自然語言處理

第 5 章 PyTorch 實作篇

講師：紀俊男

本章大綱

- PyTorch實作原理說明
- PyTorch實作範例（多選一分類）
- 以快樂版實作 PyTorch 範例
- 課後作業
- 本章總結

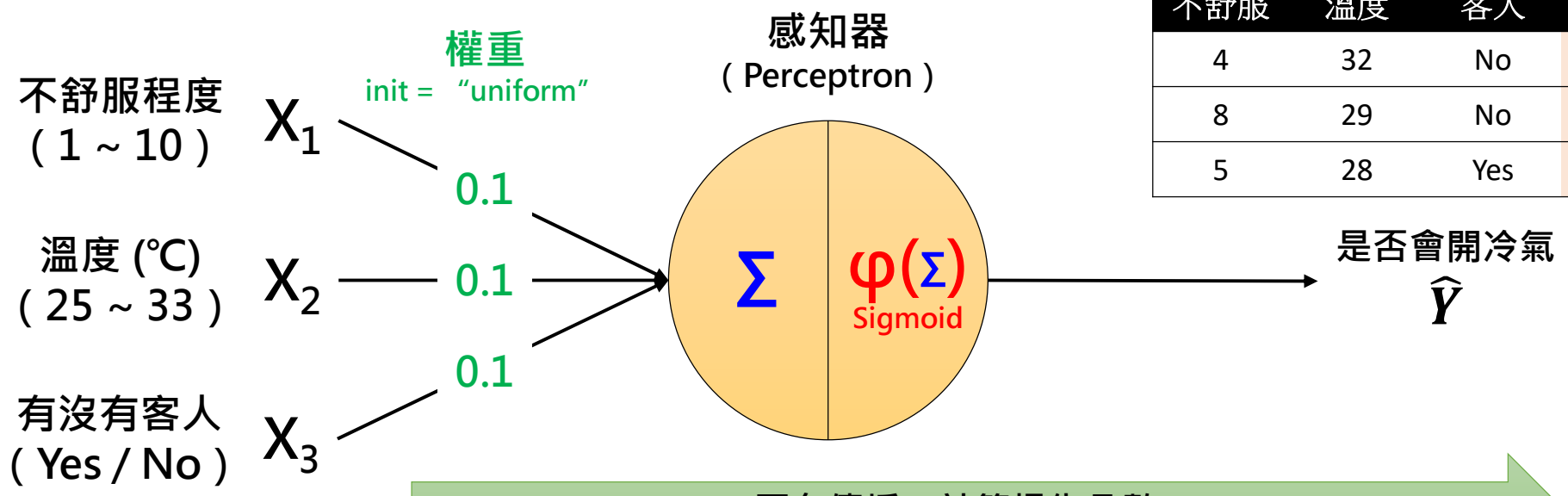
PyTorch
實作原理說明

神經元運作原理複習

平常需要做
「特徵縮放」

5	8	4
28	29	32
1	0	0

不舒服	溫度	客人	開冷氣
4	32	No	No
8	29	No	Yes
5	28	Yes	Yes



正向傳播（計算損失函數）

損失函數

$C = \frac{1}{2}(\hat{Y} - Y)^2$

一樣本點修正一次：隨機梯度下降

K 樣本點修正一次：批次隨機梯度下降

全體樣本點修正一次：一般梯度下降

Σ=4*0.1+32*0.1+0*0.1=3.6 $\hat{Y} = \varphi(3.6) = \frac{1}{1+e^{-3.6}} = 0.9734 = Yes$ ½ (1-0)²=0.5

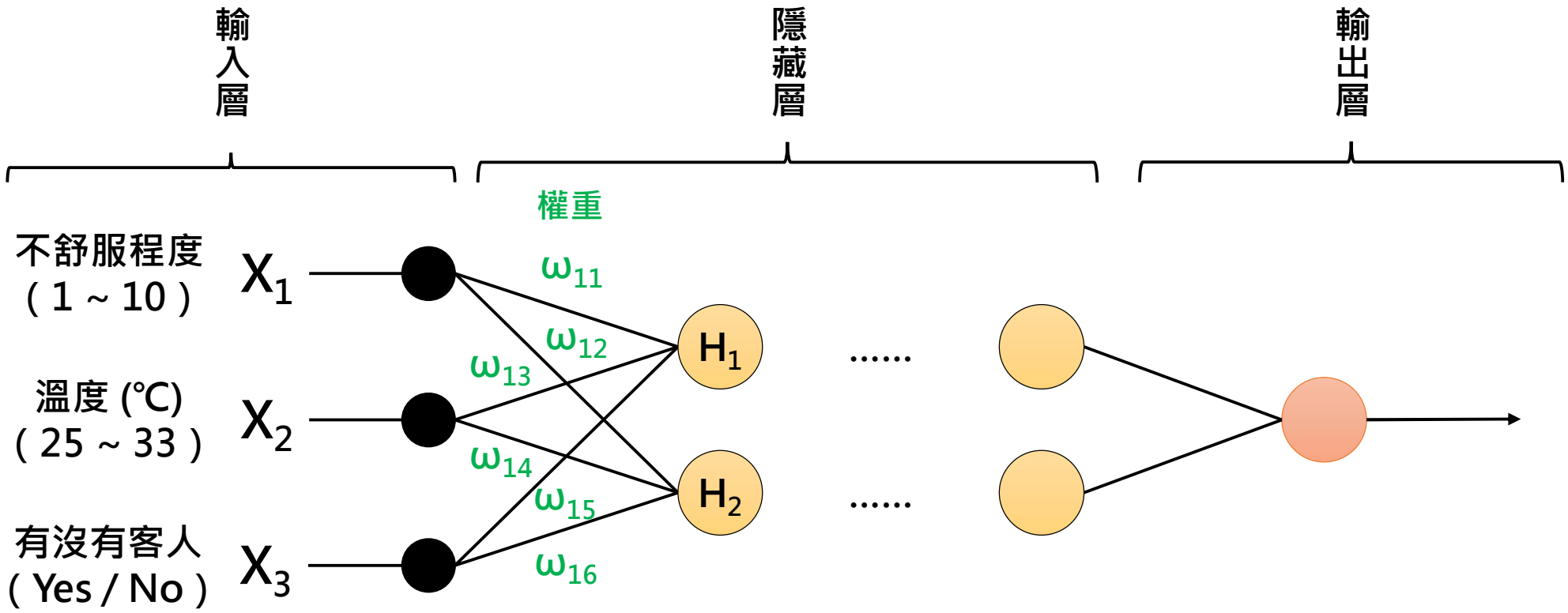
Σ=8*0.1+29*0.1+0*0.1=3.7 $\hat{Y} = \varphi(3.7) = \frac{1}{1+e^{-3.7}} = 0.9758 = Yes$ ½ (1-1)²=0.0

Σ=5*0.1+28*0.1+1*0.1=3.4 $\hat{Y} = \varphi(3.4) = \frac{1}{1+e^{-3.4}} = 0.9677 = Yes$ ½ (1-1)²=0.0

反向傳播（修正權重，讓損失函數有最小（偏微分））

全體樣本點訓練一次 = 一期（Epoch）

神經網路運作原理複習



隱藏層作用

- 增加抽象概念（H1：以不舒服程度為主。H2：以客人有無為主）
- 將模型提昇至「能解決線性不可分問題」的等級

神經網路節點計算公式複習

公式一：算數平均數

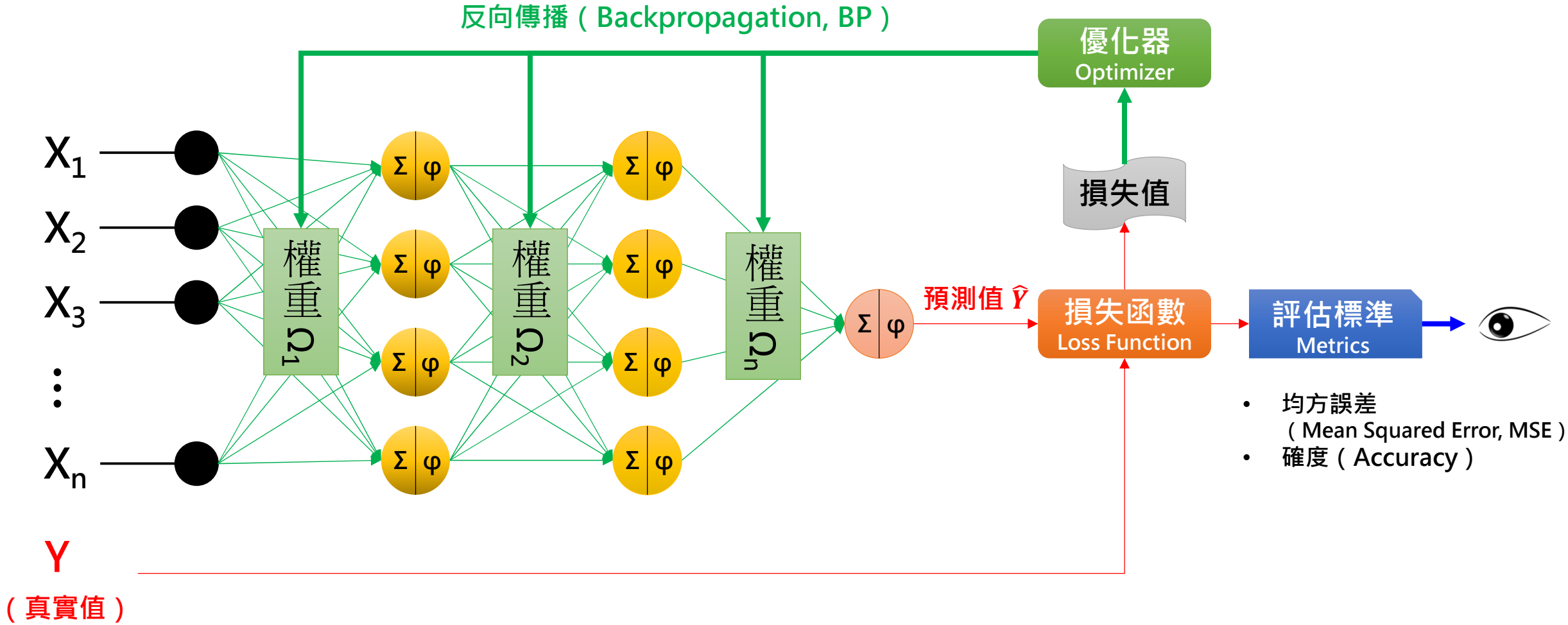
$$\frac{\text{上一層節點數} + \text{下一層節點數}}{2}$$

公式二：經驗法則公式

$$\frac{\text{樣本點個數}}{\alpha \times (\text{上一層節點數} + \text{下一層節點數})}$$

α=2~10（Scaling Factor）
（=2可防止過擬合，一般=5）

神經網路五大元素複習


反向傳播（Backpropagation, BP）
優化器
Optimizer
損失值
X_1
X_2
X_3
X_n
權重 Ω_1
權重 Ω_2
權重 Ω_n
Σ φ
預測值 $\hat{Y}$
損失函數
Loss Function
評估標準
Metrics
均方誤差
（Mean Squared Error, MSE）
確度（Accuracy）
Y
（真實值）

PyTorch 開發步驟

環境設定

- 載入套件
- 檢查 GPU
- 設定超參數

資料前處理

- 資料載入
- 斷句斷詞
- 詞彙表建立
- 資料數位化
- 大小統一

模型訓練

- 定義模型
- 訓練模型
- 模型優化

評估與部署

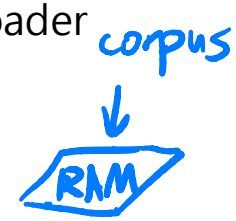
- 模型評估
- 參數調整
- 模型部署

影像與文字的前處理
步驟不盡相同

PyTorch 常用套件

環境設定

- **torch**
  - PyTorch 的核心套件，提供了多維張量的操作、自動微分系統以及其他基本工具。
- **torch.nn**
  - 包含了構建神經網路所需的模組和類別。
  - 如各種神經層（全連接層、卷積層等）、激活函數、損失函數等 fully connected layer　convolutional layer
- **torch.optim**
  - 提供了多種優化器。如 SGD、Adam、RMSprop 等。
- **torchvision**　(包含一些 dataset)
  - 一個用於計算機視覺的套件，提供了常見的圖像資料集、預訓練模型以及轉換工具。
- **torchtext**
  - 一個用於自然語言前處理與資料載入的套件。
- **torch.utils.data**
  - 包含了用於數據載入和處理的工具，如 DataLoader 類別。
  - 可以方便地進行批次載入和資料洗牌。



- **torch.autograd**
  - 實作自動微分，用於計算梯度，找到參數的最佳解。
- **torch.cuda**　↘ 權重優化的方向
  - 用於 GPU 上，進行張量運算和神經網路訓練。
- **torch.nn.functional**　device driver
  - 提供了一些不具可學習參數的函數，這些函數可以用於構建自訂層或執行特定操作。
- **torch.distributed**
  - 用於分散式訓練，支援如 MPI、Gloo 等多種平台。

PyTorch 常見之內建資料集

資料前處理

- torchvision.datasets.MNIST
  - 包含 0~9 共 10 個類別的手寫數字資料集。
  - 每個類別包含數千張 28x28 灰階圖像。
- torchvision.datasets.CIFAR10
  - 包含 10 個類別的小型圖像分類資料集。
  - 每個類別包含 6000 張 32x32 彩色圖像。
- torchvision.datasets.ImageNet
  - 包含超過 1000 個類別和數百萬張圖像的大型資料集。
  - 總數約 1400 萬張圖片，每張圖片大小不一。

Graph (不介紹)

- torchtext.datasets.IMDB

internet movie database

  - 來自 IMDB、用於情感分析的電影評論資料集。
  - 每條評論被標記為正面或負面。
- torchtext.datasets.WikiText2
  - 一個用於語言模型訓練的資料集，包含維基百科上的文章。
- torchtext.datasets.AG_NEWS
  - 一個新聞主題分類的資料集，包含四個類別：世界、體育、商業和科技。
- torchtext.datasets.Multi30k
  - 一個用於機器翻譯的資料集，包含英語到德語的句子對。

Text (本課)

定義 PyTorch 模型

模型定義

1 自定義一個類別 (class)　2 一定要繼承 torch.nn.Module

```
1  class FullyConnectedNet(nn.Module):
2      def __init__(self):
3          super(FullyConnectedNet, self).__init__()
4          self.fc1 = nn.Linear(28 * 28, 512)
5          self.fc2 = nn.Linear(512, 256)
6          self.fc3 = nn.Linear(256, 128)
7          self.fc4 = nn.Linear(128, 10)
8
9      def forward(self, x):
10         x = x.view(-1, 28 * 28)
11         x = torch.relu(self.fc1(x))
12         x = torch.relu(self.fc2(x))
13         x = torch.relu(self.fc3(x))
14         x = self.fc4(x)
15         return x
```

3 透過 __init__() 定義各神經層

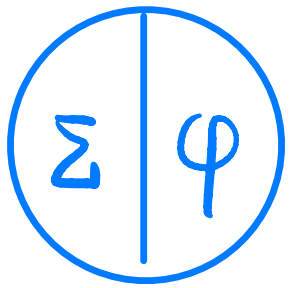
神經層變數名稱可以任取

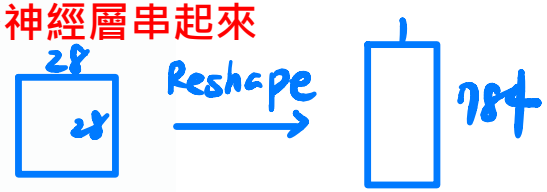
定義各層的輸出入節點數

nn.Linear(fan-in, fan-out)

override

4 透過 forward() 將各神經層串起來





x → ReLU(fc1) → ReLU(fc2) → ReLU(fc3) → fc4 → 應變數回傳

PyTorch 常見的神經層

模型定義

- 全連階層
  - nn.Linear()
- 激活函數層
  - nn.ReLU(), nn.LeakyReLU()
  - nn.Sigmoid(), nn.Softmax()
  - nn.Tanh()
- 文字嵌入層
  - nn.Embedding()
- 卷積層（CNN 模型用）
  - nn.Conv2d()
- 池化層（CNN 模型用）
  - nn.MaxPool2d()
  - nn.AvgPool2d()
- 循環神經網路層（RNN 模型用）
  - nn.RNN()
  - nn.LSTM()
  - nn.GRU()
- 正規化層
  - nn.BatchNorm1d()
  - nn.BatchNorm2d()
  - nn.BatchNorm3d()
  - nn.LayerNorm()
- 丟棄層
  - nn.Dropout()
  - nn.Dropout2d()
  - nn.Dropout3d()

PyTorch 常見的權重初始器

加速收斂

模型定義

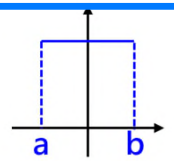
超深度網路 (layer > 50) → 梯度消失/爆炸

- 全零初始化（Zeros Initialization）
  - nn.init.zeros_(tensor)
- 單位初始化（Ones Initialization）
  - nn.init.ones_(tensor)
- 常數初始化（Constant Initialization）
  - nn.init.constant_(tensor, val)

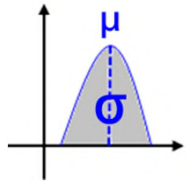
不常用 ↑

2000s

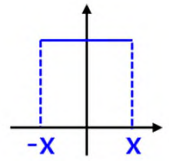
- 均勻分佈（Uniform Distribution）
  - nn.init.uniform_(tensor, a=0.0, b=1.0)



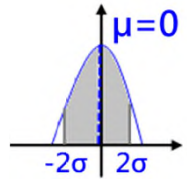
- 常態分佈（Normal Distribution）
  - nn.init.normal_(tensor, mean=0.0, std=1.0)



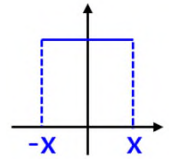
- Xavier Glorot Uniform
  - nn.init.xavier_uniform_(tensor, gain=1.0)
  - gain = α in $x = \alpha \cdot \sqrt{6/(Fan_{in} + Fan_{out})}$



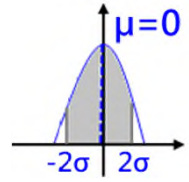
- Xavier Glorot Normal
  - nn.init.xavier_normal_(tensor, gain=1.0)
  - 抽樣 ~ N(μ=0, $\sigma = \sqrt{2/(Fan_{in} + Fan_{out})}$)



- 何愷明均勻分佈（Kaiming Uniform）
  - nn.init.kaiming_uniform_(tensor, a=0, mode='fan_in', nonlinearity='relu')
  - 抽樣 ~ $[-x, x]$ $x = \sqrt{3/mode}$ ，mode 為 Fan_{in} 或 Fan_{out} 個數
  - a 為 nonlinearity = 'leaky_relu' 時的負向斜率



- 何愷明常態分佈（Kaiming Normal）
  - nn.init.kaiming_normal_(tensor, a=0, mode='fan_in', nonlinearity='relu')
  - 抽樣 ~ $N(\mu = 0, \sigma = \sqrt{1/mode})$ ，mode 為 Fan_{in} 或 Fan_{out} 個數

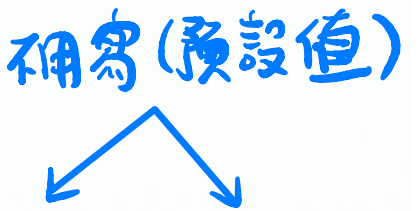


PyTorch 常見的權重初始器

模型定義

- 套用**權重初始器**的程式碼：

```
1  class FullyConnectedNet(nn.Module):
2      def __init__(self):
3          super(FullyConnectedNet, self).__init__()
4          self.fc1 = nn.Linear(28 * 28, 512)
5          self.fc2 = nn.Linear(512, 256)
6          self.fc3 = nn.Linear(256, 128)
7          self.fc4 = nn.Linear(128, 10)
8
9          # 初始化權重為隨機正態分佈
10         init.normal_(self.fc1.weight, mean=0.0, std=1.0)
11         init.normal_(self.fc2.weight, mean=0.0, std=1.0)
12         init.normal_(self.fc3.weight, mean=0.0, std=1.0)
13         init.normal_(self.fc4.weight, mean=0.0, std=1.0)
```



PyTorch 常見的權重初始器

模型定義

- **為何坊間 PyTorch 程式碼鮮少使用權重初始器？**
  - PyTorch 各種神經層，已有內建權重初始器。大部份情況都夠用了
  - 除非想自訂權重初始器，否則通常不會撰寫權重初始器相關程式碼。
  - 以下是各種神經層的內建權重初始器：

- **Linear() 預設權重初始器**
  - nn.init.normal_()
- **Conv2d() 預設權重初始器**
  - nn.init.kaiming_uniform_()
- **RNN() 預設權重初始器**
  - 均勻分佈初始化，初值 ~ $[-\sqrt{k} \quad \sqrt{k}]$，
    $k = \frac{1}{\text{該層節點數}}$
- **LSTM() 預設權重初始器**
  - 均勻分佈初始化，初值 ~ $[-\sqrt{k} \quad \sqrt{k}]$，
    $k = \frac{1}{\text{該層節點數}}$
- **GRU() 預設權重初始器**
  - 均勻分佈初始化，初值 ~ $[-\sqrt{k} \quad \sqrt{k}]$，
    $k = \frac{1}{\text{該層節點數}}$

訓練模型

- (1) 定義「模型訓練」三神器

1. 初始化剛剛自己定義的 PyTorch 模型（可能的話，將它丟入 GPU）

```
1 model = FullyConnectedNet().to(device)
2 
3 criterion = nn.CrossEntropyLoss()
4 
5 optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

2. 定義本模型想要使用的「損失函數」
3. 優化器，負責找到最佳權重

MSELoss()
- 均方差損失函數，迴歸問題時使用

BCELoss()
- 二元交叉熵損失函數，二選一分類問題時使用

CrossEntropyLoss()
- 類別交叉熵損失函數，多選一分類問題時使用

常見優化器
- optim.SGD()
- optim.Adagrad()
- optim.Adadelta()
- optim.RMSprop()
- optim.Adam()

自動代入模型各項參數

學習速率 learning rate

訓練模型

- (2) 定義訓練迴圈

期數, 回合 ≈ 50~100

1 製造一個可以迭代每個 epoch 的迴圈

```
1  for epoch in range(epochs):
2      model.train()
3      for images, labels in train_loader:
4          images, labels = images.to(device), labels.to(device)
5
6          optimizer.zero_grad()
7          outputs = model(images)
8          loss = criterion(outputs, labels)
9          loss.backward()
10         optimizer.step()
11
12     print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}')
```

2 開啟訓練模式，允許權重被更改 → PyTorch 預設權重是開啟 → 好習慣

3 迭代訓練集裡面的每一筆記錄

4 將資料上送 GPU（選做）

5 梯度（=權重優化方向）先歸零（沒有合成向量）

6 以神經網路算出預測值 $\hat{Y}$

7 以損失函數，計算 $Y - \hat{Y}$ 的差異值

8 對差異值實施偏微分，計算梯度（=權重優化方向）autograd

9 將梯度真正套用到現在的權重上面

forward (Propagation)

back (residual 殘差)

印出這一個 epoch，以當下權重，所計算出來的損失值

評估模型

```
1  model.eval()
2  correct = 0
3  total = 0
4  with torch.no_grad():
5      for images, labels in test_loader:
6          images, labels = images.to(device), labels.to(device)
7          outputs = model(images)
8          _, predicted = torch.max(outputs.data, 1)
9          total += labels.size(0)
10         correct += (predicted == labels).sum().item()
11
12 print(f'Accuracy of the model on the 10000 test images: {100 * correct / total:.2f}%')
```

1. 開啟模型的「評估模式」，凍結所有權重
2. 計算 Accuracy = correct / total x 100%
3. 關閉整個神經網路的梯度計算機制（評估時不需要此功能）
4. 取出一個 Batch，丟到 GPU
5. 計算猜對數量（下一個小節詳述）
6. 計算整個模型之正確率（Accuracy）

PyTorch 實作範例

分類問題（多選一）

範例完整原始碼：
https://lurl.cc/aeOvyO

「迴歸問題」vs.「分類問題」

- 迴歸問題 regression

年齡	月薪	購買額
19	19000	0
32	150000	15638
25	33000	0
47	25000	4752
45	26000	1244
46	28000	3677
32	18000	0
18	82000	0
47	49000	6221
48	41000	4576
45	22000	598
35	65000	0

連續數字

X　Y

- 分類問題 classification

年齡	月薪	是否購買
19	19000	0
32	150000	1
25	33000	0
47	25000	1
45	26000	1
46	28000	1
32	18000	0
18	82000	0
47	49000	1
48	41000	1
45	22000	1
35	65000	0

離散數字

X　Y

分類問題 = 應變數 Y 是離散數字的問題

NLP < classification / clustering

本範例問題說明

- 分類「鳶尾花（Iris）資料集」

sklearn.datasets

load_iris()

	花萼長度	花萼寬度	花瓣長度	花瓣寬度	種類
Index	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Iris_Type
0	5.10	3.50	1.40	0.20	0
1	4.90	3.00	1.40	0.20	0
2	4.70	3.20	1.30	0.20	0
3	4.60	3.10	1.50	0.20	0
4	5.00	3.60	1.40	0.20	0
5	5.40	3.90	1.70	0.40	0
6	4.60	3.40	1.40	0.30	0
7	5.00	3.40	1.50	0.20	0
8	4.40	2.90	1.40	0.20	0
9	4.90	3.10	1.50	0.10	0
10	5.40	3.70	1.50	0.20	0

0: Setosa
1: Versicolor
2: Virginica

環境設定

```python
# 下載講師自製的 HappyML
import os

if not os.path.isdir("HappyML"):
  os.system("git clone https://github.com/cnchi/HappyML.git")
```

- **何謂「快樂版」函式庫？**
  - 講師自製、讓你以更短時間，完成相同工作的函式庫。

- **我可以任意使用「快樂版」函式庫嗎？**
  - GitHub 公開原始碼： https://github.com/cnchi/HappyML
  - 自由修改、自由使用，註明出處。

環境設定

```python
# 載入必要套件
import pandas as pd
from sklearn.datasets import load_iris

import HappyML.preprocessor as pp

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
from torch.utils.data import DataLoader, TensorDataset
```

1. 載入資料集之用
2. 載入快樂版前處理函數
3. 載入 PyTorch 相關函式庫

```python
# 檢查是否有可用的 GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

隨堂練習：環境設定

- 請先撰寫好下列原始碼，並且執行看看：

```python
# 下載講師自製的 HappyML
import os

if not os.path.isdir("HappyML"):
  os.system("git clone https://github.com/cnchi/HappyML.git")

# 載入必要套件
import pandas as pd
from sklearn.datasets import load_iris

import HappyML.preprocessor as pp

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
from torch.utils.data import DataLoader, TensorDataset

# 檢查是否有可用的 GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

載入「鳶尾花（Iris）」資料集

```
1  from sklearn.datasets import load_iris
2
3  dataset = load_iris()
```

dataset - Dictionary (6 elements)

Key	Type	Size	Value
DESCR	str	1	.. _iris_dataset:
data	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
feature_names	list	4	['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal ...
filename	str	1	D:\bin\anaconda3\lib\site-packages\sklearn\datasets\data\iris.csv
target	int32	(150,)	[0 0 0 ... 2 2 2]
target_names	str320	(3,)	ndarray object of numpy module

Close

- **DESCR**
  - 本資料集的文字敘述。
- **data**
  - NDArray，所有自變數 X。
- **feature_names**
  - 串列（list），自變數欄位名稱。
- **target**
  - NDArray，應變數 Y。
- **target_name**
  - NDArray，應變數 Y 數字所對應的意義。
  - 如： 0=setosa, 1=versicolor... 等。

載入「自變數」與「應變數」

載入 NDArray，
並包裹成 DataFrame

```
1  import pandas as pd
2
3  X = pd.DataFrame(dataset.data, columns=dataset.feature_names)
4  Y = pd.DataFrame(dataset.target, columns=["Iris_Type"])
5  Y_name = dataset.target_names.tolist()
```

↑ 雖然有載入，但我的程式碼沒用到

自變數 X

Index	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5	3.6	1.4	0.2

應變數 Y

Index	Iris_Type
0	0
1	0
2	0
3	0
4	0

Y_name

Index	Type	Size	Value
0	str	1	setosa
1	str	1	versicolor
2	str	1	virginica

其它資料前處理程式碼

切分訓練集、測試集

特徵縮放

將資料轉成 PyTorch 張量

```python
# 切分訓練集、測試集
X_train, X_test, Y_train, Y_test = pp.split_train_test(x_ary=X, y_ary=Y)

# 特徵縮放
X_train, X_test = pp.feature_scaling(fit_ary=X_train, transform_arys=(X_train, X_test))

# 將資料轉換成 PyTorch 張量
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
Y_train_tensor = torch.tensor(Y_train.values, dtype=torch.long)
Y_test_tensor = torch.tensor(Y_test.values, dtype=torch.long)
```

執行結果

X_train_tensor

Index	sepal length (cm)	petal length (cm)	petal width (cm)
96	-0.21	0.20	0.08
102	1.43	1.15	1.14
75	0.84	0.31	0.21
115	0.61	0.81	1.41
10	-0.57	-1.32	-1.38

Y_train_tensor

Index	Iris_Type
96	1
102	2
75	1
115	2
10	0

X_test _tensor

Index	sepal length (cm)	petal length (cm)	petal width (cm)
63	0.26	0.48	0.21
97	0.37	0.25	0.08

Y_test _tensor

Index	Iris_Type
63	1
97	1

隨堂練習：資料前處理

- 請先撰寫好下列原始碼，並且執行看看：

```
# 載入資料集
dataset = load_iris()

# 切分自變數與應變數
X = pd.DataFrame(dataset.data, columns=dataset.feature_names)
Y = pd.DataFrame(dataset.target, columns=['Iris_Type'])
Y_name = dataset.target_names.tolist()

# 切分訓練集、測試集
X_train, X_test, Y_train, Y_test = pp.split_train_test(x_ary=X, y_ary=Y)

# 特徵縮放
X_train, X_test = pp.feature_scaling(fit_ary=X_train, transform_arys=(X_train, X_test))

# 將資料轉換成 PyTorch 張量
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
Y_train_tensor = torch.tensor(Y_train.values, dtype=torch.long)
Y_test_tensor = torch.tensor(Y_test.values, dtype=torch.long)
```

定義 PyTorch 模型（1）

```python
class IrisModel(nn.Module):

  # 定義神經網路每層架構
  def __init__(self):
    super(IrisModel, self).__init__()

    # 先定義每個神經層
    self.fc1 = nn.Linear(4, 16)
    self.fc2 = nn.Linear(16, 16)
    self.fc3 = nn.Linear(16, 3)

    # 接著初始化每個神經層的權重
    init.xavier_normal_(self.fc1.weight)
    init.xavier_normal_(self.fc2.weight)
    init.xavier_normal_(self.fc3.weight)

  # 定義輸入值 x 如何一路計算到輸出值（正向傳播）
  def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)

    return x
```

1. 自定義一個類別，並繼承 nn.Module
2. 透過 __init__()，定義各神經層，並初始化「父物件」
3. 定義三個全連接層（nn.Linear()）
4. 定義各層的權重初始器（可省略，並用預設值）
5. 透過定義「正向傳播」函數 forward()，
   將各神經層真正連接起來（含激活函數定義）

定義 PyTorch 模型（2）

1 初始化剛剛自己定義的 PyTorch 模型，並將它丟入 GPU

```
1  # 將模型實體化
2  model = IrisModel().to(device)
3
4  # 定義損失函數
5  criterion = nn.CrossEntropyLoss()
6
7  # 定義優化器
8  optimizer = optim.Adam(model.parameters(), lr=0.001)
```

2 本模型為「多選一」問題，使用「類別交叉熵」為損失函數

自動代入模型各項參數

學習速率

MSELoss()
- 均方差損失函數，迴歸問題時使用

BCELoss()
- 二元交叉熵損失函數，二選一分類問題時使用

CrossEntropyLoss()
- 類別交叉熵損失函數，多選一分類問題時使用

3 使用 Adam() 做為優化器，找出最佳權重

常見優化器
- optim.SGD()
- optim.Adagrad()
- optim.Adadelta()
- optim.RMSprop()
- optim.Adam()

隨堂練習：定義 PyTorch 模型

- 請先撰寫好下列原始碼，並且執行看看：

```python
class IrisModel(nn.Module):

  # 定義神經網路每層架構
  def __init__(self):
    super(IrisModel, self).__init__()

    # 先定義每個神經層
    self.fc1 = nn.Linear(4, 16)
    self.fc2 = nn.Linear(16, 16)
    self.fc3 = nn.Linear(16, 3)

    # 接著初始化每個神經層的權重
    init.xavier_normal_(self.fc1.weight)
    init.xavier_normal_(self.fc2.weight)
    init.xavier_normal_(self.fc3.weight)

  # 定義輸入值 x 如何一路計算到輸出值（正向傳播）
  def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)

    return x
```

```python
# 將模型實體化
model = IrisModel().to(device)

# 定義損失函數
criterion = nn.CrossEntropyLoss()

# 定義優化器
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

訓練模型（1）

- 將資料集依照批次數切分好：

torch.utils.data

1 指定一個批次，要有幾筆資料（目前一批 = 32 筆）

```
1  # 將資料集切割成數個 batch
2  batch_size = 32
3
4  train_dataset = TensorDataset(X_train_tensor, Y_train_tensor)
5  test_dataset = TensorDataset(X_test_tensor, Y_test_tensor)
6
7  train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
8  test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

2 將自變數、應變數打成一包，方便一起切割

Dataset wrapping tensors, indexing tensors along the 1st dimension

3 用 DataLoader() 切割資料集

- 訓練集為求公正且不偏，會用 shuffle=True 加以洗牌。
- 測試集不負責訓練模型參數，無公正且不偏問題，故不洗牌。

訓練模型（2）

- 模型訓練迴圈

```python
# 開始訓練
epochs = 100

for epoch in range(epochs):
    # 將模型設定為訓練模式
    model.train()

    # 儲存猜對的數量 & 完整數量
    correct = 0
    total = 0

    # 取出一個 Batch 開始訓練
    for X_batch, Y_batch in train_loader:
        # 將 Batch 的資料轉換到 GPU 上
        X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

        # 先把上一個 Batch 的梯度歸零
        optimizer.zero_grad()
```

1. 指定要訓練 100 epochs
2. 開啟模型的「訓練模式」，讓權重可以被更動
3. 方便未來用 correct / total x 100% 計算 Accuracy
4. 取出一個 Batch（目前=32），丟到 GPU
5. 梯度 = 權重優化方向，每次都要歸零重新計算

訓練模型（2）

- 模型訓練迴圈（續）

```
Epoch 1/100, Loss: 1.1080, Acc: 0.3214
Epoch 2/100, Loss: 1.0511, Acc: 0.3750
Epoch 3/100, Loss: 1.0458, Acc: 0.4464
Epoch 4/100, Loss: 1.0161, Acc: 0.5000
Epoch 5/100, Loss: 0.9469, Acc: 0.5446
Epoch 6/100, Loss: 1.0180, Acc: 0.5893
...
Epoch 97/100, Loss: 0.0716, Acc: 0.9732
Epoch 98/100, Loss: 0.1014, Acc: 0.9732
Epoch 99/100, Loss: 0.1024, Acc: 0.9732
Epoch 100/100, Loss: 0.0903, Acc: 0.9732
```

Next Batch

```
20    # 計算輸出值
21    Y_pred = model(X_batch)
22    # 計算損失值
23    loss = criterion(Y_pred, Y_batch.squeeze())
24    # 根據損失值求微分找損失極小的權重（反向傳播）
25    loss.backward()
26    # 將求出來的權重實際更新上去
27    optimizer.step()
28
29    # 計算第 1 軸的每一列，最大值之索引為何
30    _, predicted = torch.max(Y_pred.data, 1)
31    # 將這一批次有幾筆資料加入到 total 中
32    total += Y_batch.size(0)
33    # 計算猜對的數量（.item() 會協助取得純量）
34    correct += (predicted == Y_batch.squeeze()).sum().item()
35
36  # 計算每個 epoch 的準確率
37  accuracy = correct / total
38  print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}, Acc: {accuracy:.4f}')
```

1 先算出 $\hat{Y}$，再用損失函數算 $\hat{Y}$ $vs.$ Y 的差距

2 .backward() 算權重優化方向（梯度），.step() 負責套用

predicted

3 計算 Y_pred 的每一列，在所有欄位（dim=1）中，最大的數值＆索引值

~~tensor(0.9267)~~, 2 =

	0	1	2
0	0.997379	0.00262133	2.35715e-08
1	2.12966e-06	0.0732935	0.926704

5 算出 Y_batch 第零軸有幾個（亦即：(32, 1) 中的 32）

.squeeze()
(32, 1) → (32,)

6 計算猜對數量

Index	0
27	0
137	2

==

Index	0
27	0
137	2

→ True, True → .sum() [2] → .item() 2

隨堂練習：訓練模型

- 請先撰寫好下列原始碼，並且執行看看：

```python
# 將資料集切割成數個 batch
batch_size = 32

train_dataset = TensorDataset(X_train_tensor, Y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, Y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```python
# 開始訓練
epochs = 100

for epoch in range(epochs):
  # 將模型設定為訓練模式
  model.train()

  # 儲存猜對的數量 & 完整數量
  correct = 0
  total = 0

  # 取出一個 Batch 開始訓練
  for X_batch, Y_batch in train_loader:
    # 將 Batch 的資料轉換到 GPU 上
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # 先把上一個 Batch 的梯度歸零
    optimizer.zero_grad()

    # 計算輸出值
    Y_pred = model(X_batch)
    # 計算損失值
    loss = criterion(Y_pred, Y_batch.squeeze())
    # 根據損失值求微分找損失極小的權重（反向傳播）
    loss.backward()
    # 將求出來的權重實際更新上去
    optimizer.step()

    # 計算第 1 軸的每一列，最大值之索引為何
    _, predicted = torch.max(Y_pred.data, 1)
    # 將這一批次有幾筆資料加入到 total 中
    total += Y_batch.size(0)
    # 計算猜對的數量（.item() 會協助取得純量）
    correct += (predicted == Y_batch.squeeze()).sum().item()

  # 計算每個 epoch 的準確率
  accuracy = correct / total
  print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}, Acc: {accuracy:.4f}')
```

評估模型好壞（1）

1 開啟模型的「評估模式」，凍結所有權重

```python
# 將模型切換為評估模式
model.eval()

# 關閉 PyTorch 的梯度計算機制（Evaluation 時期不需要它）
with torch.no_grad():

    # 儲存猜對的數量 & 完整數量
    correct = 0
    total = 0

    # 取出測試集的一個批次，開始測試
    for X_batch, Y_batch in test_loader:
        # 將 Batch 的資料轉換到 GPU 上
        X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)
```

2

3 方便未來用 correct / total x 100% 計算 Accuracy

4 取出一個 Batch（目前=32），丟到 GPU

評估模型好壞（2）

Next Batch

```
16      # 計算輸出值
17      Y_pred = model(X_batch)
18
19      # 找到每一列預測機率最高的數值與其索引，即模型的預測類別。
20      _, predicted = torch.max(Y_pred.data, 1)
21      # 將這一批次有幾筆資料加入到 total 中
22      total += Y_batch.size(0)
23      # 計算猜對的數量（.item() 會協助取得純量）
24      correct += (predicted == Y_batch.squeeze()).sum().item()
25
26  # 計算每個 epoch 的準確率
27  print(f'Test Accuracy: {correct / total:.4f}')
```

1 算出 $\hat{Y}$，準備拿來跟 Y 對比之用

2 計算猜對數量

3

```
Test Accuracy: 0.9474
```

隨堂練習：評估模型好壞

- 請先撰寫好下列原始碼，並且執行看看：

```python
# 將模型切換為評估模式
model.eval()

# 關閉 PyTorch 的梯度計算機制（Evaluation 時期不需要它）
with torch.no_grad():

  # 儲存猜對的數量 & 完整數量
  correct = 0
  total = 0

  # 取出測試集的一個批次，開始測試
  for X_batch, Y_batch in test_loader:
    # 將 Batch 的資料轉換到 GPU 上
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # 計算輸出值
    Y_pred = model(X_batch)

    # 找到每一列預測機率最高的數值與其索引，即模型的預測類別。
    _, predicted = torch.max(Y_pred.data, 1)
    # 將這一批次有幾筆資料加入到 total 中
    total += Y_batch.size(0)
    # 計算猜對的數量（.item() 會協助取得純量）
    correct += (predicted == Y_batch.squeeze()).sum().item()

  # 計算每個 epoch 的準確率
  print(f'Test Accuracy: {correct / total:.4f}')
```

以快樂版實作 PyTorch範例

分類問題（多選一）

範例完整原始碼：
https://lurl.cc/NXiaX9

快樂版做了哪些事？

- 定義了一個行為類似 tensorflow.keras.Sequential 的類別

HappyML.pytorch.Sequential

```
1   class Sequential(nn.Module):
2       def __init__(self): ···
5
6       def add(self, layer, kernel_initializer=None, activation=None): ···
46
47      def forward(self, x): ···
51
52      def compile(self, optimizer="adam", loss="categorical_crossentropy"): ···
87
88      def summary(self): ···
90
91      def fit(self, x=None, y=None, validation_split=0.0, batch_size=None, epochs=1, shuffle=True): ···
156
157     def predict(self, x, batch_size=None): ···
180
181     def evaluate(self, x=None, y=None, batch_size=None): ···
```

1 負責初始化此模型

2 將神經層加入此模型中

3 用模型計算輸入→輸出的「正向傳播」值

4 設定「優化器」與「損失函數」

5 印出當前模型各神經層的資訊

6 訓練模型

7 用模型與輸出值，計算預測值

8 評估模型效能

環境設定

```python
# 下載講師自製的 HappyML
import os

if not os.path.isdir("HappyML"):
    os.system("git clone https://github.com/cnchi/HappyML.git")
```

```python
# 載入必要套件
import pandas as pd

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

import torch
import torch.nn as nn
from HappyML.pytorch import Sequential
```

1 資料「前處理」相關函數

2 建造「神經網路」相關函數

```python
# 檢查是否有可用的 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

資料集前處理

```
1  # 載入資料集
2  dataset = load_iris()
3  X, Y = dataset.data, dataset.target
4
5  # 切分訓練集、測試集
6  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
7
8  # 特徵縮放
9  scaler = StandardScaler()
10 X_train = scaler.fit_transform(X_train)
11 X_test = scaler.transform(X_test)
12
13 # 轉換成 PyTorch tensor
14 X_train, Y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(Y_train, dtype=torch.long)
15 X_test, Y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(Y_test, dtype=torch.long)
```

隨堂練習：環境設定與前處理

- 請先撰寫好前兩頁的原始碼（拷貝貼上可），並且執行看看：

```python
# 下載講師自製的 HappyML
import os

if not os.path.isdir("HappyML"):
  os.system("git clone https://github.com/cnchi/HappyML.git")

# 載入必要套件
import pandas as
from sklearn.dat
from sklearn.moc
from sklearn.pre

import torch
import torch.nn
from HappyML.pyt
```

```python
# 檢查是否有可用的 GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")

# 載入資料集
dataset = load_iris()
X, Y = dataset.data, dataset.target

# 切分訓練集、測試集
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

# 特徵縮放
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 轉換成 PyTorch tensor
X_train, Y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(Y_train, dtype=torch.long)
X_test, Y_test = torch.tensor(X_test, dtype=torch.float32), torch.tensor(Y_test, dtype=torch.long)
```

定義模型

支援任何 torch.nn 神經網路層

支援下列權重初始器
- “random_uniform”
- “random_normal”
- “glorot_uniform”
- “glorrot_normal”
- “he_uniform”
- “he_normal”
- 其它任何 init.* 之初始器

支援下列激活函數
- “relu”
- “leaky_relu”
- “sigmoid”
- “softmax”
- “tanh”
- “linear”
- 其它任何 nn.* 之激活函數

1
2
3
4

```
1  # 以快樂版建立模型，並嘗試丟入 GPU 中
2  model = Sequential().to(device)
3
4  # 定義模型的每個神經層、權重初始器、與激活函數
5  model.add(nn.Linear(4, 16), kernel_initializer="glorot_normal", activation="relu")
6  model.add(nn.Linear(16, 16), kernel_initializer="glorot_normal", activation="relu")
7  model.add(nn.Linear(16, 3), kernel_initializer="glorot_normal", activation="softmax")
8
9  # 加入優化器、損失函數，並將模型編譯好
10 model.compile(optimizer="adam", loss="categorical_crossentropy")
11
12 # 印出模型當前樣貌（非必要）
13 model.summary()
```

```
Sequential(
  (layers): ModuleList(
    (0): Linear(in_features=4, out_features=16, bias=True)
    (1): ReLU()
    ...
    (4): Linear(in_features=16, out_features=3, bias=True)
    (5): Softmax(dim=-1)
  )
  (criterion): CrossEntropyLoss()
)
```

支援下列優化器
- “sgd”
- “adagrad”
- “adadelta”
- “rmsprop”
- “adam”
- “adamax”
- “nadam”
- 其它任何 optim.* 之優化器

支援下列損失函數
- “mse”
- “binary_crossentropy”
- “categorical_crossentropy”
- 其它任何 nn.* 之損失函數

隨堂練習：定義模型

- 請先撰寫好前一頁的原始碼 ，並且執行看看：

```python
# 以快樂版建立模型，並嘗試丟入 GPU 中
model = Sequential().to(device)

# 定義模型的每個神經層、權重初始器、與激活函數
model.add(nn.Linear(4, 16), kernel_initializer="glorot_normal", activation="relu")
model.add(nn.Linear(16, 16), kernel_initializer="glorot_normal", activation="relu")
model.add(nn.Linear(16, 3), kernel_initializer="glorot_normal", activation="softmax")

# 加入優化器、損失函數，並將模型編譯好
model.compile(optimizer="adam", loss="categorical_crossentropy")

# 印出模型當前樣貌（非必要）
model.summary()
```

```
Sequential(
  (layers): ModuleList(
    (0): Linear(in_features=4, out_features=16, bias=True)
    (1): ReLU()
    ...
    (4): Linear(in_features=16, out_features=3, bias=True)
    (5): Softmax(dim=-1)
  )
  (criterion): CrossEntropyLoss()
)
```

模型訓練

- 程式碼

```
# 訓練模型
model.fit(x=X_train, y=Y_train, validation_split=0.2, batch_size=32, epochs=500)
```

```
Epoch 1/500 - loss: 1.0305 - acc: 0.3125 - val_loss: 0.9902 - val_acc: 0.4167
Epoch 2/500 - loss: 1.0297 - acc: 0.3125 - val_loss: 0.9894 - val_acc: 0.4167
Epoch 3/500 - loss: 1.0290 - acc: 0.3125 - val_loss: 0.9886 - val_acc: 0.4167
...
Epoch 499/500 - loss: 0.7792 - acc: 0.8646 - val_loss: 0.7540 - val_acc: 0.9167
Epoch 500/500 - loss: 0.7787 - acc: 0.8646 - val_loss: 0.7537 - val_acc: 0.9167
```

預測答案

- 程式碼

```
# 模型預測
Y_pred = model.predict(X_test)

df = pd.DataFrame({'Y_test': Y_test.numpy(), 'Y_pred': Y_pred.numpy()})
print(df)
```

真實值
Y_test

	Y_test	Y_pred
0	1	1
1	0	0
2	2	2
3	1	2
4	1	2
...		
25	2	2
26	2	2
27	2	2
28	0	0
29	0	0

預測值
Y_pred

模型評估

- 程式碼

```
test_loss, test_acc = model.evaluate(X_test, Y_test)
print(f"Loss of Testing Set: {test_loss:.4f}")
print(f"Accuracy of Testing Set: {test_acc:.4f}")
```

```
Loss of Testing Set: 0.7601
Accuracy of Testing Set: 0.8667
```

隨堂練習：模型訓練、預測、評估

- 請先撰寫好下列原始碼，並且執行看看：

```python
# 訓練模型
model.fit(x=X_train, y=Y_train, validation_split=0.2, batch_size=32, epochs=500)

# 模型預測
Y_pred = model.predict(X_test)

df = pd.DataFrame({'Y_test': Y_test.numpy(), 'Y_pred': Y_pred.numpy()})
print(df)

# 模型評估
test_loss, test_acc = model.evaluate(X_test, Y_test)
print(f"Loss of Testing Set: {test_loss:.4f}")
print(f"Accuracy of Testing Set: {test_acc:.4f}")
```

課後作業：幫玻璃劃分等級

- **資料集介紹：玻璃會根據下列九個屬性，劃分成 1~7 等級**
  - **RI**：折射率（Refractive Index）
  - **Na**：鈉含量
  - **Mg**：鎂含量
  - **Al**：鋁含量
  - **Si**：矽含量
  - **K**：鉀含量
  - **Ca**：鈣含量
  - **Ba**：鋇含量
  - **Fe**：鐵含量
  - **Type**：等級，一共有 1~7 等
- **程式要求**
  - 請用 **PyTorch 神經網路**，撰寫一個**「多選一」**的分類器，根據**屬性**自動幫玻璃**分級**。
  - 要能夠做到**訓練**你的 **PyTorch 模型**，並用 **X_test** 做**預測**，將預測值與真實值**並排顯示**。
  - 最後評估一下，您模型的**正確率**有多少？
  - 本作業可以使用標準 **PyTorch 函式庫**，或 **HappyML 快樂版**函式庫作答。

本章總結

- **神經網路運作原理**
  - 輸入值 x **權重** + **激活函數** → **預測值 Y_pred**
  - **損失函數** = f(**預測值 Y_pred**, **真實值 Y**)
  - **梯度** = 權重**優化方向** = **損失函數**微分求得
  - 正向傳播=計算**預測值 Y_pred** vs. **反向傳播**=計算**梯度**＆優化權重
  - 資料集的「**批次（Batch）**」 vs.「**期數（Epochs）**」
- **神經網路五大元素**
  - 神經層：nn.Linear(), nn.Conv2d()...
  - 權重初始器：nn.normal(), nn.xavier_normal()...
  - 損失函數：MSELoss(), BCELoss(), CrossEntropyLoss()
  - 優化器：optim.SGD(), optim.Adam()...
  - 評估標準：Accuracy, Recall, Precision, F1-score, MSE
- **PyTorch 神經網路模型開發流程**
  - 載入資料集
  - 資料集前處理
  - 定義 PyTorch 模型
    - 繼承 nn.Module、__init__() → 定義神經層、forward() → 連接各層
    - 選擇損失函數、選擇優化器
  - 訓練迴圈：model.train(), optim.zero_grad(), loss.backward(), optim.step()
  - 評估模型：model.eval(), torch.no_grad()