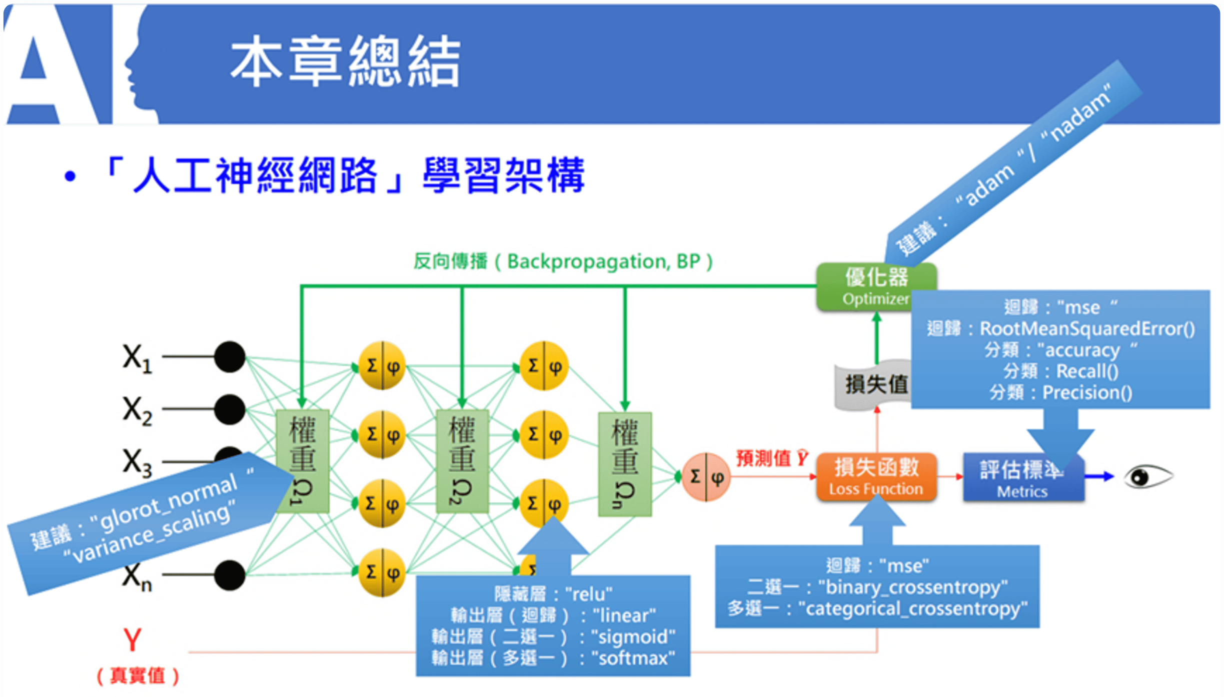
PyTorch in Practice

Tech. dept in NTU AI Club
Stan Wang

Contents

1. Neural Network
2. Pytorch overview
3. Regression
4. Classification
5. Classification competition in Kaggle
6. Regression competition in Kaggle

Neural Network



圖片取自紀俊男(RobertChi)老師

Pytorch overview

What is pytorch?

PyTorch 是由 Facebook AI Research 開發的開源深度學習框架，具有動態計算圖功能，使模型建構、訓練和調試更加靈活和直觀。 PyTorch 採用 Python 開發，在學術界和工業界廣泛應用。

What is Tensor?

- Tensor 是一個類似 NumPy 的 ndarray 的多維資料數組，但它可以在 CPU 或 GPU 上有效運作。
- tensor不僅支援多種數學演算法和矩陣運算，還利用自動引導使得神經網路的反向傳播計算更加簡單且有效率。

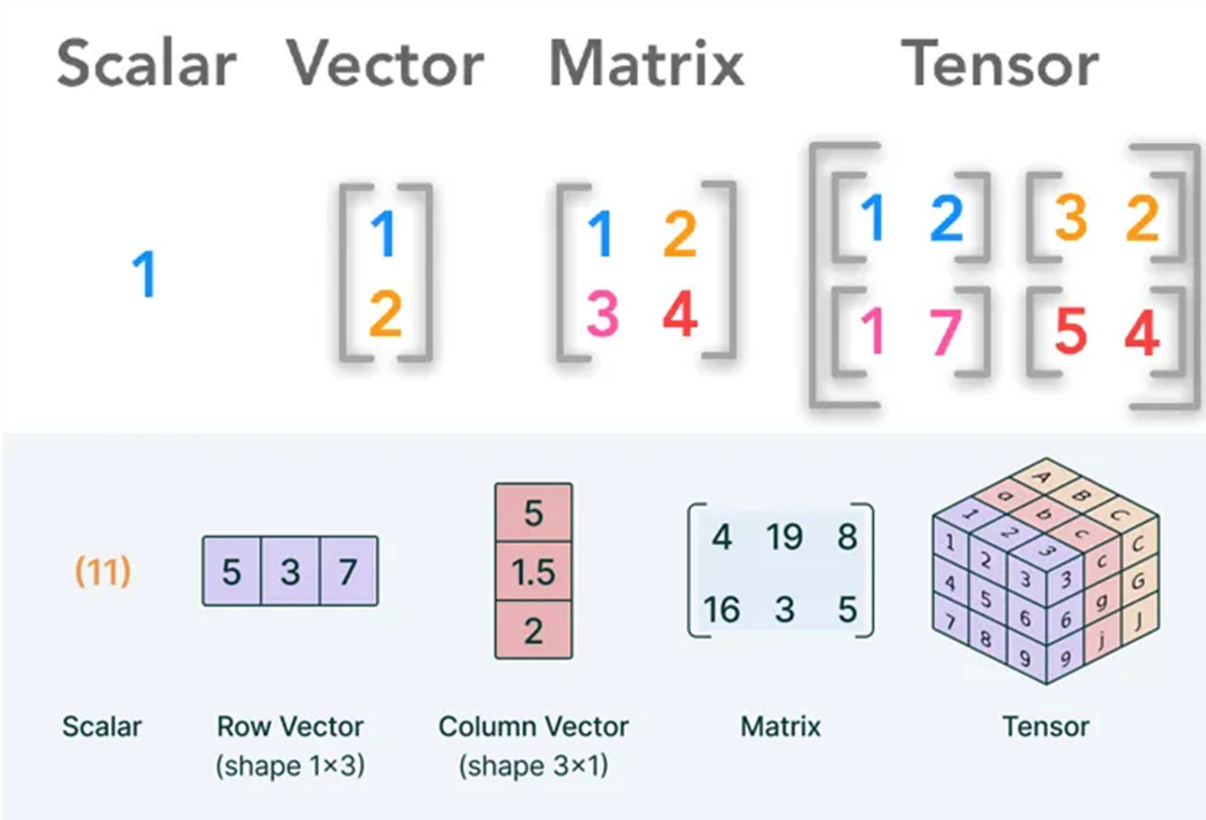


Image Source

建立簡單的張量

input

```
# 1D Floating Point Tensor
tensor_1D = torch.Tensor([1, 2, 3])
print("Tensor 1D:", tensor_1D)
print(tensor_1D.type())

# 2D Floating Point Tensor
tensor_2D = torch.Tensor([[1, 2, 3], [4, 5, 6]])
print("Tensor 2D:", tensor_2D)
print(tensor_2D.type())
```

output

```
Tensor 1D: tensor([1., 2., 3.])
torch.FloatTensor
Tensor 2D: tensor([[1., 2., 3.],
        [4., 5., 6.]])
torch.FloatTensor
```

張量運算

input

```
# Tensor operation
tensor_1 = torch.tensor([[1, 2, 3], [4, 5, 6]])
tensor_2 = torch.tensor([[6, 5, 4], [3, 2, 1]])

print(f"Addition: {tensor_1 + tensor_2}")
print(f"Subtraction: {tensor_1 - tensor_2}")
print(f"Multiplication: {tensor_1 * tensor_2}")
print(f"Division: {tensor_1 / tensor_2}")
```

---

output

```
Addition: tensor([[7, 7, 7],
        [7, 7, 7]])
Subtraction: tensor([[-5, -3, -1],
        [ 1,  3,  5]])
Multiplication: tensor([[ 6, 10, 12],
        [12, 10,  6]])
Division: tensor([[0.1667, 0.4000, 0.7500],
        [1.3333, 2.5000, 6.0000]])
```

與其它資料結構轉換

input

```
# Python List to Tensors
list_1D = [1, 2, 3]
tensor_1D = torch.tensor(list_1D)
print("Tensor 1D:", tensor_1D)

list_1D = tensor_1D.tolist()
print("List 1D:", list_1D)

ndarray_1D = np.array([1, 2, 3])
tensor_1D = torch.from_numpy(ndarray_1D)
print("Tensor 1D:", tensor_1D)

ndarray_1D = tensor_1D.numpy()
print("NDArray 1D:", ndarray_1D)
```

---

output

```
Tensor 1D: tensor([1, 2, 3])
List 1D: [1, 2, 3]
Tensor 1D: tensor([1, 2, 3])
NDArray 1D: [1 2 3]
```

將張量在 CPU 與 GPU 間互轉

input

```
tensor_1D = torch.tensor([1, 2, 3])
print("Tensor 1D:", tensor_1D)
print(tensor_1D.type())

# Transfer tensor from CPU to GPU
tensor_1D_GPU = tensor_1D.to(device)
print("Tensor 1D GPU:", tensor_1D_GPU)
print(tensor_1D_GPU.type())

# Transfer tensor from GPU to CPU
tensor_1D_CPU = tensor_1D_GPU.to(torch.device("cpu"))
print("Tensor 1D CPU:", tensor_1D_CPU)
print(tensor_1D_CPU.type())
```

---

output

```
Tensor 1D: tensor([1, 2, 3])
torch.LongTensor
Tensor 1D GPU: tensor([1, 2, 3], device='cuda:0')
torch.cuda.LongTensor
Tensor 1D CPU: tensor([1, 2, 3])
torch.LongTensor
```

張量資訊

input

```python
# Get the "axes" of a tensor
tensor_2D = torch.tensor([[1, 2, 3], [4, 5, 6]])
print("Tensor 2D:", tensor_2D)
print("Axes:", tensor_2D.dim())

# Get the "dimension/shape" of a tensor
print("Shape:", tensor_2D.shape)
print("Size:", tensor_2D.size())

# Get the "Number of Elements" of a tensor
print("Number of Elements:", tensor_2D.numel())

# Get the "Data Type" of a tensor
print("Data Type:", tensor_2D.dtype)
```

---

output

```
Tensor 2D: tensor([[1, 2, 3],
        [4, 5, 6]])
Axes: 2
Shape: torch.Size([2, 3])
Size: torch.Size([2, 3])
Number of Elements: 6
Data Type: torch.int64
```

張量維度操作

input

```python
tensor_2D = torch.tensor([[1, 2, 3], [4, 5, 6]])
print("Tensor 2D:", tensor_2D)
print("Initial Shape:", tensor_2D.shape)

# Use unsqueeze to insert a new dimension at the specified dimension
tensor_unsqueezed = tensor_2D.unsqueeze(dim=1)
print("Tensor after unsqueeze(dim=1):", tensor_unsqueezed)
print("New Shape after unsqueeze:", tensor_unsqueezed.size())

# Use squeeze to remove dimensions of size 1
tensor_squeezed = tensor_unsqueezed.squeeze(dim=1)
print("Tensor after squeeze(dim=1):", tensor_squeezed)
print("New Shape after squeeze:", tensor_squeezed.size())
```

---

output

```
Tensor 2D: tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
Initial Shape: torch.Size([2, 3])
Tensor after unsqueeze(dim=1): tensor([[[1, 2, 3]],

        [[4, 5, 6]]])
New Shape after unsqueeze: torch.Size([2, 1, 3])
Tensor after squeeze(dim=1): tensor([[1, 2, 3],
        [4, 5, 6]])
New Shape after squeeze: torch.Size([2, 3])
```

Regression

Environment

```
import pandas as pd
import numpy as np
# from tqdm import tqdm
# import matplotlib.pyplot as plt

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

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
```

Dataset

特徵名稱	說明
age	年齡
sex	性別
bmi	身體質量指數
bp	平均血壓
s1~s6	六項血液生化指標

目標：預測資料病患一年後的糖尿病進展指數

步驟

1. 前處理
2. 建立模型
3. 訓練模型
4. 測試模型

前處理

1. 載入資料集
2. 切分自變數、應變數
   ~~3. 處理缺失資料~~
   ~~4. 類別資料數位化~~
3. 切分訓練集、測試集
   ~~6. 特徵縮放(已做過)~~
4. 轉成張量

下載資料

```
# Load dataset
from sklearn.datasets import load_diabetes

dataset = load_diabetes()
```

拆分自變數與應變數

```
# Split independent variable and dependent variable
X = pd.DataFrame(dataset.data, columns=dataset.feature_names)
Y = pd.DataFrame(dataset.target, columns=['Diabete_Value'])
```

拆分資料

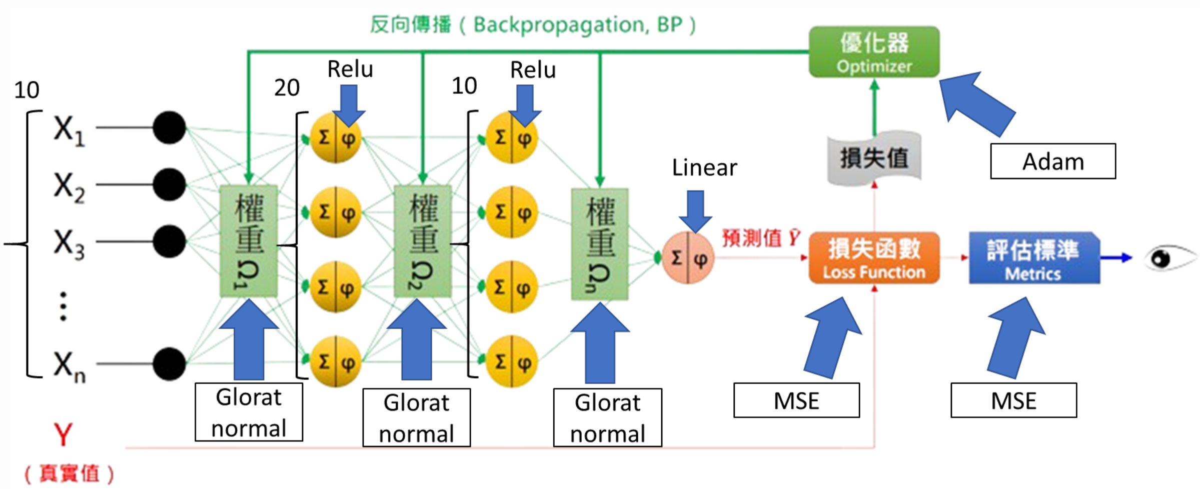
```
# Split train dataset and test dataset variable
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, ra
```

轉成向量

```
# Transform into Tensor
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float)
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float)
Y_train_tensor = torch.tensor(Y_train.values, dtype=torch.float)
Y_test_tensor = torch.tensor(Y_test.values, dtype=torch.float)
```

~~1. 前處理~~
2. 建立模型
3. 訓練模型
4. 測試模型

建立模型(示意圖)



圖片取自紀俊男(RobertChi)老師

建立模型(程式碼)

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

  # Define the architecture of each layer of the neural network
  def __init__(self):
    super(DiabeteModel, self).__init__()

    # Define each neural layer
    self.fc1 = nn.Linear(10, 20)
    self.fc2 = nn.Linear(20, 10)
    self.fc3 = nn.Linear(10, 1)

    # Initialize the weights of each neural layer
    init.xavier_normal_(self.fc1.weight)
    init.xavier_normal_(self.fc2.weight)
    init.xavier_normal_(self.fc3.weight)

  # Define forward propagation
  def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)

    return x

# Construct the model
model = DiabeteModel().to(device)

# Define the loss function
criterion = nn.MSELoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

訓練模型

1. 建立 Dataset 與 DataLoader
2. 將張量放到GPU
3. 清除舊的梯度
4. 計算損失
5. 反向傳播
6. 更新優化器權重

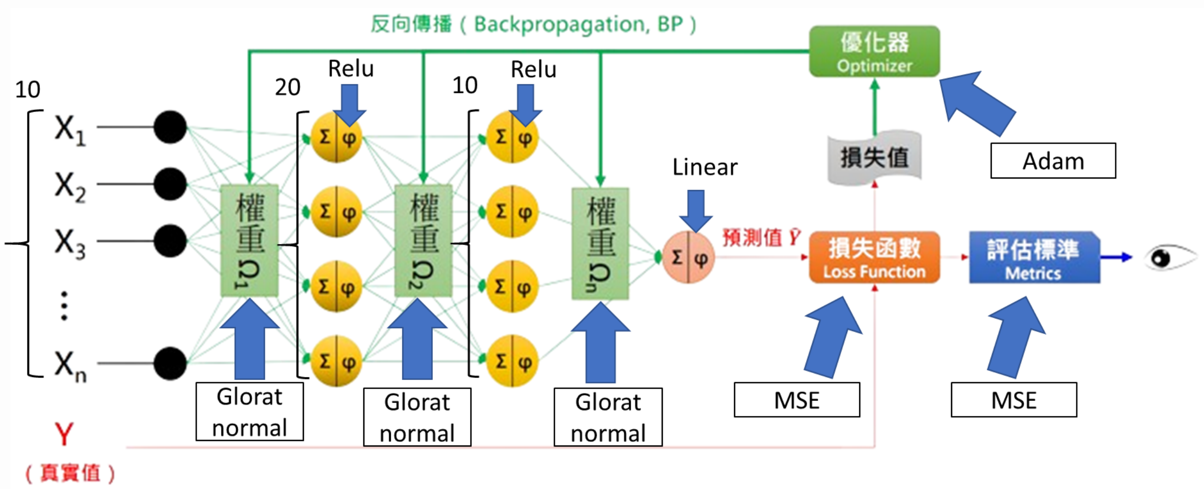
建立 Dataset 與 DataLoader

```
batch_size = 15

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=Tru
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False
```

訓練迴圈(示意圖)



圖片取自紀俊男(RobertChi)老師

訓練迴圈(程式碼)

```python
epochs = 150

for epoch in range(epochs):
  # Set the model to training mode
  model.train()

  running_loss = 0.0
  total_samples = 0

  # Take a Batch and start training
  for X_batch, Y_batch in train_loader:
    # Translate the data from the Batch to the GPU
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Zero out the gradient of the previous Batch
    optimizer.zero_grad()

    # Calculate the output
    Y_pred = model(X_batch)
    # Calculate the loss
    loss = criterion(Y_pred, Y_batch)
    # Backward propagation
    loss.backward()
    # Update the weight
    optimizer.step()

    running_loss += loss.item() * X_batch.size(0)
    total_samples += X_batch.size(0)

  # Calculate the accuracy for each epoch
  epoch_loss = running_loss / total_samples
  print(f'Epoch {epoch+1}/{epochs},MSE Loss: {epoch_loss:.4f}')
```

訓練結果(前面十行與最後十行)

```
Epoch 1/150,MSE Loss: 29051.5199
Epoch 2/150,MSE Loss: 29011.7012
Epoch 3/150,MSE Loss: 28965.5169
Epoch 4/150,MSE Loss: 28904.0876
Epoch 5/150,MSE Loss: 28822.1680
Epoch 6/150,MSE Loss: 28713.4460
Epoch 7/150,MSE Loss: 28572.2816
Epoch 8/150,MSE Loss: 28381.4172
Epoch 9/150,MSE Loss: 28141.3310
Epoch 10/150,MSE Loss: 27830.3267
```

```
Epoch 141/150,MSE Loss: 3043.6555
Epoch 142/150,MSE Loss: 3033.1254
Epoch 143/150,MSE Loss: 3027.5503
Epoch 144/150,MSE Loss: 3023.3534
Epoch 145/150,MSE Loss: 3017.9371
Epoch 146/150,MSE Loss: 3013.7183
```

```
Epoch 147/150,MSE Loss: 3008.9037
Epoch 148/150,MSE Loss: 3004.4907
Epoch 149/150,MSE Loss: 2998.2981
Epoch 150/150,MSE Loss: 2993.6150
```

測試模型

```python
# Switch to evaluation mode
model.eval()

test_loss = 0.0
total_samples = 0

# Turn off PyTorch's gradient calculation
with torch.no_grad():
  # Take a Batch and start testing
  for X_batch, Y_batch in test_loader:
    # Transform the data from CPU into GPU.
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Calculate output values
    Y_pred = model(X_batch)

    # Calculate loss for the batch
    loss = criterion(Y_pred, Y_batch)

    # Accumulate total loss and sample count
    test_loss += loss.item() * X_batch.size(0)
    total_samples += X_batch.size(0)

# Compute the average loss for the test dataset
avg_test_loss = test_loss / total_samples
print(f'Test MSE Loss: {avg_test_loss:.4f}')
```

結果

```
Test MSE Loss: 3257.9897
```

Classification

Dataset

特徵名稱	說明
sepal length (cm)	花萼長度（公分）

sepal width (cm)	花萼寬度（公分）
petal length (cm)	花瓣長度（公分）
petal width (cm)	花瓣寬度（公分）

目標：預測鳶尾花的品種

- 0:Setosa
- 1:Versicolor
- 2:Virginica

前處理

1. 載入資料集
2. 切分自變數、應變數
   ~~3. 處理缺失資料~~
   ~~4. 類別資料數位化~~
3. 切分訓練集、測試集
4. 特徵縮放
5. 轉成張量

載入資料集

```
# Load dataset
from sklearn.datasets import load_iris

dataset = load_iris()
```

拆分自變數與應變數

```
# Split independent variable and dependent variable
X = pd.DataFrame(dataset.data, columns=dataset.feature_names)
Y = pd.DataFrame(dataset.target, columns=['Iris_Type'])
Y_name = dataset.target_names.tolist()
```

拆分資料

```
# Split train dataset and test dataset variable
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, ra
```

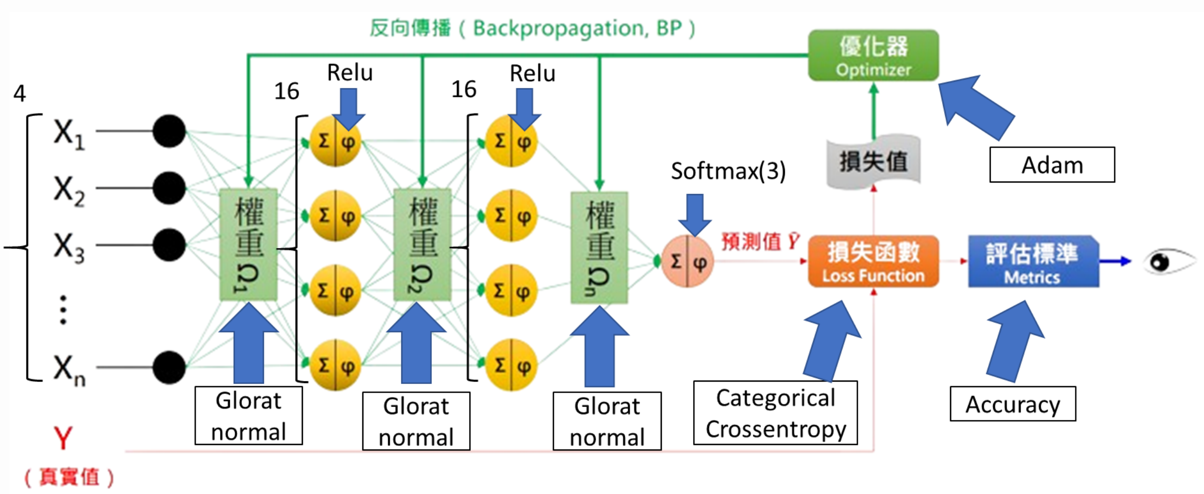
特徵縮放

```
# Feature Scaling
sc_X = StandardScaler().fit(X_train)
X_train = sc_X.transform(X_train)
X_test = sc_X.transform(X_test)
```

轉成向量

```
# Transform into Tensor
X_train_tensor = torch.from_numpy(X_train).float()
X_test_tensor = torch.from_numpy(X_test).float()
Y_train_tensor = torch.tensor(Y_train.values, dtype=torch.long)
Y_test_tensor = torch.tensor(Y_test.values, dtype=torch.long)
```

建立模型(示意圖)



圖片取自紀俊男(RobertChi)老師

建立模型(程式碼)

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

  # Define the architecture of each layer of the neural network
  def __init__(self):
    super(IrisModel, self).__init__()

    # Define each neural layer
    self.fc1 = nn.Linear(4, 16)
    self.fc2 = nn.Linear(16, 16)
    self.fc3 = nn.Linear(16, 3)

    # Initialize the weights of each neural layer
    init.xavier_normal_(self.fc1.weight)
    init.xavier_normal_(self.fc2.weight)
    init.xavier_normal_(self.fc3.weight)

  # Define forward propagation
  def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)

    return x

# Construct the model
model = IrisModel().to(device)

# Define the loss function
criterion = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

訓練模型

1. 建立 Dataset 與 DataLoader
2. 將張量放到GPU
3. 清除舊的梯度
4. 計算損失
5. 反向傳播
6. 更新優化器權重

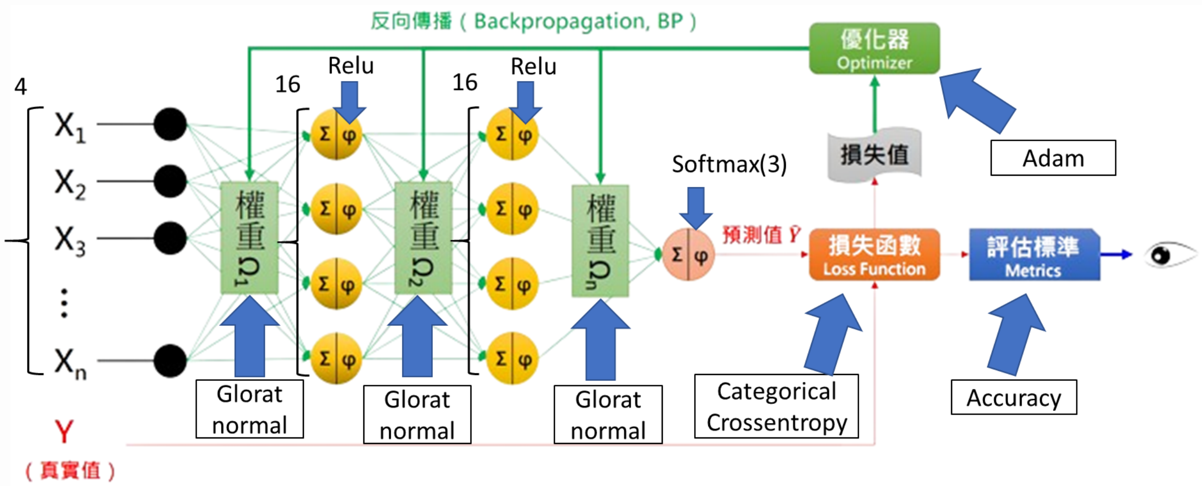
建立 Dataset 與 DataLoader

```
batch_size = 15

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=Tru
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False
```

訓練迴圈(示意圖)



圖片取自紀俊男(RobertChi)老師

訓練迴圈(程式碼)

```python
epochs = 50

for epoch in range(epochs):
  # Set the model to training mode
  model.train()

  # Store the number of correct guesses & the full number of guesses
  correct = 0
  total = 0

  # Take a Batch and start training
  for X_batch, Y_batch in train_loader:
    # Translate the data from the Batch to the GPU
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Zero out the gradient of the previous Batch
    optimizer.zero_grad()

    # Calculate the output
    Y_pred = model(X_batch)
    # Calculate the loss
    loss = criterion(Y_pred, Y_batch.squeeze())
    # Backward propagation
    loss.backward()
    # Update the weight
    optimizer.step()

    # Find the biggest index
    _, predicted = torch.max(Y_pred.data, 1)
    # Add the data of this batch to total
    total += Y_batch.size(0)
    # Calculate the number of correct guesses (.item() will help get the pu
    correct += (predicted == Y_batch.squeeze()).sum().item()

    # Calculate the accuracy for each epoch
    accuracy = correct / total
  print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}, Acc: {accuracy
```

訓練結果(前面十行與最後十行)

```
Epoch 1/50, Loss: 1.0151, Acc: 0.4917
Epoch 2/50, Loss: 0.9234, Acc: 0.6417
Epoch 3/50, Loss: 0.8413, Acc: 0.6833
Epoch 4/50, Loss: 0.6903, Acc: 0.6917
Epoch 5/50, Loss: 0.9573, Acc: 0.6917
Epoch 6/50, Loss: 0.6670, Acc: 0.7000
Epoch 7/50, Loss: 0.6463, Acc: 0.7000
Epoch 8/50, Loss: 0.6023, Acc: 0.7167
Epoch 9/50, Loss: 0.7148, Acc: 0.7167
Epoch 10/50, Loss: 0.5522, Acc: 0.7167
```

```
Epoch 41/50, Loss: 0.1101, Acc: 0.9583
Epoch 42/50, Loss: 0.0760, Acc: 0.9583
Epoch 43/50, Loss: 0.1394, Acc: 0.9583
Epoch 44/50, Loss: 0.2037, Acc: 0.9583
Epoch 45/50, Loss: 0.1376, Acc: 0.9583
Epoch 46/50, Loss: 0.1379, Acc: 0.9583
Epoch 47/50, Loss: 0.1396, Acc: 0.9667
Epoch 48/50, Loss: 0.0747, Acc: 0.9667
Epoch 49/50, Loss: 0.0288, Acc: 0.9667
Epoch 50/50, Loss: 0.1737, Acc: 0.9667
```

測試模型

```
# Switch to evaluation mode
model.eval()

# Turn off PyTorch's gradient calculation
with torch.no_grad():

  correct = 0
  total = 0

  # Take a Batch and start testing
  for X_batch, Y_batch in test_loader:
    # Transform the data from CPU into GPU.
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Calculate output values
    Y_pred = model(X_batch)

    # Find the biggest perbentage of index
    _, predicted = torch.max(Y_pred.data, 1)
    # Add the data of this batch to total
    total += Y_batch.size(0)
    # Calculate the number of correct guesses
    correct += (predicted == Y_batch.squeeze()).sum().item()

  print(f'Test Accuracy: {correct / total:.2%}')
```

結果

```
Test Accuracy: 100.00%
```

Classification competition in Kaggle

資料集簡介

連結

- 我們有出生年份、性別等資訊，希望能夠預測最終是否存活

載入資料集

```
# Load dataset

dataset = pd.read_csv('/content/1132_NTUAI_DL_Resource/titanic/train.csv')
dataset_test = pd.read_csv('/content/1132_NTUAI_DL_Resource/titanic/test.cs
```

刪除部分欄位為空的資料

```
dataset = dataset.dropna(subset=['Embarked'])
```

拆分自變數與應變數

```
# Split independent variable and dependent variable
X = dataset.drop(columns=['PassengerId', 'Name', 'Ticket', 'Cabin', 'Surviv
Y = pd.DataFrame(dataset, columns=['Survived'])
X_pred = dataset_test.drop(columns=['PassengerId', 'Name', 'Ticket', 'Cabin
```

類別資料數位化

```
# One-Hot encoder
X_mod = pd.get_dummies(X, columns=['Sex', 'Embarked'], drop_first=True).ast
X_pred_mod = pd.get_dummies(X_pred, columns=['Sex', 'Embarked'], drop_first
```

拆分資料

```
# Split train dataset and test dataset variable
X_train, X_test, Y_train, Y_test = train_test_split(X_mod, Y, test_size=0.2
```

特徵縮放

```python
# Feature Scaling
sc_X = StandardScaler().fit(X_train)
X_train = sc_X.transform(X_train)
X_test = sc_X.transform(X_test)
X_pred_scale = sc_X.transform(X_pred_mod)
```

轉成向量

```python
# Transform into Tensor
X_train_tensor = torch.from_numpy(X_train).float()
X_test_tensor = torch.from_numpy(X_test).float()
X_pred_tensor = torch.from_numpy(X_pred_scale).float()
Y_train_tensor = torch.tensor(Y_train.values, dtype=torch.float)
Y_test_tensor = torch.tensor(Y_test.values, dtype=torch.float)
```

建立模型

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

  # Define the architecture of each layer of the neural network
  def __init__(self, input_len):
    super(TitanicModel, self).__init__()

    # Define each neural layer
    self.fc1 = nn.Linear(input_len, 13)
    self.fc2 = nn.Linear(13, 13)
    self.fc3 = nn.Linear(13, 1)

    # Initialize the weights of each neural layer
    init.xavier_normal_(self.fc1.weight)
    init.xavier_normal_(self.fc2.weight)
    init.xavier_normal_(self.fc3.weight)

  # Define forward propagation
  def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = torch.sigmoid(self.fc3(x))

    return x

# Construct the model
model = TitanicModel(X_train_tensor.shape[1]).to(device)

# Define the loss function
criterion = nn.BCELoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

訓練模型

1. 建立 Dataset 與 DataLoader
2. 將張量放到GPU
3. 清除舊的梯度
4. 計算損失
5. 反向傳播
6. 更新優化器權重

建立 Dataset 與 DataLoader

```
batch_size = 15

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=Tru
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False
```

訓練迴圈(程式碼)

```python
epochs = 50

for epoch in range(epochs):
  # Set the model to training mode
  model.train()

  # Store the number of correct guesses & the full number of guesses
  correct = 0
  total = 0

  # Take a Batch and start training
  for X_batch, Y_batch in train_loader:
    # Translate the data from the Batch to the GPU
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Zero out the gradient of the previous Batch
    optimizer.zero_grad()

    # Calculate the output
    Y_pred = model(X_batch)
    # Calculate the loss
    loss = criterion(Y_pred.squeeze(), Y_batch.squeeze())
    # Backward propagation
    loss.backward()
    # Update the weight
    optimizer.step()

    # Add the data of this batch to total
    total += Y_batch.size(0)

    # Calculate the number of correct guesses (.item() will help get the pu
    predicted = (Y_pred.squeeze() >= 0.5).float()
    correct += (predicted == Y_batch.squeeze()).sum().item()

  # Calculate the accuracy for each epoch
  accuracy = correct / total
  print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}, Acc: {accuracy
```

訓練結果(前面十行與最後十行)

```
Epoch 1/50, Loss: 0.6352, Acc: 0.5556
Epoch 2/50, Loss: 0.4765, Acc: 0.7103
Epoch 3/50, Loss: 0.5724, Acc: 0.6962
Epoch 4/50, Loss: 0.7052, Acc: 0.7482
Epoch 5/50, Loss: 0.3761, Acc: 0.7904
Epoch 6/50, Loss: 0.5385, Acc: 0.8200
Epoch 7/50, Loss: 0.4009, Acc: 0.8186
Epoch 8/50, Loss: 0.4260, Acc: 0.8143
Epoch 9/50, Loss: 0.5415, Acc: 0.8087
Epoch 10/50, Loss: 0.7234, Acc: 0.8073
```

```
Epoch 41/50, Loss: 0.1802, Acc: 0.8411
Epoch 42/50, Loss: 0.2659, Acc: 0.8439
Epoch 43/50, Loss: 0.5901, Acc: 0.8467
Epoch 44/50, Loss: 0.2018, Acc: 0.8439
Epoch 45/50, Loss: 0.6142, Acc: 0.8453
Epoch 46/50, Loss: 0.2658, Acc: 0.8453
Epoch 47/50, Loss: 0.3154, Acc: 0.8467
Epoch 48/50, Loss: 0.3701, Acc: 0.8453
Epoch 49/50, Loss: 0.4778, Acc: 0.8467
Epoch 50/50, Loss: 0.4226, Acc: 0.8481
```

測試模型

```
# Switch to evaluation mode
model.eval()

# Turn off PyTorch's gradient calculation
with torch.no_grad():

  correct = 0
  total = 0

  # Take a Batch and start testing
  for X_batch, Y_batch in test_loader:
    # Transform the data from CPU into GPU.
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Calculate output values
    Y_pred = model(X_batch)

    # Find the biggest perbentage of index
    predicted = (Y_pred.squeeze() >= 0.5).float()

    # Add the data of this batch to total
    total += Y_batch.size(0)
    # Calculate the number of correct guesses
    correct += (predicted == Y_batch.squeeze()).sum().item()

  print(f'Test Accuracy: {correct / total:.2%}')
```

結果

```
Test Accuracy: 73.03%
```

預測模型

```
model.eval()
with torch.no_grad():
    Y_pred = model(X_pred_tensor.to(device))

    Y_pred_label = (Y_pred >= 0.5).long()

Y_pred_numpy = Y_pred_label.squeeze().cpu().tolist()

submission = pd.DataFrame({'PassengerId': dataset_test['PassengerId'], 'Sur
submission.to_csv('submission_titanic.csv', index=False)
```

Regression competition in Kaggle

資料集簡介

連結

- 我們有房子的銷售年、月份等資訊，希望能夠預測房價

載入資料集

```
# Load dataset

dataset = pd.read_csv('/content/1132_NTUAI_DL_Resource/house-prices-advance
dataset_test = pd.read_csv('/content/1132_NTUAI_DL_Resource//house-prices-a
```

拆分自變數與應變數

```
# Split independent variable and dependent variable
X = pd.DataFrame(dataset, columns=['MSSubClass', 'MSZoning', 'LotArea', 'Ov
Y = pd.DataFrame(dataset, columns=['SalePrice'])
X_pred = pd.DataFrame(dataset_test, columns=['MSSubClass', 'MSZoning', 'Lot
```

類別資料數位化

```
# One-Hot encoder
X_mod = pd.get_dummies(X, columns=['MSZoning', 'CentralAir', 'SaleType', 'S
X_pred_mod = pd.get_dummies(X_pred, columns=['MSZoning', 'CentralAir', 'Sal
```

拆分資料

```
# Split train dataset and test dataset variable
X_train, X_test, Y_train, Y_test = train_test_split(X_mod, Y, test_size=0.2
```

轉成向量

```
# Transform into Tensor
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float)
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float)
X_pred_tensor = torch.tensor(X_pred_mod.values, dtype=torch.float)
Y_train_tensor = torch.tensor(Y_train.values, dtype=torch.float)
Y_test_tensor = torch.tensor(Y_test.values, dtype=torch.float)
```

建立模型

```
class PriceModel(nn.Module):

  # Define the architecture of each layer of the neural network
  def __init__(self, input_len):
    super(PriceModel, self).__init__()

    # Define each neural layer
    self.fc1 = nn.Linear(input_len, 61)
    self.fc2 = nn.Linear(61, 30)
    self.fc3 = nn.Linear(30, 1)

    # Initialize the weights of each neural layer
    init.xavier_normal_(self.fc1.weight)
    init.xavier_normal_(self.fc2.weight)
    init.xavier_normal_(self.fc3.weight)

  # Define forward propagation
  def forward(self, x):
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)

    return x

# Construct the model
model = PriceModel(X_train_tensor.shape[1]).to(device)

# Define the loss function
criterion = nn.MSELoss()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

訓練模型

1. 建立 Dataset 與 DataLoader
2. 將張量放到GPU
3. 清除舊的梯度
4. 計算損失
5. 反向傳播
6. 更新優化器權重

建立 Dataset 與 DataLoader

```
batch_size = 15

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=Tru
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False
```

訓練迴圈(程式碼)

```python
epochs = 70

for epoch in range(epochs):
  # Set the model to training mode
  model.train()

  running_loss = 0.0
  total_samples = 0

  # Take a Batch and start training
  for X_batch, Y_batch in train_loader:
    # Translate the data from the Batch to the GPU
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Zero out the gradient of the previous Batch
    optimizer.zero_grad()

    # Calculate the output
    Y_pred = model(X_batch)
    # Calculate the loss
    loss = torch.sqrt(criterion(Y_pred, Y_batch))
    # Backward propagation
    loss.backward()
    # Update the weight
    optimizer.step()

    running_loss += loss.item() * X_batch.size(0)
    total_samples += X_batch.size(0)

  # Calculate the accuracy for each epoch
  epoch_loss = running_loss / total_samples
  print(f'Epoch {epoch+1}/{epochs},RMSE Loss: {epoch_loss:.4f}')
```

訓練結果(前面十行與最後十行)

```
Epoch 1/70,RMSE Loss: 190295.3514
Epoch 2/70,RMSE Loss: 161153.2240
Epoch 3/70,RMSE Loss: 110397.1278
Epoch 4/70,RMSE Loss: 97744.3532
Epoch 5/70,RMSE Loss: 94279.7636
Epoch 6/70,RMSE Loss: 92779.6218
Epoch 7/70,RMSE Loss: 89694.3305
Epoch 8/70,RMSE Loss: 87927.7034
Epoch 9/70,RMSE Loss: 85346.8407
Epoch 10/70,RMSE Loss: 83919.2306
```

```
Epoch 61/70,RMSE Loss: 68016.3338
Epoch 62/70,RMSE Loss: 67869.1597
Epoch 63/70,RMSE Loss: 67421.2056
Epoch 64/70,RMSE Loss: 67955.5566
Epoch 65/70,RMSE Loss: 68029.8467
Epoch 66/70,RMSE Loss: 67129.8483
```

```
Epoch 67/70,RMSE Loss: 67244.2728
Epoch 68/70,RMSE Loss: 67066.5131
Epoch 69/70,RMSE Loss: 67346.6508
Epoch 70/70,RMSE Loss: 67331.6777
```

測試模型

```python
# Switch to evaluation mode
model.eval()

test_loss = 0.0
total_samples = 0

# Turn off PyTorch's gradient calculation
with torch.no_grad():
  # Take a Batch and start testing
  for X_batch, Y_batch in test_loader:
    # Transform the data from CPU into GPU.
    X_batch, Y_batch = X_batch.to(device), Y_batch.to(device)

    # Calculate output values
    Y_pred = model(X_batch)

    # Calculate model predictions
    Y_pred = model(X_batch)
    # Calculate loss for the batch
    loss = torch.sqrt(criterion(Y_pred, Y_batch))

    # Accumulate total loss and sample count
    test_loss += loss.item() * X_batch.size(0)
    total_samples += X_batch.size(0)

# Compute the average loss for the test dataset
avg_test_loss = test_loss / total_samples
print(f'Test RMSE Loss: {avg_test_loss:.4f}')
```

結果

```
Test RMSE Loss: 69788.8924
```

預測模型

```
model.eval()
with torch.no_grad():
    Y_pred = model(X_pred_tensor.to(device))

Y_pred_numpy = Y_pred.squeeze().cpu().tolist()

submission = pd.DataFrame({'Id': dataset_test['Id'], 'SalePrice': Y_pred_nu
submission.to_csv('submission_price.csv', index=False)
```

Thank you for your listening