National Tsing Hua University
Fall 2023 11210IPT 553000
Deep Learning in Biomedical Optical Imaging
Homework 3

GAO WEL LUN[1]

[1]*Institute of Photonics Technologies, National Tsing Hua University, Hsinchu 30013, Taiwan*

Student ID:110066511

1. Task A

過度擬合(Overfitting)是一種不理想的模型訓練行為[1]。當學習模型在訓練中呈現良好表現，在訓練資料中提供精準的準確度，但在新數據的資料中提供不準確預測，會呈現糟糕的表現，此模型即為過度擬合模型。 過度擬合模型無法對所有類型的新資料中表現良好的預測結果。

過度擬合的發生，是由以下幾個原因發生而造成的:

1. 當訓練資料太少，且沒有足夠的資料範例，無法準確預測輸入資料值。
2. 模型複雜度較高，模型可能學習訓練資料中的雜訊。
3. 模型在單一訓練資料上訓練過長，導致模型可能適應雜訊的微小變化。

在原始的 Lab 4 的檔案中，使用了 ConvModel 進行 30 次迭代模型訓練，在第 15 次迭代後，訓練精準度維持在 100%，驗證精準度則維持在 97%，並沒後續變化，在測試的新數據資料進行預測，測試精準度只有 73.75，這意味著模型過度擬合的表現。

為了減少過度擬合的發生，以下是解決過度擬合的常見方法:

1. 增加訓練數據:更多的訓練樣本，使模型訓練能學習到更多有效的特徵。
2. 降低模型複雜度:在訓練數據較少時，過於複雜的模型，可能會採樣到雜訊進行學習，適當降低模型複雜度，能避免模型擬合到雜訊。
3. L1,L2 正規化:在損失函數添加權重值，藉由調整限制權重值，避免模型過於複雜，減少過度擬合發生機率。
4. 提早暫停:當監控模型時，當模型性能開始降低時停止訓練，能避免過度擬合。

本篇使用降低模型複雜度來減少過度擬合發生機率，而降低模型複雜度也就是減少模型中的網路層、神經元個數。這裡使用 nn.Dropout(0.5)這段程式碼，隨機關閉模型中 50%的神經元或通道，使模型過於依賴特定神經元，更好的學習未見過的數據。其修改程式碼如 Fig1.。

```
class ConvModel(nn.Module):
    def __init__(self):
        super().__init__()

        # 1 channel, and using 3x3 kernels for simplicity, 256*256
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding='same')
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2) # 128*128

        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 128*128
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 64*64

        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 64*64
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2) # 32*32



        # Adjust flattened dimensions based on the output size of your last pooling layer
        flattened_dim = 32 * 32 * 32

        self.fc1 = nn.Linear(flattened_dim, 32)
        self.fc2 = nn.Linear(32, 1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        x = F.relu(self.conv2(x))
        x = self.pool2(x)

        x = F.relu(self.conv3(x))
        x = self.pool3(x)

        # Flatten the output for the fully connected layers
        x = x.reshape(x.size(0), -1)  # x.size(0) is the batch size

        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

```
class ConvModel(nn.Module):
    def __init__(self):
        super().__init__()

        # 1 channel, and using 3x3 kernels for simplicity, 256*256
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding='same')
        # self.bn1 = nn.BatchNorm2d(32)                                    # Batch Normalization
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2) # 128*128
        # self.drop1 = nn.Dropout(0.5)                                     # Dropout

        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 128*128
        # self.bn2 = nn.BatchNorm2d(32)                                    # Batch Normalization
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 64*64
        # self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 64*64
        # self.bn3 = nn.BatchNorm2d(32)                                    # Batch Normalization
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2) # 32*32
        # self.drop3 = nn.Dropout(0.5)

        # Adjust flattened dimensions based on the output size of your last pooling layer
        flattened_dim = 32 * 32 * 32

        self.fc1 = nn.Linear(flattened_dim, 32)
        self.fc2 = nn.Linear(32, 1)
        self.dropout = nn.Dropout(0.5)                                     # Dropout

    def forward(self, x):
        x = F.relu(self.conv1(x))
        # x = self.bn1(x)                                                  # Batch Normalization
        x = self.pool1(x)
        # x = self.drop1(x)                                                # dropout

        x = F.relu(self.conv2(x))
        # x = self.bn2(x)                                                  # Batch Normalization
        x = self.pool2(x)
        # x = self.drop2(x)                                                # dropout

        x = F.relu(self.conv3(x))
        # x = self.bn3(x)                                                  # Batch Normalization
        x = self.pool3(x)
        # x = self.drop3(x)                                                # dropout

        # Flatten the output for the fully connected layers
        x = x.reshape(x.size(0), -1)  # x.size(0) is the batch size

        x = F.relu(self.fc1(x))
        x = self.dropout(x)                                                # dropout
        return self.fc2(x)
```

Fig. 1. 左圖為 Lab 4ConvModel 原始程式碼，右圖為添加 nn.Dropout(0.5)的修改後程式碼。

從執行添修後改程式的結果觀察，在其他參數不變情況下，訓練精準度落在 97.3%，而驗證精準度則維持在 97.25%，兩者之間的變異數變小，且在測試的新數據資料進行預測中，測試精準度提升至 78.5。從結果來說，nn.Dropout(0.5)減少了模型的複雜度，減緩了模型的過度擬合，若之後想進一步優化，之後會改變 Dropout 參數或加入 L1、L2
Regularization，進一步提升模型效能。

2. Task B

從 Lab 4 的程式碼分別執行了 ANN 和 CNN 兩種不同的神經網路架構，觀察到 ANN 訓練時間快於 CNN，而訓練、驗證和測試精準度則是 CNN 優於 ANN。

而一般在處理圖像分類任務時，通常 CNN 會比 ANN 訓練速度更快，由於 CNN 的捲積層會捕捉數據局部的特徵值，在池化層中進一步縮小數據，並降低數據複雜度，保留重要特徵，再加上 CNN 神經網路元共享同一個權重參數，能大幅減少 GPU 的負擔，加快模型的訓練速度。然而本次觀察到 ANN 訓練時間快於 CNN，由於本次程式碼是 X 光胸腔照健康與否，任務類型相較簡單，以及 ANN 結構相較簡單，結構中的隱藏層和參數較少，訓練時的計算負擔較小，導致某些情況下 ANN 的訓練速度比 CNN 更快。

在圖像的任務分類中，CNN 的捲積層和池化層，從圖像中提取部分特徵圖，在減小特徵圖解析度的同時，並保留重要特徵，有助於減少過度擬合，也提升擷取特徵的多樣性，使得能夠在複雜圖形中，獲得有效的特徵值。因此在本次的程式執行結果下，證明了 CNN 對於圖像處理的部分比 ANN 更為出色。因此對於圖像分類、物體檢測或其他視覺任務上，CNN 通常會是首要選擇。

ANN 和 CNN 是屬於兩種不同神經網路架構，在不同的任務和應用場景，會具有不同工作性能和運行速度。ANN 為一個通用的神經網路結構，而 CNN 則專為處理圖像、

視覺、多維數據設計的神經網路結構。根據分類任務和數據型態來選擇適合神經網路結構。

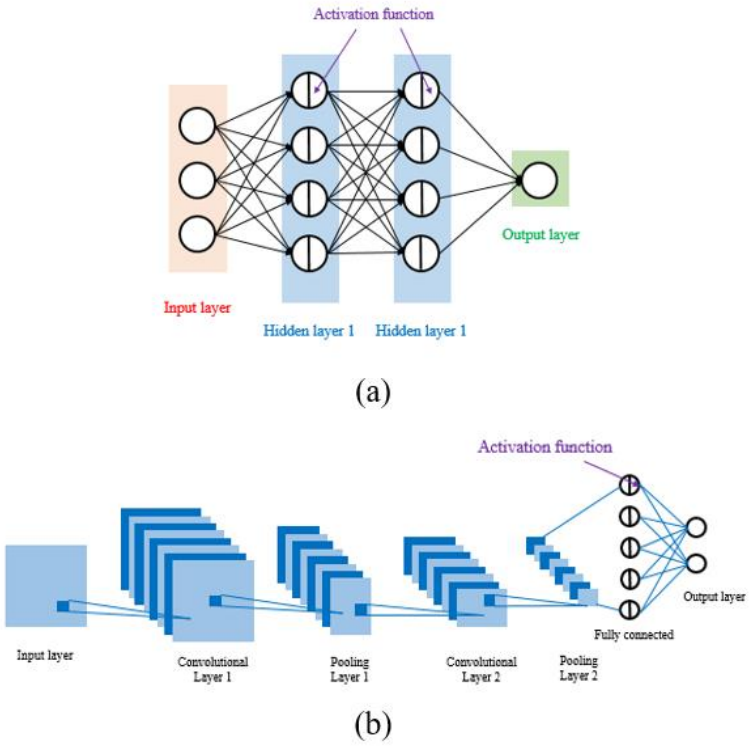


Fig. 2. (a) ANN(Artificial Neural Networks) 神經網路架構流程圖 ，(b) CCN(Convolution Neural Networks) 神經網路架構流程圖。

ANN 和 CNN 兩者屬於深度學習的神經網路模型，根據 Fig 2.神經網路流程圖進行簡述:

1. ANN(Artificial Neural Networks) 神經網路架構流程:

- 輸入層(Input layer) : 輸入數據的特徵向量。
- 隱藏層(Hidden layer) : 用於學習和提取特徵的中間層。
- 激活函數(Activation function) : 產生最終輸出或預測。
- 輸出層(Output layer) : 引入非線性函數，使模型簡單化。

2. CCN(Convolution Neural Networks) 神經網路架構流程:

- 輸入層(Input layer) : 輸入數據的特徵向量
- 捲積層(Convolutional layer) : 使用捲積核以檢測圖像中的特徵，來產生特徵圖。
- 池化層(Pooling layer) : 降低特徵圖的空間解析度，減少電腦計算時間，並同時保留重要特徵。
- 全連接層(Fully connected layer) : 一般位於神經網路圖中最後幾層，可能包含一個或多個全連接層，將特徵投射至輸出層之前，用於產生最終預測。
- 激活函數(Activation function) : 引入非線性函數，使模型簡單化，一般會在捲積層和全連接層後。
- 輸出層(Output layer) : 產生最終輸出或預測。

ANN 為一個通用的神經網路結構，能處理各種類型的分類任務;而 CNN 則專為處理圖像、視覺、多維數據設計的神經網路結構，由於具有捲積層和池化層，能更好的從數據中提取特徵。總結來說，根據任務類型，選擇適當的神經網路結構，才能更有效的訓練模型，以實現最佳的模型性能。

3. Task C

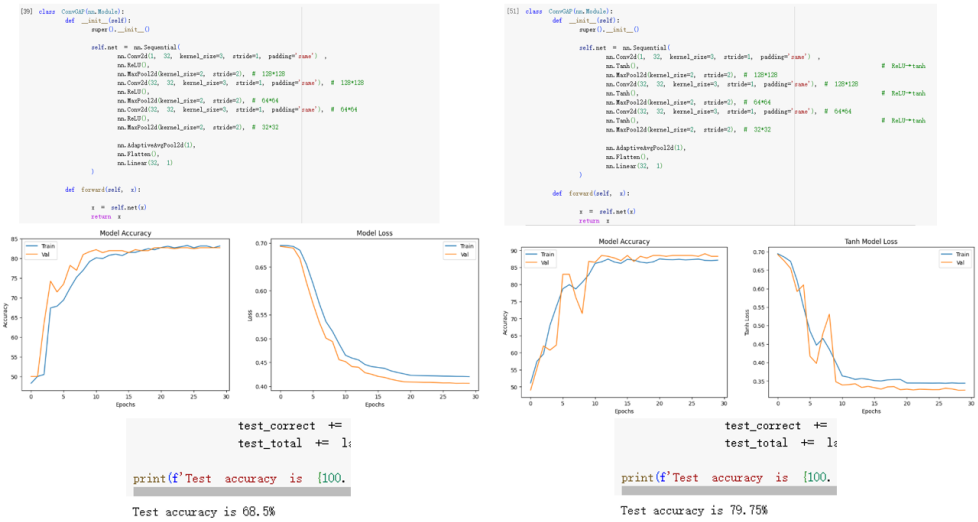
GAP (Global Average Pooling)是屬於 CNN 神經網路的一種，其作用如下:

1. 維度減小:GAP 會把每張特徵圖取平均，將每張特徵值轉化為單一值。
2. 位置不變性:GAP 不考慮特徵在數據中的空間位置，意指在提取特徵時，不受特徵在圖形數據上的確切位置影響，只需要關注重要特徵存在與否。
3. 減少過度擬合:通過將特徵圖所小，有助於降低模型的參數量，減少過度擬合的風險。

傳統的 CNN 神經網路結構中，通常需要手動計算每一層特徵圖尺寸，確保全連接層的特徵維度與全連接層輸入匹配。而 GAP 哲是消除全連接層匹配的過程，由於 GAP 將特徵圖轉為單一數值，不需要手動計算每一層特徵尺寸。因此使用 GAP 神經網路架構，不用擔心特徵尺寸不匹配的問題，因為 GAP 能生成統一尺寸的特徵，使得模型設計更加方便靈活。

本篇使用 Lab 4ConvGAP 原始程式碼，由於在經過 30 次迭代後，模型的性能有下降的跡象，因此對於這部進行優化。這裡修改的部分，是將原本的 ReLU 函數換成了 Tanh 函數，其餘參數都沒有做更動，其程式修改和驗證結果如 Fig.3 所示。從 Fig.3 精準度和損失圖表能觀察到，使用 ReLU 函數的訓練和驗證精準度落在 84% 和 84.5%，而使用 Tanh 函數的訓練精準度落在 87.12% 和 88.25%，兩者的變異數都在 1%以內，然而後者的偏差比前者偏差低，再加上使用訓練好的模型測試在新數據上， ReLU 函數的測試精準度為 68.5%，而 Tanh 函數的精準度為 79.75%，從結果上使用 Tanh 函數會比使用 ReLU 函數性能好。

一般深度學習中，在 CNN 神經網路模型中，通常使用 ReLU 作為激活函數，在訓練擁有良好的表現，具有更快的收斂，會比 Tanh 函數更容易獲得更良好的性能。而在某些情況下，在圖像的分類中，Tanh 函數會比 ReLU 函數在 ConvGAP 中表現得更好。通常會因為數據的範圍和輸出值。數據分布或神經元死亡所導致。



(a) (b)

Fig. 3. (a) Lab 4ConvGAP 原始程式碼 ，使用 ReLU 作為激活函數，(b)為修改後的程式碼，將 ReLU 函數替換成 Tanh 函數，其下圖為訓練和驗證精準度和損失，以及測試新數據的結果。使用 ReLU 函數的測試精準度為 68.5%，而使用 Tanh 函數的精準度為 79.75%。

References

1. https://aws.amazon.com/tw/what-is/overfitting/