5-1 使用 tf.keras

左方有分頁

5-1 使用 tf.keras

5-1 使用 pytorch

5-1 使用 pytorch\_Lightning

5-2 手寫辨認

5-3 Cifar 圖片分類

使用 tf.keras

### **1. 安裝 TensorFlow**

| # 安裝 TensorFlow (如果還沒有安裝)  !pip install tensorflow  import tensorflow as tf  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.datasets import load\_iris  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import OneHotEncoder  from sklearn.preprocessing import StandardScaler  from sklearn.metrics import accuracy\_score |
| --- |

### **2. 載入並處理 Iris 資料集**

我們將使用 sklearn 來載入 Iris 資料集，並將資料集進行處理。

| # 載入 Iris 資料集  iris = load\_iris()  X = iris.data  y = iris.target  # 標準化特徵  scaler = StandardScaler()  X\_scaled = scaler.fit\_transform(X)  # 將標籤進行 one-hot 編碼  encoder = OneHotEncoder(sparse=False)  y\_onehot = encoder.fit\_transform(y.reshape(-1, 1))  # 切分資料集為訓練集和測試集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y\_onehot, test\_size=0.2, random\_state=42) |
| --- |

### **3. 建立並訓練神經網路模型**

現在我們使用 tf.keras 建立一個簡單的神經網路模型，並配置 TensorBoard 回調函數以便於監控訓練過程。

| # 定義神經網路模型  model = tf.keras.Sequential([  tf.keras.layers.Dense(32, activation='relu', input\_shape=(X\_train.shape[1],)),  tf.keras.layers.Dense(32, activation='relu'),  tf.keras.layers.Dense(3, activation='softmax') # 因為是三分類問題  ])  # 編譯模型  model.compile(optimizer='adam',  loss='categorical\_crossentropy',  metrics=['accuracy'])  # 設定 TensorBoard 回調函數  tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir='logs', histogram\_freq=1)  # 訓練模型  history = model.fit(X\_train, y\_train, epochs=50, batch\_size=8,  validation\_data=(X\_test, y\_test),  callbacks=[tensorboard\_callback]) |
| --- |

### **4. 使用 TensorBoard 可視化訓練過程**

| # 啟動 TensorBoard  %load\_ext tensorboard  %tensorboard --logdir logs |
| --- |

### **5. 評估模型**

| # 在測試資料上評估模型  test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test)  print(f"測試損失: {test\_loss}")  print(f"測試準確度: {test\_accuracy}") |
| --- |

### **步驟：**

1. **資料載入與預處理**：
   * 使用 sklearn.datasets 載入 Iris 資料集，並將資料進行標準化處理。
   * 將目標變數（y）進行 one-hot 編碼，因為這是多分類問題。
   * 使用 train\_test\_split 將資料分成訓練集與測試集。
2. **建立神經網路模型**：
   * 使用 tf.keras.Sequential 建立一個包含兩層隱藏層的簡單全連接神經網路。
   * 每一層隱藏層使用 ReLU 激活函數，輸出層使用 Softmax 激活函數來處理多分類問題。
3. **訓練模型並啟用 TensorBoard**：
   * 設定 TensorBoard 回調函數，並在模型訓練過程中啟用它，這樣你可以在 Colab 中查看訓練過程的指標。
   * 使用 model.fit() 進行訓練，並指定 validation\_data 來進行測試集驗證。
4. **啟動 TensorBoard**：
   * 在 Colab 中執行 %load\_ext tensorboard 和 %tensorboard --logdir logs，這樣會在 Colab 介面中啟動 TensorBoard，讓你可以看到訓練過程中的指標。
5. **模型評估**：
   * 最後，在測試集上評估模型的準確度，並印出結果。

**15/15** ━━━━━━━━━━━━━━━━━━━━ **2s** 25ms/step - accuracy: 0.4252 - loss: 0.8490 - val\_accuracy: 0.8000 - val\_loss: 0.7122

Epoch 2/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.7600 - loss: 0.7103 - val\_accuracy: 0.9000 - val\_loss: 0.5901

Epoch 3/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.8008 - loss: 0.6110 - val\_accuracy: 0.9333 - val\_loss: 0.5061

Epoch 4/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - accuracy: 0.8625 - loss: 0.5690 - val\_accuracy: 0.9333 - val\_loss: 0.4362

Epoch 5/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.8356 - loss: 0.4796 - val\_accuracy: 0.9333 - val\_loss: 0.3766

Epoch 6/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9245 - loss: 0.3582 - val\_accuracy: 0.9333 - val\_loss: 0.3251

Epoch 7/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.8322 - loss: 0.4364 - val\_accuracy: 0.9333 - val\_loss: 0.2867

Epoch 8/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.8709 - loss: 0.3594 - val\_accuracy: 0.9333 - val\_loss: 0.2549

Epoch 9/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.8846 - loss: 0.3521 - val\_accuracy: 0.9333 - val\_loss: 0.2290

Epoch 10/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9214 - loss: 0.2569 - val\_accuracy: 0.9333 - val\_loss: 0.2066

Epoch 11/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9435 - loss: 0.2448 - val\_accuracy: 0.9333 - val\_loss: 0.1893

Epoch 12/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9324 - loss: 0.2094 - val\_accuracy: 0.9667 - val\_loss: 0.1709

Epoch 13/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9719 - loss: 0.2252 - val\_accuracy: 1.0000 - val\_loss: 0.1576

Epoch 14/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9702 - loss: 0.1994 - val\_accuracy: 0.9667 - val\_loss: 0.1450

Epoch 15/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9726 - loss: 0.1649 - val\_accuracy: 1.0000 - val\_loss: 0.1295

Epoch 16/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9474 - loss: 0.1856 - val\_accuracy: 1.0000 - val\_loss: 0.1200

Epoch 17/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9768 - loss: 0.1409 - val\_accuracy: 1.0000 - val\_loss: 0.1095

Epoch 18/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 13ms/step - accuracy: 0.9795 - loss: 0.1307 - val\_accuracy: 1.0000 - val\_loss: 0.1029

Epoch 19/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 14ms/step - accuracy: 0.9501 - loss: 0.1566 - val\_accuracy: 1.0000 - val\_loss: 0.0928

Epoch 20/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 13ms/step - accuracy: 0.9882 - loss: 0.1109 - val\_accuracy: 1.0000 - val\_loss: 0.0873

Epoch 21/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 12ms/step - accuracy: 0.9524 - loss: 0.1308 - val\_accuracy: 1.0000 - val\_loss: 0.0825

Epoch 22/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 12ms/step - accuracy: 0.9422 - loss: 0.1245 - val\_accuracy: 1.0000 - val\_loss: 0.0738

Epoch 23/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 11ms/step - accuracy: 0.9510 - loss: 0.1224 - val\_accuracy: 1.0000 - val\_loss: 0.0718

Epoch 24/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 15ms/step - accuracy: 0.9855 - loss: 0.1015 - val\_accuracy: 1.0000 - val\_loss: 0.0638

Epoch 25/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 14ms/step - accuracy: 0.9491 - loss: 0.1158 - val\_accuracy: 1.0000 - val\_loss: 0.0597

Epoch 26/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 14ms/step - accuracy: 0.9357 - loss: 0.1226 - val\_accuracy: 1.0000 - val\_loss: 0.0567

Epoch 27/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 14ms/step - accuracy: 0.9769 - loss: 0.0840 - val\_accuracy: 1.0000 - val\_loss: 0.0562

Epoch 28/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 16ms/step - accuracy: 0.9906 - loss: 0.0588 - val\_accuracy: 1.0000 - val\_loss: 0.0507

Epoch 29/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 14ms/step - accuracy: 0.9467 - loss: 0.1168 - val\_accuracy: 1.0000 - val\_loss: 0.0535

Epoch 30/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9780 - loss: 0.0756 - val\_accuracy: 1.0000 - val\_loss: 0.0447

Epoch 31/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - accuracy: 0.9831 - loss: 0.0666 - val\_accuracy: 1.0000 - val\_loss: 0.0426

Epoch 32/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9674 - loss: 0.1005 - val\_accuracy: 1.0000 - val\_loss: 0.0449

Epoch 33/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9865 - loss: 0.0730 - val\_accuracy: 1.0000 - val\_loss: 0.0410

Epoch 34/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9937 - loss: 0.0494 - val\_accuracy: 1.0000 - val\_loss: 0.0406

Epoch 35/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9549 - loss: 0.0927 - val\_accuracy: 1.0000 - val\_loss: 0.0388

Epoch 36/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - accuracy: 0.9720 - loss: 0.0695 - val\_accuracy: 1.0000 - val\_loss: 0.0379

Epoch 37/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9937 - loss: 0.0393 - val\_accuracy: 1.0000 - val\_loss: 0.0343

Epoch 38/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9815 - loss: 0.0619 - val\_accuracy: 1.0000 - val\_loss: 0.0336

Epoch 39/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9810 - loss: 0.0515 - val\_accuracy: 1.0000 - val\_loss: 0.0328

Epoch 40/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9878 - loss: 0.0566 - val\_accuracy: 1.0000 - val\_loss: 0.0338

Epoch 41/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - accuracy: 0.9700 - loss: 0.0856 - val\_accuracy: 1.0000 - val\_loss: 0.0316

Epoch 42/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9927 - loss: 0.0542 - val\_accuracy: 1.0000 - val\_loss: 0.0323

Epoch 43/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9801 - loss: 0.0636 - val\_accuracy: 1.0000 - val\_loss: 0.0304

Epoch 44/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - accuracy: 0.9869 - loss: 0.0519 - val\_accuracy: 1.0000 - val\_loss: 0.0288

Epoch 45/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9649 - loss: 0.0769 - val\_accuracy: 1.0000 - val\_loss: 0.0285

Epoch 46/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 11ms/step - accuracy: 0.9979 - loss: 0.0359 - val\_accuracy: 1.0000 - val\_loss: 0.0268

Epoch 47/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9666 - loss: 0.0826 - val\_accuracy: 1.0000 - val\_loss: 0.0301

Epoch 48/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9821 - loss: 0.0434 - val\_accuracy: 1.0000 - val\_loss: 0.0230

Epoch 49/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 8ms/step - accuracy: 0.9770 - loss: 0.0650 - val\_accuracy: 1.0000 - val\_loss: 0.0260

Epoch 50/50

**15/15** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - accuracy: 0.9709 - loss: 0.0506 - val\_accuracy: 1.0000 - val\_loss: 0.0257

The tensorboard extension is already loaded. To reload it, use:

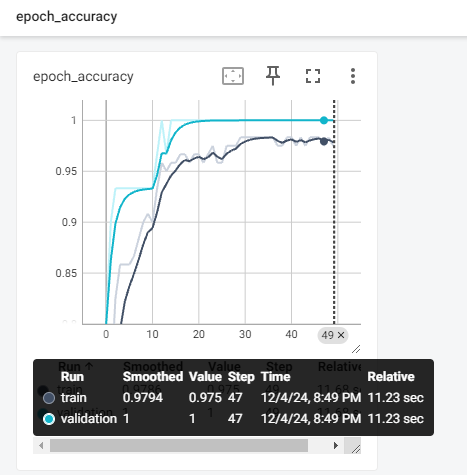
%reload\_ext tensorboard

Reusing TensorBoard on port 6006 (pid 3065), started 0:47:03 ago. (Use '!kill 3065' to kill it.)

**1/1** ━━━━━━━━━━━━━━━━━━━━ **0s** 35ms/step - accuracy: 1.0000 - loss: 0.0257

測試損失: 0.02570234425365925

測試準確度: 1.0



5-1 使用 pytorch

左方有分頁

5-1 使用 tf.keras

5-1 使用 pytorch

5-1 使用 pytorch\_Lightning

5-2 手寫辨認

5-3 Cifar 圖片分類

使用 pytorch

| import torch  import torch.nn as nn  import torch.optim as optim  import numpy as np  from sklearn.datasets import load\_iris  from sklearn.preprocessing import OneHotEncoder  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler  from torch.utils.data import DataLoader, TensorDataset  from torch.utils.tensorboard import SummaryWriter # 引入 TensorBoard  # 1. 載入 Iris 資料集  data = load\_iris()  X = data.data  y = data.target  # 2. 將標籤進行 one-hot 編碼  encoder = OneHotEncoder(sparse\_output=False)  y\_onehot = encoder.fit\_transform(y.reshape(-1, 1))  # 3. 資料標準化  scaler = StandardScaler()  X\_scaled = scaler.fit\_transform(X)  # 4. 分割訓練集與測試集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y\_onehot, test\_size=0.2, random\_state=42)  # 5. 將資料轉換為 Tensor  X\_train\_tensor = torch.tensor(X\_train, dtype=torch.float32)  X\_test\_tensor = torch.tensor(X\_test, dtype=torch.float32)  y\_train\_tensor = torch.tensor(y\_train, dtype=torch.float32)  y\_test\_tensor = torch.tensor(y\_test, dtype=torch.float32)  # 6. 創建 DataLoader  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=16, shuffle=True)  test\_loader = DataLoader(test\_dataset, batch\_size=16, shuffle=False)  # 7. 定義神經網絡模型  class IrisModel(nn.Module):  def \_\_init\_\_(self):  super(IrisModel, self).\_\_init\_\_()  self.fc1 = nn.Linear(4, 64) # 輸入層 4 個特徵，64 個神經元  self.fc2 = nn.Linear(64, 64) # 隱藏層  self.fc3 = nn.Linear(64, 3) # 輸出層 3 類別  # Dropout 防止過擬合  self.dropout = nn.Dropout(0.3)  def forward(self, x):  x = torch.relu(self.fc1(x))  x = self.dropout(x) # Dropout  x = torch.relu(self.fc2(x))  x = self.dropout(x) # Dropout  x = self.fc3(x)  return x  # 8. 創建模型實例  model = IrisModel()  # 9. 設定損失函數和優化器  criterion = nn.CrossEntropyLoss() # CrossEntropyLoss 用於分類問題  optimizer = optim.AdamW(model.parameters(), lr=0.001) # 使用 AdamW 優化器  # 10. 設置學習率調度器  scheduler = optim.lr\_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5, factor=0.5)  # 11. 設定 TensorBoard 寫入器  writer = SummaryWriter(log\_dir='logs') # 設定日誌目錄  # 12. 訓練過程  epochs = 50  for epoch in range(epochs):  model.train()  running\_loss = 0.0  correct\_predictions = 0  total\_samples = 0  for inputs, labels in train\_loader:  optimizer.zero\_grad()  outputs = model(inputs)  loss = criterion(outputs, torch.max(labels, 1)[1]) # CrossEntropy需要整數標籤  loss.backward()  optimizer.step()  running\_loss += loss.item()  \_, predicted = torch.max(outputs, 1)  total\_samples += labels.size(0)  correct\_predictions += (predicted == torch.max(labels, 1)[1]).sum().item()  # 計算訓練集準確度  train\_accuracy = correct\_predictions / total\_samples  avg\_loss = running\_loss / len(train\_loader)  # 驗證過程  model.eval()  correct\_predictions = 0  total\_samples = 0  with torch.no\_grad():  for inputs, labels in test\_loader:  outputs = model(inputs)  \_, predicted = torch.max(outputs, 1)  total\_samples += labels.size(0)  correct\_predictions += (predicted == torch.max(labels, 1)[1]).sum().item()  # 計算測試集準確度  test\_accuracy = correct\_predictions / total\_samples  # 更新學習率調度器  scheduler.step(avg\_loss)  # 記錄訓練和測試指標到 TensorBoard  writer.add\_scalar('Loss/train', avg\_loss, epoch)  writer.add\_scalar('Accuracy/train', train\_accuracy, epoch)  writer.add\_scalar('Accuracy/test', test\_accuracy, epoch)  # 印出訓練與測試結果  print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg\_loss:.4f}, Train Accuracy: {train\_accuracy \* 100:.2f}%, Test Accuracy: {test\_accuracy \* 100:.2f}%")  # 測試最終準確度  model.eval()  correct\_predictions = 0  total\_samples = 0  with torch.no\_grad():  for inputs, labels in test\_loader:  outputs = model(inputs)  \_, predicted = torch.max(outputs, 1)  total\_samples += labels.size(0)  correct\_predictions += (predicted == torch.max(labels, 1)[1]).sum().item()  final\_accuracy = correct\_predictions / total\_samples  print(f"Final Test Accuracy: {final\_accuracy \* 100:.2f}%")  # 記得關閉 TensorBoard 寫入器  writer.close()  # 9. 啟動 TensorBoard（在 Colab 上執行）  %reload\_ext tensorboard  %tensorboard --logdir logs |
| --- |

**Epoch [1/50], Loss: 1.0548, Train Accuracy: 49.17%, Test Accuracy: 80.00%**

**Epoch [2/50], Loss: 0.9292, Train Accuracy: 69.17%, Test Accuracy: 96.67%**

**Epoch [3/50], Loss: 0.8186, Train Accuracy: 81.67%, Test Accuracy: 93.33%**

**Epoch [4/50], Loss: 0.7071, Train Accuracy: 76.67%, Test Accuracy: 93.33%**

**Epoch [5/50], Loss: 0.6407, Train Accuracy: 80.00%, Test Accuracy: 90.00%**

**Epoch [6/50], Loss: 0.5590, Train Accuracy: 80.83%, Test Accuracy: 90.00%**

**Epoch [7/50], Loss: 0.5059, Train Accuracy: 83.33%, Test Accuracy: 90.00%**

**Epoch [8/50], Loss: 0.4376, Train Accuracy: 85.00%, Test Accuracy: 90.00%**

**Epoch [9/50], Loss: 0.4384, Train Accuracy: 84.17%, Test Accuracy: 90.00%**

**Epoch [10/50], Loss: 0.3907, Train Accuracy: 85.00%, Test Accuracy: 93.33%**

**Epoch [11/50], Loss: 0.3339, Train Accuracy: 90.00%, Test Accuracy: 93.33%**

**Epoch [12/50], Loss: 0.3242, Train Accuracy: 88.33%, Test Accuracy: 93.33%**

**Epoch [13/50], Loss: 0.3084, Train Accuracy: 88.33%, Test Accuracy: 93.33%**

**Epoch [14/50], Loss: 0.2842, Train Accuracy: 90.00%, Test Accuracy: 93.33%**

**Epoch [15/50], Loss: 0.2501, Train Accuracy: 90.00%, Test Accuracy: 96.67%**

**Epoch [16/50], Loss: 0.2479, Train Accuracy: 91.67%, Test Accuracy: 96.67%**

**Epoch [17/50], Loss: 0.2287, Train Accuracy: 89.17%, Test Accuracy: 96.67%**

**Epoch [18/50], Loss: 0.2034, Train Accuracy: 91.67%, Test Accuracy: 96.67%**

**Epoch [19/50], Loss: 0.1725, Train Accuracy: 95.00%, Test Accuracy: 100.00%**

**Epoch [20/50], Loss: 0.1795, Train Accuracy: 94.17%, Test Accuracy: 100.00%**

**Epoch [21/50], Loss: 0.1783, Train Accuracy: 95.83%, Test Accuracy: 100.00%**

**Epoch [22/50], Loss: 0.1655, Train Accuracy: 96.67%, Test Accuracy: 100.00%**

**Epoch [23/50], Loss: 0.1638, Train Accuracy: 95.83%, Test Accuracy: 100.00%**

**Epoch [24/50], Loss: 0.1808, Train Accuracy: 94.17%, Test Accuracy: 100.00%**

**Epoch [25/50], Loss: 0.1623, Train Accuracy: 94.17%, Test Accuracy: 100.00%**

**Epoch [26/50], Loss: 0.1444, Train Accuracy: 94.17%, Test Accuracy: 96.67%**

**Epoch [27/50], Loss: 0.1313, Train Accuracy: 95.00%, Test Accuracy: 96.67%**

**Epoch [28/50], Loss: 0.1709, Train Accuracy: 94.17%, Test Accuracy: 100.00%**

**Epoch [29/50], Loss: 0.1233, Train Accuracy: 95.83%, Test Accuracy: 100.00%**

**Epoch [30/50], Loss: 0.1161, Train Accuracy: 96.67%, Test Accuracy: 96.67%**

**Epoch [31/50], Loss: 0.1330, Train Accuracy: 94.17%, Test Accuracy: 100.00%**

**Epoch [32/50], Loss: 0.1577, Train Accuracy: 94.17%, Test Accuracy: 100.00%**

**Epoch [33/50], Loss: 0.1084, Train Accuracy: 95.83%, Test Accuracy: 96.67%**

**Epoch [34/50], Loss: 0.1337, Train Accuracy: 94.17%, Test Accuracy: 96.67%**

**Epoch [35/50], Loss: 0.1125, Train Accuracy: 95.00%, Test Accuracy: 96.67%**

**Epoch [36/50], Loss: 0.1522, Train Accuracy: 94.17%, Test Accuracy: 100.00%**

**Epoch [37/50], Loss: 0.0939, Train Accuracy: 98.33%, Test Accuracy: 96.67%**

**Epoch [38/50], Loss: 0.1220, Train Accuracy: 95.83%, Test Accuracy: 96.67%**

**Epoch [39/50], Loss: 0.1490, Train Accuracy: 95.00%, Test Accuracy: 96.67%**

**Epoch [40/50], Loss: 0.1082, Train Accuracy: 95.83%, Test Accuracy: 96.67%**

**Epoch [41/50], Loss: 0.0972, Train Accuracy: 95.83%, Test Accuracy: 96.67%**

**Epoch [42/50], Loss: 0.1075, Train Accuracy: 95.83%, Test Accuracy: 100.00%**

**Epoch [43/50], Loss: 0.1330, Train Accuracy: 95.83%, Test Accuracy: 96.67%**

**Epoch [44/50], Loss: 0.1069, Train Accuracy: 97.50%, Test Accuracy: 96.67%**

**Epoch [45/50], Loss: 0.0795, Train Accuracy: 97.50%, Test Accuracy: 96.67%**

**Epoch [46/50], Loss: 0.0884, Train Accuracy: 95.83%, Test Accuracy: 96.67%**

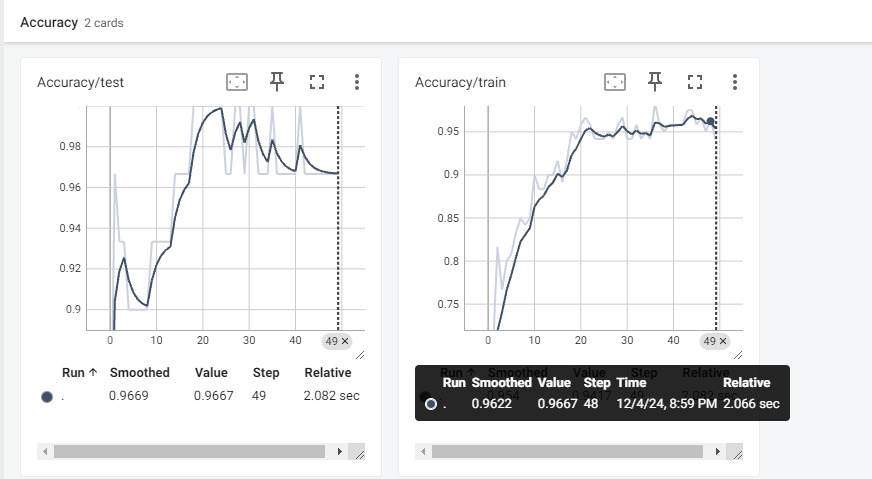
**Epoch [47/50], Loss: 0.1024, Train Accuracy: 96.67%, Test Accuracy: 96.67%**

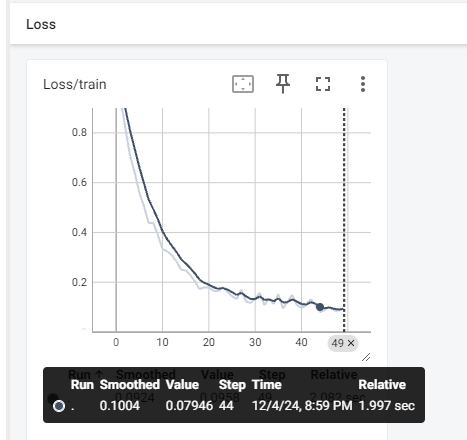
**Epoch [48/50], Loss: 0.0862, Train Accuracy: 95.00%, Test Accuracy: 96.67%**

**Epoch [49/50], Loss: 0.0852, Train Accuracy: 96.67%, Test Accuracy: 96.67%**

**Epoch [50/50], Loss: 0.0958, Train Accuracy: 94.17%, Test Accuracy: 96.67%**

**Final Test Accuracy: 96.67%**

****

****

5-1 使用 pytorch\_Lightning

左方有分頁

5-1 使用 tf.keras

5-1 使用 pytorch

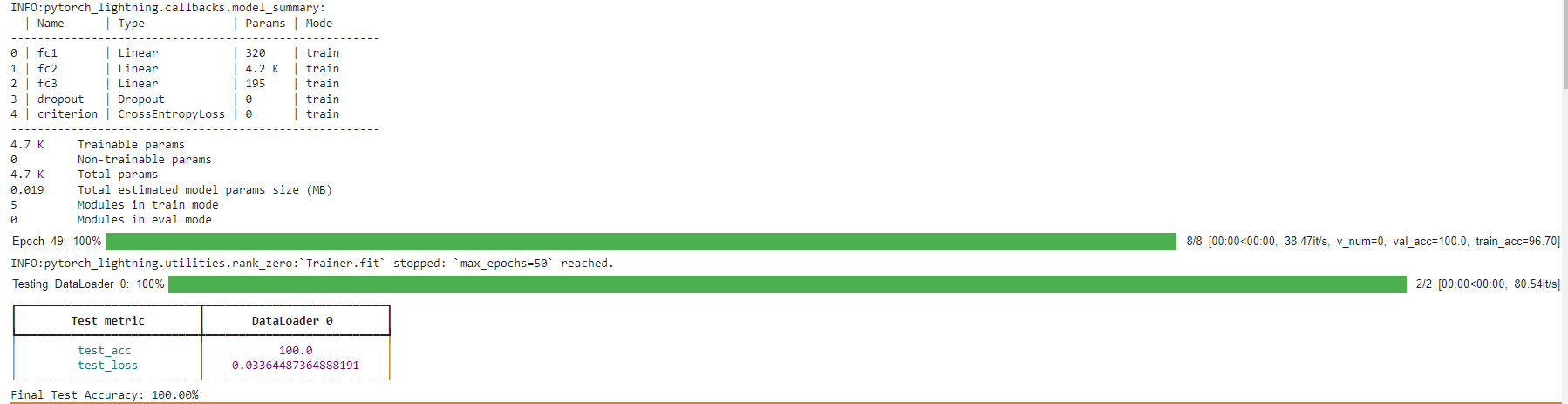
5-1 使用 pytorch\_Lightning

5-2 手寫辨認

5-3 Cifar 圖片分類

使用 pytorch\_Lightning

| !pip install pytorch-lightning  import pytorch\_lightning as pl  import torch  import torch.nn as nn  import torch.optim as optim  from torchmetrics.functional import accuracy  from sklearn.datasets import load\_iris  from sklearn.preprocessing import OneHotEncoder, StandardScaler  from sklearn.model\_selection import train\_test\_split  from torch.utils.data import DataLoader, TensorDataset  # 1. 載入和處理資料  data = load\_iris()  X = data.data  y = data.target  # One-hot 編碼  encoder = OneHotEncoder(sparse\_output=False)  y\_onehot = encoder.fit\_transform(y.reshape(-1, 1))  # 標準化數據  scaler = StandardScaler()  X\_scaled = scaler.fit\_transform(X)  # 分割訓練集與測試集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y\_onehot, test\_size=0.2, random\_state=42)  # 轉換為 Tensor  X\_train\_tensor = torch.tensor(X\_train, dtype=torch.float32)  X\_test\_tensor = torch.tensor(X\_test, dtype=torch.float32)  y\_train\_tensor = torch.tensor(y\_train, dtype=torch.float32)  y\_test\_tensor = torch.tensor(y\_test, dtype=torch.float32)  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=16, shuffle=True)  test\_loader = DataLoader(test\_dataset, batch\_size=16, shuffle=False)  # 2. 定義模型和 LightningModule  class IrisModel(pl.LightningModule):  def \_\_init\_\_(self):  super(IrisModel, self).\_\_init\_\_()  self.fc1 = nn.Linear(4, 64)  self.fc2 = nn.Linear(64, 64)  self.fc3 = nn.Linear(64, 3)  self.dropout = nn.Dropout(0.3)  self.criterion = nn.CrossEntropyLoss()  def forward(self, x):  x = torch.relu(self.fc1(x))  x = self.dropout(x)  x = torch.relu(self.fc2(x))  x = self.dropout(x)  x = self.fc3(x)  return x  def training\_step(self, batch, batch\_idx):  inputs, labels = batch  outputs = self(inputs)  loss = self.criterion(outputs, torch.max(labels, 1)[1])  acc = accuracy(outputs.softmax(dim=-1), torch.max(labels, 1)[1], task="multiclass", num\_classes=3) \* 100 # 轉成百分比  self.log("train\_loss", loss, on\_step=False, on\_epoch=True)  self.log("train\_acc", acc, on\_step=False, on\_epoch=True, prog\_bar=True, logger=True)  return loss  def validation\_step(self, batch, batch\_idx):  inputs, labels = batch  outputs = self(inputs)  loss = self.criterion(outputs, torch.max(labels, 1)[1])  acc = accuracy(outputs.softmax(dim=-1), torch.max(labels, 1)[1], task="multiclass", num\_classes=3) \* 100 # 轉成百分比  self.log("val\_loss", loss, on\_step=False, on\_epoch=True)  self.log("val\_acc", acc, on\_step=False, on\_epoch=True, prog\_bar=True, logger=True)  def test\_step(self, batch, batch\_idx):  inputs, labels = batch  outputs = self(inputs)  loss = self.criterion(outputs, torch.max(labels, 1)[1])  acc = accuracy(outputs.softmax(dim=-1), torch.max(labels, 1)[1], task="multiclass", num\_classes=3) \* 100 # 轉成百分比  self.log("test\_loss", loss, on\_step=False, on\_epoch=True)  self.log("test\_acc", acc, on\_step=False, on\_epoch=True, prog\_bar=True, logger=True)  def configure\_optimizers(self):  optimizer = optim.AdamW(self.parameters(), lr=0.001)  scheduler = optim.lr\_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=5)  return {  "optimizer": optimizer,  "lr\_scheduler": {  "scheduler": scheduler,  "monitor": "val\_loss"  }  }  # 3. 定義 PyTorch Lightning DataModule  class IrisDataModule(pl.LightningDataModule):  def \_\_init\_\_(self, train\_dataset, test\_dataset):  super(IrisDataModule, self).\_\_init\_\_()  self.train\_dataset = train\_dataset  self.test\_dataset = test\_dataset  def train\_dataloader(self):  return DataLoader(self.train\_dataset, batch\_size=16, shuffle=True)  def val\_dataloader(self):  return DataLoader(self.test\_dataset, batch\_size=16, shuffle=False)  def test\_dataloader(self):  return DataLoader(self.test\_dataset, batch\_size=16, shuffle=False)  # 4. 訓練模型  model = IrisModel()  data\_module = IrisDataModule(train\_dataset, test\_dataset)  # 定義 TensorBoard Logger  logger = pl.loggers.TensorBoardLogger("logs", name="iris")  # 訓練器  trainer = pl.Trainer(max\_epochs=50, logger=logger, log\_every\_n\_steps=1)  trainer.fit(model, data\_module)  # 測試模型  result = trainer.test(model, datamodule=data\_module)  # 打印最終測試準確度  final\_accuracy = result[0]['test\_acc'] # 獲取第一個測試結果  print(f"Final Test Accuracy: {final\_accuracy:.2f}%")  # 5. 啟動 TensorBoard（在 Colab 上執行）  %reload\_ext tensorboard  %tensorboard --logdir logs |
| --- |



INFO:pytorch\_lightning.callbacks.model\_summary:

| Name | Type | Params | Mode

-------------------------------------------------------

0 | fc1 | Linear | 320 | train

1 | fc2 | Linear | 4.2 K | train

2 | fc3 | Linear | 195 | train

3 | dropout | Dropout | 0 | train

4 | criterion | CrossEntropyLoss | 0 | train

-------------------------------------------------------

4.7 K Trainable params

0 Non-trainable params

4.7 K Total params

0.019 Total estimated model params size (MB)

5 Modules in train mode

0 Modules in eval mode

Epoch 49: 100%

 8/8 [00:00<00:00, 38.47it/s, v\_num=0, val\_acc=100.0, train\_acc=96.70]

INFO:pytorch\_lightning.utilities.rank\_zero:`Trainer.fit` stopped: `max\_epochs=50` reached.

Testing DataLoader 0: 100%

 2/2 [00:00<00:00, 80.54it/s]

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┓

┃ **Test metric** ┃ **DataLoader 0** ┃

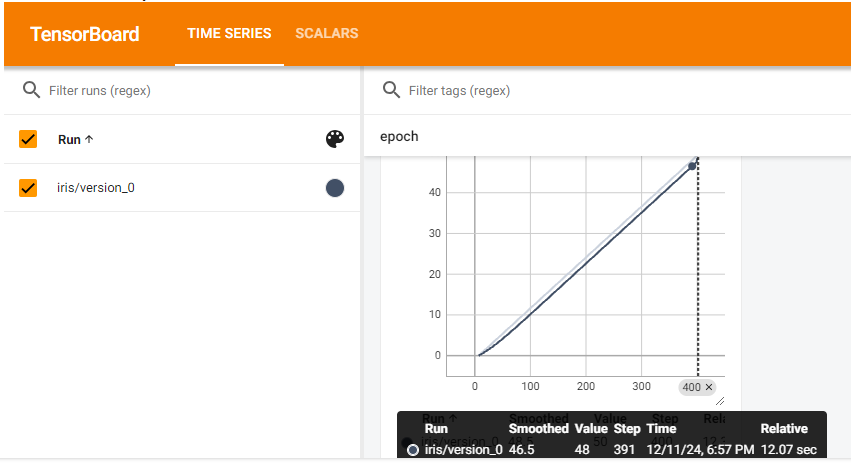
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━┩

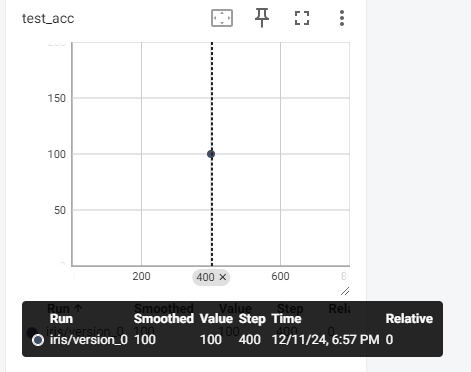
│ test\_acc │ 100.0 │

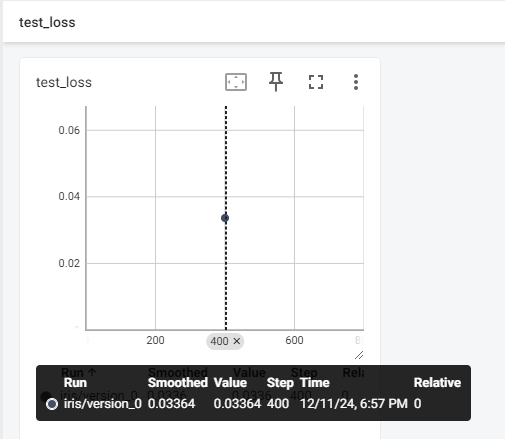
│ test\_loss │ 0.03364487364888191 │

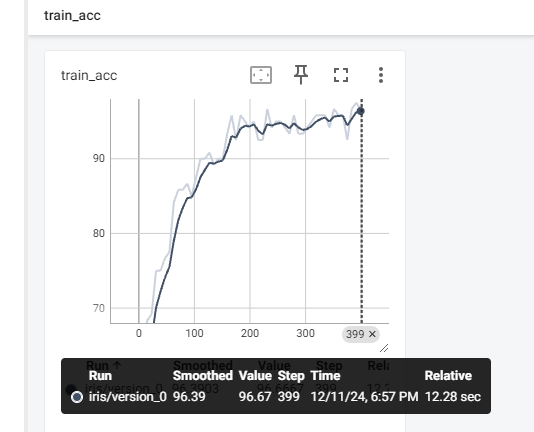
└───────────────────────────┴───────────────────────────┘

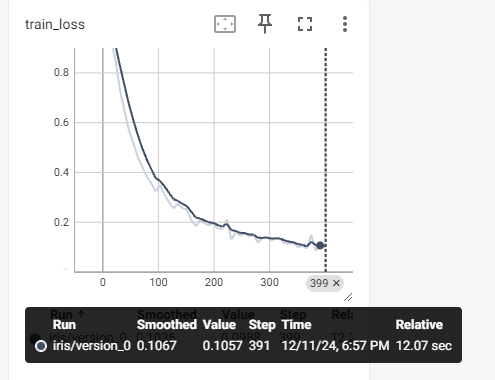
Final Test Accuracy: 100.00%

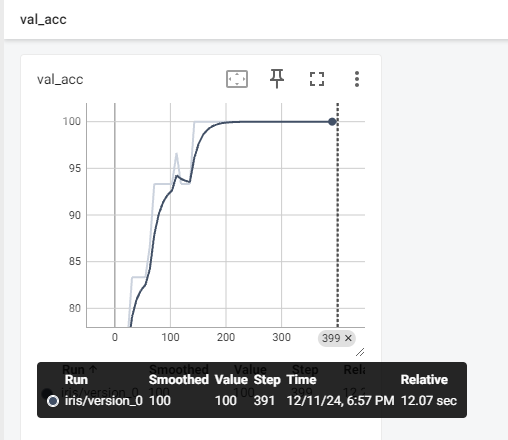


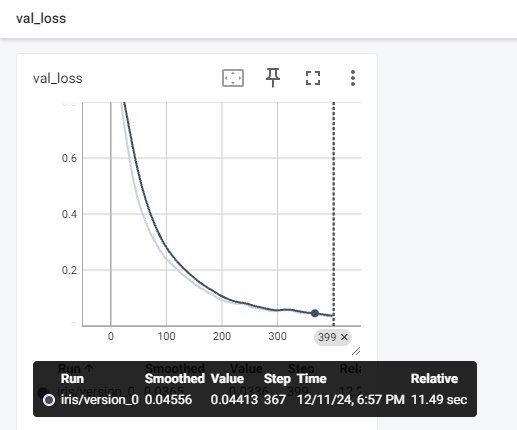












5-2 手寫辨認

左方有分頁

5-1 使用 tf.keras

5-1 使用 pytorch

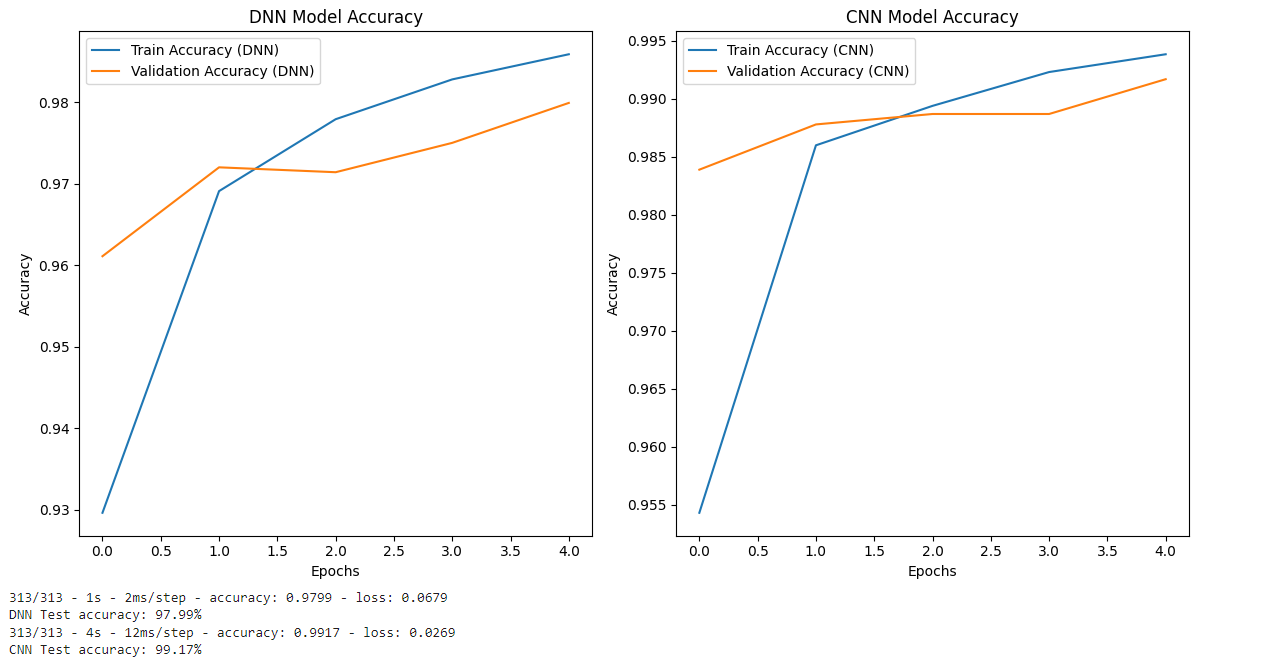
5-1 使用 pytorch\_Lightning

5-2 手寫辨認

5-3 Cifar 圖片分類

手寫辨認使用DNN/CNN 並且比較(Keras)

| # 引入必要的庫  import tensorflow as tf  from tensorflow.keras import layers, models  import matplotlib.pyplot as plt  # 載入 MNIST 數據集  (x\_train, y\_train), (x\_test, y\_test) = tf.keras.datasets.mnist.load\_data()  # 資料預處理：將圖像數據縮放到 [0, 1] 區間  x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  # 重塑資料形狀 (將 28x28 的圖像變為 28x28x1 的形式，這樣 CNN 可以處理)  x\_train = x\_train.reshape(-1, 28, 28, 1)  x\_test = x\_test.reshape(-1, 28, 28, 1)  # 將標籤轉換為 one-hot 編碼  y\_train = tf.keras.utils.to\_categorical(y\_train, 10)  y\_test = tf.keras.utils.to\_categorical(y\_test, 10)  # 顯示一些樣本圖像  plt.figure(figsize=(10, 5))  for i in range(6):  plt.subplot(2, 3, i+1)  plt.imshow(x\_train[i].reshape(28, 28), cmap='gray')  plt.title(f"Label: {y\_train[i].argmax()}")  plt.axis('off')  plt.show()  # --------------------- 選擇模型類型 ----------------------  # 在這裡選擇你要使用的模型，設置為 True 表示使用該模型  use\_dnn = True # 設為 True 使用 DNN，設為 False 使用 CNN  # ------------------------------------------------------  if use\_dnn:  # 建立 DNN 模型  model = models.Sequential([  layers.Flatten(input\_shape=(28, 28)), # 扁平化圖像為 1D 向量  layers.Dense(128, activation='relu'), # 隱藏層  layers.Dense(10, activation='softmax') # 輸出層 (10 顆神經元對應 10 個類別)  ])  else:  # 建立 CNN 模型  model = models.Sequential([  layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),  layers.MaxPooling2D((2, 2)),  layers.Conv2D(64, (3, 3), activation='relu'),  layers.MaxPooling2D((2, 2)),  layers.Conv2D(64, (3, 3), activation='relu'),  layers.Flatten(),  layers.Dense(64, activation='relu'),  layers.Dense(10, activation='softmax')  ])  # 編譯模型  model.compile(optimizer='adam',  loss='categorical\_crossentropy',  metrics=['accuracy'])  # 訓練模型  model.fit(x\_train, y\_train, epochs=5, batch\_size=32, validation\_data=(x\_test, y\_test))  # 評估模型  test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)  # 顯示正確率，以百分比顯示  print(f"Test accuracy: {test\_acc \* 100:.2f}%")  # 預測測試集中的圖像  predictions = model.predict(x\_test)  # 顯示一些測試圖像和預測結果  plt.figure(figsize=(10, 5))  for i in range(6):  plt.subplot(2, 3, i+1)  plt.imshow(x\_test[i].reshape(28, 28), cmap='gray')  plt.title(f"Predicted: {predictions[i].argmax()}, True: {y\_test[i].argmax()}")  plt.axis('off')  plt.show()  # ----------------- 保存模型（使用 .keras 格式） ----------------  # 保存模型  model.save("handwriting\_model.keras")  # 重新加載模型  loaded\_model = tf.keras.models.load\_model("handwriting\_model.keras")  # 重新編譯模型（如果需要）  loaded\_model.compile(optimizer='adam',  loss='categorical\_crossentropy',  metrics=['accuracy'])  # 評估重新加載的模型  loaded\_test\_loss, loaded\_test\_acc = loaded\_model.evaluate(x\_test, y\_test, verbose=2)  # 顯示加載模型的正確率，以百分比顯示  print(f"Re-loaded model Test accuracy: {loaded\_test\_acc \* 100:.2f}%") |
| --- |



313/313 - 1s - 2ms/step - accuracy: 0.9799 - loss: 0.0679

DNN Test accuracy: 97.99%

313/313 - 4s - 12ms/step - accuracy: 0.9917 - loss: 0.0269

CNN Test accuracy: 99.17%

5-3 Cifar 圖片分類

左方有分頁

5-1 使用 tf.keras

5-1 使用 pytorch

5-1 使用 pytorch\_Lightning

5-2 手寫辨認

5-3 Cifar 圖片分類

CIFAR-10 圖片分類 使用VGG19(Keras)

| # 引入必要的庫  import tensorflow as tf  from tensorflow.keras import layers, models  import matplotlib.pyplot as plt  from tensorflow.keras.datasets import cifar10  # 1. 載入 CIFAR-10 數據集  (x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()  # 資料預處理：將圖像數據縮放到 [0, 1] 區間  x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  # 顯示一些樣本圖像  class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']  plt.figure(figsize=(10, 5))  for i in range(6):  plt.subplot(2, 3, i+1)  plt.imshow(x\_train[i])  plt.title(f"Label: {class\_names[y\_train[i][0]]}")  plt.axis('off')  plt.show()  # 2. 載入 VGG19 預訓練模型（不包括頂層分類器）  base\_model = tf.keras.applications.VGG19(include\_top=False, input\_shape=(32, 32, 3))  # 3. 凍結 VGG19 模型的卷積層  base\_model.trainable = False  # 4. 增加頂層分類器  model = models.Sequential([  base\_model, # 預訓練的 VGG19 模型  layers.Flatten(), # 扁平化層  layers.Dense(128, activation='relu'), # 隱藏層  layers.Dense(10, activation='softmax') # 輸出層 (10 顆神經元對應 10 個類別)  ])  # 5. 編譯模型  model.compile(optimizer='adam',  loss='sparse\_categorical\_crossentropy',  metrics=['accuracy'])  # 6. 訓練模型  model.fit(x\_train, y\_train, epochs=10, batch\_size=64, validation\_data=(x\_test, y\_test))  # 7. 評估模型  test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)  # 顯示正確率，以百分比顯示  print(f"Test accuracy: {test\_acc \* 100:.2f}%")  # 8. 預測測試集中的圖像  predictions = model.predict(x\_test)  # 顯示一些測試圖像和預測結果  plt.figure(figsize=(10, 5))  for i in range(6):  plt.subplot(2, 3, i+1)  plt.imshow(x\_test[i])  plt.title(f"Predicted: {class\_names[predictions[i].argmax()]}, True: {class\_names[y\_test[i][0]]}")  plt.axis('off')  plt.show()  # 9. 解凍部分層進行微調  # 解凍 VGG19 模型中的部分層，開始微調  base\_model.trainable = True  fine\_tune\_at = 15 # 設定從第15層開始微調  # 再次編譯模型  for layer in base\_model.layers[:fine\_tune\_at]:  layer.trainable = False  model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=0.0001), # 設置較小的學習率  loss='sparse\_categorical\_crossentropy',  metrics=['accuracy'])  # 10. 微調模型  model.fit(x\_train, y\_train, epochs=5, batch\_size=64, validation\_data=(x\_test, y\_test))  # 11. 評估微調後的模型  test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)  print(f"Test accuracy after fine-tuning: {test\_acc \* 100:.2f}%") |
| --- |



