國立清華大學

工業工程與工程管理學系

IIE 期末 project

使用 DCGAN 進行 Augmentation

對腦腫瘤圖像進行癌症分類與辨識

學生 ： 110034532 白哲睿

指導教授 ： 邱銘傳教授

一、 背景介紹

(一) 背景說明

腦腫瘤是世界上最致命的癌症之一。這是成人和兒童常見的癌症。它具有最低的存活率，並且根據它們的位置、質地和形狀而具有多種類型。腦腫瘤的錯誤分類會導致不良後果。因此，在早期階段確定正確的腫瘤類型和分級對於選擇精確的治療計劃具有重要作用。

隨著技術的進步，醫學影像領域得到了突飛猛進的發展。腫瘤可能是良性或惡性，良性腫瘤僅限於大腦，因此本質上是靜態的。它們對大腦的鄰近組織施加了巨大的壓力。另一方面，惡性組織會從您身體任何部位的癌細胞群開始擴散，然後遷移到您的大腦。故我們主要關注的是在早期發現腦腫瘤以採取適當的治療，且根據這些信息，可以確定最合適的治療方法，因此，如果檢測準確和詳細，則可以大大增加患者的生存和治療機會。由最新研究所示圖像辨識處理已被證明有助於診斷腫瘤例如 MRI 圖像。

(二) 研究目的

檢查患者大腦的磁共振成像 (MRI) 圖像是一種區分腦腫瘤的有效技術。但是腦腫瘤很一種複雜症狀，且腦腫瘤的大小和位置有很多異常，這使得完全了解腫瘤的性質非常困難，此外 MRI 分析需要專業的神經外科醫生。很多時候，發展中國家缺乏熟練的醫生和缺乏對腫瘤的知識，使得從 MRI 生成報告變得非常具有挑戰性和耗時。且由於腦腫瘤的 MRI 資料集每一個類別的數據量不平衡，故我們會透過 DCGAN 對數據量相對較少的類別進行數據增強，在進行 CNN 的分類，以期望能夠提高 CNN 的分類準確率。

(三) 問題描述(5W1H)

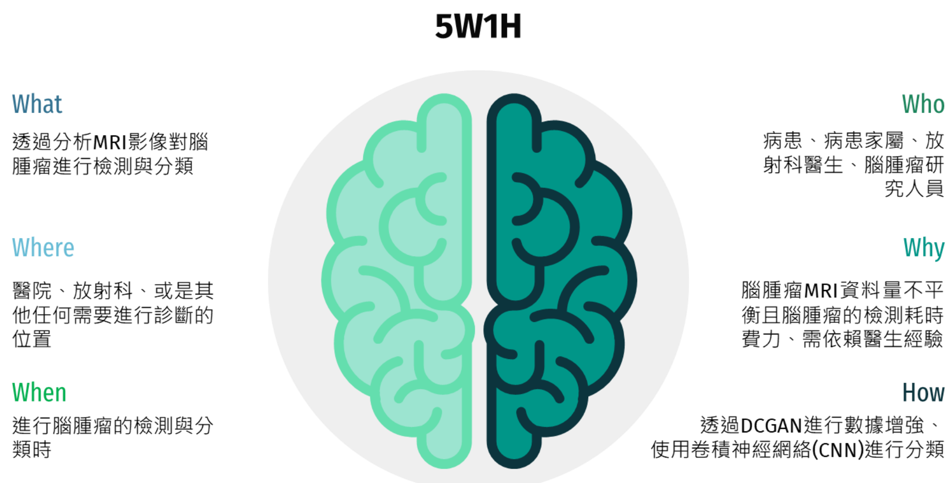


圖 1、5W1H

二、 方法介紹

(一) DenseNet121

今天我們要介紹的是 DenseNet 模型，它的基本思路與 ResNet 一致，但是它建立的是前面所有層與後面層的密集連接（dense connection），它的名稱也是由此而來。DenseNet 的另一大特色是通過特徵在 channel 上的連接來實現特徵重用（feature reuse）。這些特點讓 DenseNet 在參數和計算成本更少的情形下實現比 ResNet 更優的性能，DenseNet 也因此斬獲 CVPR 2017 的最佳論文獎。

(二) DCGAN

DCGAN 全稱 Deep Convolutional Generative Adversarial Networks，中文名曰深度卷積對抗網路。

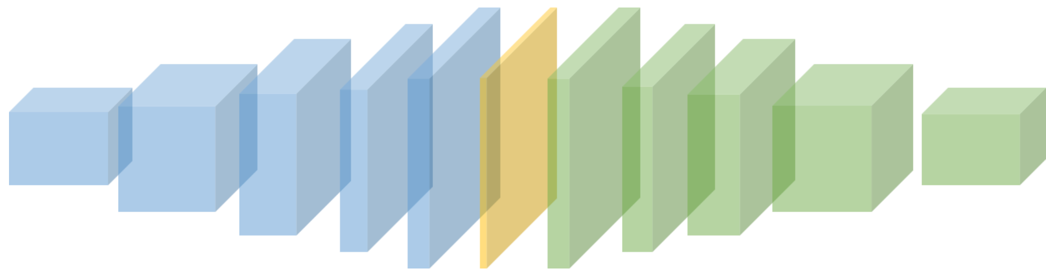


圖 2、DCGAN 結構

我們知道深度學習中對影象處理應用最好的模型 CNN，那麼我們將 CNN 與 GAN 結合就是我們看到 DCGAN，其結構可分為 Generator 與 Discriminator，其中 Generator 為圖三中 藍色之結構，而 Discriminator 為圖三中綠色之結構，最後生成出的圖像結果為黃色之結構。Generator 之程式碼以圖 4 所示，而 Discriminator 之程式碼以圖 5 所示。

圖 3、Generator 程式碼

```
def generator(z, output_channel_dim, training):
    with tf.variable_scope("generator", reuse= not training):

        # 8x8x1024
        fully_connected = tf.layers.dense(z, 8*8*1024)
        fully_connected = tf.reshape(fully_connected, (-1, 8, 8, 1024))
        fully_connected = tf.nn.leaky_relu(fully_connected)

        # 8x8x1024 -> 16x16x512
        trans_conv1 = tf.layers.conv2d_transpose(inputs=fully_connected, filters=512, kernel_size=[5,5], strides=[2,2], padding="SAME", kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV), name="trans_conv1")
        batch_trans_conv1 = tf.layers.batch_normalization(inputs = trans_conv1, training=training, epsilon=EPSILON, name="batch_trans_conv1")
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1, name="trans_conv1_out")

        # 16x16x512 -> 32x32x256
        trans_conv2 = tf.layers.conv2d_transpose(inputs=trans_conv1_out, filters=256, kernel_size=[5,5], strides=[2,2], padding="SAME", kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV), name="trans_conv2")
        batch_trans_conv2 = tf.layers.batch_normalization(inputs = trans_conv2, training=training, epsilon=EPSILON, name="batch_trans_conv2")
        trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2, name="trans_conv2_out")

        # 32x32x256 -> 64x64x128
        trans_conv3 = tf.layers.conv2d_transpose(inputs=trans_conv2_out, filters=128, kernel_size=[5,5], strides=[2,2], padding="SAME", kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV), name="trans_conv3")
        batch_trans_conv3 = tf.layers.batch_normalization(inputs = trans_conv3, training=training, epsilon=EPSILON, name="batch_trans_conv3")
        trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3, name="trans_conv3_out")

        # 64x64x128 -> 128x128x64
        trans_conv4 = tf.layers.conv2d_transpose(inputs=trans_conv3_out, filters=64, kernel_size=[5,5], strides=[2,2], padding="SAME", kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV), name="trans_conv4")
        batch_trans_conv4 = tf.layers.batch_normalization(inputs = trans_conv4, training=training, epsilon=EPSILON, name="batch_trans_conv4")
        trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4, name="trans_conv4_out")

        # 128x128x64 -> 128x128x3
        logits = tf.layers.conv2d_transpose(inputs=trans_conv4_out, filters=3, kernel_size=[5,5], strides=[1,1], padding="SAME", kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV), name="logits")
        out = tf.tanh(logits, name="out")
        return out
```

圖 4、Discriminator 程式碼

```
def discriminator(x, reuse):
    with tf.variable_scope("discriminator", reuse=reuse):

        # 128*128*3 -> 64x64x64
        conv1 = tf.layers.conv2d(inputs=x, filters=64, kernel_size=[5, 5], strides=[2, 2], padding="SAME",
                                 kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='conv1')
        batch_norm1 = tf.layers.batch_normalization(conv1, training=True, epsilon=EPSILON, name='batch_norm1')
        conv1_out = tf.nn.leaky_relu(batch_norm1, name="conv1_out")

        # 64x64x64-> 32x32x128
        conv2 = tf.layers.conv2d(inputs=conv1_out, filters=128, kernel_size=[5, 5], strides=[2, 2], padding="SAME",
                                 kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='conv2')
        batch_norm2 = tf.layers.batch_normalization(conv2, training=True, epsilon=EPSILON, name='batch_norm2')
        conv2_out = tf.nn.leaky_relu(batch_norm2, name="conv2_out")

        # 32x32x128 -> 16x16x256
        conv3 = tf.layers.conv2d(inputs=conv2_out, filters=256, kernel_size=[5, 5], strides=[2, 2], padding="SAME",
                                 kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='conv3')
        batch_norm3 = tf.layers.batch_normalization(conv3, training=True, epsilon=EPSILON, name='batch_norm3')
        conv3_out = tf.nn.leaky_relu(batch_norm3, name="conv3_out")

        # 16x16x256 -> 16x16x512
        conv4 = tf.layers.conv2d(inputs=conv3_out, filters=512, kernel_size=[5, 5], strides=[1, 1], padding="SAME",
                                 kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                 name='conv4')
        batch_norm4 = tf.layers.batch_normalization(conv4, training=True, epsilon=EPSILON, name='batch_norm4')
        conv4_out = tf.nn.leaky_relu(batch_norm4, name="conv4_out")

        # 16x16x512 -> 8x8x1024
        conv5 = tf.layers.conv2d(inputs=conv4_out, filters=1024, kernel_size=[5, 5], strides=[2, 2], padding="SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=WEIGHT_INIT_STDDEV),
                                name='conv5')
        batch_norm5 = tf.layers.batch_normalization(conv5, training=True, epsilon=EPSILON, name='batch_norm5')
        conv5_out = tf.nn.leaky_relu(batch_norm5, name="conv5_out")

        flatten = tf.reshape(conv5_out, (-1, 8*8*1024))
        logits = tf.layers.dense(inputs=flatten,
                                 units=1,
                                 activation=None)
        out = tf.sigmoid(logits)
        return out, logits
```

DCGAN 的原理和 GAN 是一樣的，這裡就不在贅述。它只是把上述的 G 和 D 換成了兩個卷積神經網路（CNN）。但不是直接換就可以了，DCGAN 對卷積神經網路的結構做了一些改變，以提高樣本的質量和收斂的速度，這些改變有：

- 取消所有 pooling 層。G 網路中使用轉置卷積（transposed convolutional layer）進行上取樣，D 網路中用加入 stride 的卷積代替 pooling。
- 在 D 和 G 中均使用 batch normalization
- 去掉 FC 層，使網路變為全卷積網路
- G 網路中使用 ReLU 作為啟用函式，最後一層使用 tanh
- D 網路中使用 LeakyReLU 作為啟用函式

(三) 遷移學習

遷移學習(Transfer Learning)是一種機器學習方法，是把一個領域(源領域)的知識，遷移到另外一個領域(目標領域)，使得目標領域能夠取得更好的學習效果。當目標域數據量較小或者數據的標籤很難獲取，可以通過數據量充 足或者容易獲取標簽且和該任務相似的任務(源域)來遷移學習 。且從頭建立模型會較複雜且耗時，因此通過遷移學習可以加快學習效率。本研究初選時

所選擇的 CNN 網路結構，皆已在 ImageNet 圖片資料集中進 行預訓練，因此大幅提升本研究的訓練效率。

三、 個案研究

(一) 研究流程

本研究將利用公開的腦腫瘤 MRI 數據集，其中我們會使用 DCGAN 對數據集數量不平衡的類別進行數據增強，接著對模型進行訓練，最後透過 Accuracy, Sensitivity, Specificity, Precision, 及 F1-Score 等指標對數據增強前後的 CNN 分類結果進行比較。

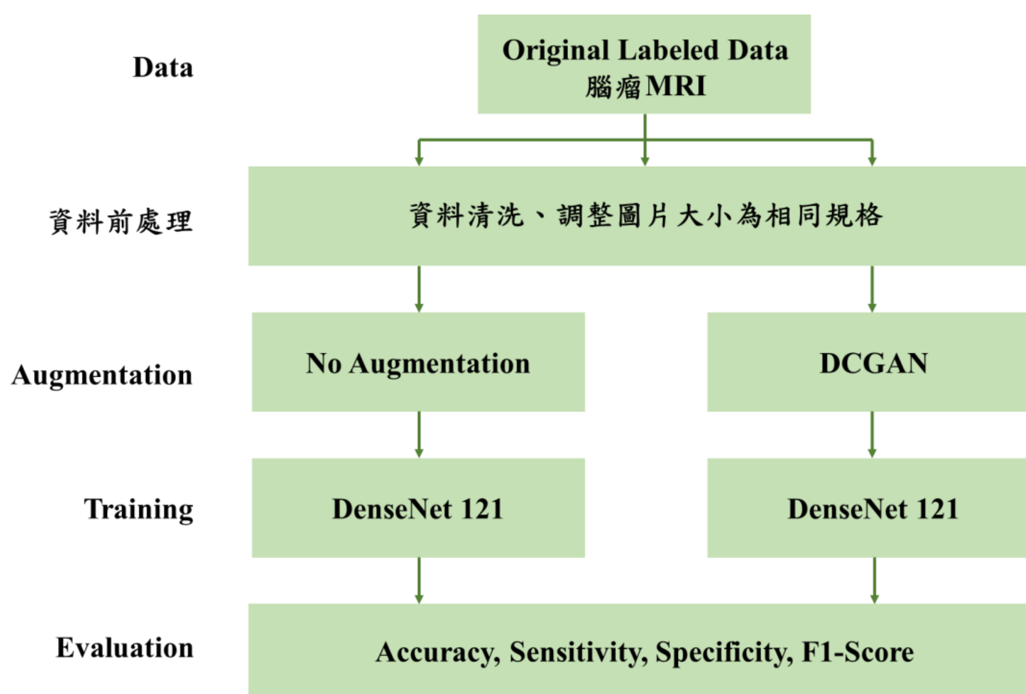


圖 5、研究流程

(二) 資料集介紹

本次研究所使用的 Dataset 在 Kaggle 上所取得，為腦部腫瘤的 MRI 圖片。總共有四種類別，分別為 glioma tumor(膠質瘤)、meningioma tumor(腦膜瘤)、no tumor(無腫瘤)與 pituitary tumor(垂體瘤)。其原檔案有 2870 張訓練集圖片與 394 張測試集圖片檔(.jpg)，皆為有標籤資料。

➢ 腦腫瘤之數據量表:

腦腫瘤類別	訓練集圖片數量	測試集圖片數量
glioma tumor(膠質瘤)	826 張	100 張
meningioma tumor(腦膜瘤)	822 張	115 張
no tumor(無腫瘤	395 張	105 張
pituitary tumor(垂體瘤)	827 張	74 張

表一、各類別圖片數量表

(三) 資料前處理

Step 1： 資料清洗(Data Cleansing)

在進行訓練前，我們需要透過人工的方式將空白或模糊的 MRI 圖片剔除在數據集外，並確認最後得到的數據集的圖像皆為清晰且可辨識的。

Step 2： 調整圖片大小為相同規格

由於原本的圖片大小不一，且大部分 CNN 網路架構皆要求 input 圖片大小須為 224*224。因此如圖 6，我們利用 resize 手法將圖片統一大小。

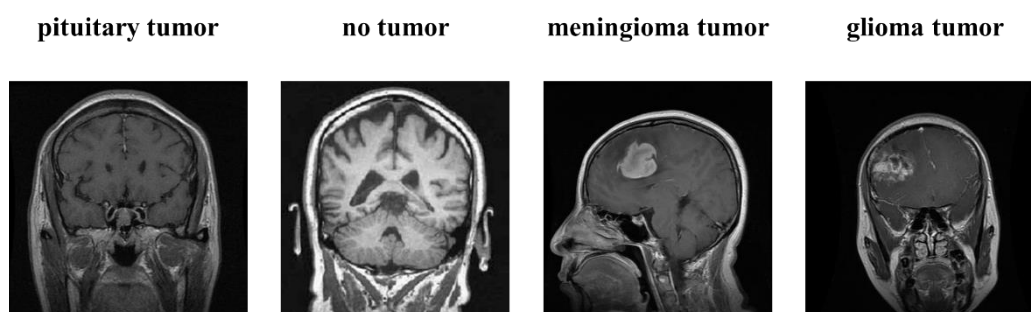


圖 6、Resize 後的 MRI 圖像，大小為 224*224

最後再將 label 利用 one hot encoding 的方式，使其能對應到各個圖片，轉碼方式如下圖

```
y_train_new = []
for i in y_train:
    y_train_new.append(labels.index(i))
y_train = y_train_new
y_train = tf.keras.utils.to_categorical(y_train)  # 將label轉成one hot encoding的形式


y_test_new = []
for i in y_test:
    y_test_new.append(labels.index(i))
y_test = y_test_new
y_test = tf.keras.utils.to_categorical(y_test)
```

圖 7、Label 轉碼程式

Step 3： 數據集平衡

為避免沒有腦瘤類別的數據資料過少，我們使用 DCGAN 的方式將此類別的資料量擴增一倍以解決資料集數據不平衡的情況。如下圖

```
1 # Training
2 print("Start!")
3 input_images = np.asarray([np.asarray(Image.open(file).convert('RGB').resize((IMAGE_SIZE, IMAGE_SIZE))) for file in glob(INPUT_DATA_DIR + '*')])
4 print ("Input: " + str(input_images.shape))
5
6 np.random.shuffle(input_images)
7
8 sample_images = random.sample(list(input_images), SAMPLES_TO_SHOW)
9 show_samples(sample_images, OUTPUT_DIR + "inputs", 0)
10
11 with tf.Graph().as_default():
12     train(get_batches(input_images), input_images.shape)
13     #saver = tf.train.Saver()
```

圖 8、DCGAN 之數據增強

Step 4： 資料重新編排

由於原先測試集圖片總數過多，且沒額外劃分驗證集圖片資料。因此我們先將訓練集與測試集圖片合併，隨機打亂後再按照訓練與測試集 9:1 的比例重新劃分。如下圖

```
[ ] #shuffle是將圖片順序隨機打亂
    X_train,  y_train  =  shuffle(X_train,y_train,  random_state=101)
    #  train_test_split為交叉驗證的函數，函數中X_train代表所要劃分的樣本特徵集，y_train代表所要劃分的樣本結果，test_size代表樣本占比
    X_train,X_test,y_train,y_test  =  train_test_split(X_train,y_train,  test_size=0.1,random_state=101)
```

圖 9、重新分割資料集

(四) DCGAN 之建立與數據增強

利用 DCGAN 進行影像生成，DCGAN 訓練模型分為兩個網路，一個為 Generator，由反捲基組成(五層反捲基)；另一個為 Discriminator，由捲基網路組成(五層捲基)，架構如圖，參數設定如表。經過 500 個 epoch 訓練後，進行影像生成與訓練損失。

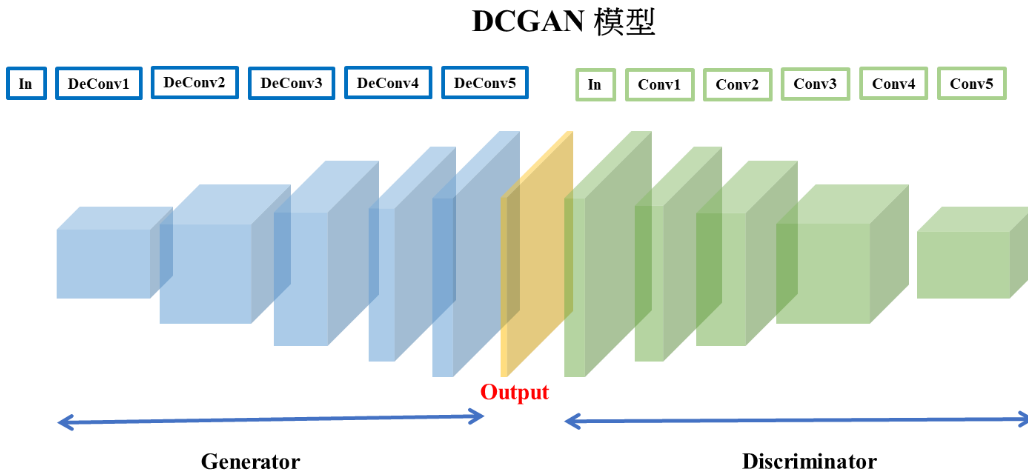


圖 10、DCGAN 完整模型架構

DCGAN 參數設定	
IMAGE_SIZE	128
NOISE_SIZE	100
LR_D	0.00001
LR_G	0.0004
BATCH_SIZE	64
EPOCHS	500
BETA1	0.5
SAMPLES_TO_SHOW	395

表二、DCGAN 之參數設定

腦腫瘤類別	訓練集圖片數量	測試集圖片數量
glioma tumor	826 張	100 張
meningioma tumor	822 張	115 張
no tumor	790 張	105 張
pituitary tumor	827 張	74 張

表三、經資料前處理後的圖片數量表

(五) DenseNet121 模型建立與訓練

本研究利用 TensorFlow 開源軟體庫進行訓練，其中我們使用經 ImageNet 預訓練過的 CNN 網絡架構-DenseNet121。值得注意的是我們為了減少模型參數而加入 2D 全局平均池化層來避免過擬合，另外我們也在模型中加入 Dropout 層減少訓練的時間、最後再利用 Softmax 函數確保能更有效的分類。模型建立的程式碼如下圖:

```
[ ]  #DenseNet121
     effnet  =  tf.keras.applications.densenet.DenseNet121(weights='imagenet',include_top=False,input_shape=(image_size,image_size,3))
     model  =  effnet.output
     model  =  tf.keras.layers.GlobalAveragePooling2D()(model)
     model  =  tf.keras.layers.Dropout(rate=0.4)(model)
     model  =  tf.keras.layers.Dense(4,activation='softmax')(model)
     model  =  tf.keras.models.Model(inputs=effnet.input,  outputs  =  model)
```

圖 11、模型建立與模型架構設定

```
history  =  model.fit(X_train,y_train,validation_split=0.1,  epochs  =30,  verbose=1,  batch_size=8,
                        callbacks=[tensorboard,checkpoint,reduce_lr])

331/331 [==============================] - ETA: 0s - loss: 0.0462 - accuracy: 0.9886
Epoch 00018: val_accuracy did not improve from 0.96259
331/331 [==============================] - 86s 261ms/step - loss: 0.0462 - accuracy: 0.9886 - val_loss: 0.1015 - val_accuracy: 0.9626 - lr: 2.4300e-06
Epoch 19/30
331/331 [==============================] - ETA: 0s - loss: 0.0417 - accuracy: 0.9890
Epoch 00019: val_accuracy did not improve from 0.96259

Epoch 00019: ReduceLROnPlateau reducing learning rate to 7.289999985005124e-07.
331/331 [==============================] - 86s 261ms/step - loss: 0.0417 - accuracy: 0.9890 - val_loss: 0.1011 - val_accuracy: 0.9626 - lr: 2.4300e-06
Epoch 20/30
331/331 [==============================] - ETA: 0s - loss: 0.0377 - accuracy: 0.9924
Epoch 00020: val_accuracy did not improve from 0.96259
331/331 [==============================] - 86s 261ms/step - loss: 0.0377 - accuracy: 0.9924 - val_loss: 0.1014 - val_accuracy: 0.9626 - lr: 7.2900e-07
Epoch 21/30
331/331 [==============================] - ETA: 0s - loss: 0.0420 - accuracy: 0.9890
Epoch 00021: val_accuracy did not improve from 0.96259

Epoch 00021: ReduceLROnPlateau reducing learning rate to 2.1870000637136398e-07.
331/331 [==============================] - 89s 268ms/step - loss: 0.0420 - accuracy: 0.9890 - val_loss: 0.0989 - val_accuracy: 0.9626 - lr: 7.2900e-07
Epoch 22/30
```

圖 12、模型訓練

(六) 參數優化

本研究為了探究如何提高 CNN 分類的精確度而設計了一個 4 因子 3 水準的實驗，期望能透過調整這幾個因子與其水準達到更高的準確度，並希望能找到最佳的水準組合在其他參數皆不變的情況下，其中我們在實驗中會分別對經過 DCGAN 數據增強與沒有經過增強的數據進行 CNN 模型的分類。

詳細因子與水準如下表:

表 4、實驗因子及水準

因子	說明	Level 1	Level 2	Level 3
A	Dropout	0.4	0.5	0.6
B	Optimizer	Adam	AdaDelta	SGD
C	Activate function	Tahn	ELU	ReLu
D	Learning rate	0.01	0.005	0.001

接下來我們需要檢測實驗設計中的所有參數組合，但如果要試驗所有的參數組合需要實驗 81 次，所以我們為了能減少實驗的總次數以達到減少時間與調整參數的總次數，並且希望能獲得與全因子實驗設計近乎相同的結果，我們透過實驗設計中的田口方法，選擇上述的四項因子與各三個水準並應用 L9 直交表來找到最佳的參數組合以達到參數優化的結果。值得注意的是針對每一次的實驗，我們統一使用 30 epoch 進行模型的訓練。
實驗設計的 L9 直交表如下:

表 5、L9 直交表

實驗	Dropout	Optimizer	Activate function	Learning rate
1	0.4	Adam	Tahn	0.005
2	0.4	AdaDelta	Tahn	0.01
3	0.4	SGD	ELU	0.001
4	0.5	Adam	SGD	0.01
5	0.5	AdaDelta	SGD	0.005
6	0.5	SGD	ELU	0.01
7	0.6	Adam	ELU	0.001
8	0.6	AdaDelta	Tahn	0.005
9	0.6	SGD	SGD	0.001

(七) 實驗設計之結果

- DenseNet121(未經過 DCGAN 數據增強)

  下表為 DenseNet121 的實驗組合與其結果，其中準確率最高的組合為第四個實驗，其中準確率達到了 0.9132，接著若我們將這 9 次的實驗結果通過統計分析後可以觀察到各個因子對於準確度的影響程度。

表 6、DenseNet121 實驗設計結果

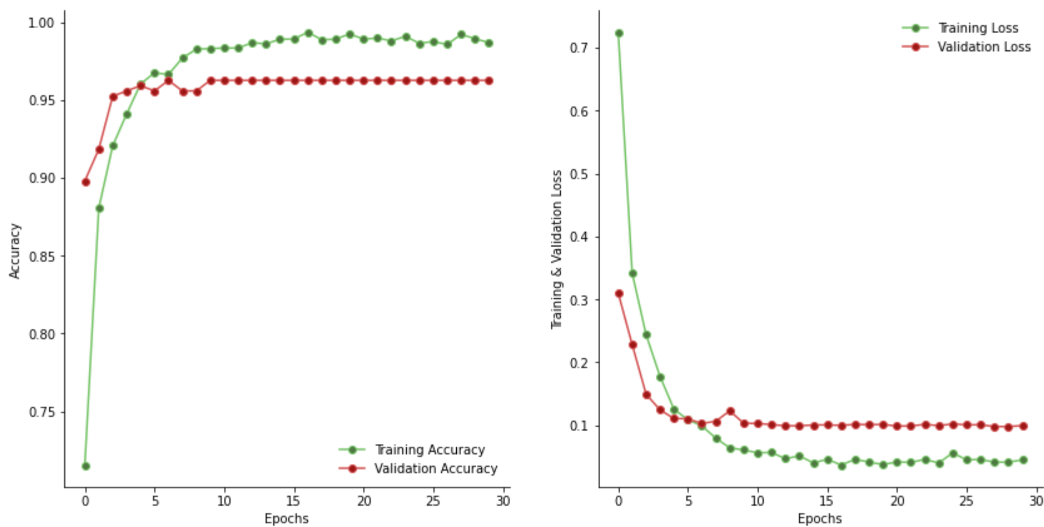
實驗	Dropout	Optimizer	Activate function	Learning rate	Accuracy
1	0.4	Adam	Tahn	0.005	0.8651
2	0.4	AdaDelta	Tahn	0.01	0.8943
3	0.4	Adagrad	ELU	0.01	0.8815
4	0.5	Adam	SGD	0.001	0.9132
5	0.5	AdaDelta	SGD	0.005	0.9013
6	0.5	Adagrad	ELU	0.01	0.912
7	0.6	Adam	ELU	0.001	0.8753
8	0.6	AdaDelta	Tahn	0.005	0.8864
9	0.6	Adagrad	SGD	0.001	0.8946

下圖為未經過 DCGAN 數據增強的損失函數結果與準確率的趨勢圖，左圖為訓練集與驗證集的準確度趨勢圖，透過觀察我可以發現 Epochs 越大則兩個準確度越大且逐漸趨於平穩並逐漸收斂。

右圖為訓練集與驗證集的損失函數趨勢圖，同樣透過觀察我們可以發現 Epochs 越大則損失函數越低，且損失函數會趨於平穩並逐漸收斂。

圖 13、No Augmentation 之 Accuracy 與 Loss

Epochs vs. Training and Validation Accuracy/Loss



- DenseNet121(經過 DCGAN 數據增強)

  下圖為經過數據增強後 DenseNet121 之實驗組合與其結果，其中準確度最高的組合為第 7 個實驗，準確度高達 0.9864，且透過觀察我們可以發現經過數據增強後的分類準確度能有明顯的上升。

表 7、DenseNet121 實驗結果

實驗	Dropout	Optimizer	Activate function	Learning rate	Accuracy
1	0.4	Adam	Tahn	0.005	0.8955
2	0.4	AdaDelta	Tahn	0.01	0.9232
3	0.4	Adagrad	ELU	0.01	0.9153
4	0.5	Adam	SGD	0.001	0.8966
5	0.5	AdaDelta	SGD	0.005	0.9472
6	0.5	Adagrad	ELU	0.01	0.9561
7	0.6	Adam	ELU	0.001	0.9864
8	0.6	AdaDelta	Tahn	0.005	0.9247
9	0.6	Adagrad	SGD	0.001	0.9311

四、 結論

(一) 實驗結果之分析

通過上述針對數據增強前後的實驗對照組，我們可以得到兩組最佳的參數組合如表 8，透過這張表我們可以明顯的觀察到有經過數據增強的最佳水準組合之準確度明顯高於未經過數據增強的最佳水準組合，所以我們可以證明使用 DCGAN 對資料及進行數據增強來平衡數據集的各類別數量對於 CNN 分類的準確度提升是有幫助的。

表 8、最佳水準組合

Data Augmentation	Dropout	Optimizer	Activate function	Learning rate	Accuracy
No	0.5	Adam	SGD	0.001	0.9132
Yes	0.6	Adam	ELU	0.001	0.9864

(二) 貢獻

本研究利用 DCGAN 的方法進行數據增強以平衡整體數據的資料量，並且使用遷移學習來預訓練使用 ImageNet 數據集之 CNN 架構的 DenseNet 模型，並且比較沒有使用數據增強時的腦腫瘤的辨識率，最後透過實驗設計的田口方法找出最佳的參數水準組合。

(三) 侷限性

因 DCGAN 之參數調整的困難與時間緊迫的限制，故我們僅參考文獻提供的參數而並未進行參數調整，另外在本研究中並未評估 DCGAN 生成出的 MRI 圖像，也並未運用不同的基於 CNN 架構之模型進行比較與分析。

由於應用於本研究的資料集圖像品質較佳，故訓練時幾乎沒有預到低畫質或是圖像模糊的情況，所以在進行臨床的檢驗時可能因圖像的缺陷造成準確度下降的可能性。

(四) 適用性

雖然有上述幾個侷限性，但因為優異的實驗結果，我認為此方法有很大的機會可以應用於臨床實驗，儘管無法完全讓醫師或專家評藉此模型的結果作為診斷依據或是研究結論，但是這個模型產生的結果仍可以做為輔助的工具幫助專家或醫生作為參考的依據，以減少時間與人力的耗費。

(五) 未來改善

未來我們可以透過將 DCGAN 的參數進行調整與優化，以幫助模型生成更加清晰或高畫質的圖像，並且評估合成影像的表現，抑或本研究可以嘗試其他最新的生成圖像模型來比較此研究的訓練結果，並運用不同的資料集來驗證模型的泛化能力。

參考資料

https://link.springer.com/article/10.1007/s11063-020-10398-2

https://link.springer.com/chapter/10.1007%2F978-981-15-0222-4_52

https://www.kaggle.com/sartajbhuvaji/brain-tumor-classification-mri