Python Programming for Deep Learning
Final Report

Group 4
105501539 吳亭葳
105501007 陳薪宇

一、過程:

我們先在網路上找別人用過的模型，包含參考台鐵的多層 CNN、DenseNet、ResNet+GRU，一開始沒有改太多參數，一方面先熟悉 python、collab、keras、tensorflow 的運作方式，一方面讓我們了解深度神經學習的流程，以及機器學習的輔助工具像是 wandb、TensorBoard 等等。最終選用的模型在最後一周才被我們選定，data01 我們是使用多層 CNN 和一層 GRU 作為模型，在 test01 達到我們所嘗試中的最高準確率；data02 我們是使用多個多層 CNN 和數個 GRU 做 Ensemble，在 test02 有達到我們所嘗試中的最高準確率。

二、能讓網路更好的幾個方向:

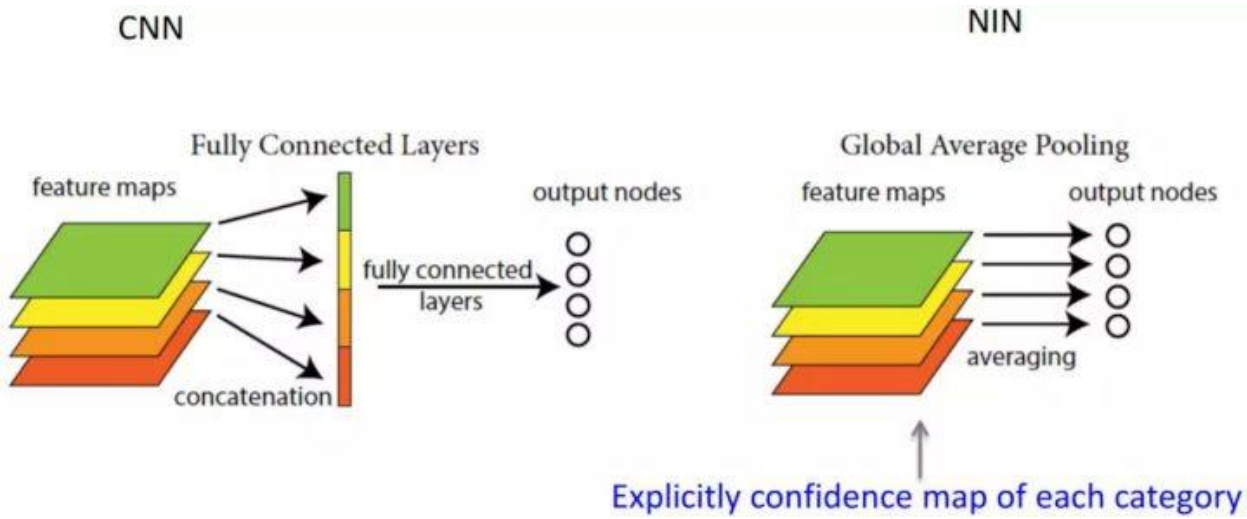
A.CNN:kener size:

影像辨識的部分，不同特徵用不同的 kernel size 會獲得不同的結果，可能原因在看特徵比較細小時用小 kernel size 較好；特徵較大時可以使用較大尺寸的 kernel size，這塊我們嘗試的不多，太早進入下一步，使前面這塊來不急做太多嘗試，看到許多其他組分享將 kernel size 從 3*3 改至 5*5，會有些許提升，可能這份資料集的特徵用 3*3 真的太小，採用 3*3 會比 5*5 來的較差。

B.GAP:

在不斷加深網路的時候，參數量也不知不覺變得過於龐大，特別是全連接層的部分。在有大量參數的情況下，training 的過程中可能會大幅增加所需時間，也有可能訓練至過擬合，需要用 dropout 去捨棄掉一些參數，這時候 GAP(Global Average Pooling)就可以發揮他的優點。

下圖是全連接層和 GAP 的簡單原理比較圖:

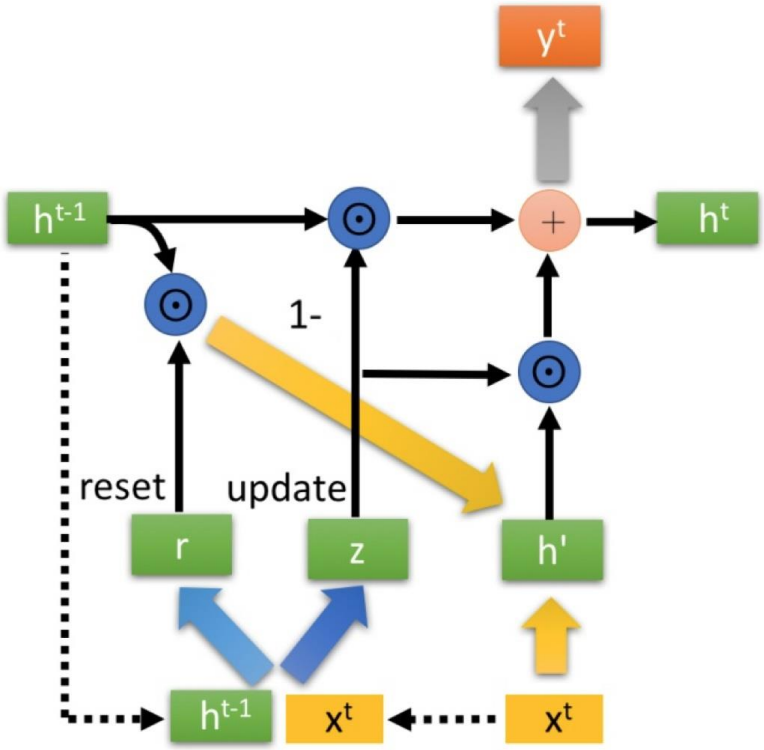


可發現 GAP 直接對不同 channel(feature map)取其平均值後用 softmax 函數做分類，而非接上大量參數的全連接層後，再 softmax 後分類出不同的結果，可防止參數過多導致的過擬合，也不失其學習的能力，但網路上有提到使用 GAP 可能收斂速度會下降。

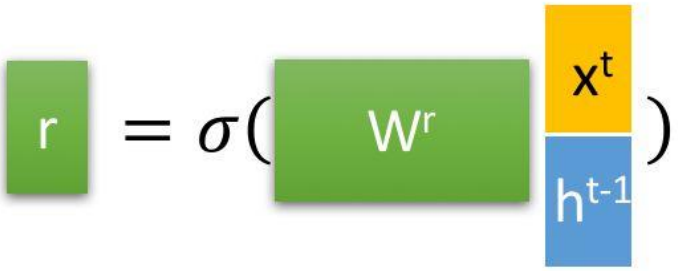
C.GRU :

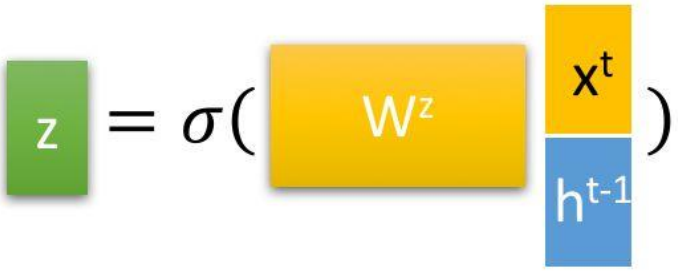
若 GAP 是 FCL 的改版，則 GRU(Gate Recurrent Unit)則是 RNN 的改版。GRU 是為了解決 RNN 長期記憶和反向傳播所遇到梯度爆炸的問題，跟 LSTM(Long-Short Term Memory)有點相像。

以下是 GRU 的數學原理圖:

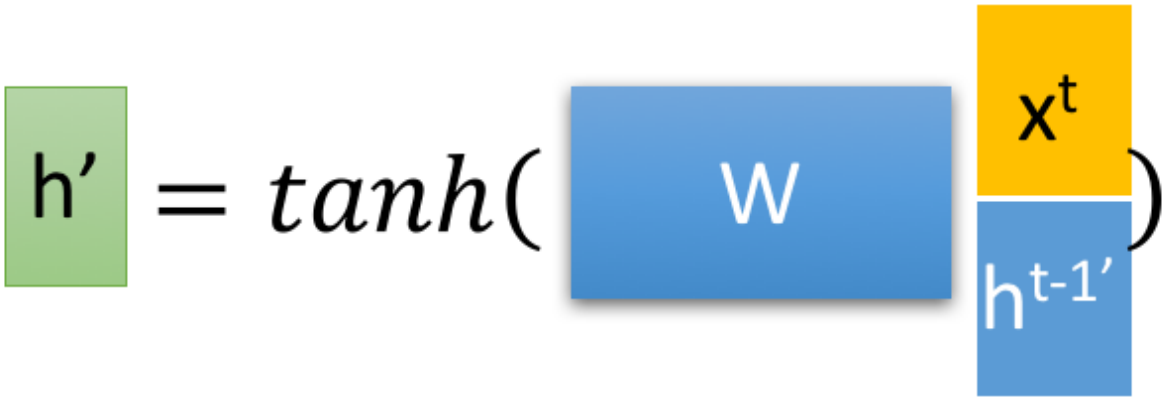


可發現他的運算較 LSTM 簡易且少量，但能力卻不會下降太多。
以下講解 GRU 運作的方式，一開始從收到從上個狀態 h(t-1)和目前節點輸入 x(t)來獲取兩個門控狀態，分別是 r 控制 reset gate 和 z 控制 update gate。(激活函數為:sigmoid)





取得門控信號後，首先透過重置門控獲得重置後的資料 h(t-1)'=h(t-1)⊙r，在將 h(t-1)'與輸入 x(t)進行拼接，然後放入 tanh 函數中激活投射至-1~1 範圍內，得到下式。



此 h'主要有目前的節點輸入 x(t)。且像是記住了當前狀態的方式，將 h'添加到當前的隱藏狀態。
最後是更新記憶的階段，方程式:h(t)=(1-z) ⊙h(t-1)+z⊙h'
(z 為 0~1 的門控訊號，若接近 1 則代表記下來的東西越多；反之，忘記的越多。好處是我們只需要一個變數便可以控制記憶和忘記，比 LSTM 少)
(1-z) ⊙h(t-1):表示對上個狀態的遺忘，忘記 h(t-1)中不太重要的資料。
z⊙h':表示對目前的記憶，可以看成對維度中某些資訊的選擇。

上兩式合在一起則為，遺忘上個狀態 h(t-1)某些資訊，加入當前的某些資訊。

D. ENSEMBLE:

在模型越加越深，不斷地以計算量或不同的模型去嘗試，但往往每個模型都有其擅長的方向，並沒有完美的模型，於是整體學習演算法(ensemble learning algorithms)的想法就可以想成大家來分工，每個不同的模型可以確保大家不會都錯，難免會有些模型可以辨識較易錯的圖片。Ensemble 不太像是找到一個很棒的模型去預測數據，而是透過多個模型完成一個像是投票或討論出它們覺得最適合的結果，那個就是預測的結果。

過程中我們選擇使用 voting，透過不同參數訓練出的 CNN+GRU 每個一票，在訓練資料集中表現最佳的那個組合。算是最為簡單也明瞭的方式，也較適合作為初學者嘗試的切入點，也有看到別組使用 bagging 或 boosting 等技巧。

至於為何是多數決，是我們以我們訓練好的模型來看，這是最佳的策略，其他組員也有提到選出所有模型中機率最高的，這是相當有趣的嘗試。

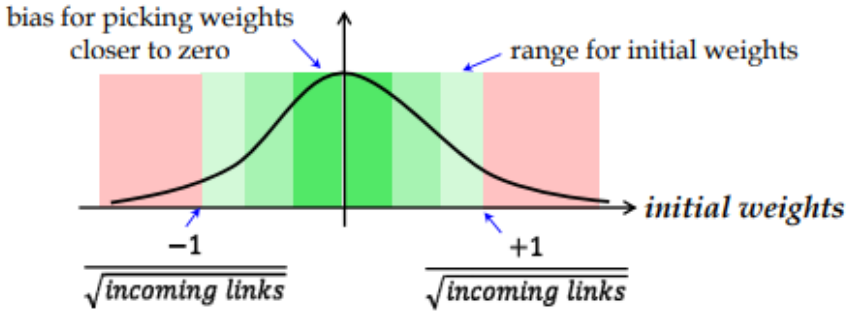
E.he_normal:

在訓練模型時最重要的是調整參數，但參數的初始化也很重要，若單純的隨機分布在-1 至 1 中，可能直接在激活函數時飽和，使倒傳遞修正回去時會有問題。下圖是中央大學電

機系教授李進福老師深度神經網路系統設計第四張的其中一頁講義:

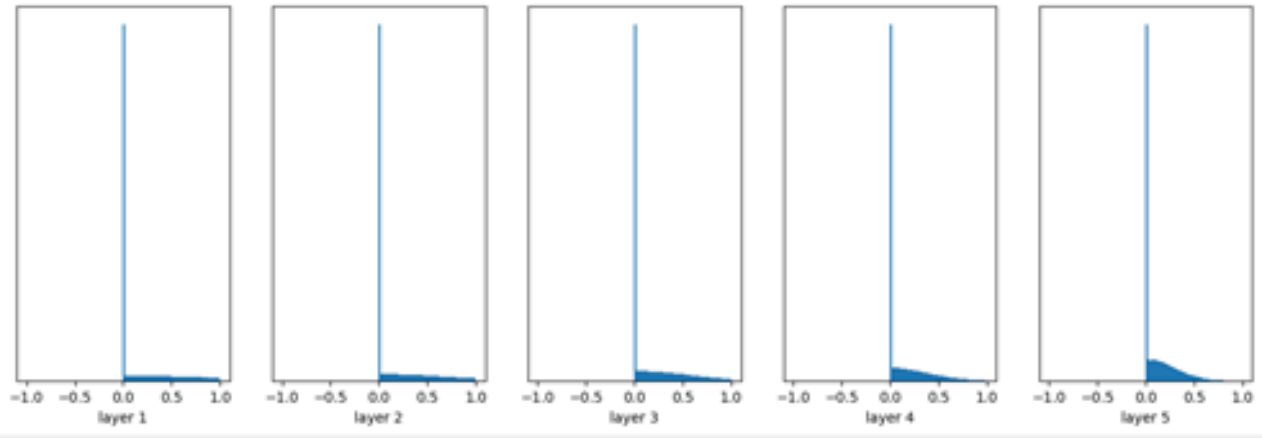
Preparing Data

- Random initial weights
  - Similarly, we should avoid large initial weight because they cause large signals into an activation function, leading to saturating a neural network and the reduced ability to learn better weights
  - We could choose initial weights randomly and uniformly from a range -1.0 to +1.0
  - Mathematicians and computer scientists have done the maths to work out a rule of thumb for setting the random initial weights given specific shapes of networks and with specific activation function. The rule of thumb is that the weights are initialized randomly sampling from a range that is roughly the inverse of the square root of the number of links into a node



發現 weight 的起始很重要，網路上有許多不同種的方式，讓每個人可以透過他們所想的去設定。我們選擇的是 he_normal 的方式，是由何鎧明先生所提出的。我們使用時主要依靠 python 中物件導向的封裝特性，所以沒有能夠講解其函數在於統計上的意義，所以放上他人實驗出來的結果，以表達為何我們選取這個。

下圖是使用 Xavier initialization，神經網路 activation function 採用 ReLU 輸出:



每層 Hidden 輸出的直方圖。

Weight 是由均勻分布隨機生成(Xavier initialization)。

input mean 0.00065 and std 0.99949

layer 1 mean 0.40986 and std 0.59949

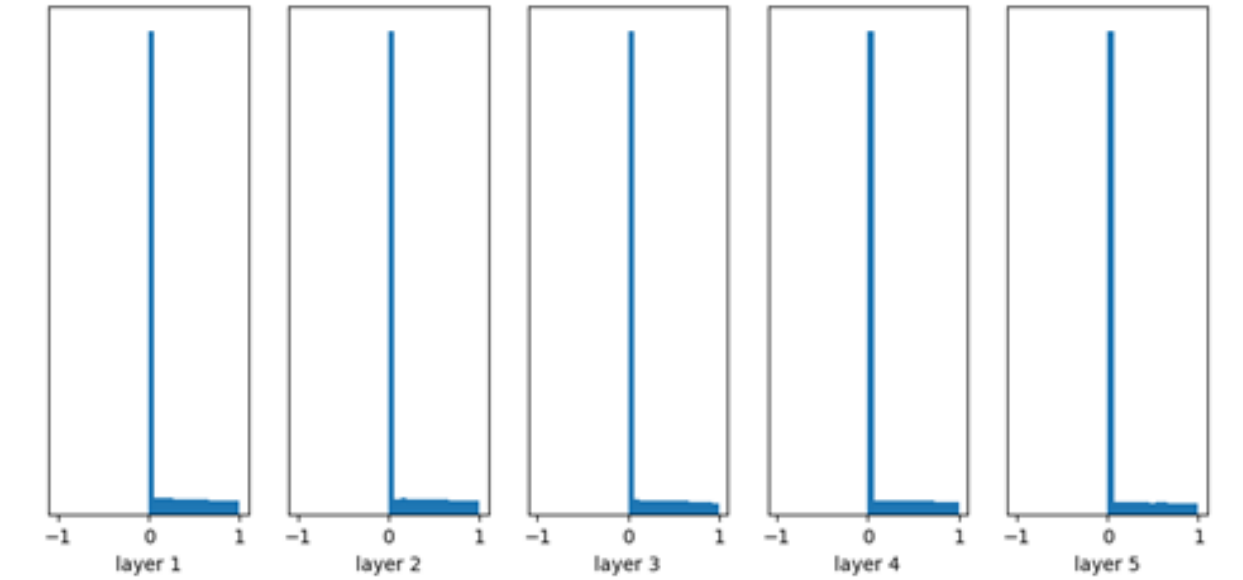
layer 2 mean 0.28339 and std 0.42536

layer 3 mean 0.22020 and std 0.31701

layer 4 mean 0.15992 and std 0.23894

layer 5 mean 0.13443 and std 0.18587

下圖是改用 He initialization，神經網路 activation function 採用 ReLU 輸出:



每層 Hidden 輸出的直方圖。

Weight 是由常態分佈隨機生成(He initialization)。

input mean 0.00065 and std 0.99949

layer 1 mean 0.59251 and std 0.86758

layer 2 mean 0.63703 and std 0.91839

layer 3 mean 0.64056 and std 0.95845

layer 4 mean 0.72482 and std 1.06023

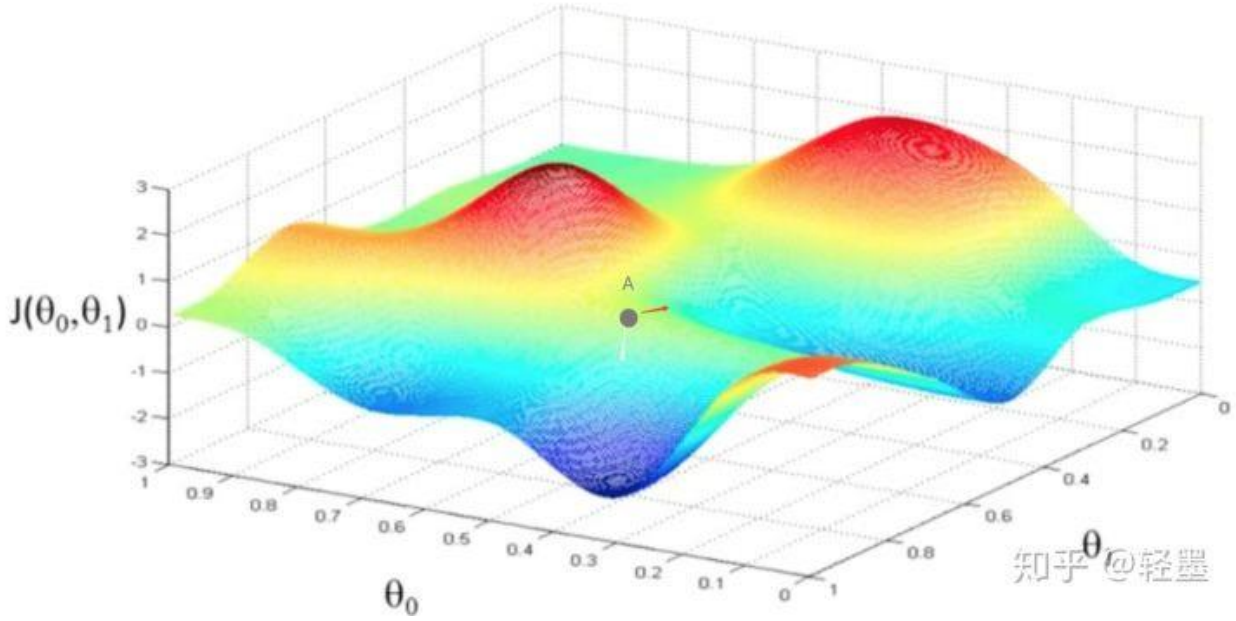
layer 5 mean 0.80317 and std 1.19583

F.SHUFFLE:

在第一次報告中，有同學提出 shuffle，我們覺得這是個可以討論的問題，以下是在我們上網查完後得出的結論。

大方向是，今天我們透過修正 weight 去找尋 loss function 的最低點，但要是每次 batch 進來都一樣，可能在那個空間上只會往固定某個方向找，但那邊可能只有 local minimum，

且在這個模型中的 learning rate 無法到達 global minimum(尤其在 batch size 比較小時)，所以透過每次 shuffle 去做訓練，能確保有不同方向的嘗試。
以下的圖能良好的表達在 loss function 找 minimum:



G.regulation:

像是在訓練時，可能會擬合出太高次方，出現 overfitting，於是我們可以透過 L1/L2 使 Loss Function 更平滑，比較能抗雜訊干擾。
其算式如下方表示:

$$y = b + \sum w_i x_i$$

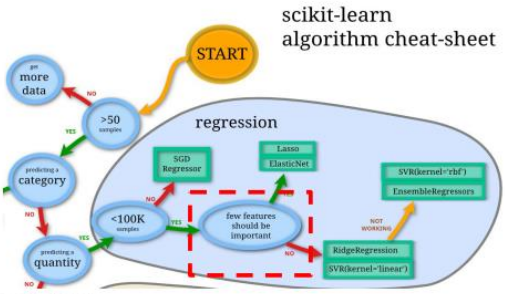
The functions with smaller w_i are better

$$L = \sum_n \left(\hat{y}^n - \left(b + \sum w_i x_i \right) \right)^2 \quad \boxed{+\lambda \sum (w_i)^2}$$

常見的 regulation 有分為 L1 和 L2，兩者明顯差異為，
如下圖:

- Ridge Regression (L2)

$$\text{Cost} = \text{Prediction error} + \alpha \sum (\text{weights})^2$$



- Lasso Regression (L1)
  --- Least Absolute Shrinkage and Selection Operator

$$\text{Cost} = \text{Prediction error} + \alpha \sum |\text{weights}|$$

前者為絕對值；後者為平方。我們選取使用 L2。

三、模型

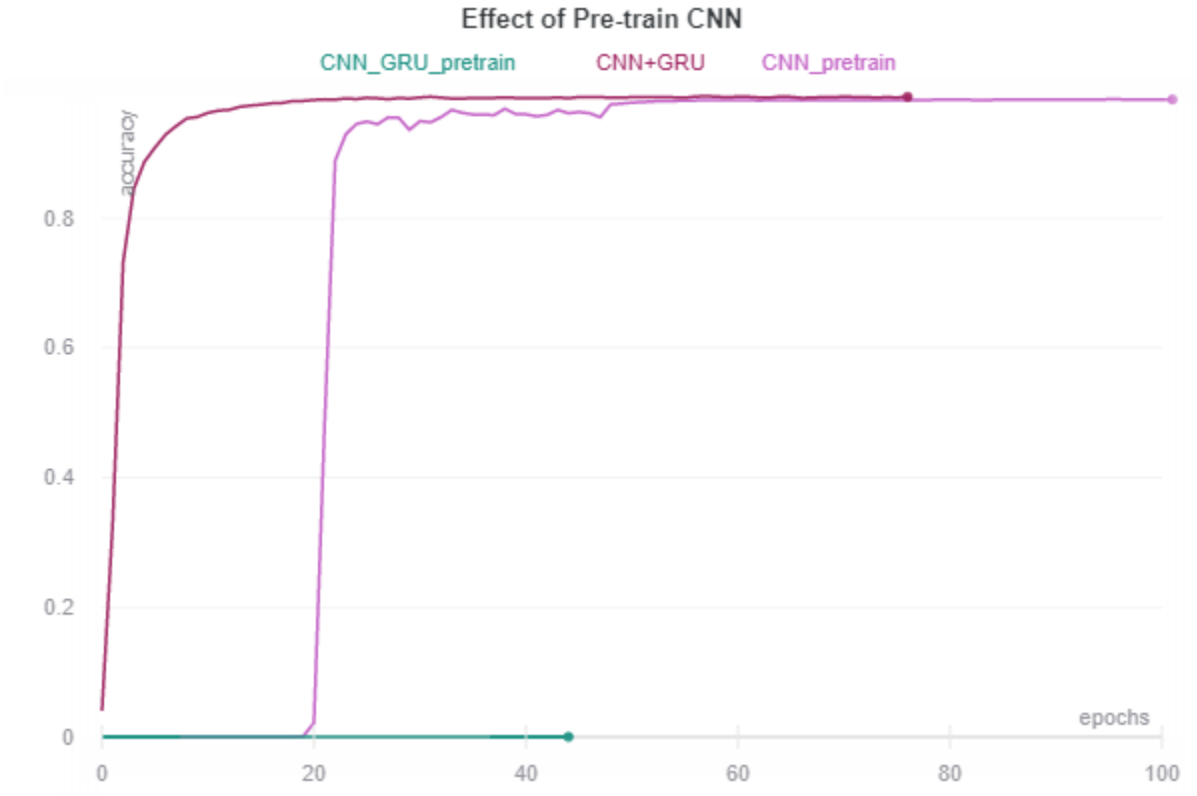
以 data 01 中使用的模型舉例，在上面註解不同部分所做之事:

```
#CNN+GRU
def build_model(config=config_default):
    #setup weight initializer and regularizer
    initializer = tf.keras.initializers.he_normal( seed = 3)
    #configuration can be set in config dictionary
    alpha = config["weight_regu"]  # weight decay coefficient
    regularizer = tf.keras.regularizers.l2(alpha)
    dropout_rate =config["dropout"]

    #multilayer Convolutional neural network
    inn = Input((60, 200, 3))
    out = inn
    out = Conv2D(filters=64, kernel_size=(3, 3), padding='valid',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    out = Conv2D(filters=64, kernel_size=(3, 3), padding='same',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    out = MaxPooling2D(pool_size=(2, 2))(out)
    out = Dropout(dropout_rate)(out)
```

```
    out = Conv2D(filters=128, kernel_size=(3, 3), padding='valid',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    out = Conv2D(filters=128, kernel_size=(3, 3), padding='same',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    out = MaxPooling2D(pool_size=(2, 2))(out)
    out = Dropout(dropout_rate)(out)

    out = Conv2D(filters=256, kernel_size=(3, 3), padding='valid',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    out = Conv2D(filters=256, kernel_size=(3, 3), padding='same',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    out = MaxPooling2D(pool_size=(2, 2))(out)
    out = Dropout(dropout_rate)(out)

    out = Conv2D(filters=512, kernel_size=(3, 3), padding='valid',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    out = Conv2D(filters=512, kernel_size=(3, 3), padding='same',
kernel_initializer = initializer,kernel_regularizer = regularizer)(out)
    out = BatchNormalization()(out)
    out = layers.Activation('relu')(out)
    #using global average pooling for reducing dimension
    out = layers.GlobalAveragePooling2D()(out)
    out = Dropout(dropout_rate)(out)
    #using repeat vector as six time step input to GRU
    out = layers.RepeatVector(6)(out)
    out = layers.GRU(128,return_sequences=True)(out)
    sep = Dropout(dropout_rate)(out)
    #Lambda layer for separating each time step output
    sep0 = layers.Lambda(lambda x: x[:, 0, :])(sep)
```

```
    sep1 = layers.Lambda(lambda x: x[:, 1, :])(sep)
    sep2 = layers.Lambda(lambda x: x[:, 2, :])(sep)
    sep3 = layers.Lambda(lambda x: x[:, 3, :])(sep)
    sep4 = layers.Lambda(lambda x: x[:, 4, :])(sep)
    sep5 = layers.Lambda(lambda x: x[:, 5, :])(sep)
    #Dense layer before output for classification
    dig1 = layers.Dense(36, name='digit1', activation='softmax',
kernel_initializer = initializer)(sep0)
    dig2 = layers.Dense(36, name='digit2', activation='softmax',
kernel_initializer = initializer)(sep1)
    dig3 = layers.Dense(36, name='digit3', activation='softmax',
kernel_initializer = initializer)(sep2)
    dig4 = layers.Dense(36, name='digit4', activation='softmax',
kernel_initializer = initializer)(sep3)
    dig5 = layers.Dense(36, name='digit5', activation='softmax',
kernel_initializer = initializer)(sep4)
    dig6 = layers.Dense(36, name='digit6', activation='softmax',
kernel_initializer = initializer)(sep5)
    model = tf.keras.models.Model(inputs=inn, outputs=[dig1,
dig2,dig3,dig4,dig5,dig6],name="test")
    model.compile(
            loss=[
                tf.keras.losses.CategoricalCrossentropy(),
                tf.keras.losses.CategoricalCrossentropy(),
                tf.keras.losses.CategoricalCrossentropy(),
                tf.keras.losses.CategoricalCrossentropy(),
                tf.keras.losses.CategoricalCrossentropy(),
                tf.keras.losses.CategoricalCrossentropy(),
            ],

optimizer=tf.keras.optimizers.Adam(learning_rate=myconfig["LR"],
beta_1=0.9),
            metrics=['accuracy'])
    return model
```

四、模型訓練情況:

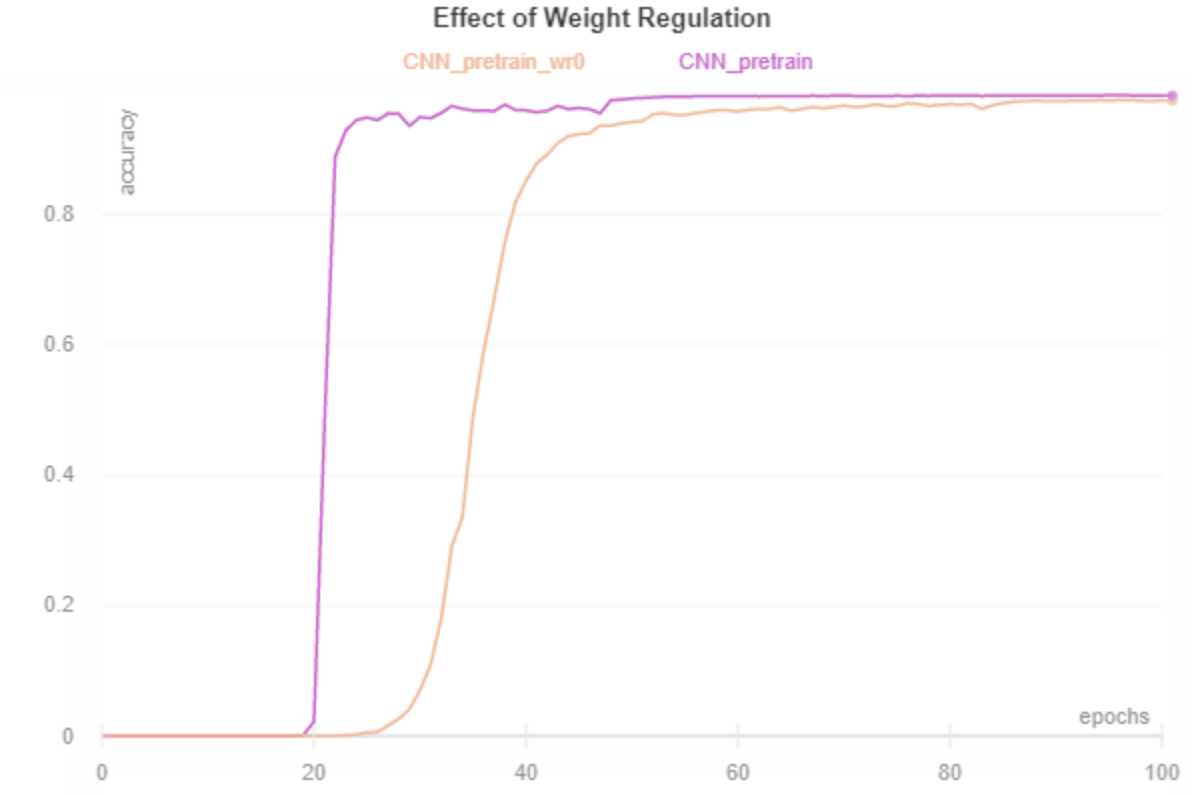


先單獨使用 CNN 模型對捲基層做 pretraining，如果使用 CNN+GRU 從頭開始訓練，收斂時間會太久，在此 y 軸 accuracy 為全對率。

五、討論:

在嘗試許多模型以及調整參數的過程中，我們認為在 CAPTCHA 辨識方面，對於準確率在 97%以上影響最多的調整是加上 weight regularization、flatten 改成 global average pooling 以及加上 GRU。

在 layer 中加上 weight regularization，藉由在 loss fuction 加上一項，會鼓勵模型參數維持在較小的區域，可以避免 gradient explotion，好處是可以讓模型權重不要走向極端值，壞處是如果某些分類需要極端的權重才分辨得出來，會影響準確率。在我們嘗試的過程中，加上 weight regularization 對於此工作的影響主要是正面的，不僅增加準確度，模型收斂速度也會更快。



由 flatten 改成 GAP 可以減少連接 GRU 或 Dense layer 的參數，節省的計算和記憶體空間可以用來增加 CNN 的層數。如果比較直觀的看 GAP 是將 CNN 輸出的 feature map 轉換成一個代表某種資訊的維度，flatten 則還是讓模型下一層去看 CNN 輸出的 feature map，並希望模型可以自己找出對於準確度有用的維度。

由於 CAPTCHA 辨識跟順序有關，某種程度可以想像成是圖形轉語句的問題，像是我們看到一張 CAPTCHA 時很難一瞬間就解析出每個字母，通常會由左而右、在連續時間中產生對於字母的解釋。例如 t=0 時會產生對於第一個字母的解釋、t=1 時會產生對於第二個字母的解釋等等，而且 t=1 的解釋會和 t=0 的解釋有空間關系。根據這樣的想法，我們在每一個 time step 對 GRU 輸入使用 CNN filtered 的圖片，而 GRU 每個 time step 的輸出就

是對於字母的解釋，理想上我們希望 GRU layer 能夠在 training 時了解每個輸出之間的時空關係。

關於訓練的過程，由於超參數的調整後果沒有辦法很有效的被預測，我們搜尋超參數的方法是像地毯式搜索，滿耗費時間和電力的，這一點是我們沒有辦法解決的。雖然結果還有更多進步的空間，尤其是資料二的部分，不過我們在這次的期末專題中學到非常多關於深度神經網路學習的知識，也期望在未來為這個領域做出貢獻。

六、參考:

1.GAP 全局池化替代全连接层的分析:
https://blog.csdn.net/Touch_Dream/article/details/79775786 -
2.人人都能看懂的 GRU
https://zhuanlan.zhihu.com/p/32481747
3.数据集 shuffle 的重要性
https://zhuanlan.zhihu.com/p/57108650
4.中央大學電機系李進福教授第四章投影片
5.深度學習: Weight initialization 和 Batch Normalization
https://medium.com/@chih.sheng.huang821/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92-weight-initialization%E5%92%8Cbatch-normalization-f264c4be37f5
6.Learning Model : L1/L2 regularization 差異
https://medium.com/ai%E5%8F%8D%E6%96%97%E5%9F%8E/learning-model-l1-l2-regularization%E5%B7%AE%E7%95%B0-8d7fc089b35c