

Lab2 - Classification task by Resnet18 on CIFAR10

Advisor : Tsai, Chia-Chi

TA : 陳喬雅

Outline



1. Lab2 task
2. Layer introduction
3. ResNet18
4. CIFAR-10
5. Sample code introduction

Outline

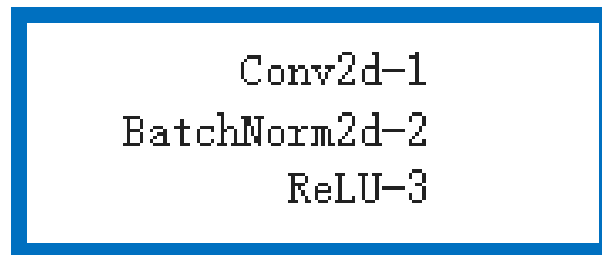


1. Lab2 task
2. Layer introduction
3. ResNet18
4. CIFAR-10
5. Sample code introduction

1. Lab2 task-1

1) Use pytorch to implement an specific classification DNN model, dataset CIFAR10

Hint:要包含兩層的藍色框layer，可在中間加上pooling layer，最後要加一層FC以達成分類任務



Layers	Input channel	Output channel	kernel	padding
Conv1	3	32	3	1
Bn1	32	32	x	x
Conv2	32	64	3	1
Bn2	64	64	x	x

1. Lab2 task-1



2) Print model summary(including parameters)

3) Print test accuracy, plot epoch-train accuracy, epoch-val accuracy, epoch-train loss, epoch-val loss.

Accuracy (20%)

Accuracy	Score
>80%	20
>70%	16
>60%	12

1. Lab2 task-2

- 1) Use **Resnet18** to train on CIFAR-10
- 2) Experiment on the following and compare the result with baseline
 - Input image normalization
 - Data augmentation
 - Different base learning rate and update strategy
- 3) Print test acc
- 4) Plot train-loss, val-loss, train-acc, val-acc

Accuracy (20%)

Report (60%)

Accuracy	Score
>85%	20
>80%	16
>70%	12

1. Lab2 task

- Recommend platform: colab with python (Use **Pytorch**)
- Upload platform: NCKU moodle
- Upload compressed **StudentID_lab2.zip** file including:
 - **學號_lab2.ipynb**
 - IPython notenook 須包含程式碼跟結果
 - **學號_lab2.pdf**
 - 各項print以及plot輸出
 - 實作所運到的困難及解決方法
- Provided file: lab2 sample code
- Q&A: course.aislabor@gmail.com

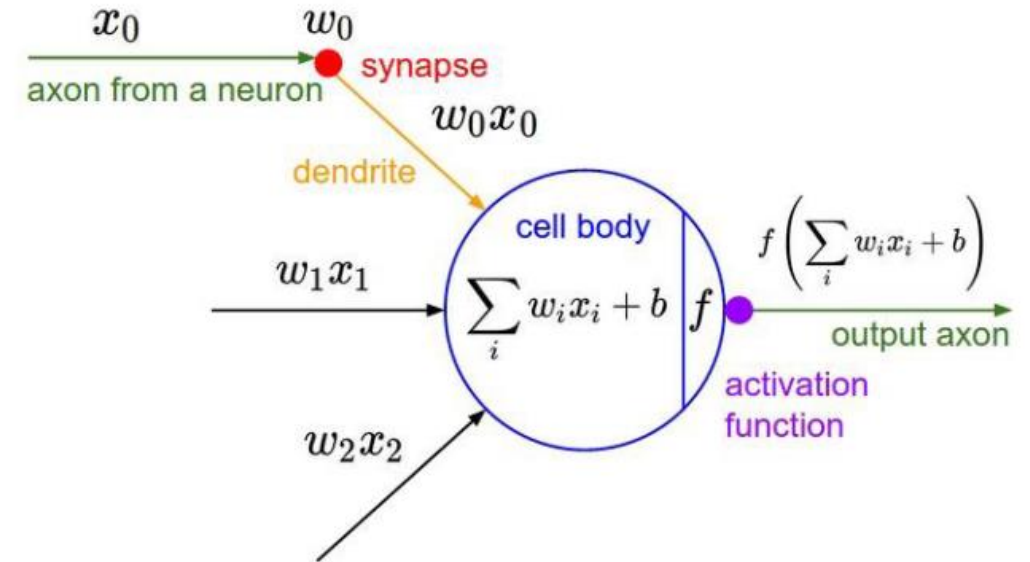
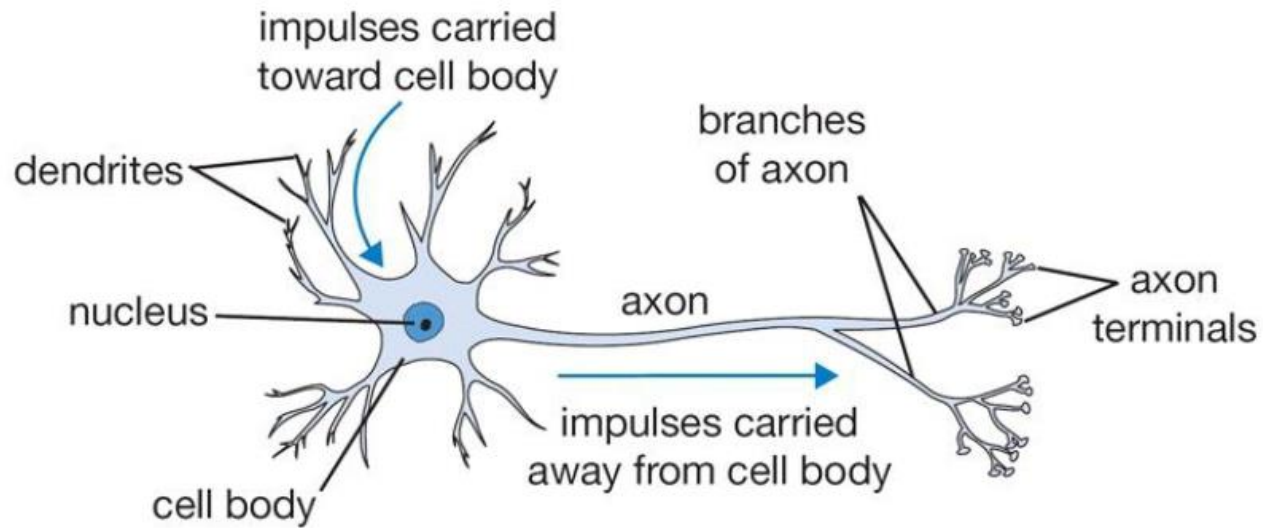
Outline



1. Lab2 task
2. Layer introduction
3. ResNet18
4. CIFAR-10
5. Sample code introduction

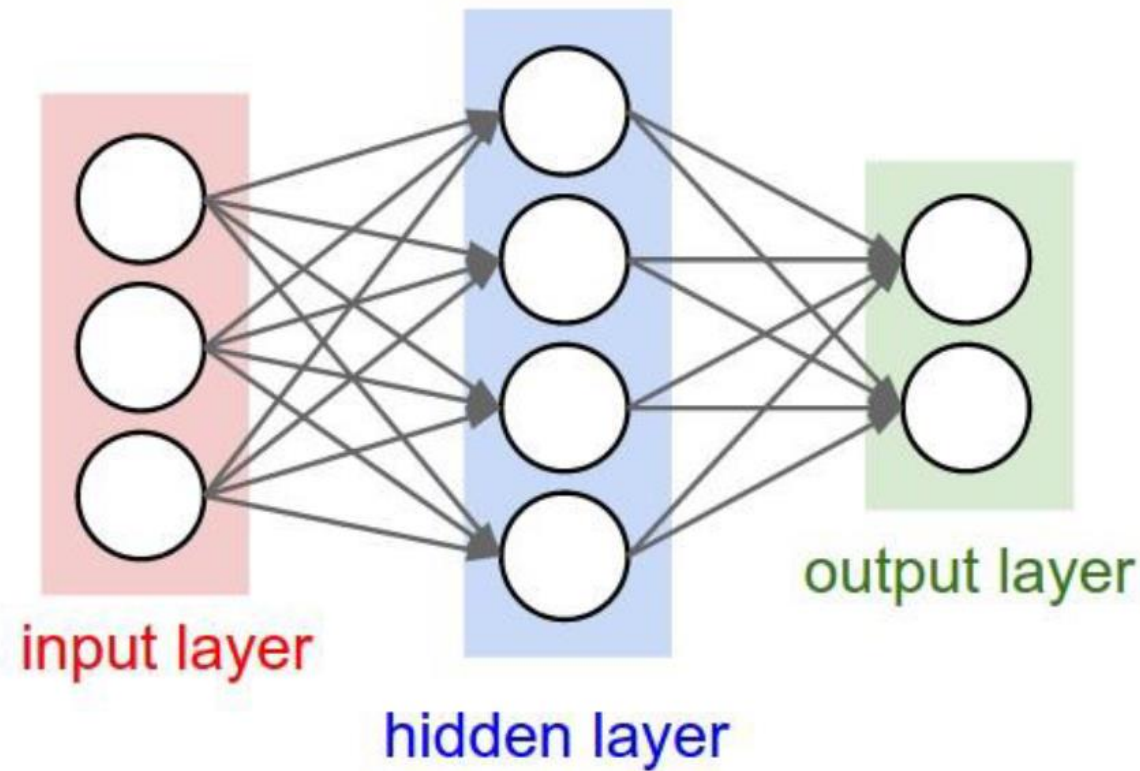
3. Layers introduction

- Single neuron.



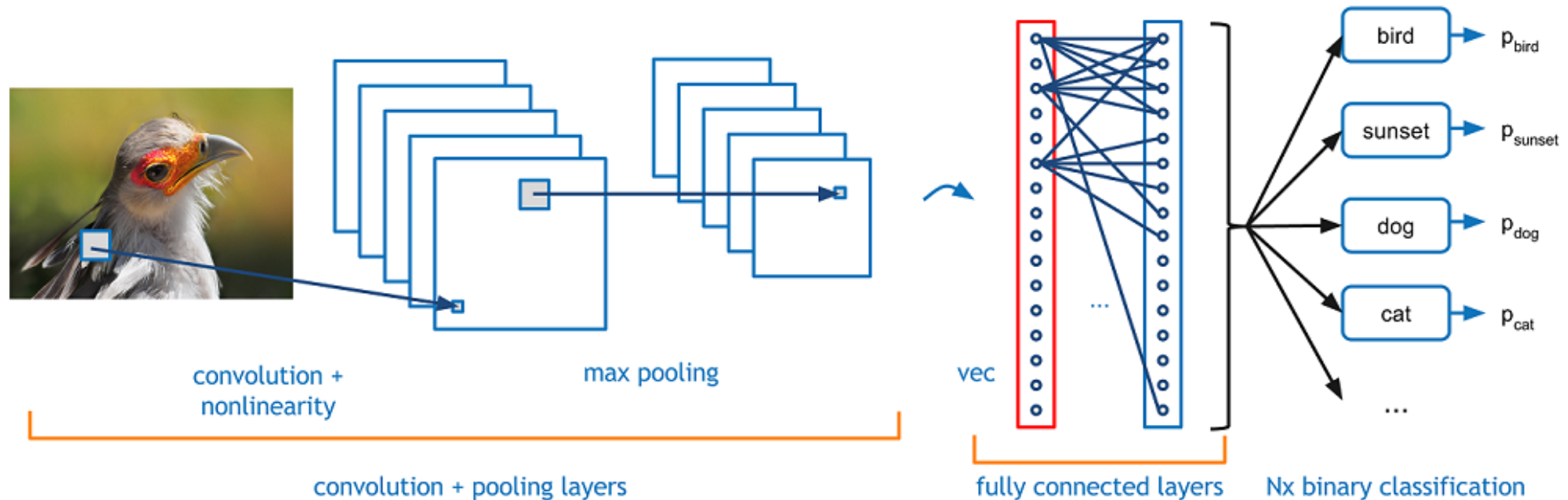
3. Layers introduction

- Multiple neurons.



3. Layers introduction

- Mimicking multiple neurons by layers.



3. Layers introduction – Activation function



```
model.add(Activation("relu"))
```

Activation Functions

1. step:
$$f(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

2. sigmoid:
$$t = \sum_{i=1}^n w_i x_i \quad s(t) = 1/(1 + e^{-t})$$

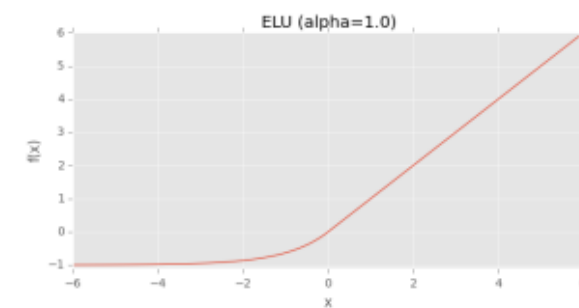
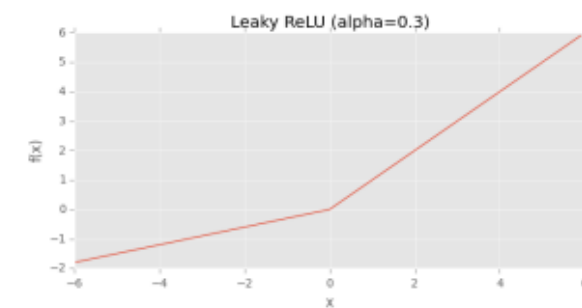
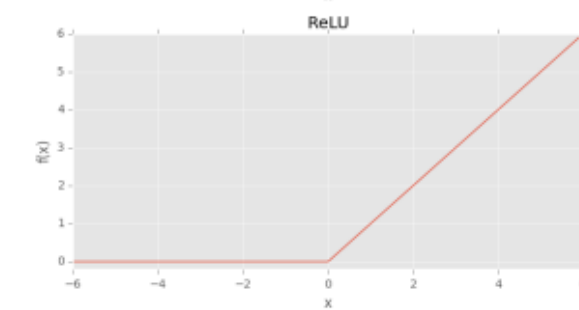
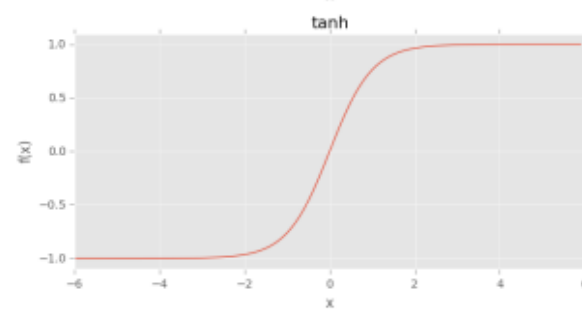
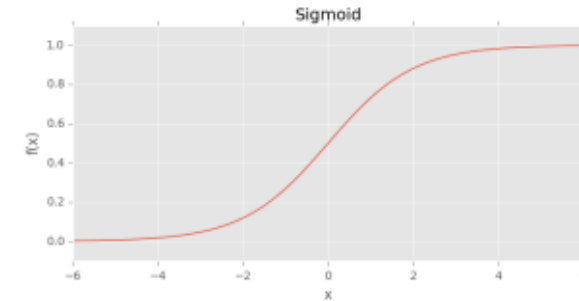
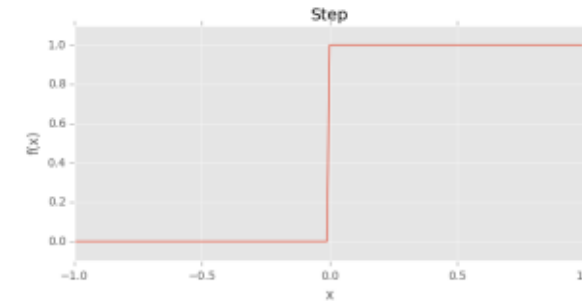
3. tanh:
$$f(z) = \tanh(z) = (e^z - e^{-z}) / (e^z + e^{-z})$$

4. Rectified Linear Unit (ReLU)
$$f(x) = \max(0, x)$$

5. Leaky ReLUs
$$f(net) = \begin{cases} net & \text{if } net \geq 0 \\ \alpha \times net & \text{otherwise} \end{cases}$$

6. Exponential Linear Units (ELUs)
$$f(net) = \begin{cases} net & \text{if } net \geq 0 \\ \alpha \times (\exp(net) - 1) & \text{otherwise} \end{cases}$$

7. Softmax:
$$\text{Softmax}(x) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$



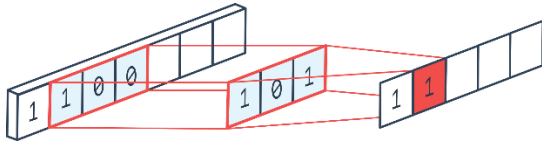
3. Layers introduction – Convolution layer



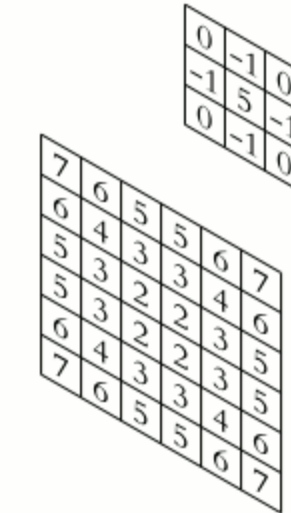
```
self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
```

- 2D Convolution is using a '**kernel**' to extract certain 'features' from an input image.

1D:

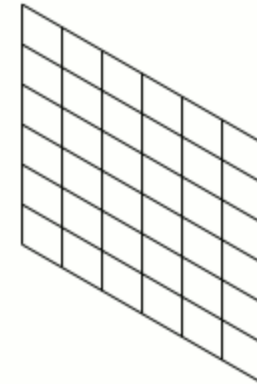


2D:



input

output



<https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37>

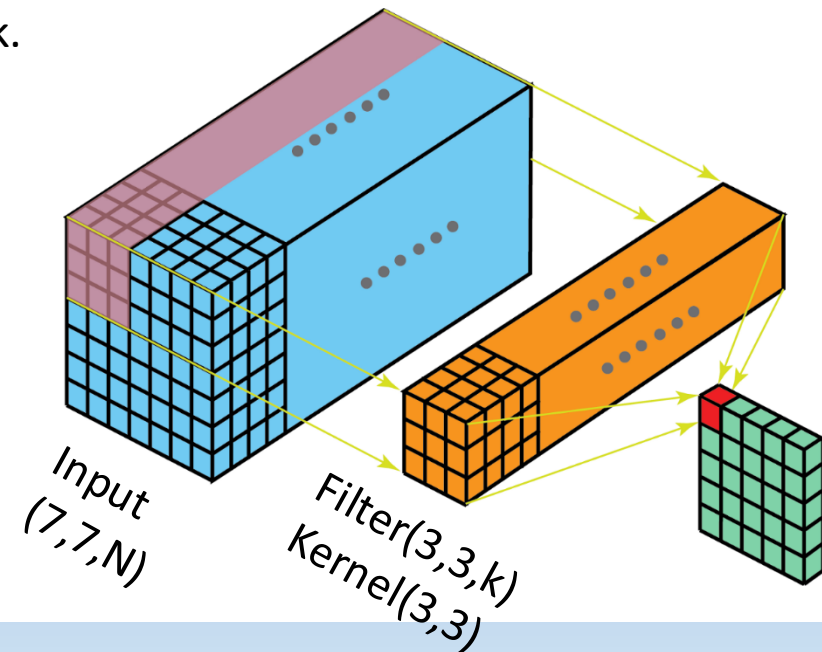
3. Layers introduction – Convolution layer



- Kernel vs Filter

A **kernel** is, as described earlier, a **matrix of weights** which are multiplied with the input **to extract relevant features**. The dimensions of the kernel matrix is *how the convolution gets its name*. For example, in 2D convolutions, the kernel matrix is a 2D matrix.

A **filter** however **is a concatenation of multiple kernels**, each kernel is assigned to a particular channel of the input. Filters are always one dimension more than the kernels. For example, in 2D convolutions, filters are 3D matrices (which is essentially a concatenation of 2D matrices i.e. the kernels). So for a CNN layer with kernel dimensions $h*w$ and input channels k , the filter dimensions are $h*w*k$.



<https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37>

3. Layers introduction – Convolution layer



- E.g. Convolution on grayscale image

Convolution/Padding

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

$X: m \times n$

$W: k \times k$

→ Y : same size with X

Padding = $p = 1$

[https://medium.com/@ayeshmanthaperera/
what-is-padding-in-cnns-71b21fb0dd7](https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7)

3. Layers introduction – Convolution layer



- E.g. Convolution on grayscale image

Convolution/**Stride** = s , only compute convolution on pixels $x_{1+i*s, 1+j*s}$

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

Padding = $p = 1$, Stride = $s = 2$

X : $m \times n$

W : $k \times k$

$$\rightarrow Y: \left(\frac{m-k+2p}{s} + 1 \right) \times \left(\frac{n-k+2p}{s} + 1 \right)$$

e.g.

X : $m \times n$, $m = n = 5$

W : $k \times k$, $k = 3$

$$\left(\frac{5-3+2*1}{2} + 1 \right) \times \left(\frac{5-3+2*1}{2} + 1 \right) = (3,3)$$

3. Layers introduction – Convolution layer

- E.g. Convolution on grayscale image

Convolution/**Stride** = s , only compute convolution on pixels $x_{1+i*s, 1+j*s}$

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

Padding = $p = 1$, Stride = $s = 2$

$X: m \times n$

$W: k \times k$

$$\rightarrow Y: \left(\frac{m-k+2p}{s} + 1 \right) \times \left(\frac{n-k+2p}{s} + 1 \right)$$

e.g.

$X: m \times n, m = n = 5$

$W: k \times k, k = 3$

$$\left(\frac{5-3+2*1}{2} + 1 \right) \times \left(\frac{5-3+2*1}{2} + 1 \right) = (3,3)$$

3. Layers introduction – Convolution layer

- E.g. Convolution on RGB image

0	0	0	0	0	0	...
0	136	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...

Input Channel #3 (Blue)

e.g.

Input X: $N \times N$

Kernel W: 3×3

Filters(channels = 3) : $3 \times 3 \times 3$

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #1

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #2

0	1	1
0	1	0
1	-1	1

Kernel Channel #3

308

+

-498

+

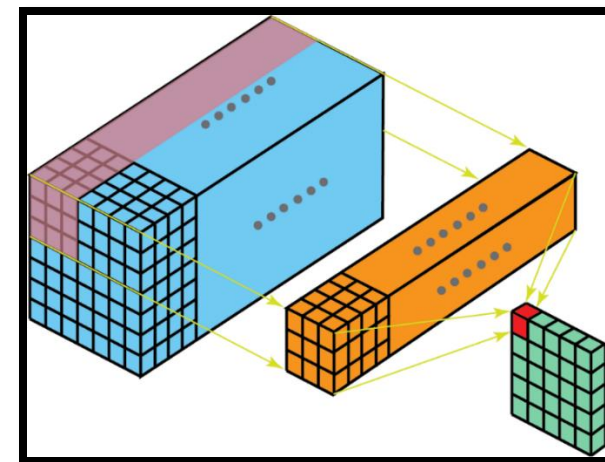
164

+ 1 = -25

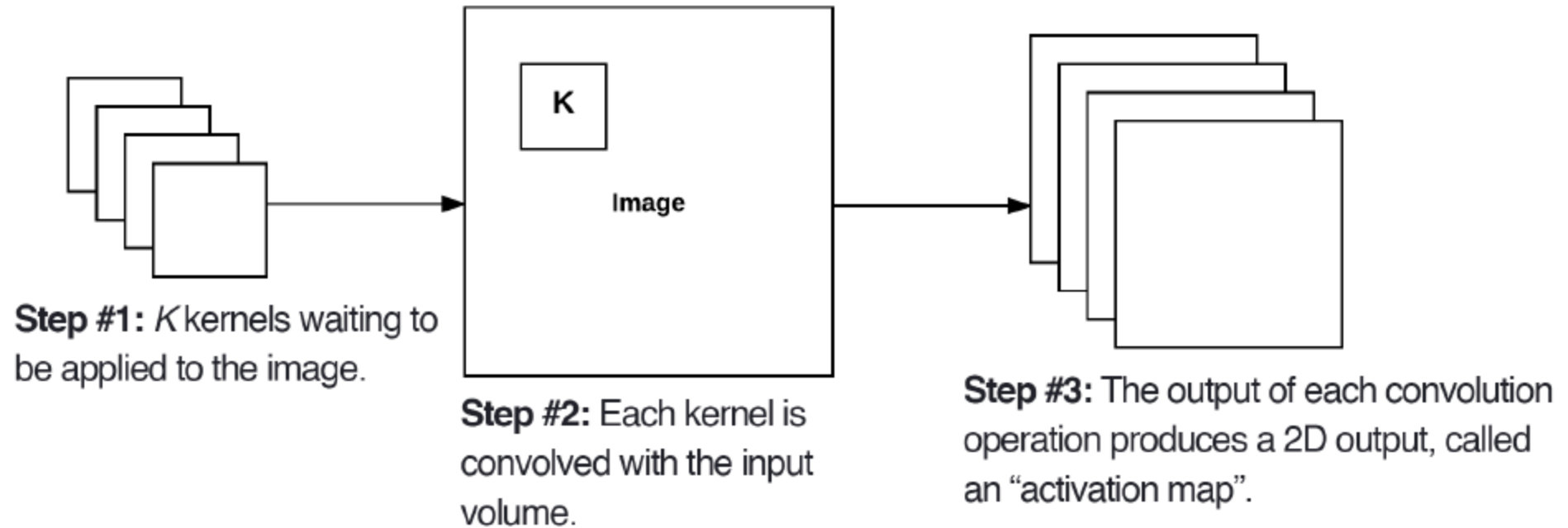
Bias = 1

-25				...
				...
				...
				...
...

Output



3. Layers introduction – Convolution layer



Left: At each convolutional layer in a CNN, there are K kernels applied to the input volume.

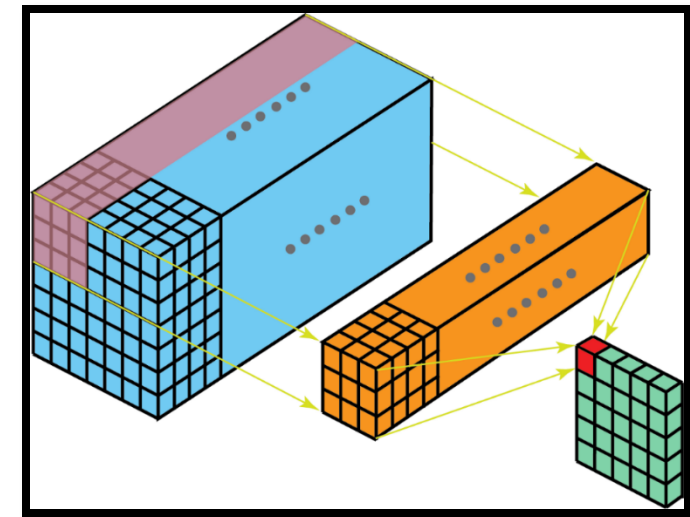
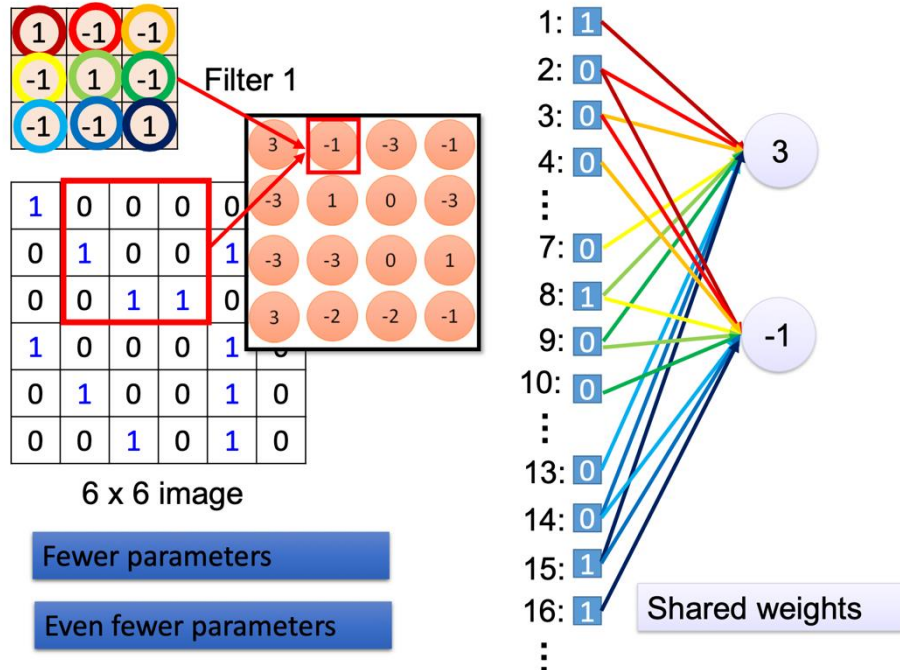
Middle: Each of the K kernels is convolved with the input volume.

Right: Each kernel produces an 2D output, called an **activation map**(**feature map**).

<https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac>

3. Layers introduction – Convolution layer

- Why do we use CNN instead of NN?
 1. reduce the number of parameters(shared weights)
 2. accelerate execution time
 3. combine image features
 4. No longer need experts to design convolutional kernels



3. Layers introduction – Convolution layer



- Calculation of parameters, take LeNet for example:

1. Output width of conv layer:

$$\begin{aligned} &= ((28-5+2*0) / 1) + 1 \\ &= 24 \text{ (conv layer output width)} \end{aligned}$$

2. Number of neurons/units within the conv layer

$$\begin{aligned} &= \text{output height} * \text{output width} * \text{number of feature maps(channels)} \\ &= 24 \times 24 \times 6 \text{ (conv output volume)} \\ &= 3,456 \text{ units} \end{aligned}$$

3. Number of training parameters or weights within the conv layer (without weight sharing)

$$\begin{aligned} &= 3456 * ((5 * 5 * 1) + 1 \text{ bias}) \\ &= 89,856 \text{ weights} \end{aligned}$$

4. Number of training parameters or weights with weight sharing (with weight sharing)

$$\begin{aligned} &= 6 * ((5 * 5 * 1) + 1 \text{ bias}) \\ &= 156 \text{ weights} \end{aligned}$$

Aa Name	LeNet (first conv layer)
Input Image Size	28x28x1
Number of Filters channels	6
Filter size kernel	5x5x1
Stride	1
Padding	0
Number of parameters for a unit (without parameter sharing)	89,856
Number of parameters for a unit (with parameter sharing)	156
Output Volume	24x24x6

<https://towardsdatascience.com/understanding-parameter-sharing-or-weights-replication-within-convolutional-neural-networks-cc26db7b645a>

3. Layers introduction – Convolution layer



- Calculation of parameters, take AlexNet for example:

1. Output width of conv layer:

$$\begin{aligned} &= ((227-11) / 4) + 1 \\ &= 55 \text{ (conv layer output width)} \end{aligned}$$

2. Number of neurons/units within the conv layer

$$\begin{aligned} &= \text{output height} * \text{output width} * \text{number of feature maps} \\ &= 55 \times 55 \times 96 \text{ (conv output volume)} \\ &= 290,400 \text{ units} \end{aligned}$$

3. Number of training parameters or weights within the conv layer (without weight sharing)

$$\begin{aligned} &= 290400 * ((11 * 11 * 3) + 1 \text{ bias}) \\ &= 105,415,600 \text{ weights} \end{aligned}$$

4. Number of training parameters or weights with weight sharing (with weight sharing)

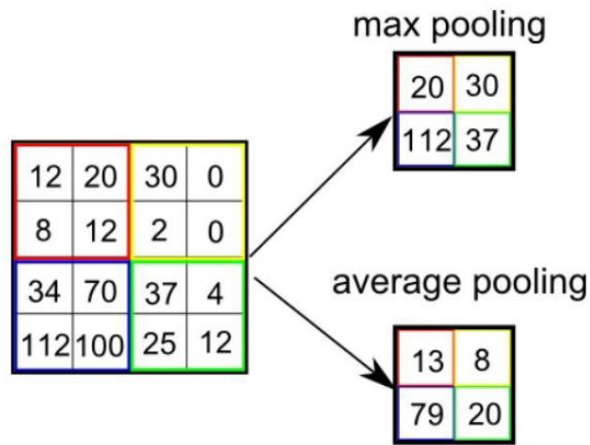
$$\begin{aligned} &= 96 * ((11 * 11 * 3) + 1 \text{ bias}) \\ &= 34,944 \text{ weights} \end{aligned}$$

<u>Aa</u> Name	☰ AlexNet (first conv layer)
Input Image Size	227x227x3
Number of Filters channels	96
Filter size kernel	11x11x3
Stride	4
Padding	0
Number of parameters for a unit (without parameter sharing)	105,705,600
Number of parameters for a unit (with parameter sharing)	34,944
Output Volume	55x55x96

3. Layers introduction – Pooling layer



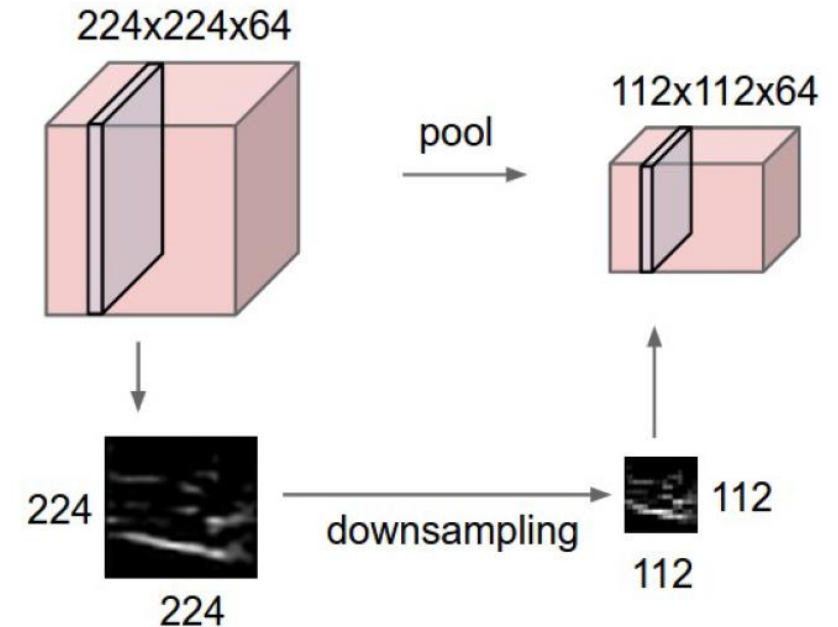
- To reduce model dimensionality by sliding window approach.
 - Max pooling: Discard unnecessary or noisy features.
 - Average pooling



Pooling layer size (2,2), stride = 2

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

max pooling layer with size=(3,3), stride=1, padding=0

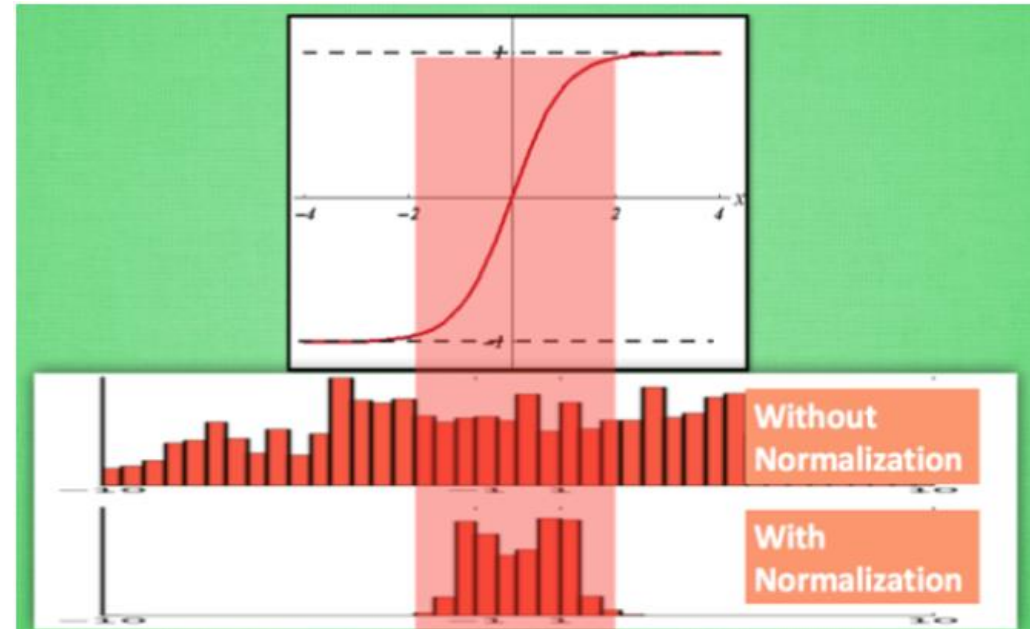
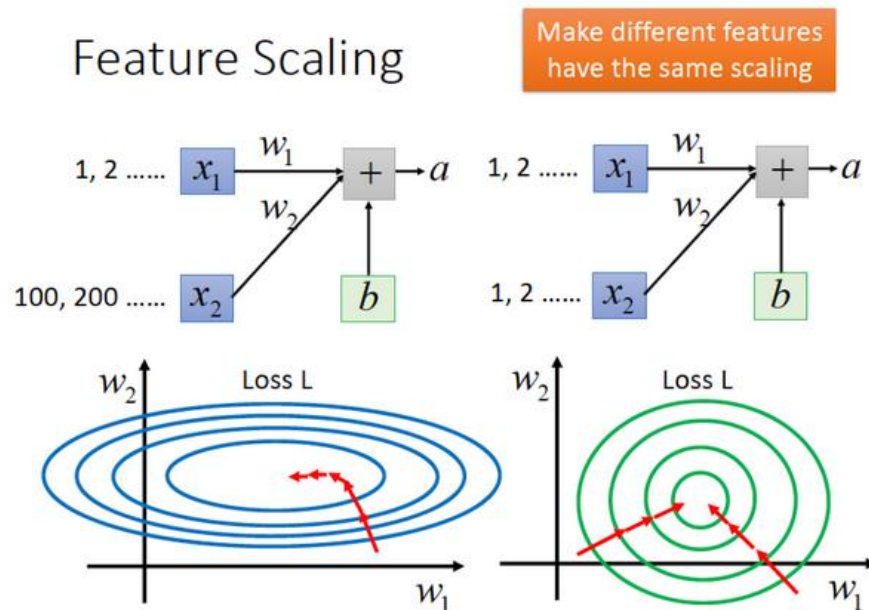


after pooling layer with size=(2,2), stride=2, padding=0

3. Layers introduction – Batch normalization layer



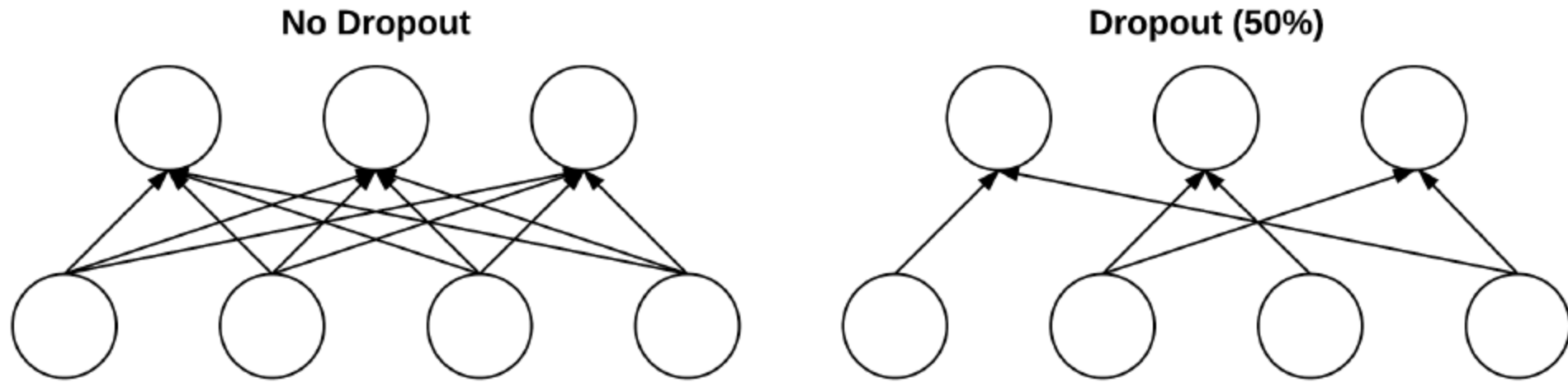
- Efficiently normalize based on loaded batch data.
- Train faster with bigger learning rate by avoiding covariant problem.
- Avoiding vanishing gradients especially effective for sigmoid, tanh.



NTU Professor Lee' s ML: <https://www.youtube.com/watch?v=BZh1ltr5Rkg>

3. Layers introduction – Dropout layer

- To generalize the model by randomly dropping multiple redundant nodes.
- Usually placed between fully-connected layers.



Left: Two layers of a neural network that are fully-connected with no dropout. Right: The same two layers after dropping 50% of the connections.

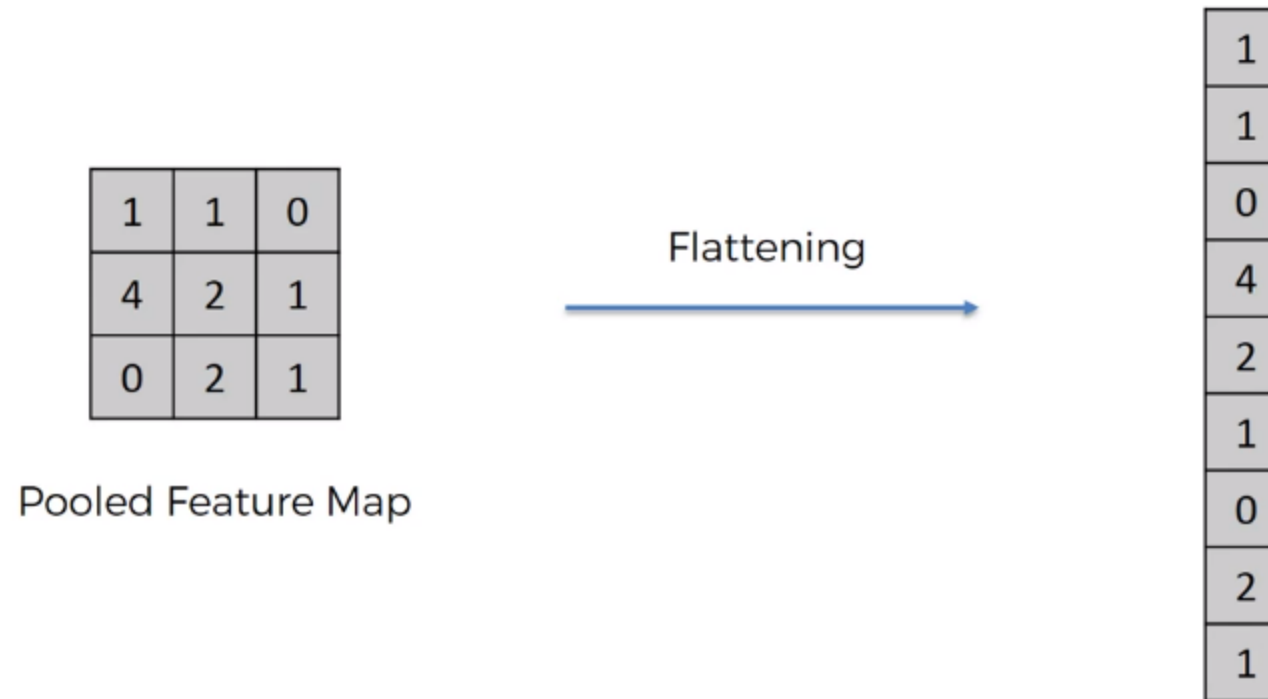
Connections: 12

$p = 0.5$

Connections: 6

3. Layers introduction – Flatten layer

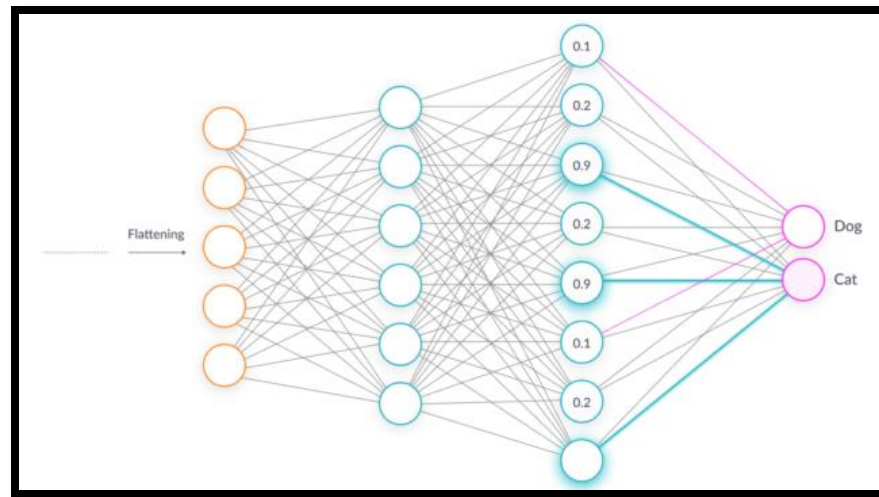
- Flatten layer is used to make the multidimensional input one-dimensional, commonly used in the transition from the convolution layer to the full connected layer.



3. Layers introduction – FC layer



- Fully connected layer.
- Each neuron in the dense layer receives input from all neurons of its previous layer, where neurons of the dense layer perform matrix-vector multiplication



https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

```
self.fc1 = nn.Linear(4096, 10)
```

Image normalization



- min/max normalization: 縮到0~1或-1~1之間，通常是input range已知的情況可用，

output = input / 255

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- transforms.ToTensor(): range(0, 255) -> range(0.0, 1.0)

Image normalization



- Standardization: 將sampled dataset的mean和std轉換成接近於0和1，以此減少偏差，避免被某部分資料支配，通常是用在Input range未知的情況，以採樣的方式來取得。
- mean -> 0 : unbiased的data更有利於model學習
- std -> 1 : 減緩梯度消失和梯度爆炸

```
# 計算normalization需要的mean & std
def get_mean_std(dataset, ratio=0.3):
    # Get mean and std by sample ratio
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=int(len(dataset)*ratio), shuffle=True, num_workers=2)

    data = next(iter(dataloader))[0] # get the first iteration data
    mean = np.mean(data.numpy(), axis=(0,2,3))
    std = np.std(data.numpy(), axis=(0,2,3))
    return mean, std

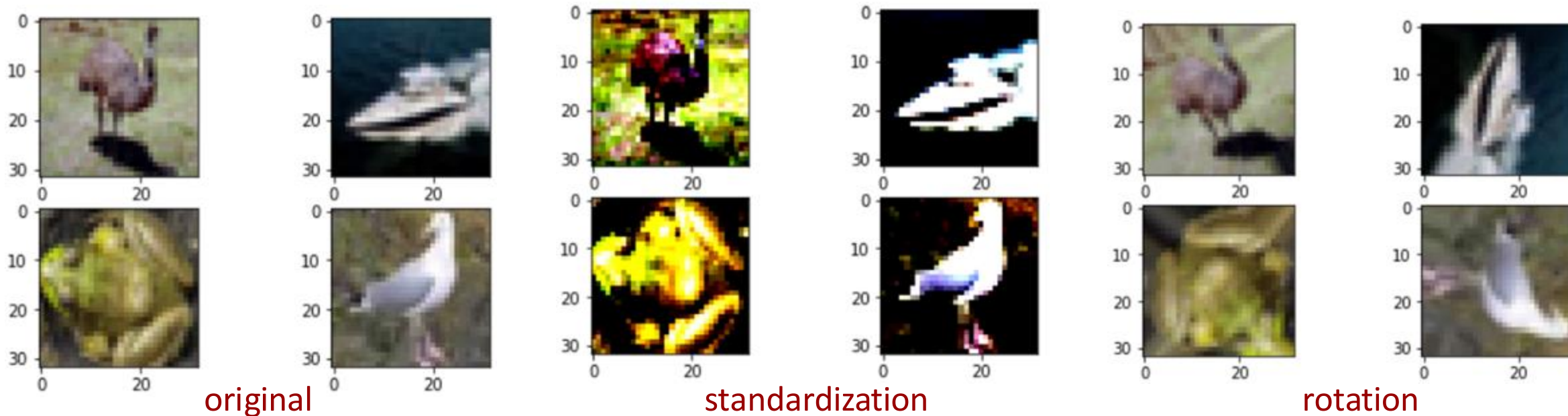
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transforms.ToTensor())
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms.ToTensor())

train_mean, train_std = get_mean_std(train_dataset)
test_mean, test_std = get_mean_std(test_dataset)
print(train_mean, train_std)
print(test_mean, test_std)
```

Data augmentation



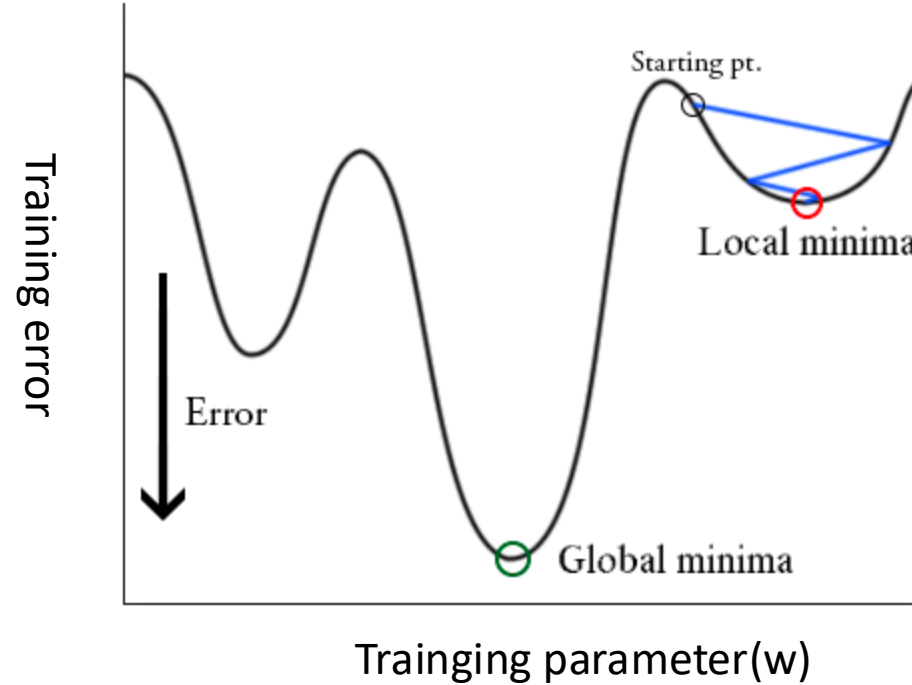
- 將圖片進行旋轉、調整大小、比例尺寸，或改變亮度色溫、翻轉、加入Gaussian noise等處理
- `transforms.RandomHorizontalFlip()`
- [Transforming and augmenting images — Torchvision main documentation](#)



Learning rate and update strategy



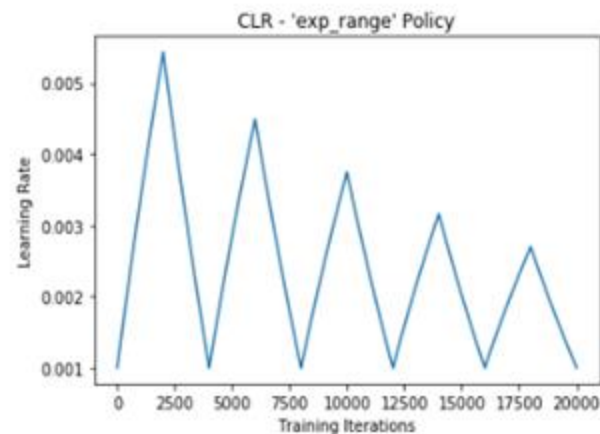
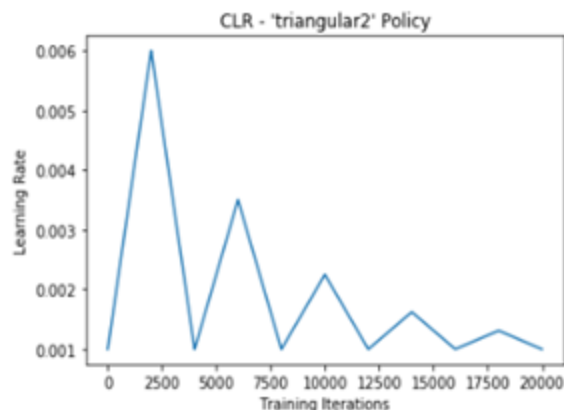
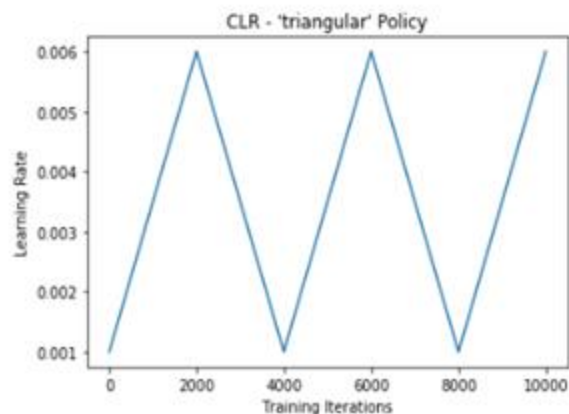
- 訓練model時，若採用固定的learning rate，容易找到local minima而非global minima



Learning rate and update strategy



- Learning Rate Decay : 通常在訓練一定epoch後，會對學習率進行衰減，從而讓model收斂得更好
- Cyclical Learning Rates: 設定學習率的上下限後，讓學習率在一個範圍內衰降或增加，在遇到saddle point不會卡住



Batch size

$$\theta^* = \arg \min_{\theta} L$$

➤ (Randomly) Pick initial values θ^0

➤ Compute gradient $\mathbf{g}^0 = \nabla L^1(\theta^0)$

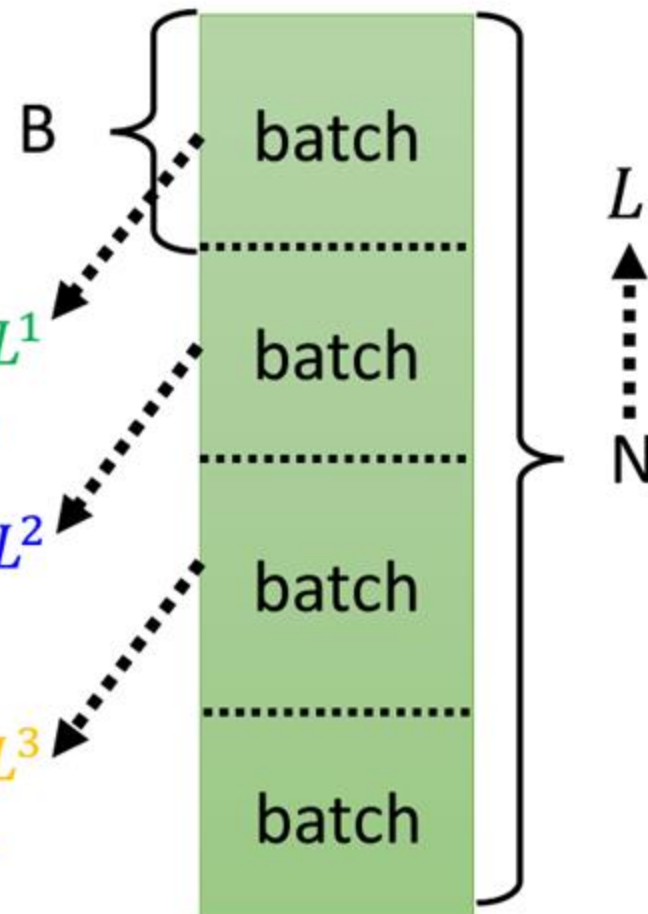
$$\text{update } \theta^1 \leftarrow \theta^0 - \eta \mathbf{g}^0$$

➤ Compute gradient $\mathbf{g}^1 = \nabla L^2(\theta^1)$

$$\text{update } \theta^2 \leftarrow \theta^1 - \eta \mathbf{g}^1$$

➤ Compute gradient $\mathbf{g}^3 = \nabla L^3(\theta^2)$

$$\text{update } \theta^3 \leftarrow \theta^2 - \eta \mathbf{g}^3$$



1 **epoch** = see all the batches once → **Shuffle** after each epoch

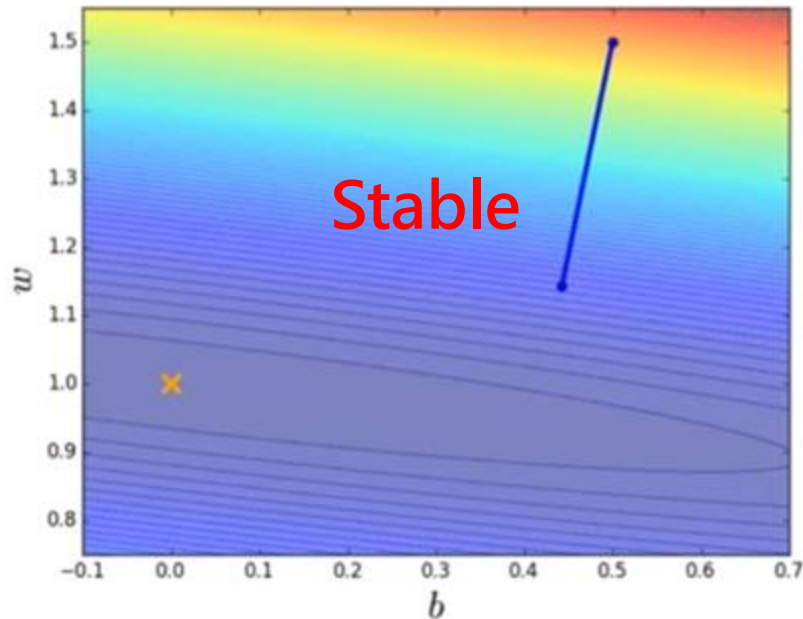
Large batch v.s. small batch



Consider 20 example($N=20$)

Batch size = N (Full batch)

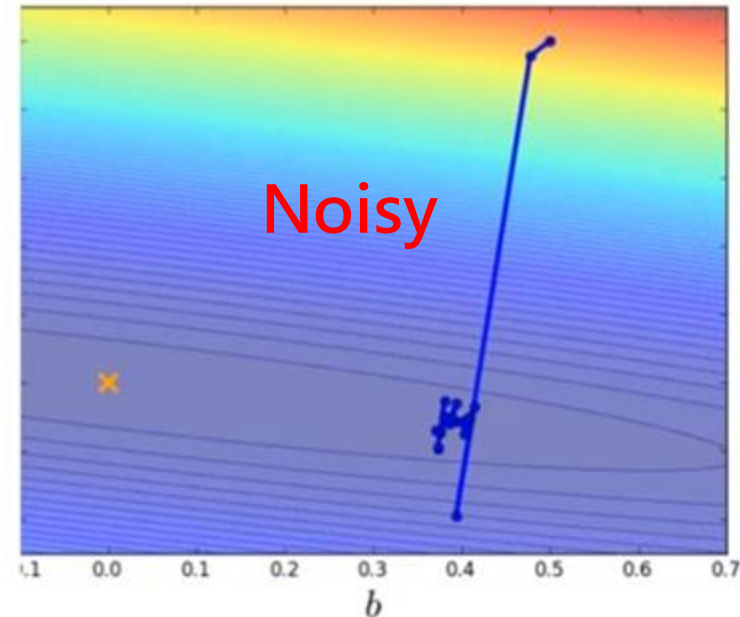
Update after seeing
all 20 examples



Batch size = 1

Update for each example

Update 20 times in an epoch

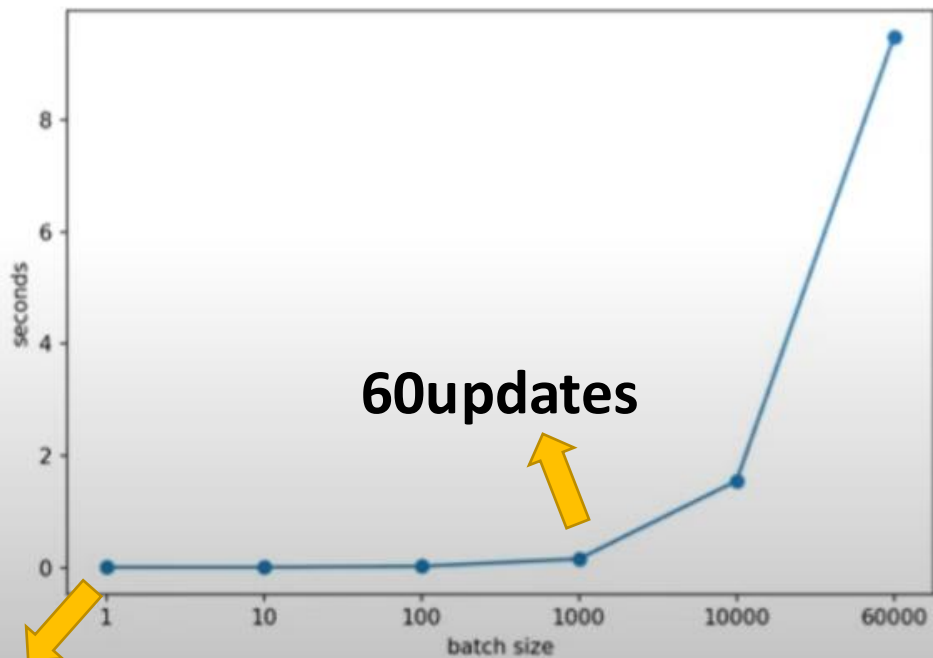


Large batch v.s. small batch



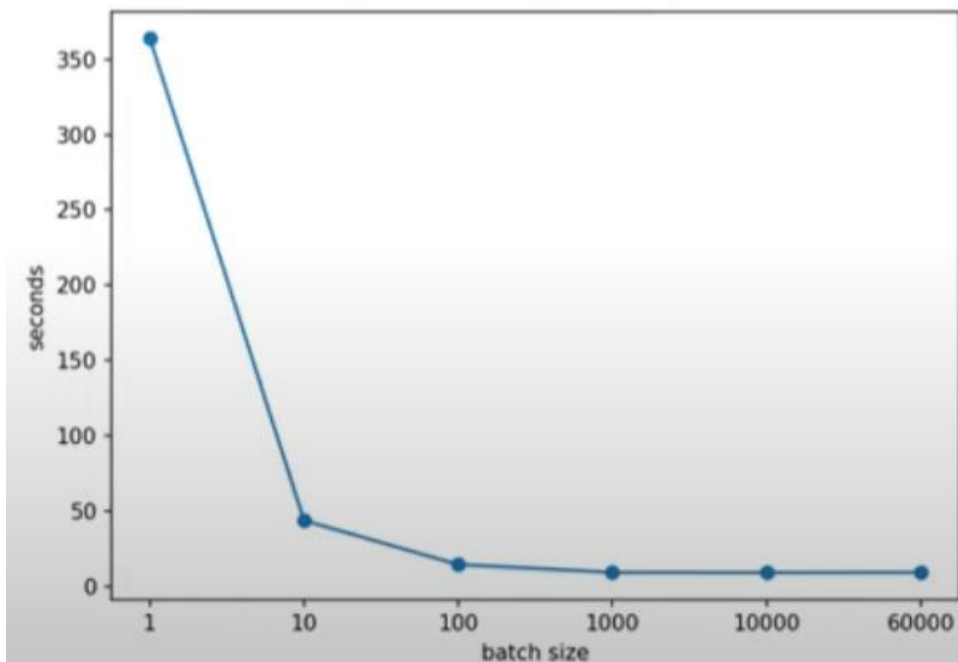
- Larger batch size doesn't require longer time to compute gradient(update)
- Smaller batch requires longer time for one epoch(考慮平行計算)

Time for one **update**



60000 updates in an epoch

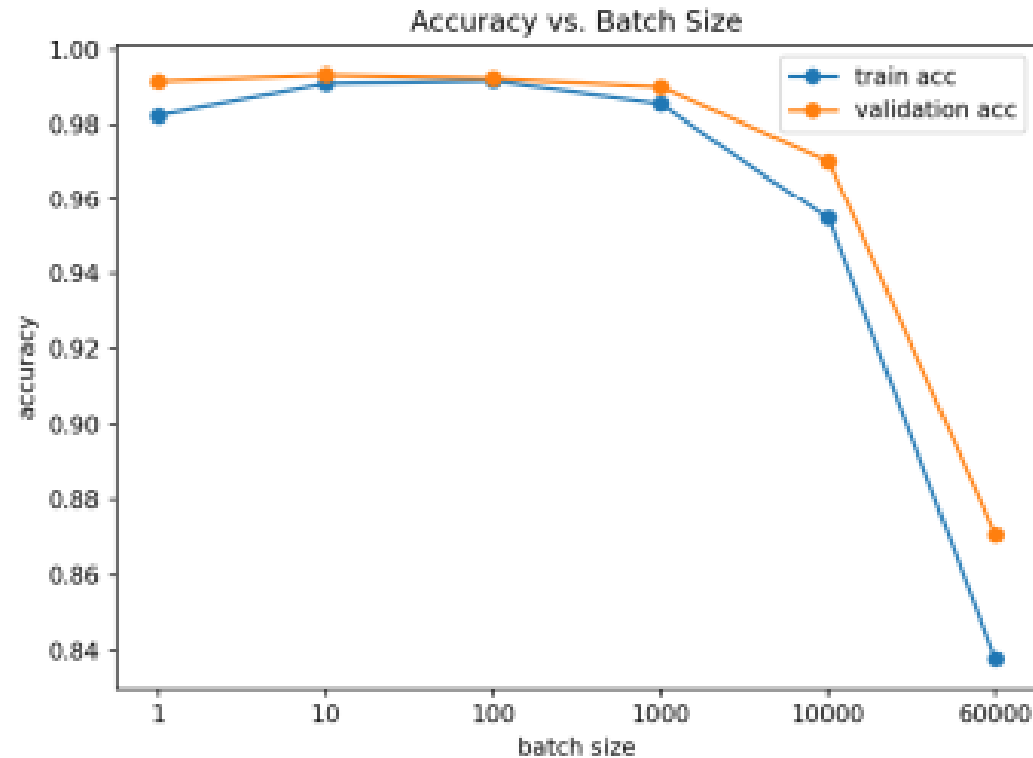
Time for one **epoch**



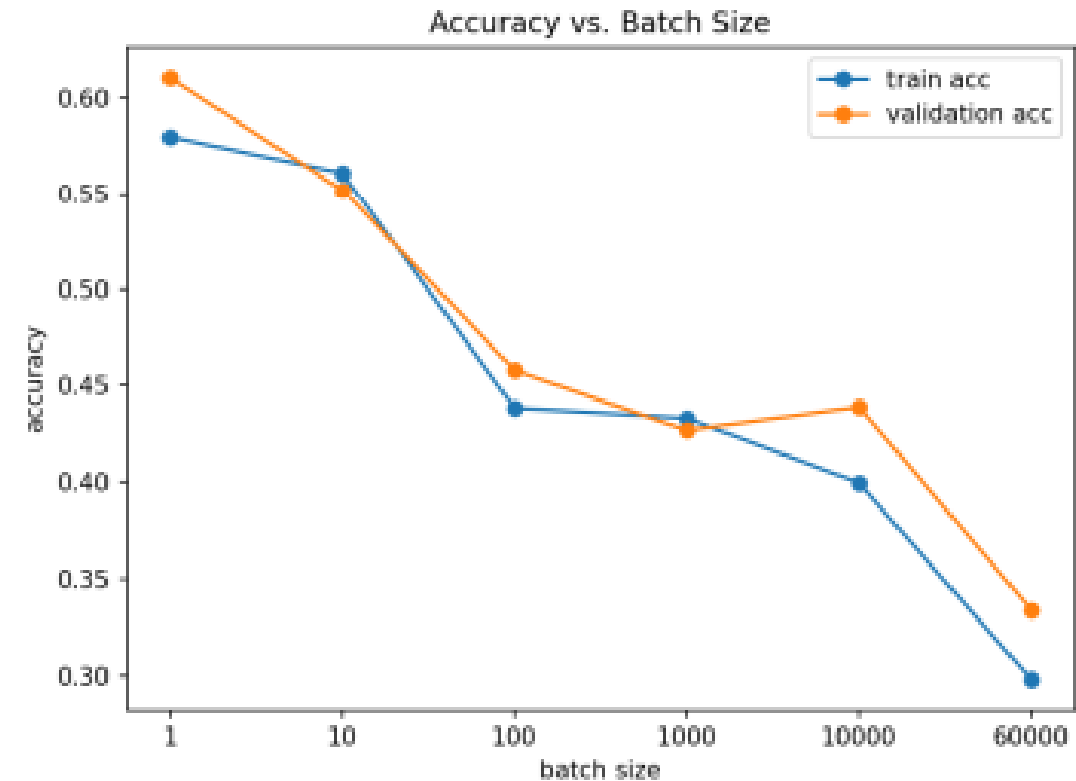
Large batch v.s. small batch



MNIST






CIFAR-10



➤ Smaller batch size has better performance

Batch size



	Small	Large
Speed for one update (no parallel)	Faster	Slower
Speed for one update (with parallel)	Same	Same (not too large)
Time for one epoch	Slower	Faster 
Gradient	Noisy	Stable
Optimization	Better 	Worse
Generalization	Better 	Worse

Batch size is a hyperparameter you have to decide

參考: <https://speech.ee.ntu.edu.tw/~hylee/ml/ml2021-course-data/small-gradient-v7.pdf>

Outline



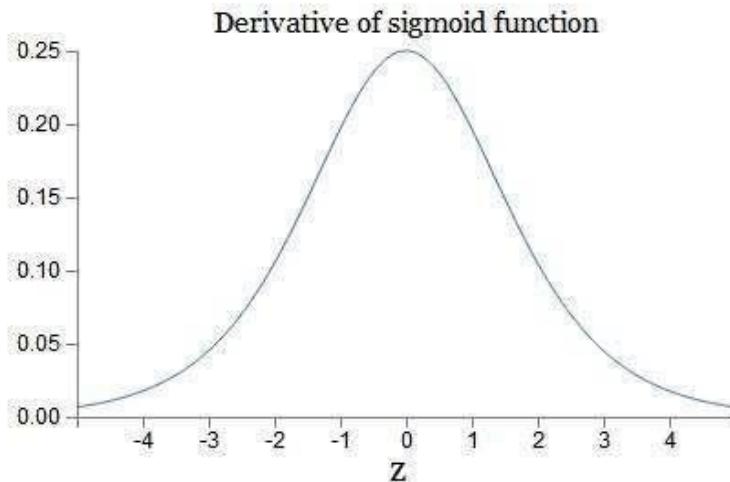
1. Lab2 task
2. Layer introduction
- 3. ResNet18**
4. CIFAR-10
5. Sample code introduction

Vanishing gradient problem



$$\begin{aligned} \frac{\partial E_{total}}{\partial w_1} &= \frac{\partial E_{total}}{\partial y_4} \frac{\partial y_4}{\partial z_4} \frac{\partial z_4}{\partial x_4} \frac{\partial x_4}{\partial z_3} \frac{\partial z_3}{\partial x_3} \frac{\partial x_3}{\partial z_2} \frac{\partial z_2}{\partial x_2} \frac{\partial x_2}{\partial z_1} \frac{\partial z_1}{\partial w_1} \\ &= \frac{\partial E_{total}}{\partial y_4} \sigma'(z_4) w_4 \sigma'(z_3) w_3 \sigma'(z_2) w_2 \sigma'(z_1) x_1 \end{aligned}$$

因為我們初始化的權重通常是在0附近的小數， $w_2 * w_3 * w_4$ 會很小，導致 w_1 的梯度消失



若我們activation function是使用sigmoid，因為sigmoid導數的閾值是(0,0.25)，導致神經網絡在反向傳播的時候，其梯度越來越小，最後甚至根本無法訓練

使用Batch Normalization或改用ReLU可以解決

Degradation



- 但當深度逐漸增加，我們發現56層的神經網路反而比20層網路結果還差。這樣的結果並非來自於 **overfitting** 和 **Vanishing gradient problem**，而是因為深度增加連帶著使得 **training error** 增加所導致的退化問題，以至於深層的特徵丟失了淺層特徵的原始模樣

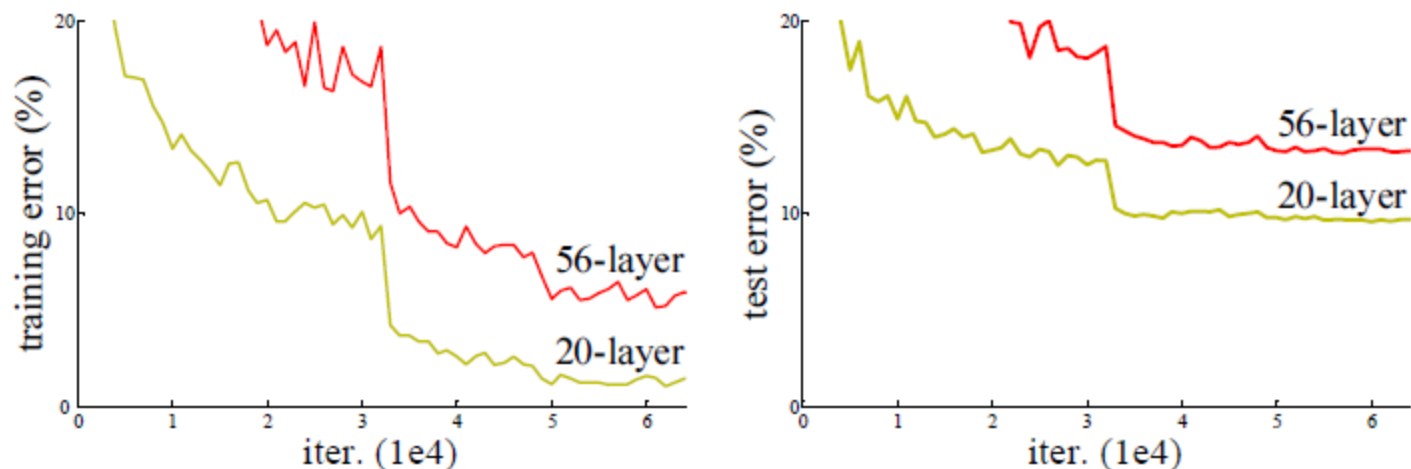
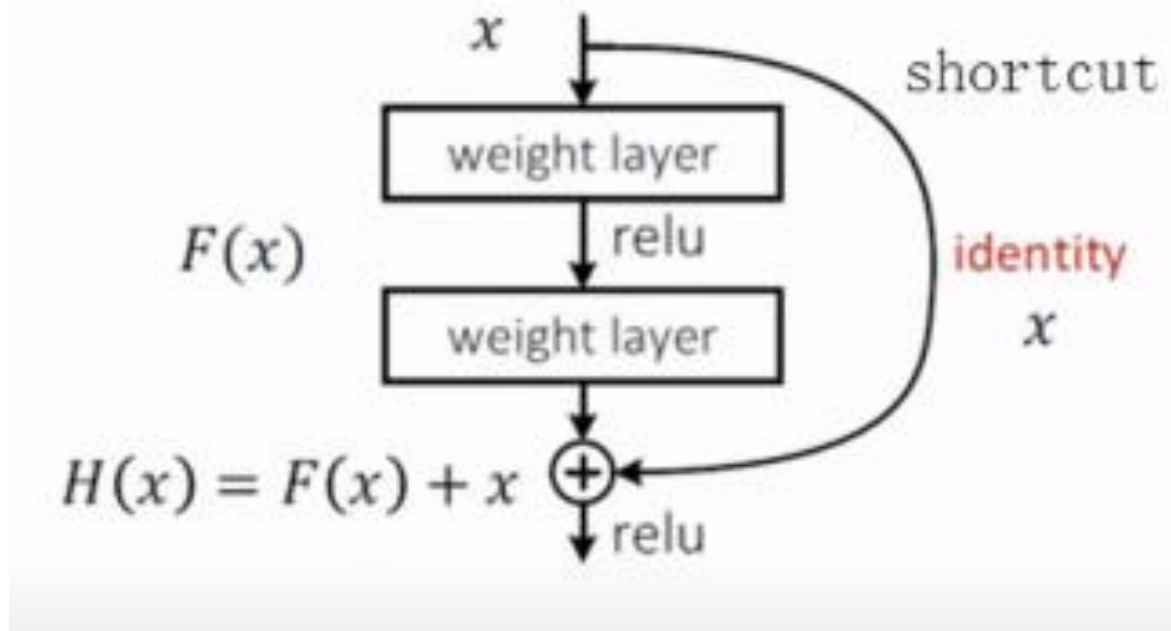


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

ResNet-18



- 用Deep residual Network來處理degradation，這樣做能在網路層加深後，正確率至少不會變的更差。



輸入是 x

原本學習是: $x \rightarrow F(x)$

改成學習: $H(x) = F(x) + x$

ResNet18-implement



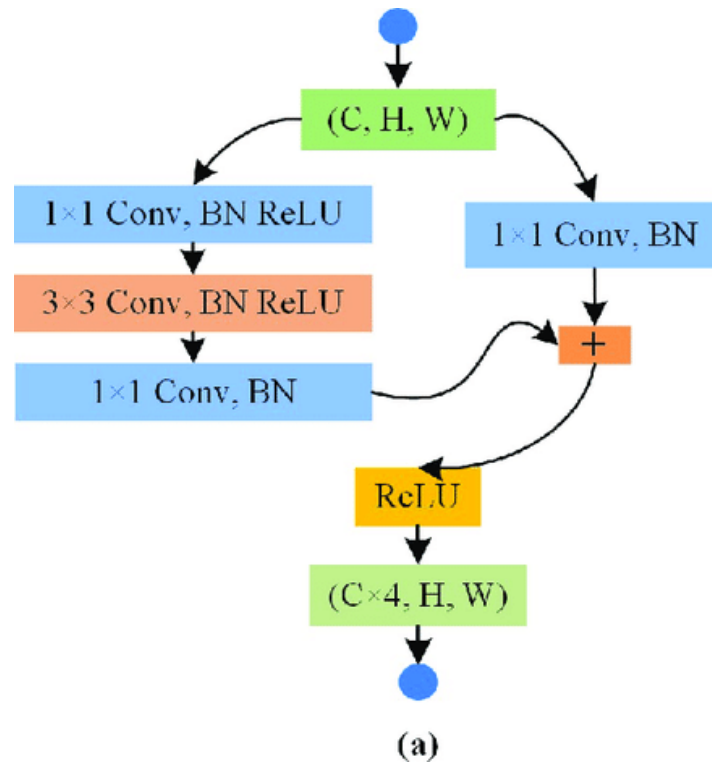
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

[ResNet paper](#)

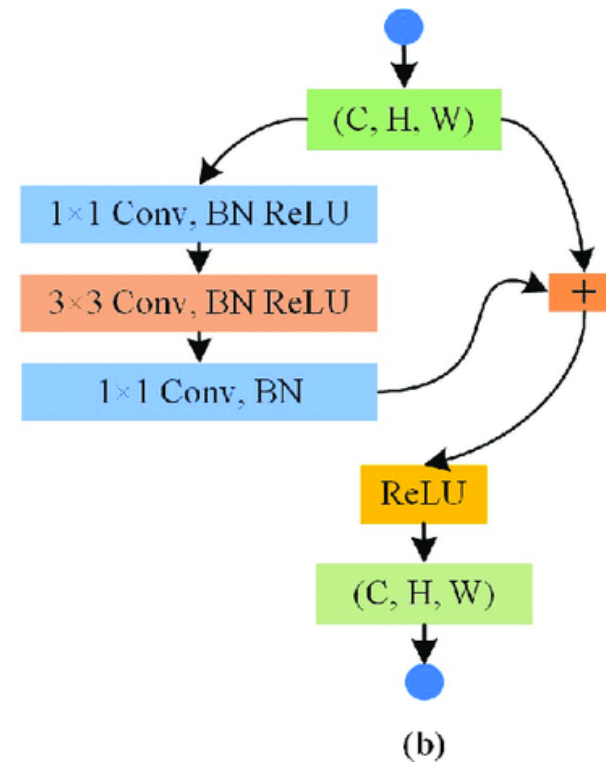
ResNet-18



- Conv block v.s identity block

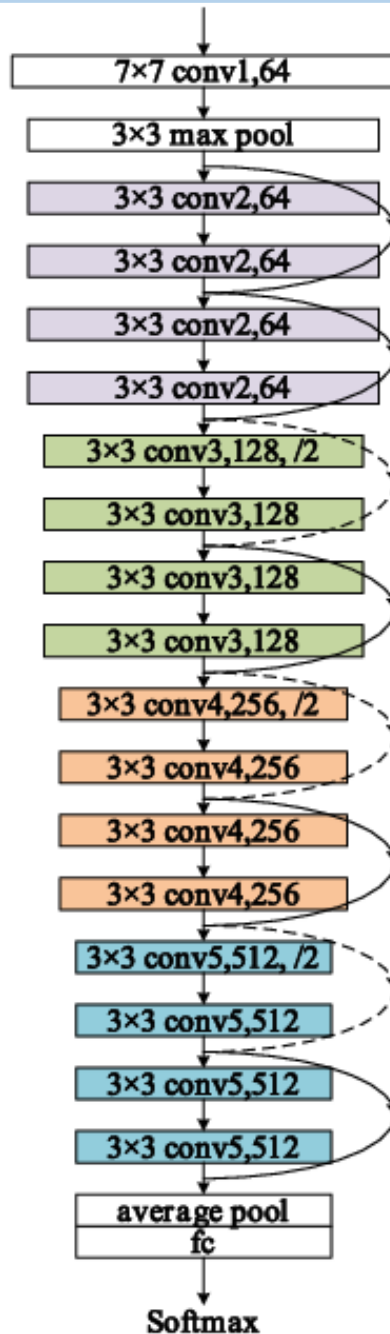


Convolution block



identity block

ResNet18



實線表示維度相同
計算方式為 $H(x)=F(x)+x$

虛線表示維度不同
計算方式為 $H(x)=F(x)+Wx$
其中 W 是 $1*1$ 的卷積，調整 x 的維度

[ResNet paper](#)

ResNet18-implement



先完成make_layer，再分別接上4層layer

```
# make layers
#self.layer1 = ...
#self.layer2 = ...
#self.layer3 = ...
#self.layer4 = ...
#self.fc = ...

#This function is primarily used to repeat the same residual block
def make_layer(self, block, channels, num_blocks, stride):
```

[ResNet paper](#)

Outline



1. Lab2 task
2. Layer introduction
3. ResNet18
4. **CIFAR-10**
5. Sample code introduction

CIFAR-10

- CIFAR-10 consists of 32x32 colour images in 10 classes
- 50000 training images + 10000 test images

airplane



automobile



bird



cat



deer



dog



frog



horse



ship



truck



Outline



1. Lab2 task
2. Layer introduction
3. ResNet18
4. CIFAR-10
5. Sample code introduction

4. Sample code introduction



Task 1

1. 搭建由{CNN,BN,ReLU}所組成的layer
2. 用兩層layer搭配pooling layer 和 FC layer創建出model
3. 進行訓練並分別繪製出train acc/train loss/val acc/ val loss 等圖

```
import torch.nn.functional as F

##### create your own model #####
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()

    def forward(self, x):
```

18

記得先上傳resnet18.py

Task 2

1. 完成resnet18.py並上傳
2. 進行訓練並分別繪製出train acc/train loss/val acc/ val loss 等圖

```
from thop import profile
from resnet18 import *
##### 使用 thop 計算 FLOPs 和參數數量 #####
```

4. Sample code introduction



- data augmentation

```
##### data augmentation & normalization #####
transform_train = transforms.Compose([
    transforms.ToTensor(),

    # data augmentation

    # data normalization    # standardization: (image - train_mean) / train_std
    transforms.Normalize(mean=train_mean, std=train_std),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    # data normalization    # standardization: (image - train_mean) / train_std
    transforms.Normalize(mean=test_mean, std=test_std),
])
```

- setting parameter

```
##### setting parameter #####
EPOCH =
pre_epoch =
lr =
device = torch.device("cuda")
```

4. Sample code introduction



Print model summary

```
from torchsummary import summary
model = SimpleCNN().to(device)
summary(model, (3, 32, 32))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 32, 32]	896
BatchNorm2d-2	[-1, 32, 32, 32]	64
ReLU-3	[-1, 32, 32, 32]	0
MaxPool2d-4	[-1, 32, 16, 16]	0
Conv2d-5	[-1, 64, 16, 16]	18,496
BatchNorm2d-6	[-1, 64, 16, 16]	128
ReLU-7	[-1, 64, 16, 16]	0
MaxPool2d-8	[-1, 64, 8, 8]	0
Dropout-9	[-1, 4096]	0
Linear-10	[-1, 10]	40,970

Total params: 60,554
Trainable params: 60,554
Non-trainable params: 0

Import thop calculate flops and params

```
# 創建一個輸入樣本
input = torch.randn(1, 3, 32, 32).to(device)

# 使用 thop 計算 FLOPs 和參數數量
flops, params = profile(model, inputs=(input, ))
```

FLOPs: 5840896.0
Params: 60554.0

4. Sample code introduction



- Train model

```
##### Train model #####
```

```
# 初始化模型損失函數與優化器
```

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer =
```

```
scheduler =
```

```
# 設定參數
```

```
best_model_path = 'best_model.pth' # 模型保存路徑
```

```
# 用於記錄 loss 和 accuracy 的列表
```

```
train_losses = []
```

```
train_accuracies = []
```

```
# 記錄訓練損失和準確率
```

```
# 驗證模型
```

```
model.eval()
```

```
print('Finished Training')
```

4. Sample code introduction



- Apply to testing data

```
# load 你的best model再跑一次testloader  
model.eval()
```

4. Sample code introduction



- Print training process.
- Print test accuracy.

```
Epoch: 1  
learning rate: 0.03  
Train loss: 1.423 | Train acc: 0.487  
Val loss: 1.279 | Val acc: 0.543
```

```
Epoch: 2  
learning rate: 0.03  
Train loss: 0.870 | Train acc: 0.691  
Val loss: 0.796 | Val acc: 0.718
```

```
Epoch: 3  
learning rate: 0.03  
Train loss: 0.640 | Train acc: 0.776  
Val loss: 0.824 | Val acc: 0.716
```

```
Finished Training  
Test Accuracy: 71.30%
```

4. Sample code introduction



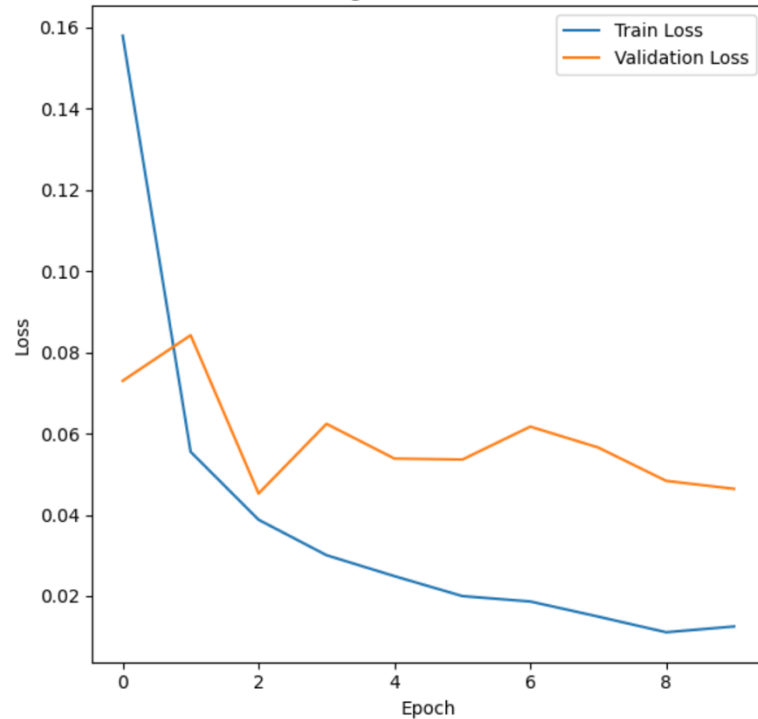
- Print and plot result.

```
import matplotlib.pyplot as plt

##### 繪製 loss 和 accuracy 的圖 #####

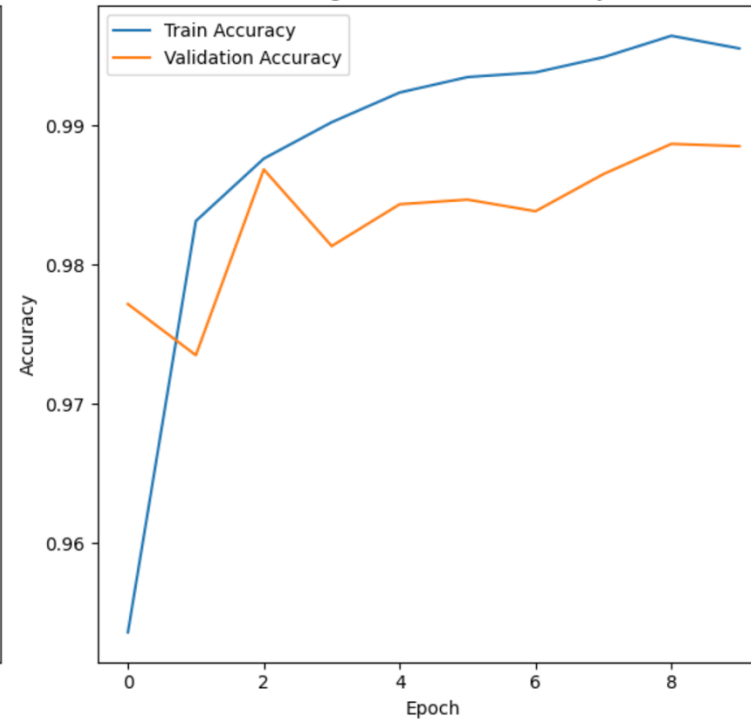
plt.show()
```

Training and Validation Loss



test accuracy: 0.9866

Training and Validation Accuracy



Thank you for listening