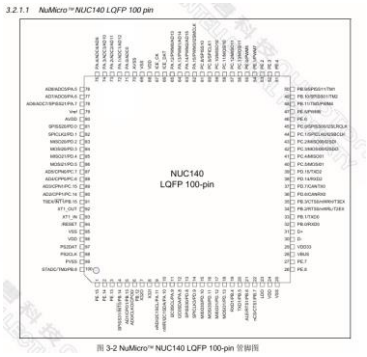
微處理機系統與介面技術 LAB 8

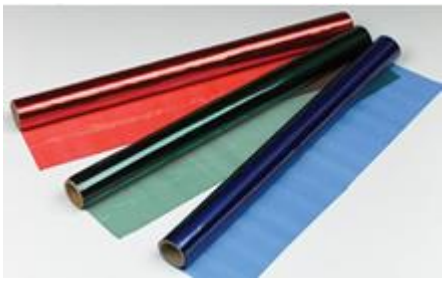
系所：電機 學號 :612415013 姓名：蕭宥羽

<實驗器材>

NUC 140 V2.0 開發板



透明紙、光敏電阻、LED 光源



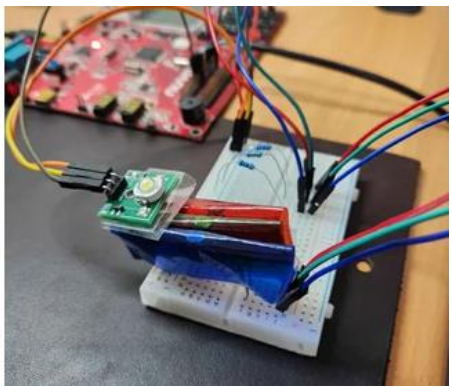
<實驗過程與方法>

✓ 實驗要求：

- Basic : Use machine learning to predict 4 outputs -red, blue, green and ambient light
- Bonus : Use machine learning to predict 7 outputs -red, blue, green, magenta, orange, yellow and ambient light

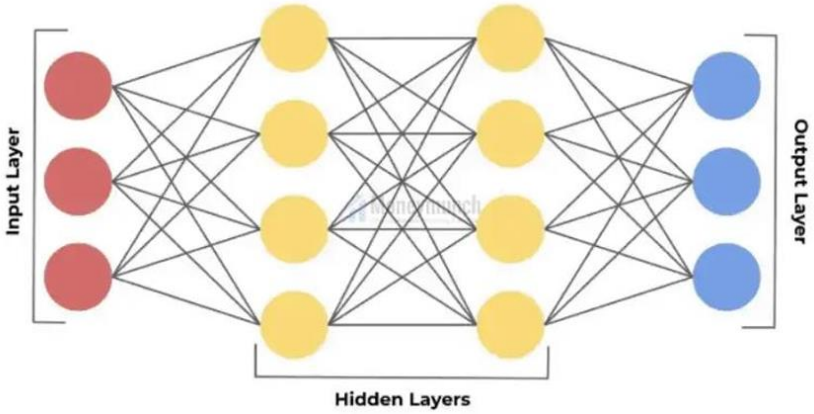
✓ 實驗環境設置：

- Cover the photoresistor with cellophane
- Align three photoresistors side by side
- Attach the LED module above the three photoresistors
- Connect the remaining circuitry (VCC, GND, ADC, Control Pin)

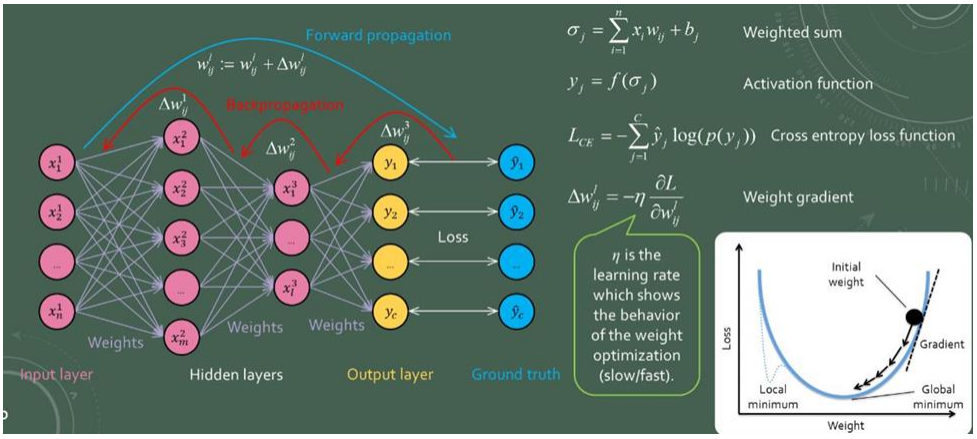


✓ **什麼是機器學習：**

機器學習 (Machine Learning) 是一種能夠自動學習並從資料中改進表現的技術。像是這次 lab 中所使用的神經網路模型，透過模擬人類大腦的結構來進行學習。這個模型由輸入層、隱藏層和輸出層組成，輸入層接收資料特徵，隱藏層則負責提取資料中的重要模式與特徵，最後輸出層產生模型的預測結果。藉由不斷學習的過程，模型會根據實際結果與預測值的差異進行優化，透過調整權重和偏差來逐漸減少誤差。隨著訓練次數的增加，模型能更準確地捕捉資料的特徵與規律，達到穩定且可靠的預測效果。



✓ **類神經網路(ANN)的架構：**



ANN 一個模仿生物神經的系統，是一種讓電腦具備學習與推理判斷能力的演算法，

ANN 的基本架構

❶ **節點 (Node)**：人工神經網路中的基本單位，每個節點模仿生物神經元的功能。節點會接收輸入訊號，經過加權計算和激活函數處理後，輸出結果到下一層節點。節點的輸出取決於**輸入值**、**權重**和**激活函數**的類型。

❷ **輸入 (Input)**：輸入是外部資料進入神經網路的起點，例如圖片的像素值、感測器的讀數或其他特徵向量。每個輸入值會被送入輸入層的節點，作為神經網路學習和預測的基礎。

❸ **權重 (Weight)**：權重是節點之間連結的重要參數，表示輸入對下一節點的重要性。每個權重在訓練過程中不斷被調整，以最小化預測誤差。權重的大小直接影響網路對輸入特徵的關注程度。

❹ **連結 (Connection)**：連結是網路中節點與節點之間的通道，負責將輸入、權重和偏差傳遞給下一節點。每條連結都有對應的權重值，並用於計算輸入數值在經過該連結後的輸出。

❺ **隱藏層 (Hidden Layer)**：隱藏層位於輸入層和輸出層之間，是網路的核心部分。隱藏層負責處理和提取輸入資料中的非線性特徵。隱藏層的節點會通過激活函數進一步增強網路表達能力，使其能夠處理複雜的資料模式。

❻ **輸出層 (Output Layer)**：輸出層是網路的最後一層，其節點數量通常與預測的類別數或目標值數量一致。輸出層將隱藏層的處理結果轉換為模型的預測結果，例如分類標籤、回歸值或機率分佈。

ANN 的學習過程

ANN 的學習過程主要透過數據流動和權重優化來實現，分為以下幾個主要階段

❶ **輸入資料**：將訓練數據輸入到類神經網路中，作為學習的起點

❷ **前向傳播**：資料從輸入層逐層向前傳遞到輸出層，生成模型的預測結果。

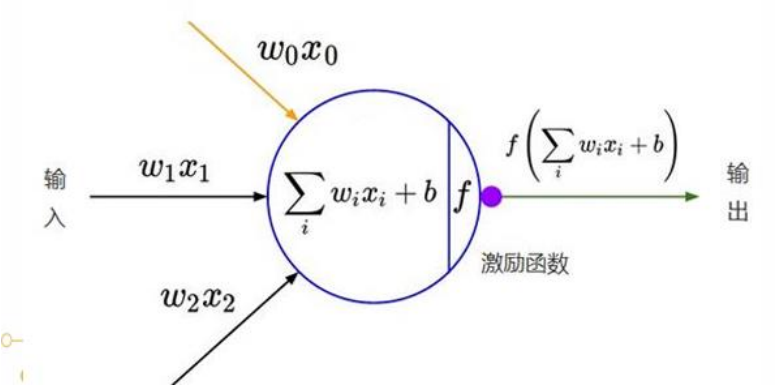
❸ **計算誤差**：根據輸出層的預測結果和實際值之間的差距，計算誤差大小

❹ **反向傳播**：將誤差從輸出層向回傳播至輸入層，計算每個權重對誤差的影響

❺ **更新權重**：根據計算的**誤差**和**梯度**，調整網路中的權重，以減少誤差並改進模型的預測能力

細節說明

❶ 前向傳播



- **Weighted Sum**

  每個節點接收來自前一層的輸入，這些輸入會被各自的權重(w) 乘以後相加，再加上偏差項(b)，這稱為加權求和，數學公式如下：

$$z = \sum_{i=1}^{n} w_i \cdot x_i + b$$

  - ✧ x_i ：前一層節點的輸出（或直接來自輸入層的數據）。
  - ✧ w_i ：每條連接對應的權重，表示輸入對該節點的影響程度。
  - ✧ b ：偏差項，用於調整節點的輸出，增加模型的靈活性。
    - ➢ 避免輸入為零時的模型僵化 ：如果所有輸入 xi=0，沒有偏差項的情況下，節點的輸出總是 z=0。但加上偏差項 b，即使輸入為零，節點仍能輸出非零的值。

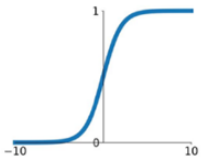
Activation Function

加權求和的結果 z 是一個線性組合，若直接輸出，最終的輸出還是一個線性組合，網路將失去處理非線性問題的能力。因此，神經網路在每個節點使用 activate function 將線性輸出轉換為非線性輸出。常見 activate function 如下:

- Sigmoid
  - 輸出範圍在 0 到 1 之間，常用於概率預測。
  - 缺點: 在極端值時，梯度趨近於 0，可能導致訓練緩慢。

Sigmoid

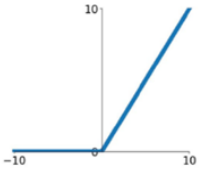
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



- ReLU
  - 當輸入為正時輸出為自身，否則輸出為 0，計算簡單且效率高。
  - 缺點: 可能出現「死亡 ReLU」現象，即某些神經元永遠輸出 0。

ReLU

$$\max(0, x)$$



- Softmax
  - 將多個輸出轉換為概率分佈，輸出值之和為 1，適用於多分類問題。

Softmax

$$S(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}}$$

❷ 反向傳播

反向傳播是神經網路訓練的核心過程，其目的是基於預測結果與實際目標之間的誤差來優化網路的參數（權重與偏差）。它主要分為三個步驟：**計算損失函數 (Loss Function)**、**優化 (Optimization Function)** 和 **更新權重 (Weight Update)**，其中**學習率 (Learning Rate)** 在更新權重的過程中扮演重要角色。

損失函數 (Loss Function)

Loss Function 用於量化神經網路的預測結果($\hat{y}$)與實際值(y)之間的誤差，作為模型學習的指標。

- Mean Square Error, MSE

  適用於回歸問題，計算預測值與實際值差異的平方平均。

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Mean Absolute Error, MAE

  誤差取絕對值後的平均值，更強調偏差的幅度。

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

✧ Cross Entropy

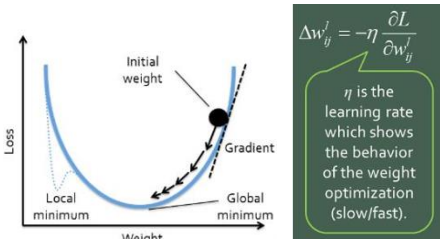
適用於分類問題，計算預測的概率分佈與實際標籤之間的差異。

$$H = -\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c} log\ (p_{i,c})$$

其中 $p_{i,c}$ 是模型對第 c 類的預測概率，$y_{i,c}$ 是實際標籤。

優化函數 (Optimization Function)

Optimization Function 負責通過損失函數的梯度來更新網路的參數，以最小化損失值。常見的優化方法包括：



✧ 梯度下降法 (Gradient Descent, GD) - conceptual

如上圖所示，Gradient Descent 會逐步沿著損失函數的梯度下降方向更新權重，直到達到局部最小值。數學式如下:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial L}{\partial w_{ij}}$$

- Δw_{ij}：表示權重 w_{ij} 的更新量（每次更新的改變值）。
- η：學習率，決定更新步伐的大小。
- $\frac{\partial L}{\partial w_{ij}}$：損失函數 L 對權重 w_{ij} 的梯度。

✧ 隨機梯度下降法 (Stochastic Gradient Descent, SGD)

SGD 是梯度下降的一種變體，它通過隨機選取一個訓練樣本來計算損失函數的梯度，並根據該梯度更新參數（權重）。這種方法避免了使用整個數據集計算梯度的高成本，從而加快了更新速度。然而，由於每次僅基於一個樣本，更新方向可能不穩定，會出現震盪的情況。

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

✧ Momentum

Momentum 是「運動量」的意思，此優化器為模擬物理動量的概念，在同方向的維度上學習速度會變快，方向改變的時候學習速度會變慢。

$$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$$

$$W \leftarrow W + V_t$$

此方法引入了一個 Vt 的參數，可視為「方向速度」，它與上一次的更新相關。當上一次梯度方向與本次相同時，|Vt| 會增大，代表學習速度變快；若方向不同，|Vt|會減小，更新幅度也會降低。動量因子 β（通常設定為 0.9）類似於空氣阻力或摩擦力，控制動量的累積與衰減。

✧ AdaGrad

AdaGrad 是一種自適應學習率的優化方法，它為每個參數計算不同的學習率，根據歷史梯度信息調整更新幅度。對於 Optimizer 來說，learning rate 相當的重要，太小會花費太多時間學習，太大有可能會造成發散

$$W \leftarrow W - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L}{\partial W}$$

$$n = \sum_{r=1}^{t} (\frac{\partial L_r}{\partial W_r})^2$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{\sum_{r=1}^{t} (\frac{\partial L_r}{\partial W_r})^2 + \epsilon}} \frac{\partial L}{\partial W}$$

在 AdaGrad 中，η 乘上 $1/\sqrt{(n+\epsilon)}$ 再做參數更新，出現了一個 n 的參數，n 為前面所有梯度值的平方和，利用前面學習的梯度值平方和來調整 learning rate，ϵ 為平滑值，加上 ϵ 的原因是為了不讓分母為 0，ϵ 一般值為 1e-8

✧ Adam 優化器

Adam 結合了 Momentum 和 AdaGrad 的優點

1. 使用 Momentum 追蹤一階動量（梯度均值），加速收斂。
2. 使用 AdaGrad 追蹤二階動量（梯度平方均值），自適應調整學習率。

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) \frac{\partial L_t}{\partial W_t} \qquad \hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$v_t = \beta_1 v_{t-1} + (1-\beta_2) (\frac{\partial L_t}{\partial W_t})^2 \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

像Momentum一樣保持了過去梯度的指數衰減平均值，像Adam一樣存了過去梯度的平方衰減平均值　　對mt跟vt做偏離校正

$$W \leftarrow W - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Momentum 提供了方向穩定性，AdaGrad 提供了自適應學習率，而 Adam 結合二者，能夠在多數情況下提供高效和穩定的優化效果。

- 學習率 (Learning Rate)

決定了梯度下降時每次更新權重的步伐大小。若學習率過大，模型可能跳過最佳解或不穩定；若學習率過小，訓練會變得緩慢甚至停滯。選擇合適的學習率能讓模型穩定且快速地收斂到最佳解。

資料處理

會將資料分為 **training data** 以及 **testing data**　(正常會分 8:2 or 7:3)

- **Training Data**：用於訓練模型，讓模型學習數據特徵與輸出之間的關係。
- **Testing Data**：用於評估模型的性能，確保模型能在未見過的數據上進行準確的預測。

通常會做 **Normalization / Standardization (正規化與標準化)**的動作

Normalization

將數據縮放到 [0, 1] 或 [-1, 1] 的範圍內，使每個特徵的數值保持在統一的比例範圍內。

- ✧ 使用場景 :1. 特徵值範圍分布相差很大 2. 模型對範圍敏感
- ✧ 如果不做會發生什麼事情 !
  1. 特徵值較大的數據主導了模型，其他特徵可能被忽略。
  2. 斂速度慢: 對於基於梯度下降的模型，數值差異大會導致梯度更新幅度不一致 ，使模型訓練變得不穩定或收斂過慢。

$$x_{norm}^{(i)} = \frac{x^{(i)} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}}$$

Standardization

將數據調整為均值為 0、標準差為 1。數據分佈呈標準正態分佈，數據點相對於均值的位置（z-score）可以更清晰地比較。

- ✧ 使用場景 :
  1. 數據分佈無上下界
  2. 對梯度敏感的模型
     使用基於梯度優化的模型（如線性回歸、邏輯回歸、神經網絡），標準化使得每個特徵的梯度更新幅度更均勻，有助於加速收斂。
- ✧ 如果不做會發生什麼事情 !
  1. 梯度更新幅度不一致：
     特徵值的範圍差異會導致梯度方向失衡，使模型在優化過程中震盪或過於緩慢。
  2. 模型訓練不穩定 :
     特別是對於需要多層運算的神經網絡，沒有標準化的輸入可能導致激活函數進入飽和區域（梯度消失）。
  3. 影響模型泛化能力：
     特徵值尺度差異大會增加模型的偏倚，降低其在測試數據上的表現。

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

<Mian function code>

這邊我會說明程式的整體流程、原理以及實作的細節，最後會說明參數的配置

✓ 機器學習流程

1. 資料準備 -training data -testing data

- ■ **Input** - train_data_input[train_data_num][input_length]
  - test_data_input[test_data_num][input_length]

```
float train_data_input[train_data_num][input_length]
{ 448,  434,  245}, { 429,  420,  236}, { 447,  433,  244},
```

二維陣列，每一個子陣列代表一筆訓練資料 {R,G,B}

■ **Output** - train_data_output[train_data_num][target_num]

- test_data_output[test_data_num][target_num]

```
int train_data_output[train_data_num][target_num]
```

```
//Ambient          //Blue
{1,0,0,0,0,0,0},{0,1,0,0,0,0,0},
```

每個子陣列是對應 data_input 的標籤，用 One-Hot Encoding 來表示資料所屬的類別。

```
Ambient:  {1, 0, 0, 0, 0, 0, 0}
Blue:     {0, 1, 0, 0, 0, 0, 0}
Magenta:  {0, 0, 1, 0, 0, 0, 0}
Red:      {0, 0, 0, 1, 0, 0, 0}
Orange:   {0, 0, 0, 0, 1, 0, 0}
Yellow:   {0, 0, 0, 0, 0, 1, 0}
```

✧ One-Hot Encoding vs Label Encoding

種類	作法	舉例	特性	優點	缺點
Label Encoding	以數值大小順序進行編碼	0, 1, 2....	不同類別有大小順序之關係	編碼方式簡單，用單一數值可代表	不同類別可能有順序之關係
One-Hot Encoding	以多數值向量方式進行編碼	[1,0,0], [0,1,0]....	不同類別為不同方向之關係，無順序之分	不同類別之間無順序大小關係，較可讓機器正確分類	當類別太多時，編碼長度會過長

2. **資料前處理** -讀取訓練與測試資料，並進行標準化處理

■ **scale_data()** -計算訓練資料的**平均值(Mean)**以及**標準差(Standard Deviation)**

✧ Mean

$$Mean_j = \frac{\sum_{i=1}^{N} train\_data\_input\,[i][j]}{N}$$

```
// Compute Data Mean
for(i = 0; i < train_data_num; i++){          // 遍歷所有訓練資料
    for(j = 0; j < input_length; j++){        // 遍歷每筆資料中的每個特徵
        sum[j] += train_data_input[i][j];     // 將每個特徵的值累加到 sum[j]
    }
}
for(j = 0; j < input_length ; j++){           // 遍歷每個特徵
    data_mean[j] = sum[j] / train_data_num;   // 計算每個特徵的平均值
    printf("MEAN: %.2f\n", data_mean[j]);     // 印出平均值
    sum[j] = 0.0;                             // 將 sum 重置為 0，為下一步計算標準差準備
}
```

✧ STD

$$STD_j = \sqrt{\frac{\sum_{i=1}^{N}(\,train\_data\_input[i][j] - Mean_j)^2}{N}}$$

```
// Compute Data STD
for(i = 0; i < train_data_num; i++){
    for(j = 0; j < input_length ; j++){
        sum[j] += pow(train_data_input[i][j] - data_mean[j], 2);    // 累加每個特徵與均值的平方差
    }
}
for(j = 0; j < input_length; j++){                                  // 遍歷每個特徵
    data_std[j] = sqrt(sum[j]/train_data_num);                      // 計算標準差
    printf("STD: %.2f\n", data_std[j]);
    sum[j] = 0.0;
}
```

- **normalize()**
  - ✧ 用 scale_data() 計算出的 Mean STD 去對數據進行標準化

$$normalized\_data[i] = \frac{data[i] - Mean_i}{STD_i}$$

```
void normalize(float *data)
{
        int i;

        // 遍歷輸入資料的每個特徵，並對每個特徵值進行標準化處理
        for(i = 0; i < input_length; i++){
            data[i] = (data[i] - data_mean[i]) / data_std[i];
        }
}
```

- **train_preprocess()　test_preprocess()**

  用 normalize()逐筆對 train_data_input、 test_data_input 中的每筆資料進行標準化

```
int train_preprocess()
{
    int i;

    for(i = 0 ; i < train_data_num ; i++)
    {
        normalize(train_data_input[i]);
    }

    return 0;
}
```

```
int test_preprocess()
{
    int i;

    for(i = 0 ; i < test_data_num ; i++)
    {
        normalize(test_data_input[i]);
    }

    return 0;
}
```

- **data_setup()**
  - ✧ 透過從 ADC 讀取模擬輸入數據的隨機波動，生成一個具有隨機性的種子值來初始化隨機數生成器。

```
/* Set the ADC operation mode as single-cycle, input mode as single-end and
                enable the analog input channel 0, 1 and 2 */
ADC_Open(ADC, ADC_ADCR_DIFFEN_SINGLE_END, ADC_ADCR_ADMD_SINGLE_CYCLE, 0x7);

/* Power on ADC module */
ADC_POWER_ON(ADC);

/* Clear the A/D interrupt flag for safe */
ADC_CLR_INT_FLAG(ADC, ADC_ADF_INT);

/* Start A/D conversion */
ADC_START_CONV(ADC);

/* Wait conversion done */
while (!ADC_GET_INT_FLAG(ADC, ADC_ADF_INT));

for (i = 0; i < 3; i++)
{
    seed *= ADC_GET_CONVERSION_DATA(ADC, i);
}
seed *= 1000;
printf("\nRandom seed: %d\n", seed);
srand(seed);
```

  - ✧ 初始化 RandomizedIndex

    初始化一個陣列 RandomizedIndex，用於存放訓練資料的 index，後續會根據這個陣列來隨機打亂訓練資料的順序。(此時 index 順序還沒被打亂)

```
ReportEvery10 = 1;
for (p = 0; p < train_data_num; p++)
{
    RandomizedIndex[p] = p;
}
```

為什麼要打亂訓練資料?

1. **避免模型記住固定模式 提升模型的泛化能力**
   如果訓練資料總是以固定順序輸入，模型可能會學到與資料順序相關的模式，而不是學習真正的特徵與輸出之間的關係。
2. **支持 SGD 有效工作**
   隨機梯度下降法要求每次從隨機選擇的樣本中計算梯度，如果資料順序固定，則會導致梯度更新不夠隨機化，影響收斂效果。

✧ 調用 scale_data()、train_preprocess()、test_preprocess()
如果標準化過程出現錯誤，返回 1 表示失敗

```
scale_data();
ret = train_preprocess();
ret |= test_preprocess();
if (ret) // Error Check
    return 1;

return 0;
```

3. **模型初始化**

對隱藏層權重 (HiddenWeights) 和輸出層權重 (OutputWeights) 做隨機初始化

- 權重變化量設為 0
- 隨機生成範圍在 [-InitialWeightMax, InitialWeightMax] 之間的初始權重
- **為什麼要這麼做?**
  - ✧ 隨機初始化權重可以打破對稱性，讓每個神經元從不同的初始值開始學習，這樣即使輸入相同，經過不同權重後，輸出會有所差異，進而使得每個神經元在反向傳播時得到不同的梯度，學習到不同的特徵。
  - ✧ 隨機性增加模型的探索能力，有助於網路更快收斂並找到更好的解。

```
for (i = 0; i < HiddenNodes; i++)
{
    for (j = 0; j <= input_length; j++)    // 遍歷每個輸入特徵，包括偏置節點 (bias)
    {
        // 將權重變化量初始化為 0（動量項的初始值）
        ChangeHiddenWeights[j][i] = 0.0;

        // 隨機生成範圍在 [-InitialWeightMax, InitialWeightMax] 的初始權重
        Rando = (float)((rand() % 100)) / 100;
        HiddenWeights[j][i] = 2.0 * (Rando - 0.5) * InitialWeightMax;
    }
}

// Initialize OutputWeights and ChangeOutputWeights
for (i = 0; i < target_num; i++)
{
    for (j = 0; j <= HiddenNodes; j++)
    {
        ChangeOutputWeights[j][i] = 0.0;
        Rando = (float)((rand() % 100)) / 100;
        OutputWeights[j][i] = 2.0 * (Rando - 0.5) * InitialWeightMax;
    }
}
```

4. **模型訓練**

- **開始訓練**

```
// Begin training
for (TrainingCycle = 1; TrainingCycle < 2147483647; TrainingCycle++)
{
    Error = 0.0;
```

■ **打亂 RandomizedIndex**

```
// Randomize order of training patterns
for (p = 0; p < train_data_num; p++)
{
    q = rand() % train_data_num;
    r = RandomizedIndex[p];
    RandomizedIndex[p] = RandomizedIndex[q];
    RandomizedIndex[q] = r;
}
```

有點像這樣　RandomizedIndex =　　{0, 1, 2, 3, 4};　->　{3, 4, 2, 0, 1}

```
for (q = 0; q < train_data_num; q++)
{
    p = RandomizedIndex[q];

    // Compute hidden layer activations
    for (i = 0; i < HiddenNodes; i++)
    {
        Accum = HiddenWeights[input_length][i];
        for (j = 0; j < input_length; j++)
        {
            Accum += train_data_input[p][j] * HiddenWeights[j][i];
        }
        Hidden[i] = 1.0 / (1.0 + exp(-Accum));
    }
```

後續就會像這樣，p 為打亂後的 index，再根據這個 index 去讀取 training data，就樣就可以以不同順序去讀取資料集了

■ **前向傳播**　　-計算隱藏層與輸出層的激活值

✧ 隱藏層 (如註解)

```
// Compute hidden layer activations
for (i = 0; i < HiddenNodes; i++)
{
    Accum = HiddenWeights[input_length][i];                    //初始化加權和(包括 bias)
    for (j = 0; j < input_length; j++)
    {
        Accum += train_data_input[p][j] * HiddenWeights[j][i];   // 累加輸入層的加權輸入
    }
    Hidden[i] = 1.0 / (1.0 + exp(-Accum));                     // Sigmoid 激活函數，計算隱藏層第 i 個神經元的輸出
}
```

✧ 輸出層

```
// Compute output layer activations and calculate errors
for (i = 0; i < target_num; i++)
{
    Accum = OutputWeights[HiddenNodes][i];
    for (j = 0; j < HiddenNodes; j++)
    {
        Accum += Hidden[j] * OutputWeights[j][i];
    }
    Output[i] = 1.0 / (1.0 + exp(-Accum));
    OutputDelta[i] = (train_data_output[p][i] - Output[i]) * Output[i] * (1.0 - Output[i]);
    Error += 0.5 * (train_data_output[p][i] - Output[i]) * (train_data_output[p][i] - Output[i]);
}
```

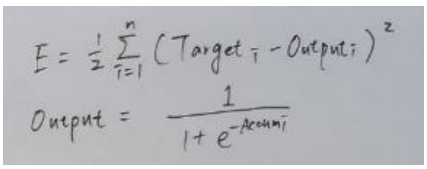
前面都跟隱藏層相同概念，只是最後多了計算誤差的部分

1.Error

使用 MSE 作為 loss function

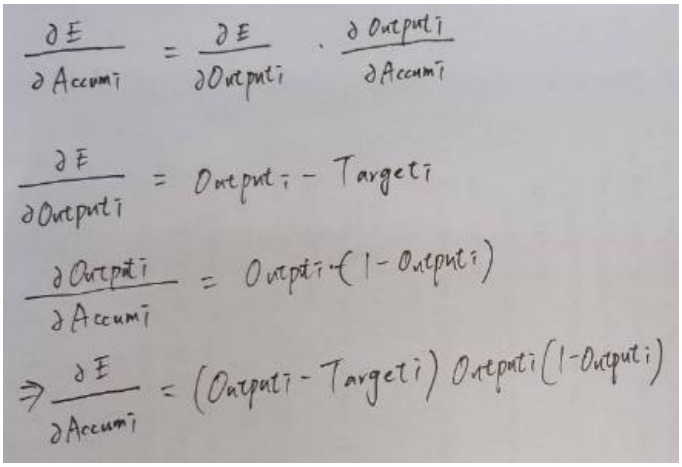
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

2. OutputDelta 在反向傳播中將誤差傳播回去



$$E = \frac{1}{2}\sum_{i=1}^{n}(Target_i - Output_i)^2$$

$$Output = \frac{1}{1+e^{-Accum_i}}$$



$$\frac{\partial E}{\partial Accum_i} = \frac{\partial E}{\partial Output_i} \cdot \frac{\partial Output_i}{\partial Accum_i}$$

$$\frac{\partial E}{\partial Output_i} = Output_i - Target_i$$

$$\frac{\partial Output_i}{\partial Accum_i} = Output_i \cdot (1 - Output_i)$$

$$\Rightarrow \frac{\partial E}{\partial Accum_i} = (Output_i - Target_i)\, Output_i (1 - Output_i)$$

我們這邊求的就是: 總誤差 E 對輸出層第 i 個神經元的激活值 Accumi 的偏導數
這個導數在反向傳播中用來計算每個輸出層神經元的誤差信號，這個誤差信號會反向傳播到隱藏層，進而計算隱藏層的誤差，最終完成整個神經網路的權重更新。
最後推導出的的結果為

$$OutputDelta_i = (Target_i - Output_i) \cdot Output_i \cdot (1 - Output_i)$$

```
OutputDelta[i] = (train_data_output[p][i] - Output[i]) * Output[i] * (1.0 - Output[i]);
```

■ 反向傳播

```
// Backpropagate errors to hidden layer
for (i = 0; i < HiddenNodes; i++)
{
    Accum = 0.0;
    for (j = 0; j < target_num; j++)
    {
        Accum += OutputWeights[i][j] * OutputDelta[j];        // 將輸出層的誤差乘以對應的權重，累加起來
    }
    HiddenDelta[i] = Accum * Hidden[i] * (1.0 - Hidden[i]);   // 計算隱藏層誤差項
}
```

i. **Accum += OutputWeights[i][j] * OutputDelta[j];**

$$Accum_i = \sum_{j=1}^{target\_num} OutputWeights\,[i][j] \cdot OutputDelta_j$$

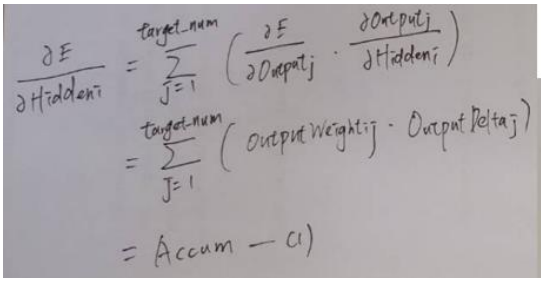
輸出層每個神經元的誤差信號會乘以對應的權重，然後累加到隱藏層的誤差信號中
計算的是輸出層誤差對隱藏層第 i 個神經元的影響

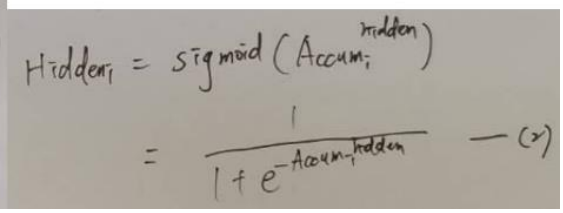
ii. **HiddenDelta[i] = Accum * Hidden[i] * (1.0 - Hidden[i]);**

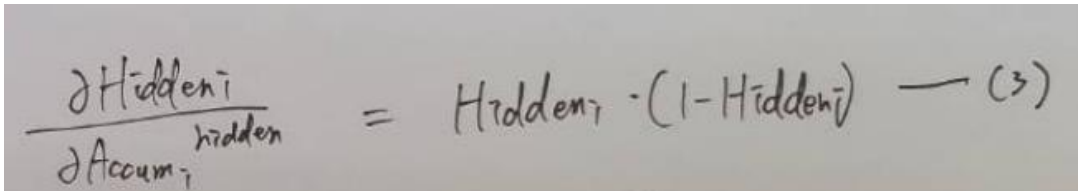
HiddenDelta 是從輸出層誤差反向傳播得到的，表示隱藏層神經元對輸出層誤差的間接影響。們這邊求的就是: 總誤差 E 對 隱藏層第 i 個神經元加權和 $Accum_i^{hidden}$ 的偏導數

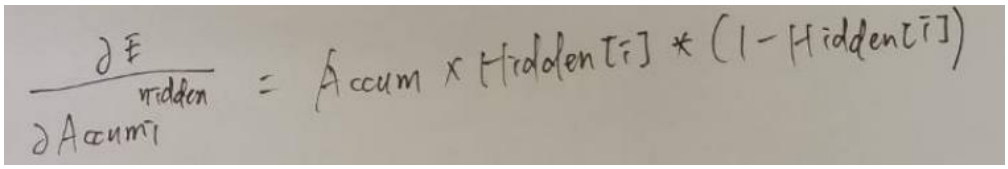
$$HiddenDelta_i = \frac{\partial E}{\partial Accum_i^{hidden}}$$

推導過程如下:









■ **更新權重** - 梯度下降(SGD) + 動量項

$$\Delta W = \eta \cdot 輸入值 \cdot \delta + \alpha \cdot \Delta W^{prev}$$

✧ η: 為學習率（learning rate）

✧ δ: 誤差項 (OutputDelta、HiddenDelta)

✧ α: 動量係數 (Momentum)

✧ ΔW^{prev}: 上一次的權重變化量 (ChangeHiddenWeights 、 ChangeOutputWeights)

Momentum 作用: 防止因為梯度方向的劇烈變化而出現震盪現象。通過引入上一次權重變化量來加速收斂，尤其是在接近最優解時能夠更快地趨近穩定。

```
// Update Input-->Hidden Weights
for (i = 0; i < HiddenNodes; i++)
{
    // 更新隱藏層的偏置權重 (輸入層到隱藏層的偏置對應的權重)
    ChangeHiddenWeights[input_length][i] = LearningRate * HiddenDelta[i] + Momentum * ChangeHiddenWeights[input_length][i];
    HiddenWeights[input_length][i] += ChangeHiddenWeights[input_length][i];
    for (j = 0; j < input_length; j++)
    {
        // 計算輸入層到隱藏層的權重變化量 (包含學習率與動量項)
        ChangeHiddenWeights[j][i] = LearningRate * train_data_input[p][j] * HiddenDelta[i] + Momentum * ChangeHiddenWeights[j][i];

        // 更新輸入層到隱藏層的權重
        HiddenWeights[j][i] += ChangeHiddenWeights[j][i];
    }
}

// Update Hidden-->Output Weights
for (i = 0; i < target_num; i++)
{
    // 更新輸出層的偏置權重 (隱藏層到輸出層的偏置對應的權重)
    ChangeOutputWeights[HiddenNodes][i] = LearningRate * OutputDelta[i] + Momentum * ChangeOutputWeights[HiddenNodes][i];
    OutputWeights[HiddenNodes][i] += ChangeOutputWeights[HiddenNodes][i];
    for (j = 0; j < HiddenNodes; j++)
    {
        // 計算隱藏層到輸出層的權重變化量 (包含學習率與動量項)
        ChangeOutputWeights[j][i] = LearningRate * Hidden[j] * OutputDelta[i] + Momentum * ChangeOutputWeights[j][i];

        // 更新隱藏層到輸出層的權重
        OutputWeights[j][i] += ChangeOutputWeights[j][i];
    }
}
```

如圖中紅色框框，第一個是計算變化量，接這第二個框框會依照這個變化量更新權重

■ 計算訓練準確率

```
accuracy = Get_Train_Accuracy();
```

Get_Train_Accuracy() 這邊就是在走一次前向傳播的過程，依照輸出的值與 **ground truth** 做比較計算出正確率

```
// get target value
max = 0;
for (i = 1; i < target_num; i++)
{
    if (train_data_output[p][i] > train_data_output[p][max])
    {
        max = i;
    }
}
target_value = max;
// get output value
max = 0;
for (i = 1; i < target_num; i++)
{
    if (Output[i] > Output[max])
    {
        max = i;
    }
}
out_value = max;
// compare output and target
if (out_value == target_value)
{
    correct++;
}
```

```
// Calculate accuracy
accuracy = (float)correct / train_data_num;
return accuracy;
```

■ 終止訓練

如果正確率大於我們設定的正確率，就會結束訓練

```
// If error rate is less than pre-determined threshold then end
if (accuracy >= goal_acc)
    break;
```

5. 模型測試

```
run_train_data();
printf("Training Set Solved!\n");
printf("--------\n");
printf("Testing Start!\n ");
run_test_data();
```

run_train_data(); run_test_data(); 跟 Get_Train_Accuracy()一樣的方法，去看 training data and testing data 經過我們訓練好模型的正確率，並把錯誤的結果印出來

```
if (out_value != target_value)
    printf("Error --> Training Pattern: %d,Target : %d, Output : %d\n", p, target_value, out_value);
else
    correct++;
```

6. 印出權重值 -load_weight();
7. 即時預測 - AdcSingleCycleScanModeTest()
   - ✧ **ADC 獲取實時輸入信號，並進行標準化處理。**
   - ✧ **將標準化後的數據輸入到訓練好的神經網路中通過前向傳播計算隱藏層和輸出層的激活值。**
   - ✧ **根據輸出層結果，找出預測的分類索引，並顯示對應的分類名稱。**
   - ✧ **不斷重複上述過程，實現實時分類預測。**

```
switch (out_value)
{

case 0:
    strcpy(output_string, "Ambient");
    break;
case 1:
    strcpy(output_string, "Blue");
    break;
case 2:
    strcpy(output_string, "Magenta");
    break;
case 3:
    strcpy(output_string, "Red");
    break;
case 4:
    strcpy(output_string, "Orange");
    break;
case 5:
    strcpy(output_string, "Yellow");
    break;
case 6:
    strcpy(output_string, "Green");
    break;
}

printf("\rPrediction output: %-8s", output_string);
CLK_SysTickDelay(500000);
```

8. 參數配置

```
const float LearningRate = 0.005; // Learning Rate
const float Momentum = 0.9;
const float InitialWeightMax = 0.5;
const float goal_acc = 0.97; // Target accuracy
```

```
#define input_length 3 // The number of input
#define HiddenNodes 17 // The number of neurons in hidden layer
#define target_num 7   // The number of output
```

參數	設置值	作用	過大影響	過小影響
LearningRate	0.005	控制每次權重更新的步長	權重更新過頭，無法收斂	收斂速度過慢
Momentum	0.9	平滑權重更新，加速收斂	權重更新過快，錯過最優解	無法減少震盪，收斂速度慢
InitialWeightMax	0.5	隨機初始化權重，打破對稱性	梯度消失，影響學習效果	收斂速度慢
goal_acc	0.97	定義訓練終止條件	可能過擬合	準確率不足，影響預測性能

<過程中遇到的困難>

在這次 Lab 的實作中，我碰到的挑戰主要是跟環境和參數設置的問題。環境方面，有時候今天蒐集到的數據和隔一天蒐集到的數據會有明顯差異，甚至實驗器材稍微碰到或位置變動都會導致數據波動，這讓結果變得不太穩定。另一方面，在隱藏層節點數和學習率的設定上也花了很

多時間，需要不斷嘗試不同的組合，透過反覆調整才找到比較好的配置。

<心得與收穫>

在這次的機器學習實驗中，我學會了如何在 MCU 上使用 C 語言進行神經網路的配置與訓練，並深入掌握神經網路的各個細節。例如，透過標準化資料的過程，我理解到這個步驟能有效確保輸入數據在模型中穩定運行，並進一步提升訓練效率。在模型訓練過程中，權重的隨機初始化則能減少初始偏置問題，進而加速模型的收斂。此外，透過實作正向傳播與誤差反向傳播演算法，我深入理解了神經網路的運算邏輯，包括隱藏層與輸出層的激活函數如何影響最終結果，以及如何根據誤差回傳更新權重。這段經驗讓我對梯度下降法及其在權重優化中的實際應用有了更具體的理解。在訓練與測試階段，我學會了如何設計有效的模型評估方法，尤其在資源有限的嵌入式環境中，透過準確率的精確計算來評估模型效能，進一步強化了我對模型表現評估的能力。這次專案不僅使我掌握了在 MCU 上實現深度學習的完整流程，更大幅提升了我在受限資源下進行 AI 應用開發的能力，為未來的嵌入式 AI 開發奠定了良好的基礎。