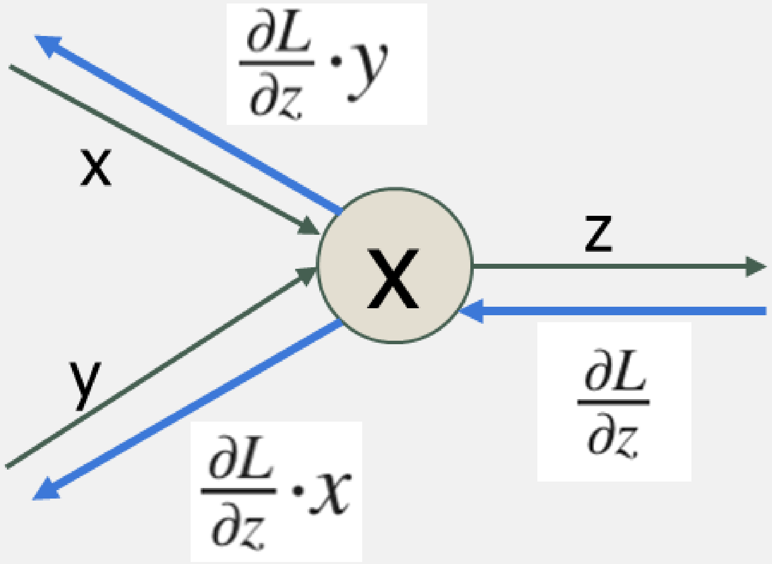
Backpropagation

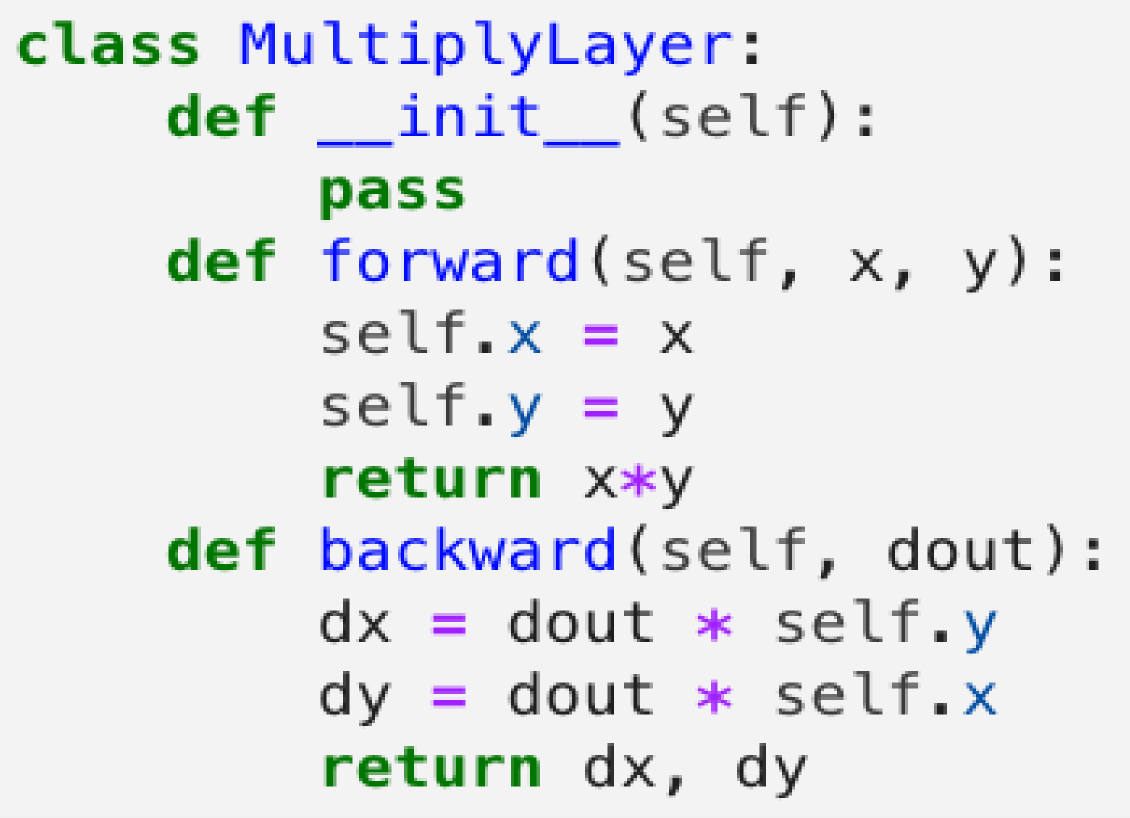
Multiply

$$z = xy$$

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

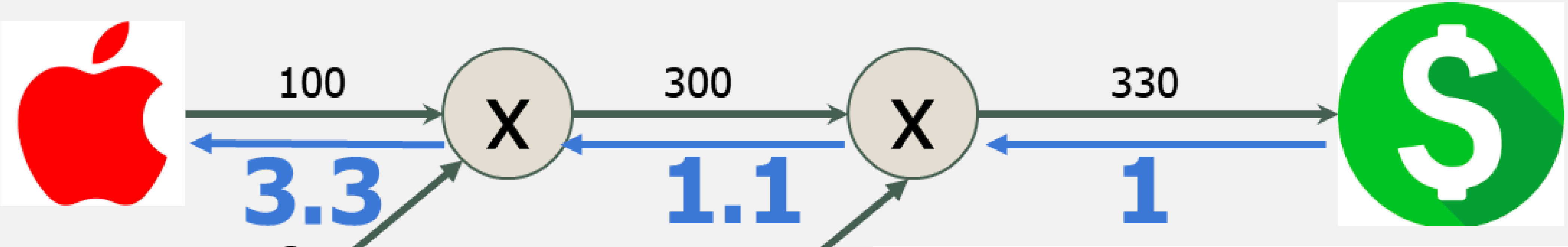
$$\frac{\partial z}{\partial y} = x$$

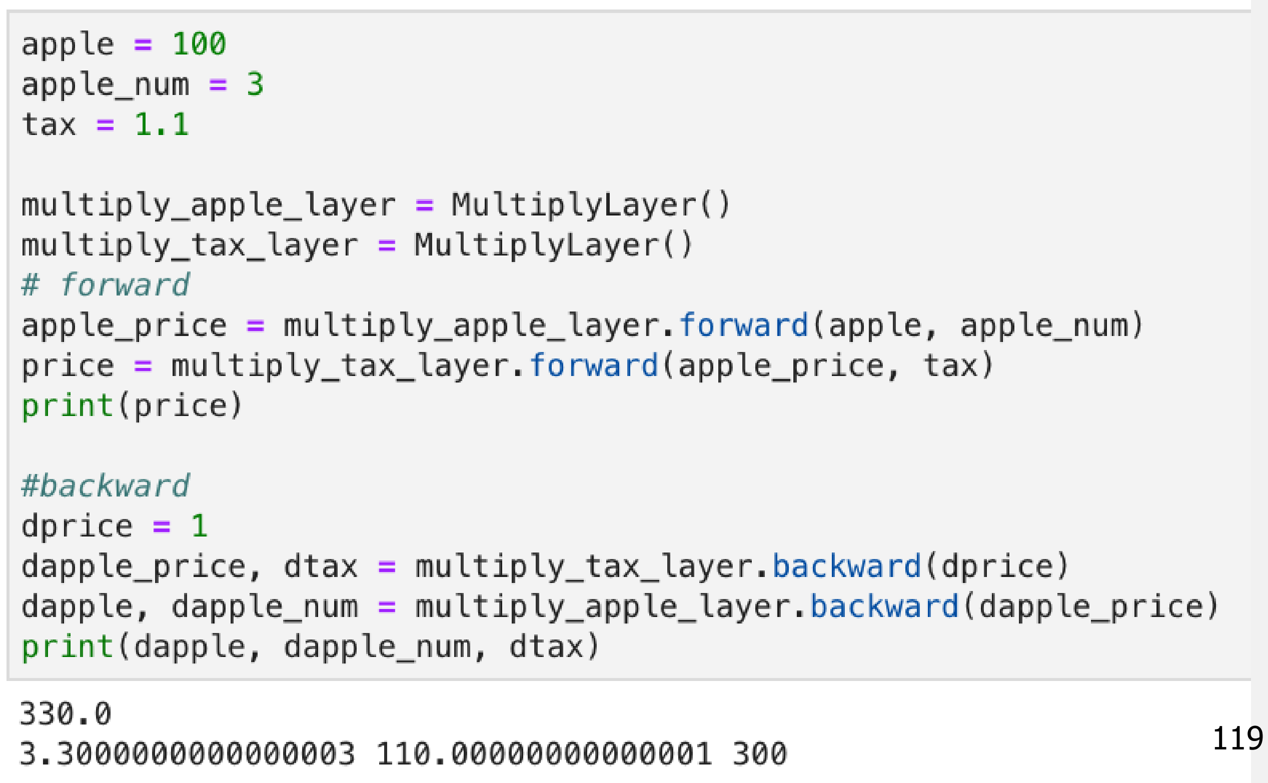




```
class MultiplyLayer:
    def __init__(self):
        pass
    def forward(self, x, y):
        self.x = x
        self.y = y
        return x*y
    def backward(self, dout):
        dx = dout * self.y
        dy = dout * self.x
        return dx, dy
```

Backpropagation



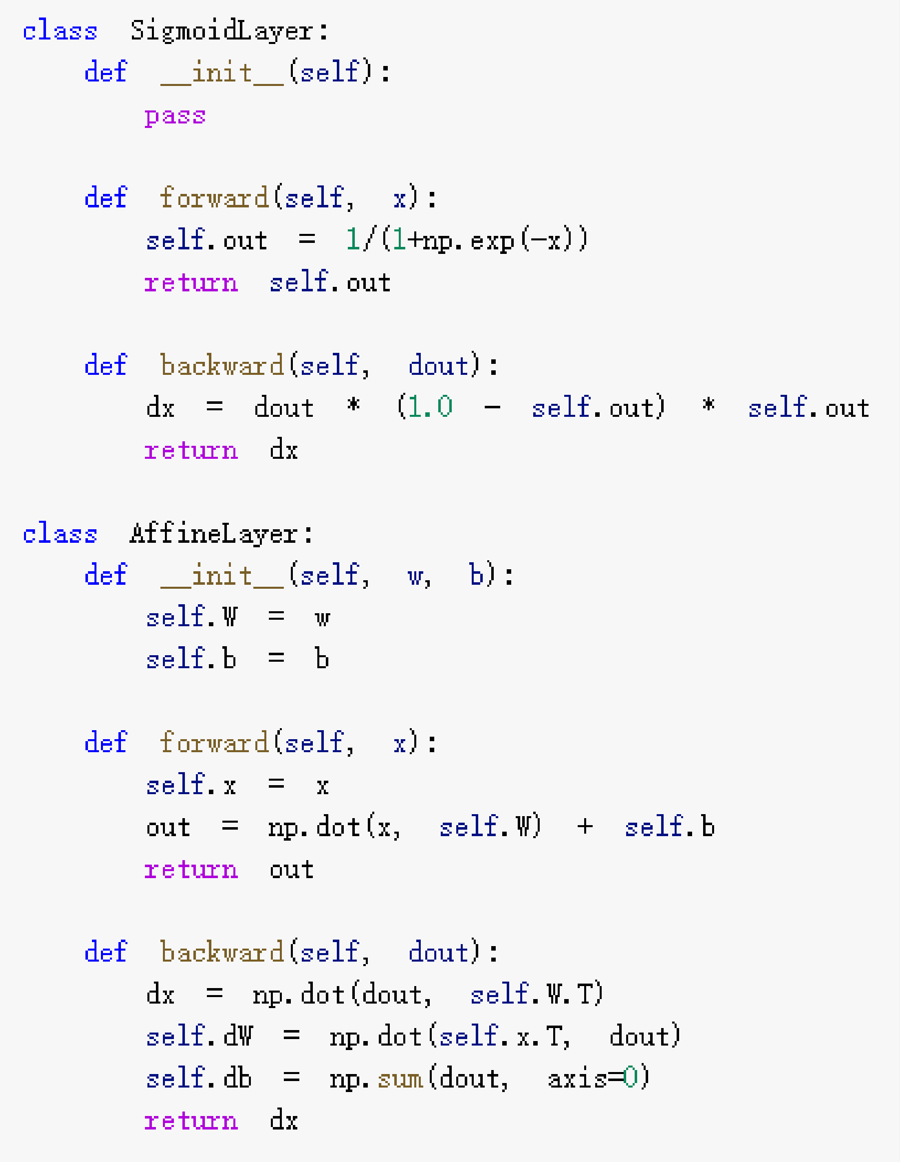


```
apple = 100
apple_num = 3
tax = 1.1

multiply_apple_layer = MultiplyLayer()
multiply_tax_layer = MultiplyLayer()
# forward
apple_price = multiply_apple_layer.forward(apple, apple_num)
price = multiply_tax_layer.forward(apple_price, tax)
print(price)

#backward
dprice = 1
dapple_price, dtax = multiply_tax_layer.backward(dprice)
dapple, dapple_num = multiply_apple_layer.backward(dapple_price)
print(dapple, dapple_num, dtax)
```

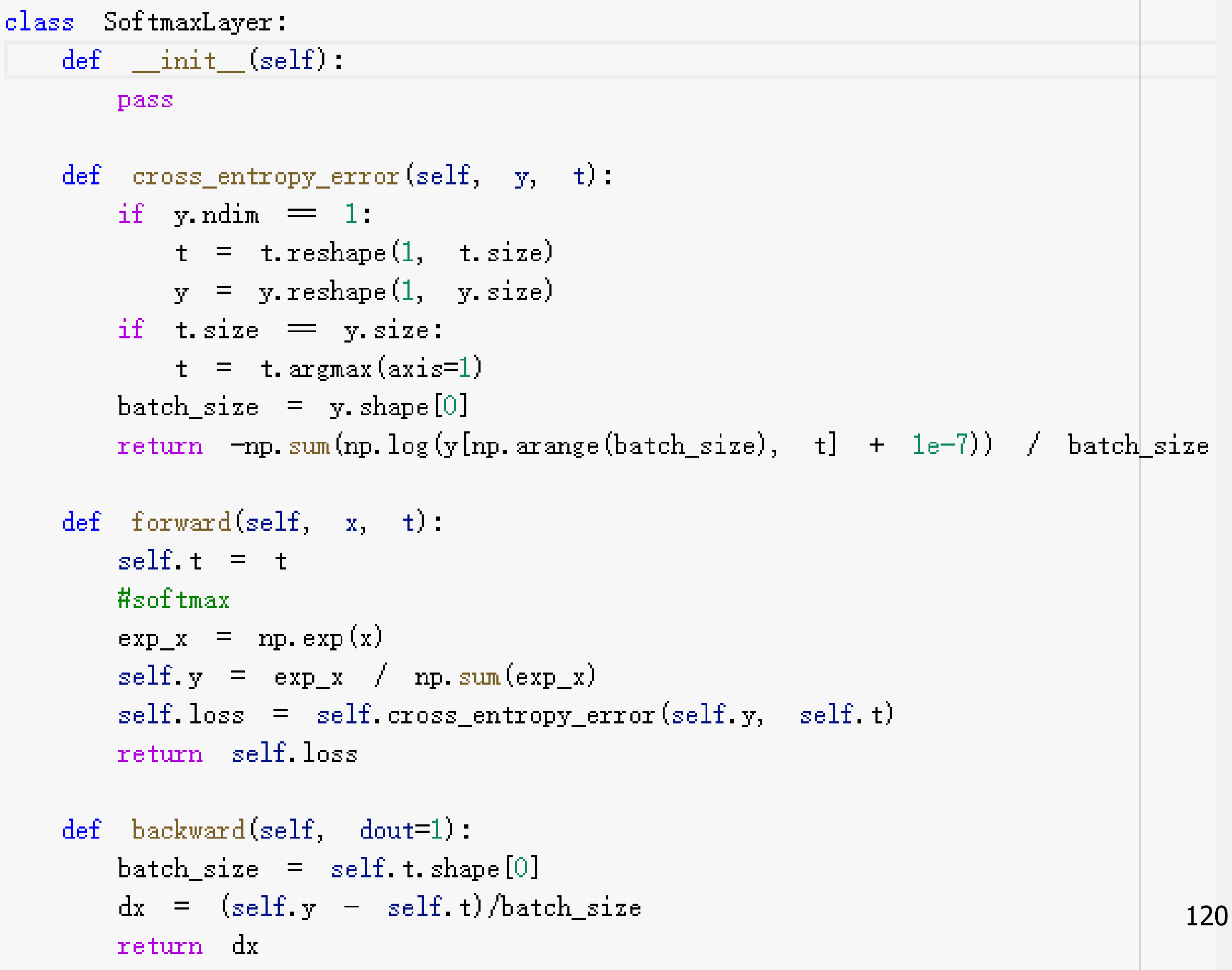
```
330.0
3.3000000000000003 110.00000000000001 300
```

Backpropagation

Rewrite



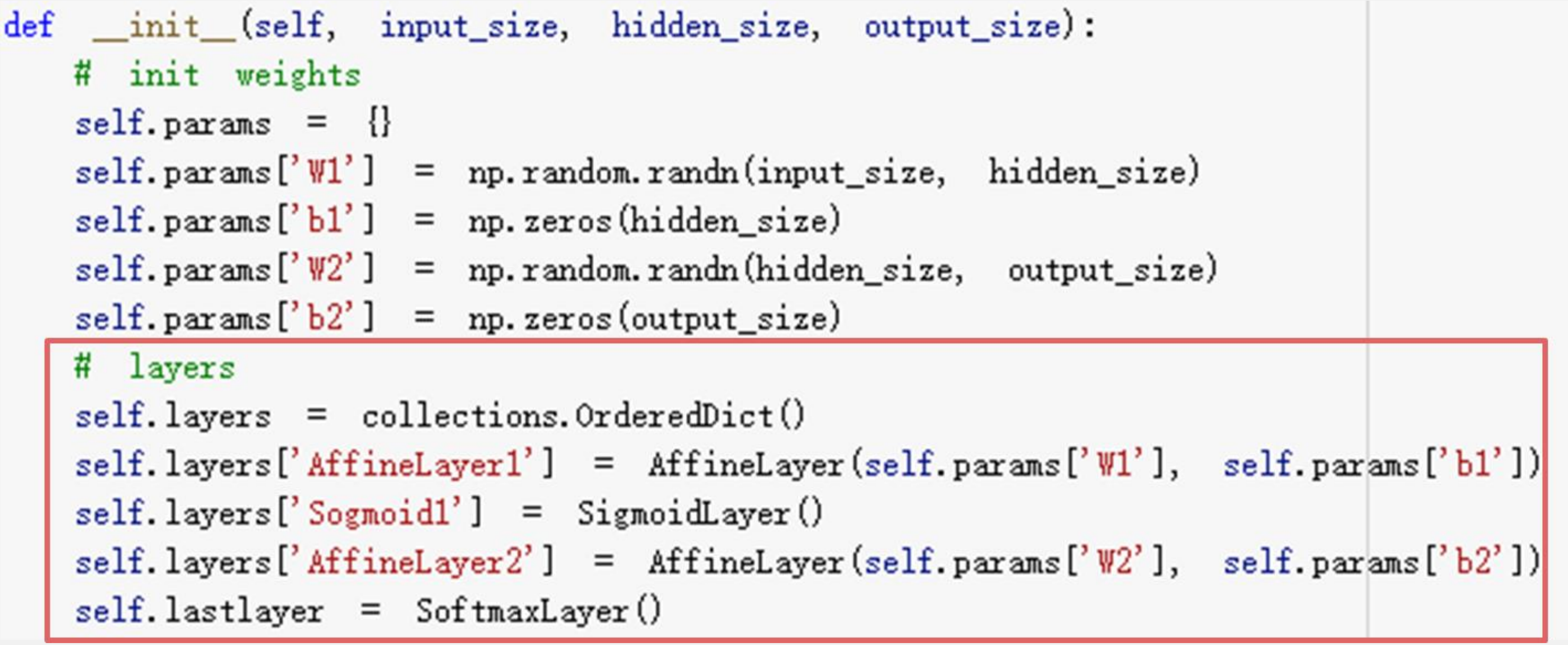
```
class SigmoidLayer:
    def __init__(self):
        pass

    def forward(self, x):
        self.out = 1/(1+np.exp(-x))
        return self.out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx

class AffineLayer:
    def __init__(self, w, b):
        self.W = w
        self.b = b

    def forward(self, x):
        self.x = x
        out = np.dot(x, self.W) + self.b
        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)
        return dx
```



```
class SoftmaxLayer:
    def __init__(self):
        pass

    def cross_entropy_error(self, y, t):
        if y.ndim == 1:
            t = t.reshape(1, t.size)
            y = y.reshape(1, y.size)
        if t.size == y.size:
            t = t.argmax(axis=1)
        batch_size = y.shape[0]
        return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size

    def forward(self, x, t):
        self.t = t
        #softmax
        exp_x = np.exp(x)
        self.y = exp_x / np.sum(exp_x)
        self.loss = self.cross_entropy_error(self.y, self.t)
        return self.loss

    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        dx = (self.y - self.t)/batch_size
        return dx
```

Backpropagation

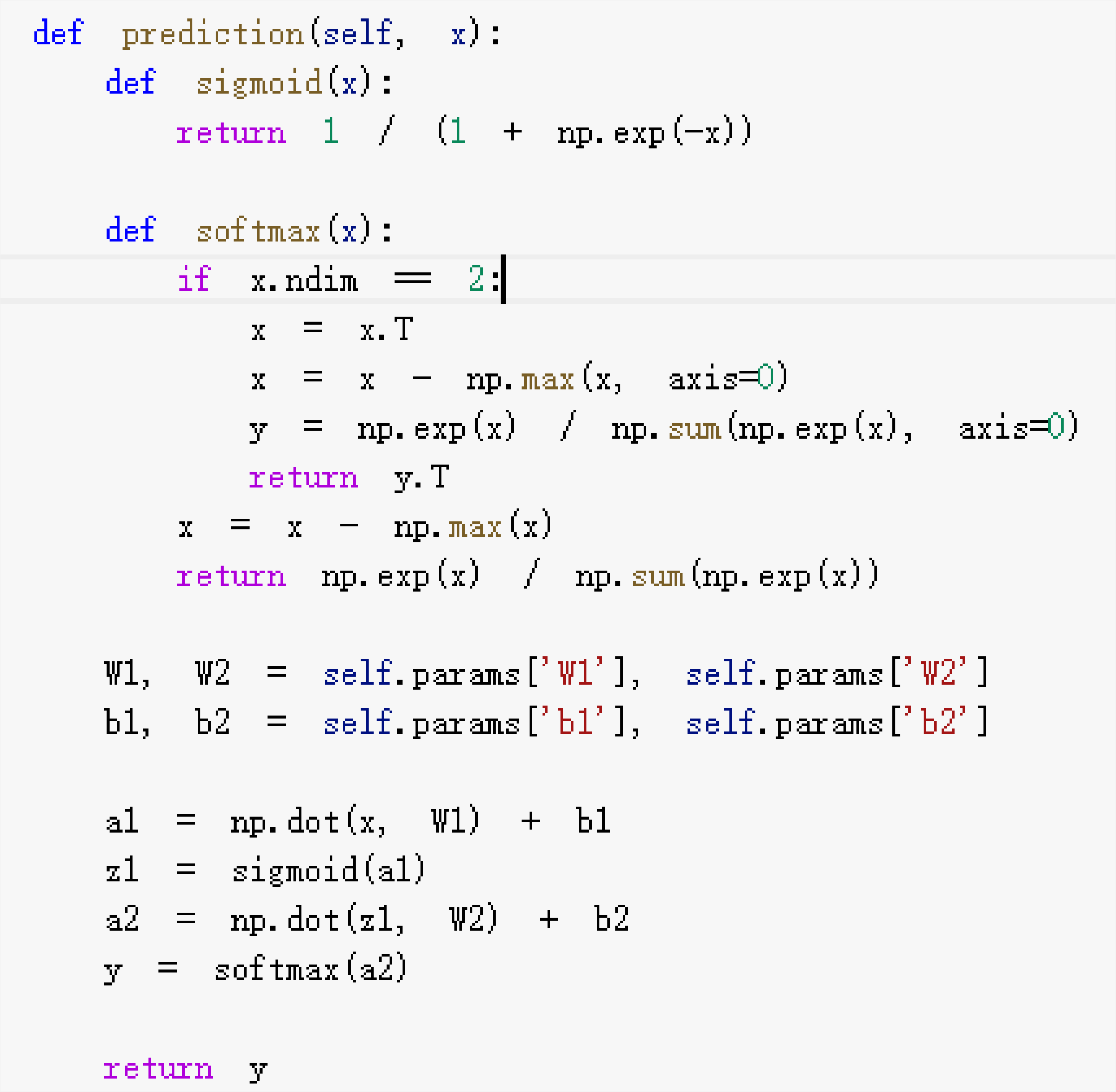
Rewrite



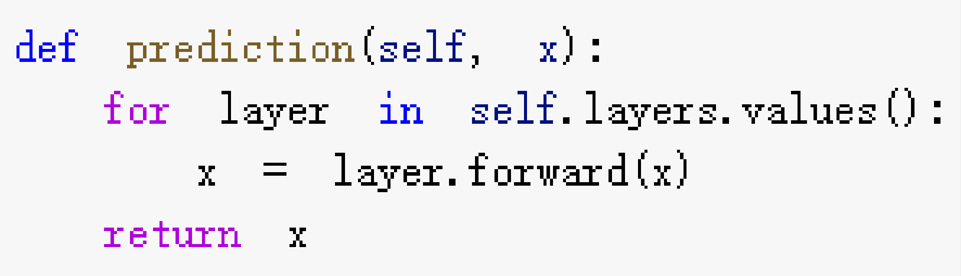
```
def __init__(self, input_size, hidden_size, output_size):
    # init weights
    self.params = {}
    self.params['W1'] = np.random.randn(input_size, hidden_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = np.random.randn(hidden_size, output_size)
    self.params['b2'] = np.zeros(output_size)
    # layers
    self.layers = collections.OrderedDict()
    self.layers['AffineLayer1'] = AffineLayer(self.params['W1'], self.params['b1'])
    self.layers['Sogmoid1'] = SigmoidLayer()
    self.layers['AffineLayer2'] = AffineLayer(self.params['W2'], self.params['b2'])
    self.lastlayer = SoftmaxLayer()
```

Backpropagation

Rewrite



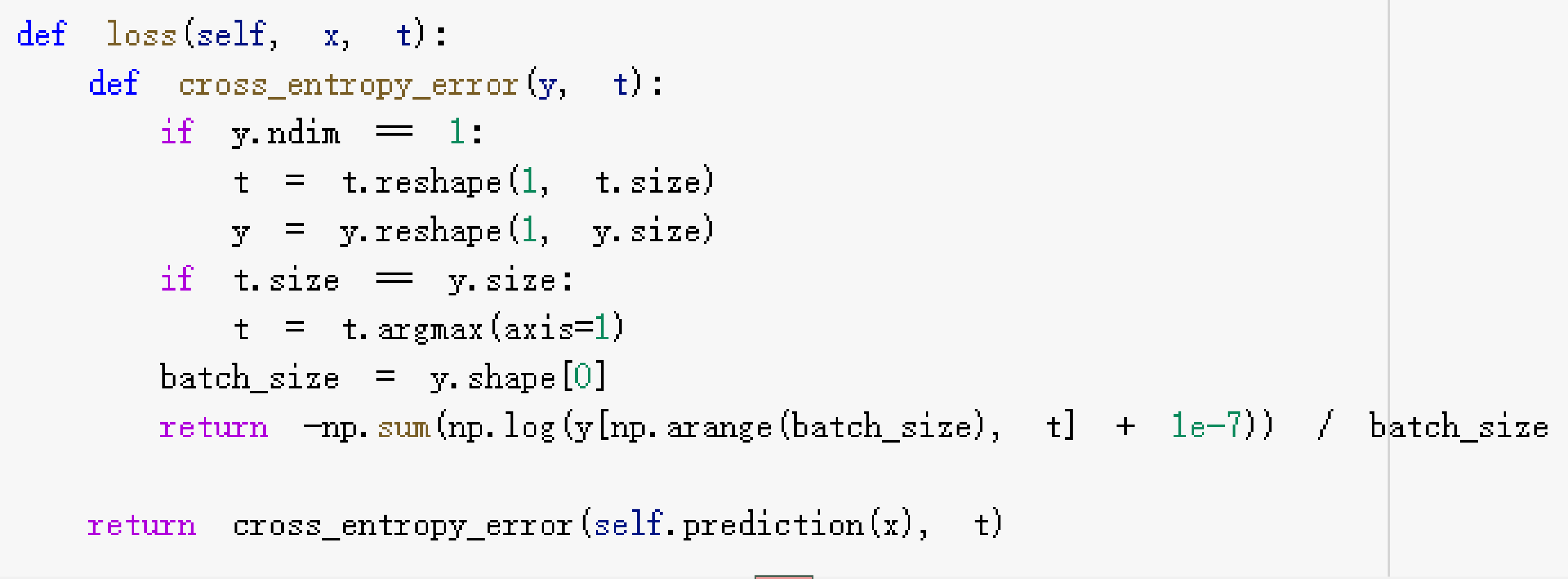
```
def prediction(self, x):
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    def softmax(x):
        if x.ndim == 2:
            x = x.T
            x = x - np.max(x, axis=0)
            y = np.exp(x) / np.sum(np.exp(x), axis=0)
            return y.T
        x = x - np.max(x)
        return np.exp(x) / np.sum(np.exp(x))

    W1, W2 = self.params[’W1’], self.params[’W2’]
    b1, b2 = self.params[’b1’], self.params[’b2’]

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    y = softmax(a2)

    return y
```



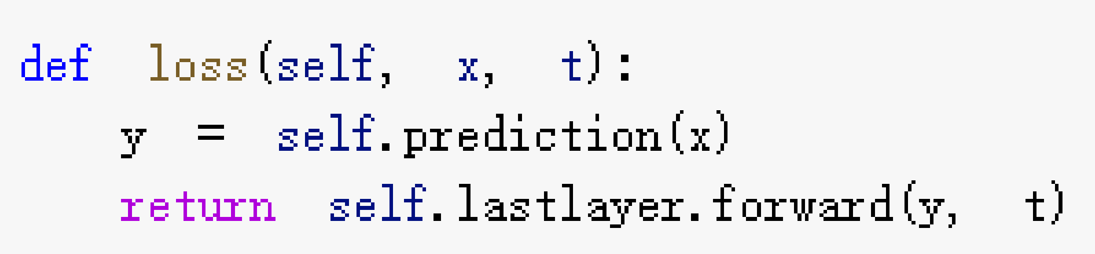
```
def prediction(self, x):
    for layer in self.layers.values():
        x = layer.forward(x)
    return x
```

Backpropagation

Rewrite



```python
def loss(self, x, t):
    def cross_entropy_error(y, t):
        if y.ndim == 1:
            t = t.reshape(1, t.size)
            y = y.reshape(1, y.size)
        if t.size == y.size:
            t = t.argmax(axis=1)
        batch_size = y.shape[0]
        return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size

    return cross_entropy_error(self.prediction(x), t)
```



```python
def loss(self, x, t):
    y = self.prediction(x)
    return self.lastlayer.forward(y, t)
```

Backpropagation

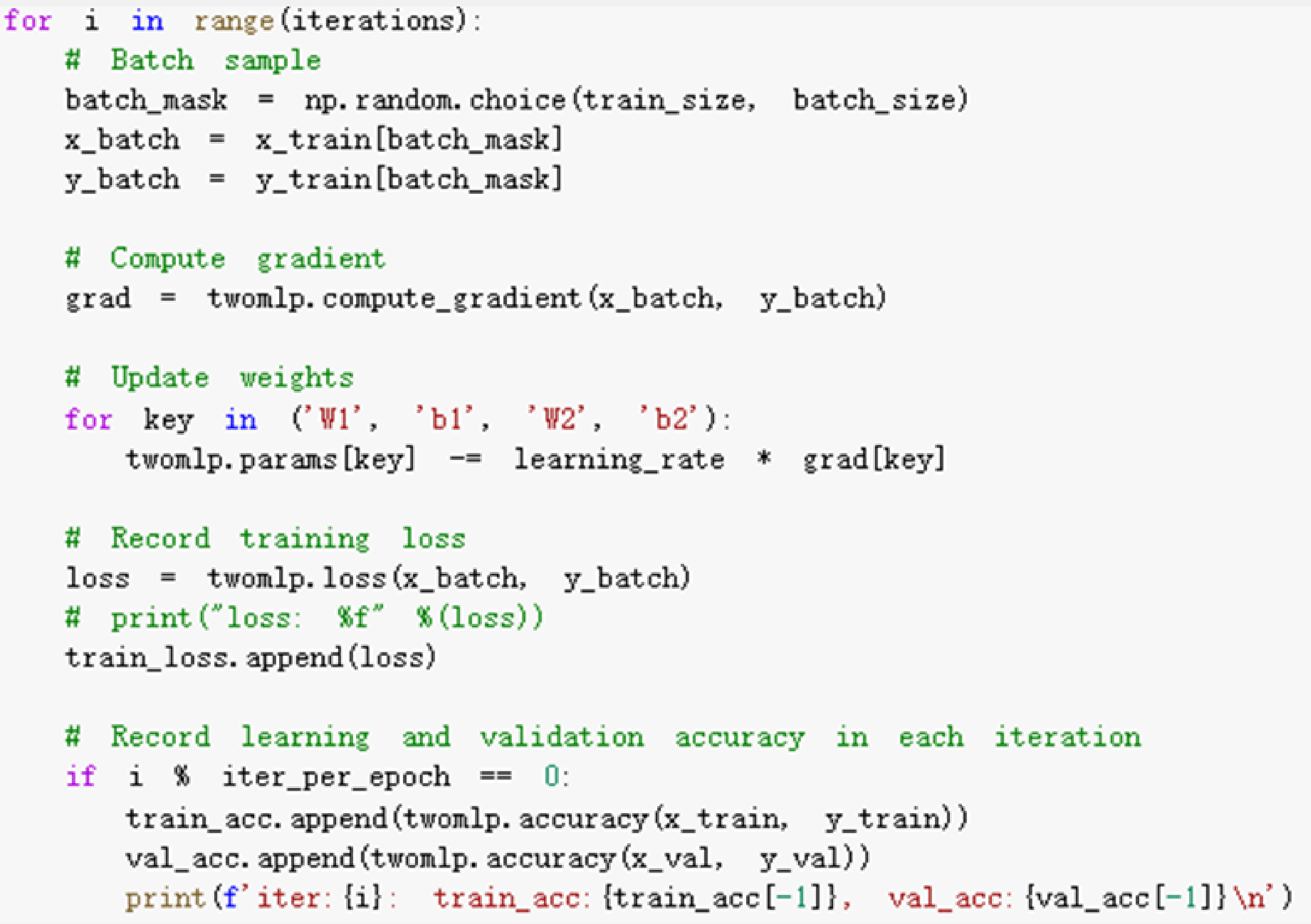
Rewrite

```
def numerical_gradient(f, x):
    h = 1e-2
    grad = np.zeros_like(x)

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x)

        x[idx] = float(tmp_val) - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val
        it.iternext()

def compute_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)
    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
    grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

    return grads
```

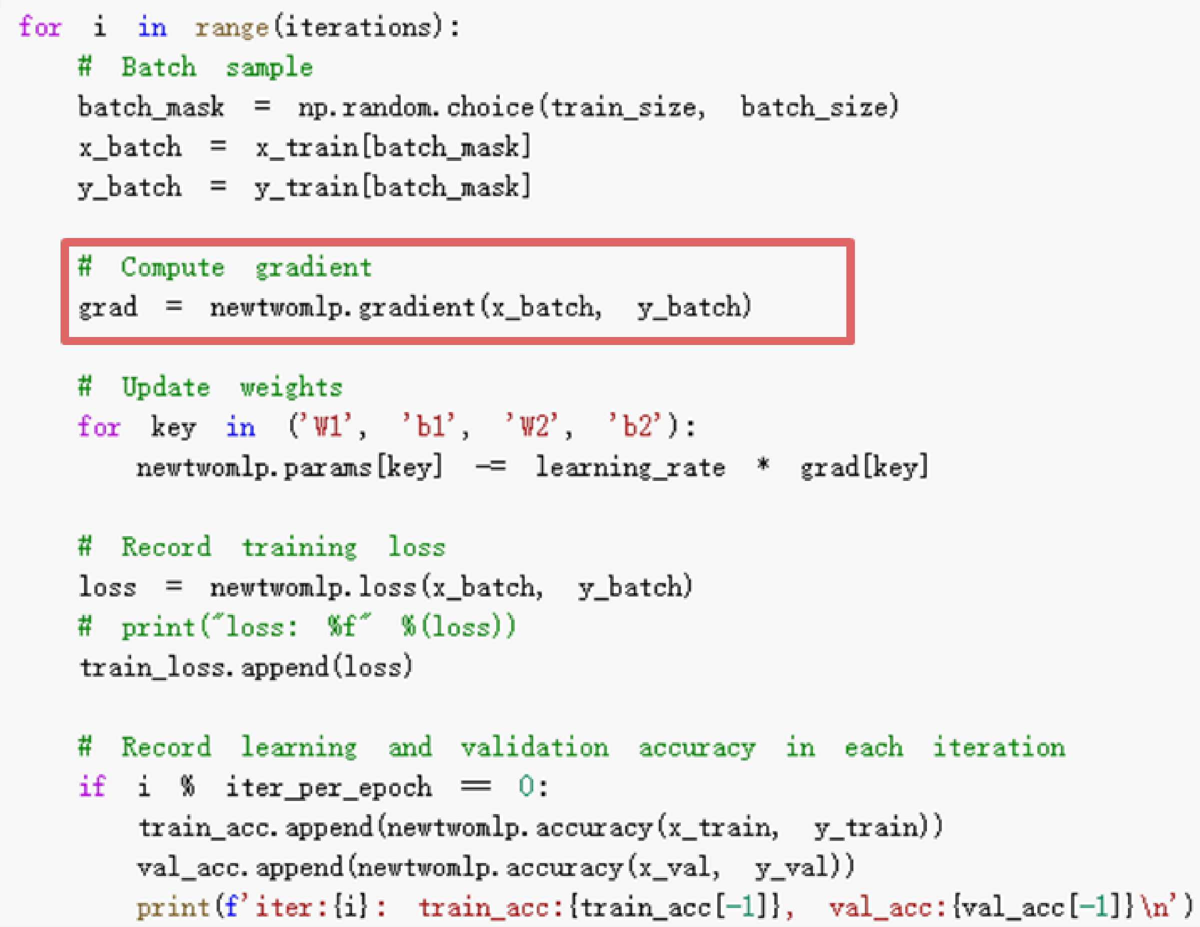
```
def gradient(self, x , t):
    #forward
    self.loss(x, t)
    #backward
    dout = 1
    dout = self.lastlayer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)
    #gradient
    grads = {}
    grads['W1'] = self.layers['AffineLayer1'].dW
    grads['b1'] = self.layers['AffineLayer1'].db
    grads['W2'] = self.layers['AffineLayer2'].dW
    grads['b2'] = self.layers['AffineLayer2'].db

    return grads
```

Backpropagation

Rewrite



```
for i in range(iterations):
    # Batch sample
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    y_batch = y_train[batch_mask]

    # Compute gradient
    grad = twomlp.compute_gradient(x_batch, y_batch)

    # Update weights
    for key in ('W1', 'b1', 'W2', 'b2'):
        twomlp.params[key] -= learning_rate * grad[key]

    # Record training loss
    loss = twomlp.loss(x_batch, y_batch)
    # print("loss: %f" %(loss))
    train_loss.append(loss)

    # Record learning and validation accuracy in each iteration
    if i % iter_per_epoch == 0:
        train_acc.append(twomlp.accuracy(x_train, y_train))
        val_acc.append(twomlp.accuracy(x_val, y_val))
        print(f'iter:{i}: train_acc:{train_acc[-1]}, val_acc:{val_acc[-1]}\n')
```



```
for i in range(iterations):
    # Batch sample
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    y_batch = y_train[batch_mask]

    # Compute gradient
    grad = newtwomlp.gradient(x_batch, y_batch)

    # Update weights
    for key in ('W1', 'b1', 'W2', 'b2'):
        newtwomlp.params[key] -= learning_rate * grad[key]

    # Record training loss
    loss = newtwomlp.loss(x_batch, y_batch)
    # print("loss: %f" %(loss))
    train_loss.append(loss)

    # Record learning and validation accuracy in each iteration
    if i % iter_per_epoch == 0:
        train_acc.append(newtwomlp.accuracy(x_train, y_train))
        val_acc.append(newtwomlp.accuracy(x_val, y_val))
        print(f'iter:{i}: train_acc:{train_acc[-1]}, val_acc:{val_acc[-1]}\n')
```

Backpropagation

Rewrite

Numerical gradient / backpropagation = 10

Numerical gradient

- Advantage: A formulas
- Disadvantage: slower and less accurate

Backpropagation

- Advantage: faster and higher accurate
- Disadvantage: Each operation needs to compute its own differential