

國立成功大學  
自動駕駛實務 Project 2 報告  
German Traffic Signs Classifier

學生：陳育政

學號：E24094198

指導教授：傅志雄、莊智清

# 大綱

## 一、前言

## 二、實作方法

1. Load the Data
2. Dataset Summary & Exploration
3. Pre-process Dataset
  - Convert to Grayscale
  - Data Augmentation
  - Normalize Dataset
4. Train the LeNet5 Model
5. Accuracy Curves, Confusion Matrix and Classification Report
6. Test LeNet5 model on New Images

## 三、問題討論

1. The Necessity of Data-Preprocessing

## 四、結論

## 五、參考資料

## 一、前言

Project 2 的目的是試圖利用由 Yann LeCun 於 1995 年所提出的 LeNet5 模型架構去解決 German Traffic Sign Recognition Benchmark(GTSRB)的問題，而 GTSRB 資料集共有 43 種不同情況的德國交通號誌 RGB 相片，例如:限速、右轉、禁止通行...等圖示。整體處理資料的 pipeline 包括資料前處理、建構訓練 LeNet5 模型以及測試模型的準確率，最後的目標就是訓練好的 LeNet5 模型可以針對不曾餵入訓練的德國交通號誌照片成功辨認種類，達成自動駕駛中不可或缺的辨識路標的功能。

## 二、實作方法

### 1. Load the Data

根據 Fig. 1.的程式碼讀取 p-code file，包括 training、validation 及 test 的 dataset。

```
# load data using pickle
import pickle

train_file='./traffic-signs-data/train.p'
valid_file='./traffic-signs-data/valid.p'
test_file='./traffic-signs-data/test.p'

with open(train_file, mode='rb') as f:
    train=pickle.load(f)
with open(valid_file, mode='rb') as f:
    valid=pickle.load(f)
with open(test_file, mode='rb') as f:
    test=pickle.load(f)

x_train, y_train=train['features'], train['labels']
x_valid, y_valid=valid['features'], valid['labels']
x_test, y_test=test['features'], test['labels']
```

Fig. 1. 讀取資料集

### 2. Dataset Summary & Exploration

印出有關 dataset 的基本資料，包括各資料集比數、圖片大小以及交通號誌種類數等數據，結果如 Fig. 2.所示。

```
Count of training data: 34799
Count of validation data: 4410
Count of testing data: 12630
Image shape: (32, 32, 3)
Count of classes: 43
```

Fig. 2. 資料集的基本資料

除了印出 dataset 的基本資料，我還嘗試隨機挑選幾張資料集內的圖片顯示，讓我對於即將處理的資料有進一步的認知，結果如 Fig. 3.

所示，可以發現影像都是 RGB，並且大小一致。

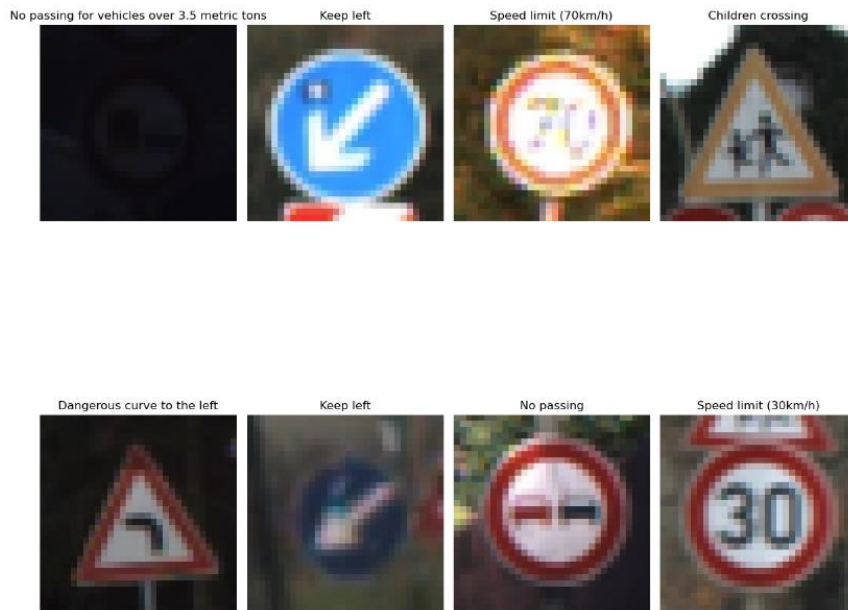


Fig. 3. 隨機挑選 8 張 traffic sign images 觀察

接著，為了更好的了解資料集中的交通號誌種類分布，我將每個資料集的各個 class 的總數以直線圖呈現，training、validation、test dataset 的直線圖依序由 Fig. 4. (a)、(b)、(c) 呈現。

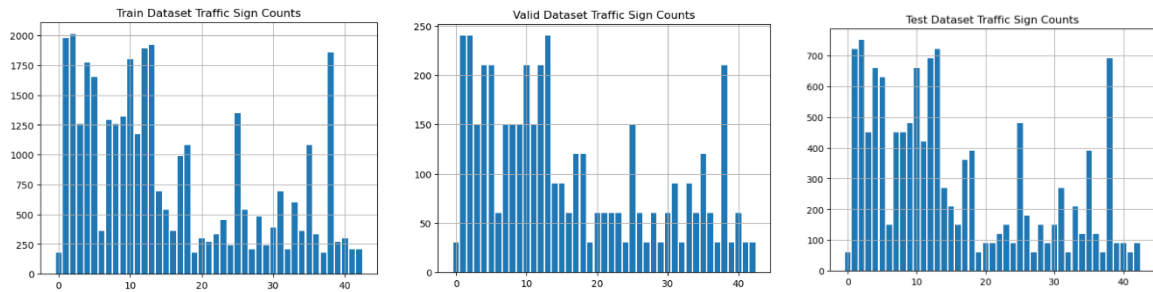


Fig. 4. (a)training dataset 各 class 資料筆數 (b)validation dataset 各 class 資料筆數  
(c)test dataset 各 class 資料筆數

### 3. Pre-process Dataset

#### ➤ Convert to Grayscale

根據 Fig. 5. 的程式碼，我將原先資料集內的 RGB 影像皆轉成灰階圖，grayscale 的效果如 Fig. 6. 呈現，而轉成灰階的原因不僅僅可以減少模型參數，讓訓練更快速，還可以讓模型專注在交通號誌圖示的輪廓差異，而非注意可能隨著光線而有顯著差異的 RGB 色階。

```

# preprocess dataset (including normalization, converting to grayscale...)
x_train_RGB=x_train
x_train_gray=np.sum(x_train/3, axis=3, keepdims=True)

x_valid_RGB=x_valid
x_valid_gray=np.sum(x_valid/3, axis=3, keepdims=True)

x_test_RGB=x_test
x_test_gray=np.sum(x_test/3, axis=3, keepdims=True)

print(f'x_train_RGB shape: {x_train_RGB.shape}')
print(f'x_train_gray shape: {x_train_gray.shape}')

x_train_RGB shape: (34799, 32, 32, 3)
x_train_gray shape: (34799, 32, 32, 1)

```

Fig. 5. 將 RGB images 轉成 grayscale images 的程式碼

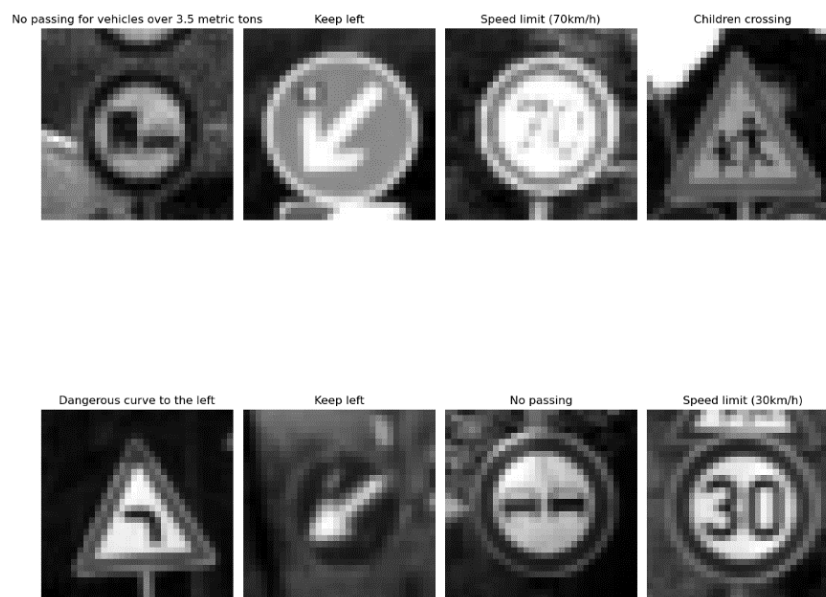


Fig. 6. Traffic sign images 轉成灰階的效果

### ➤ Data Augmentation

藉由觀察 Fig. 4. 的資料集各 class 總數可以發現不同 class 之間的數量差異相當顯著，而這將會使訓練的效果因為資料的不平均性而不如預期，為了使資料筆數較少的 class 增加一些數據，我採用了常見的 data augmentation，藉由平移、旋轉及縮放...等技術，在基於原先的資料下，產生更多筆訓練資料，達到各 class 的資料數平均的目標。

其中，我使用 Affine Transformation、Perspective Transformation 及 Rotation 共 3 種 data augmentation 的技術增加 train dataset 資料數，確保每個 class 都至少有 3000 筆資料。而 3 種 data augmentation 技術的 code implementation 可依序參考 Fig. 7, Fig. 8 及 Fig. 9。另外我也將 data augmentation 產生的圖片和原先的圖片顯示出來做比較，可參考 Fig. 10. 發現影像有輕微更動，但卻不失去本身原有的特徵。

```
# Affine Transformation(random translation)
dx, dy=np.random.randint(-1.7, 1.8, 2)
M=np.float32([[1, 0, dx],[0, 1, dy]])
dst=cv2.warpAffine(x_train[i], M, (x_train[i].shape[0], x_train[i].shape[1]))
dst=dst[:, :, None]
affine_x.append(dst)
affine_y.append(y_train[i])
```

Fig. 7. Affine Transformation 實作程式碼

```
# Perspective Transformation
high_bound=random.randint(22, 32)
low_bound=random.randint(0, 12)
points_one=np.float32([[0,0], [32,0], [0,32], [32,32]])
points_two=np.float32([[0,0], [high_bound, low_bound], [low_bound, 32], [32, high_bound]])
M=cv2.getPerspectiveTransform(points_one, points_two)
dst=cv2.warpPerspective(x_train[i], M, (32, 32))
perspective_x.append(dst)
perspective_y.append(y_train[i])
```

Fig. 8. Perspective Transformation 實作程式碼

```
# Rotation
tilt=random.randint(-15, 15)
M=cv2.getRotationMatrix2D((x_train[i].shape[0]/2, x_train[i].shape[1]/2), tilt, 1)
dst=cv2.warpAffine(x_train[i], M, (x_train[i].shape[0], x_train[i].shape[1]))
rotate_x.append(dst)
rotate_y.append(y_train[i])
```

Fig. 9. Rotation 實作程式碼

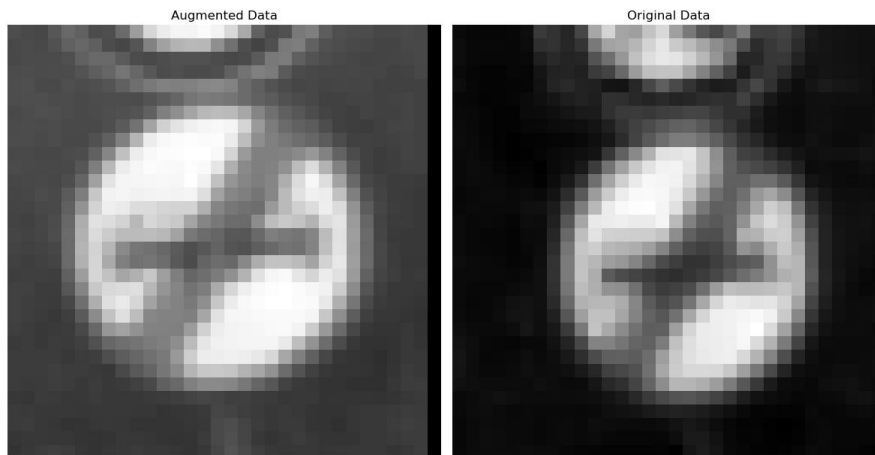


Fig. 10. 右圖為原本資料集的圖片，而左側則為 data augmentation 產出的圖片

針對原有的資料做完 data augmentation 之後，我再次把 dataset 的各 class 資料數以直線圖呈現，更新過的 training、validation dataset 的直線圖依序為 Fig. 11. (a)、(b)。我們可以發現 training 及 validation 的資料數相較於 Fig. 4. 因為使用 data augmentation 技術，各 class 的資料筆數都顯著提升，且 class 之間的數量平均。

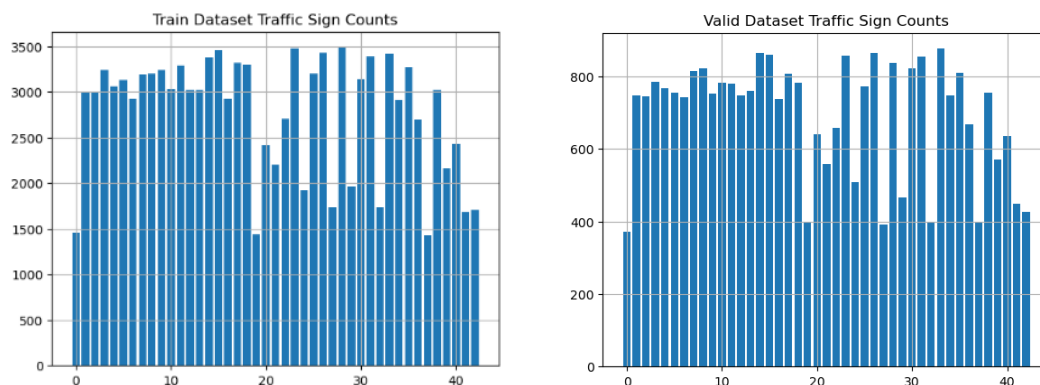


Fig. 11. (a) data augmentation 後的 training dataset 各 class 資料筆數  
(b) data augmentation 後的 validation dataset 各 class 資料筆數

### ➤ Normalize Dataset

對原先的資料集 images 做完 grayscale 轉換和 data augmentation 之後，我又把資料做 Normalization 正規化的處理，將所有 pixel value 按比例轉換至  $[-1, 1]$  區間，詳細程式碼如 Fig. 12. 所示。

```
# divided by 127.5 -> let the pixel value between 0~2, since the original pixel value is 255
# minus by 1 -> let the pixel value between -1~1
x_train_normalized=x_train/127.5-1
x_valid_normalized=x_valid/127.5-1
x_test_normalized=x_test/127.5-1

count_to_show=8
figures={}
count=0
for i in random_signs:
    labels[count]=name_values[y_train[i]][1].decode('ascii')
    figures[count]=x_train_normalized[i].squeeze()
    count+=1
```

Fig. 12. 對資料集圖片做 Normalization 正規化

## 4. Train the LeNet5 Model

我先依據 Yann LeCun 所提出的 LeNet5 架構建構出一個模型，詳細的 LeNet5 參數及架構如 Fig. 13. 所示，並開始訓練這個 LeNet5 model。我把 batch size 設為 128，epochs 設為 200 次，為了防止不必要的多餘訓練造成 over-fitting 的反效果，也有另外設定 Early-Stopping 的限制，詳細的訓練 model 的程式碼(包括 Early-Stopping 及 optimizers 的設置)可參考 Fig. 14.。至於不同的 optimizer 一定也會對 model 的訓練效果有不同的影響，為了找出較適合的 optimizer，我將嘗試 RMSProp、Adam、Adagrad 及 SGD 共 4 種 optimizer，最後的訓練結果將呈現於 Table 1.。

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 6)	156
max_pooling2d (MaxPooling2D)	(None, 14, 14, 6)	0
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2416
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 120)	48120
dense_1 (Dense)	(None, 84)	10164
dropout (Dropout)	(None, 84)	0
dense_2 (Dense)	(None, 43)	3655

=====  
 Total params: 64,511  
 Trainable params: 64,511  
 Non-trainable params: 0  
 =====

Fig. 13. LeNet5 詳細架構

```
# setup Early-Stopping criteria
from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# import different optimizers
from keras.optimizers import Adam, Adagrad, RMSprop, SGD
optimizer_Adam=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)
optimizer_RMSprop = RMSprop(learning_rate=0.001, rho=0.9, momentum=0.0, epsilon=1e-07, centered=False)
optimizer_Adagrad = Adagrad(learning_rate=0.01, initial_accumulator_value=0.1, epsilon=1e-07)
optimizer_SGD = SGD(learning_rate=0.01, momentum=0.9, decay=1e-6, nesterov=False)

# specify optimizer, loss function and metric
model.compile(optimizer=optimizer_Adagrad, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# start training with specified batch_size and epochs
batch_size=128
epochs=200
LeNet5=model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_valid, y_valid), callbacks=[early_stopping])
```

Fig. 14. Model training 程式碼(包括 Early-Stopping 及 optimizers 的參數設定)

Different Optimizers' Effect on GTSRB Dataset

Optimizer Name	<i>Adam</i>	<i>RMSProp</i>	<i>SGD</i>	<i>Adagrad</i>
Training Epochs	17	12	24	112
Training Accuracy	0.97	0.96	0.97	0.95
Validation Accuracy	0.96	0.96	0.97	0.95
Prediction Accuracy	0.93	0.93	0.93	0.91

**Note:** Adagrad 的訓練所需 epochs 最多且 accuracy 最低，因此之後將排除該 optimizer

Table 1. 考慮不同 optimizers 對於 GTSRB 辨識的表現差異

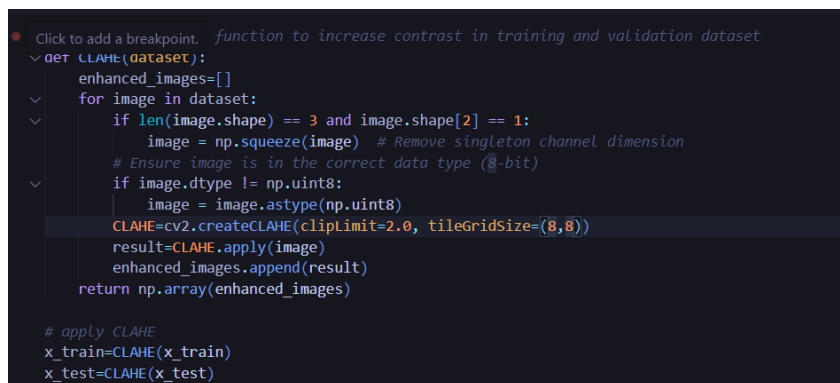
觀察 Table 1.的結果可以發現 Adagrad 的 training epochs 遠比其他



三個 optimizers 高，代表訓練所需花的時間相對多，accuracy 也不如其他三者高，因此可以將其從最終的 optimizer 選擇名單中汰除。至於 Adam、RMSProp 及 SGD 的 accuracy 大致上沒有太大的差別，因此我接下來將再想其他方法嘗試是否能提高準確率，並由這三個 optimizers 中試圖找出最適合此次 project 的 LeNet5 model。

在參考網路上的文章及詢問 ChatGPT 的意見後得到了可以嘗試 learning rate scheduler 的概念，learning rate 會依據 epochs 或是 accuracy 的變化動態更新 learning rate 的值，於是我試著增加名為 ReduceLROnPlateau 的 callback，但效果並沒有像預期的一樣變好，training 和 validation 的 accuracy 的確有增加，但是 testing accuracy 依舊卡在 0.93，無法再提高，因此這個方法只能作罷。

接著，我又考慮了把原始 images 的對比度加強，因為我注意到原始 images 的解析度都不太好，甚至有些看不清楚交通號誌的詳細內容。於是我上網搜尋了加強解析度的方法，發現了有一個叫做 Contrast Limited Adaptive Histogram Equalization(CLAHE)的演算法，可以加強灰階圖的對比度，而且 OpenCV 已經有內建好的 function 可以直接使用，因此我便針對 dataset 多增加 CLAHE 的前處理動作，程式碼呈現於 Fig. 15。



```

def CLAHE(dataset):
    enhanced_images=[]
    for image in dataset:
        if len(image.shape) == 3 and image.shape[2] == 1:
            image = np.squeeze(image) # Remove singleton channel dimension
            # Ensure image is in the correct data type (8-bit)
            if image.dtype != np.uint8:
                image = image.astype(np.uint8)
            CLAHE=cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
            result=CLAHE.apply(image)
            enhanced_images.append(result)
    return np.array(enhanced_images)

# apply CLAHE
x_train=CLAHE(x_train)
x_test=CLAHE(x_test)

```

Fig. 15. CLAHE 加強灰階圖對比度的程式碼

針對 CLAHE 的 parameters 調整，我分別嘗試(4,4)、(6,6)及(8,8)共 3 種不同大小的 tileGridSize，optimizer 先預設為 Adam，詳細的 accuracy 比較可參考 Table 2。

Different CLAHE tileGridSize's Effect on Accuracy

tileGridSize	(4,4)	(6,6)	(8,8)
Training Accuracy	0.98	0.98	0.97
Validation Accuracy	0.97	0.97	0.96
Prediction Accuracy	0.95	0.94	0.94

Note: optimizer 先預設為 Adam，其餘程式碼皆無作調整，只有多增加 CLAHE 加強對比

Table 2. 考慮不同 tileGridSize 對於 CLAHE 的效果表現

基於 Table 2.的結果，最終我選擇了(4,4)大小的 tileGridSize。目前的 prediction accuracy 已經由最初的 0.93 進步到 0.95，我又想到之前用 Data Augmentation 增加更多的 data，那如果我再讓其產生的資料更多應該會讓訓練效果更好。於是我把原先的每個 class 至少有 3000 筆資料改成至少有 5000 筆，期望可以提高準確率，optimizer 一樣先預設 Adam，而訓練出的結果也的確使 prediction accuracy 增加至 0.96，後來我還有試著讓每個 class 有 7000 筆資料，但 prediction accuracy 依然保持在 0.96，因此最終就決定讓每個 class 至少有 5000 筆資料。最後，我針對目前的狀態討論最後 optimizer 要從 Adam、RMSProp 和 SGD 選何者比較適合，結果呈現於 Table 3.。

Final Decision of Optimizers			
Optimizer name	<i>Adam</i>	<i>RMSProp</i>	<i>SGD</i>
Training Accuracy	0.98	0.97	0.97
Validation Accuracy	0.97	0.97	0.97
Prediction Accuracy	0.96	0.95	0.94
Note: 最終選擇 Adam 作為 optimizer(learning rate 固定為 0.001)			

Table 3. 在使用 CLAHE 和增加更多資料後，不同 optimizer 的準確率

## 5. Accuracy Curves, Confusion Matrix, and Classification Report

最後，我的 model 訓練模式及參數設定已經確定，所有相關資料將列在 Table 4.，而最後訓練結果的 Accuracy/Loss Curve(Fig. 16.(a)、(b))、Confusion Matrix(Fig. 17.)及 Classification Report(Fig. 18.)將依序列在下方。

Final Model and Other Parameters Configure	
Preprocessing Techniques	Grayscale Conversion
	Data Augmentation
	CLAHE Contrast Enhancement
	Dataset Normalize
NN Structure	LeNet5
Callbacks	Early Stopping
Optimizer	Adam(lr=0.001)
Batch Size	128
Training Epochs	18(大約 4min)
Prediction Accuracy	0.96

Table 4. 最終選用的 model、相關參數及前處理技術的總整理

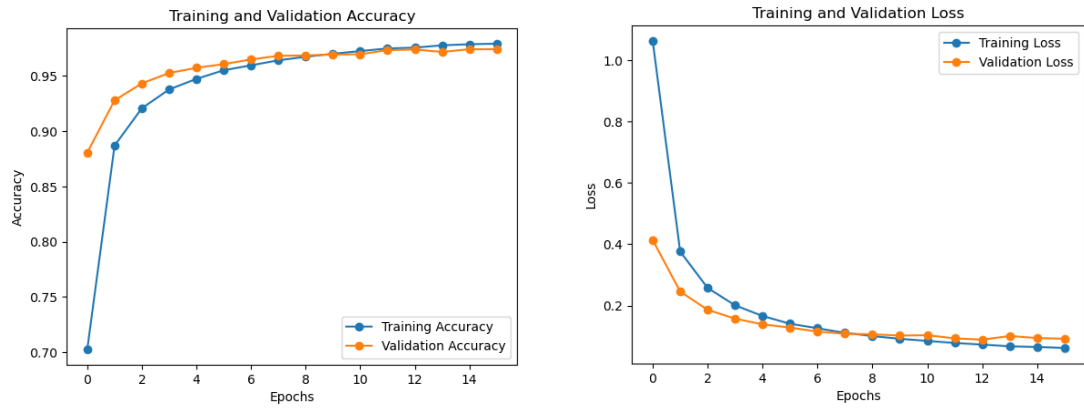


Fig. 16. (a) Accuracy Curves (b) Loss Curves

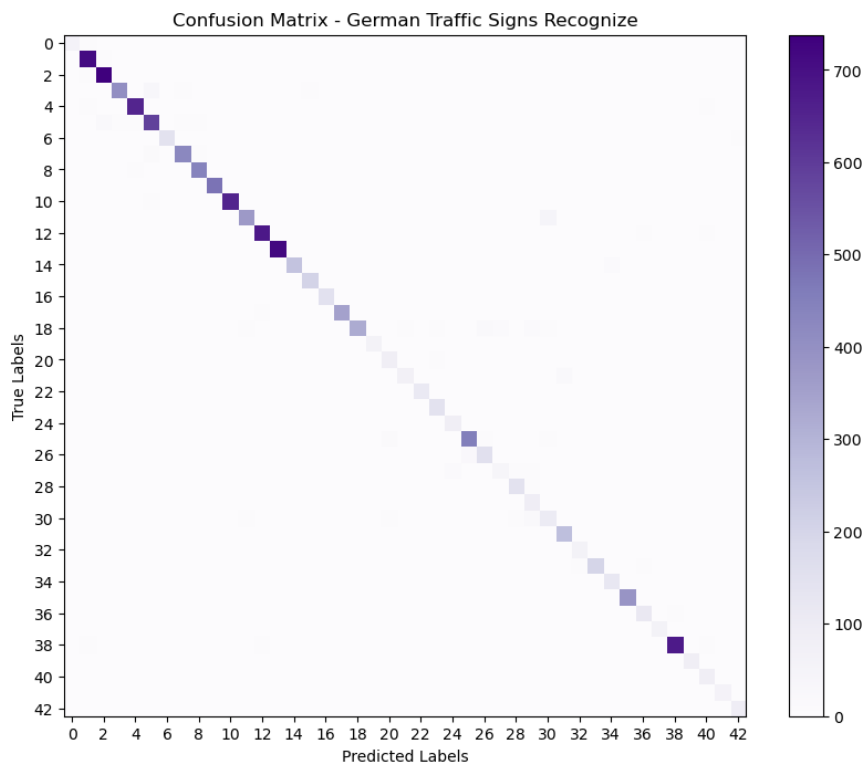


Fig. 17. Confusion Matrix of LeNet5 prediction result

accuracy			0.96	12630
macro avg	0.93	0.94	0.93	12630
weighted avg	0.96	0.96	0.96	12630

Fig. 18. Classification Report

## 6. Test LeNet5 model on New Images

以下將針對訓練完的 model 用於預測沒看過的 10 張照片做測試，檢視這個 model 的廣泛性是否足夠。我先使用 Fig. 19. 的程式碼把 testImages/ 內的照片讀出，並同樣做前處理(gray-scale conversion、CLAHE 及 normalize)，最後處理完的影像如 Fig. 20. 所示。

```
# Load the images from the testImages directory
import glob
import matplotlib.image as mpimg

fig, axes=plt.subplots(2,5,figsize=(5,2))
fig.subplots_adjust(hspace=0.2, wspace=0.01)
axes=axes.ravel()

test_images=[]
for i, img in enumerate(glob.glob('./testImages/*.png')):
    image=cv2.imread(img)
    axes[i].axis('off')
    test_images.append(image)

test_images=np.array(test_images)

# convert to grayscale
test_images=np.sum(test_images/3, axis=3, keepdims=True)

# apply CLAHE to test images
test_images=CLAHE(test_images)

# normalize the test images
test_images=test_images/127.5-1

for i in range(10): # Adjusted index range
    axes[i].imshow(test_images[i].squeeze()) # Squeezed to remove channel dimension for grayscale
```

Fig. 19. 讀出 testImages 內影像，並做前處理



Fig. 20. 前處理後的影像

接著，我根據已知的 43 類 traffic signs，手動把 testImages 的影像對應的 label 輸入至 test\_labels 中，如 Fig. 21 所示，最後用 evaluate function 評估的表現呈現於 Fig. 22.，準確率為 0.9。仔細比對後，可以發現是最後一個應該是第 25 種的 road work，而反而被誤判為 30 的 Beware of ice/snow。我經過觀察發現應該是因為解析度太低，而兩者的圖案輪廓在模糊的狀態下看起來又十分接近，因而有誤判的情況產生。

```
# Based on the signnames.csv, provide the 10 test images' labels
test_labels=[17,13,12,14,11,38,4,35,33,25]

# evaluate the accuracy
result=np.argmax(model.predict(test_images), axis=1)
print(f'predict label: {result}')
print()
accuracy=model.evaluate(test_images, np.array(test_labels))
✓ 0.2s
```

Fig. 21. 手動輸入 label，並用 evaluate 評估模型表現

```
1/1 [=====] - 0s 26ms/step
predict label: [17 13 12 14 11 38 4 35 33 30]

1/1 [=====] - 0s 126ms/step - loss: 1.0591 - accuracy: 0.9000
```

Fig. 22. 預測準確率為 0.9，誤判的 traffic sign 為編號 25 的 road work

### 三、問題討論

#### 1. The Necessity of Data-Preprocessing

從這次的 project 實作中，我體認到 data-preprocessing 對於 machine learning 的訓練是不可或缺的，因為這次使用到的前處理技術對於 model training 的成果提升有顯著的幫助，data augmentation 可以讓我在資料不足或不平均的狀況下，產生足夠的資料訓練 model，避免過低的 accuracy 或是偏向特定 class 的預測結果。Normalization 也使 model 在訓練過程的收斂速度較快，至於 CLAHE 則是在 GTSRB dataset 的影像大多模糊不清的狀況下，最大限度地加強影像對比度，使原本看起來輪廓類似的影像可以被正確區分開來。

### 四、結論

在這次的 Project 我使用了 LeNet5 搭配一些資料前處理的技巧去訓練並預測 GTSRB 資料集共分為 43 類的交通號誌影像，最終達到的預測準確率可以達到 96%。至於未來若是要提高 GTSRB 的辨識率，我認為可以改變不同的 NN 架構，例如常見的 ResNet、VGG16 及 GoogLeNet...等。而我認為提升準確率也可以從另一個方向下手，就是著重在影像處理，試圖利用一些演算法，針對模糊不清的影像加強輪廓及對比度，像這次我使用的 CLAHE 就是其中一個方法。

最後，以下將附上我該次 Project 的 GitHub Link，裡面有附上原始的 Jupyter Notebook 程式碼檔案。

GTSRB Project Repo: [Project 2 GitHub Link](#)

## 五、參考資料

1. <https://github.com/Goddard/udacity-traffic-sign-classifier/>
2. [data-augmentation-explanation](#)
3. [perspective-transformation-explanation](#)
4. [learning-rate-scheduler-idea](#)
5. [https://en.wikipedia.org/wiki/Adaptive\\_histogram\\_equalization](https://en.wikipedia.org/wiki/Adaptive_histogram_equalization)
6. [CLAHE-OpenCV-docs](#)