Deep Learning/Deep Learning Labs

Experiment Report

Lab1: back-propagation

資科工碩

郭建廷　313551072

目錄

[目錄 I](#_Toc171984799)

[1. Introduction 1](#_Toc171984800)

[2. Experiment setups 2](#_Toc171984801)

[**A. Sigmoid functions** 2](#_Toc171984802)

[**B. Neural network** 3](#_Toc171984803)

[**C. Backpropagation** 6](#_Toc171984804)

[3. Results of testing 7](#_Toc171984805)

[**A. Screenshot and comparison figure** 9](#_Toc171984806)

[**B. Show the accuracy of your prediction** 19](#_Toc171984807)

[**C. Learning curve (loss, epoch curve)** 28](#_Toc171984808)

[**D. Anything you want to present** 29](#_Toc171984809)

[4. Discussion 47](#_Toc171984810)

[**A. Try different learning rates** 47](#_Toc171984811)

[**B. Try different numbers of hidden units** 47](#_Toc171984812)

[**C. Try without activation functions** 48](#_Toc171984813)

[**D. Anything you want to present** 48](#_Toc171984814)

[5. Extra 49](#_Toc171984815)

[**A. Implement different optimizers** 49](#_Toc171984816)

[**B. Implement different activation functions** 50](#_Toc171984817)

1. Introduction

在這份報告中在Experiment setups中，首先帶出最基本的Artificial Neural，說明不帶有激活函數的Artificial Neural無法處理非線性的問題，而代出Sigmoid當作例子，並且表示用python實作Sigmoid function和算其導函數的式子。

接著在Neural network時會先講解最基本的Machine Learning概念，帶出需要去修正的模型參數與其原理，接著說明Artificial Neural的概念與本實驗報告中Artificial Neural和Neural Network model的設計方式。Backpropagation則是會說明計算梯度和修正模型參數的算法，會以一個簡單的例子做說明，最後以Python程式碼的實例做結尾。

在Results of testing章節中會表示基本測試資料的訓練成果，比較不同learning rate、不同數量的 hidden unit、和使用不同的 activation function。

在Results of testing章節中的D子項，則是說明了使用不同的資料集，除了原先所規定的Linear與Non-Linear(XOR)版本，也有調整 linear的樣本數量(n)，額外設計一個帶有Noise的資料集來測試難度較高的non-linear問題。在測試這些不同的資料集時，也會針對調整超參數(hyperparameter)、網路架構、Optimizer等情況，來記錄訓練的loss、 training和testing的效果。

在Discussion時，則會針對Results of testing所做的所有測試去討論，並且在Discussion的D子項中，會講解測試下來不同參數、架構之間的差異。

最後的Extra章節則是講解實作不同activation function(ReLU)與Optimizer(SGD、Momentum-GD)。

2. Experiment setups

**A. Sigmoid functions**

一個基本的Artificial Neural屬於一種線性的方程式，表示方式如下：

由於在處理非線性問題時，如XOR問題時會沒辦法用來區分，而激活函數本身帶有非線性的關係，因此在設計解決XOR問題時，都會在人工神經網路中加入一個激活函數，來解決非線性的問題，甚至是更加複雜的問題。而Sigmoid算是一個很常見的激活函數(Activation function)，除了達到非線性的狀態，Sigmoid也可以做到正規化的輸出，下方圖2.1表示了他是一個S型曲線的函數圖形，可以將輸出限制在[0,1]之間。

在Sigmoid的設計上使用下方的程式碼，列印出來的function如圖2.1：

|  |
| --- |
| def sigmoid(x):  return 1.0/(1.0+np.exp(-x)) |
| 一張含有 繪圖, 行, 圖表, 文字 的圖片  自動產生的描述  圖2.1、Sigmoid函數圖 |

Sigmoid function的微分推倒如下：

由原先的function可以推出 因此

微分的Sigmoid function實作如下，於Backpropagation時會提到這樣實作的目的

|  |
| --- |
| def derivative\_sigmoid(x):  return np.multiply(x,1.0 - x) |

**B. Neural network**

先帶出最基本的Machine Learning概念，圖2.2是一個Machine Learning 的Learning Algorithm。

一張含有 文字, 行, 字型, 圖表 的圖片

自動產生的描述

圖2.2、Learning Algorithms

其中 可以是任何數值、圖片甚至是音檔，透過 Representation轉成 讓 Model 可以訓練的數據，而這個Model則是我們要訓練的模型，他會輸出一個預測的值 ，要和實際地(ground-truth)值做 loss function求出Cost，而目標則是降低Cost讓Model能預測的跟實際值相近。下方數學式子可以簡易的表示一個簡單的模型：

可以看到輸入是而他會乘上一個權重矩陣，透過修正來達到降低 與的loss。

然而一般的問題很難用一層的Artificial Neural解決，圖2.3展示了多層的複雜模型，可以看到要同時修正多個模型參數。

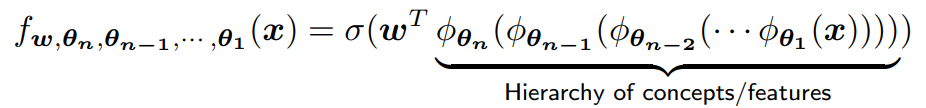


圖2.3、多層架構的Model

常見的Artificial Neural設計如下圖2.4所示，包含多個權重的和多個輸入資料與一個偏量和一個激活函數。

一張含有 文字, 螢幕擷取畫面, 行, 字型 的圖片

自動產生的描述

圖2.4、Artificial Neural，此圖中代表 進入激活函數前的數值

本章節要介紹Neural Network，是由一個Input、一個Output層，和多個Hidden Layer所組成的人工神經網路(Artificial Neural Network)，如圖2.5所示，每個Artificial Neural中都有權重和偏量要去訓練。

一張含有 螢幕擷取畫面, 圓形, 行, 圖表 的圖片

自動產生的描述

圖2.5、Neural network，圖中每個圓形節點都是一個Artificial Neural

在本實驗報告設計兩個Class來實作一個 Neural Network，分別設計nn與model，nn為一個獨立的Artificial Neural，可以設定的參數有

* 輸入
* 輸出
* 激活函數
* 微分的激活函數
* 權重矩陣的優化器
* 偏移向量的優化器

如下方程式碼：

|  |
| --- |
| neural=nn(input,output,activate\_function,derivative\_function,  weight\_optimizer,bias\_optimizer) |

其中激活函數(Activation function)和優化器(Optimizer)如果不特別給值時，預設都是None。

此Class包含了一層Artificial Neural和一個Optimizer，如題目所要求的Two-layer neural network，的新增方式可以如下：(假設2,4,4,1，sigmoid)

|  |
| --- |
| network = []  network.append(nn(2,4,sigmoid,derivative\_sigmoid))  network.append(nn(4,4,sigmoid,derivative\_sigmoid))  network.append(nn(4,1,sigmoid,derivative\_sigmoid)) |

此外也實作的Neural Network 所需的forward、backward、update功能，在training時步驟如下：

1. 根據現在的網絡，輸入的data進行前向傳播 (forward propagation)，每一層的輸出會作為下一層的輸入，依次類推。
2. nn的實例中每一層forward包含輸入值、權重、偏量、激活函數  
   數學式子表示為：。
3. 最後一層通過損失函數(loss function)計算出損失值(loss)，並對整個模型進行反向傳播(back-propagation)，以計算出模型參數的梯度。
4. 計算所有要修正的梯度之後再來對整個模型參數做修正(update)。

具體的程式碼如下：

|  |
| --- |
| def forward(self,feature):  # give feature with size input then result size output feature  # linear function + activation function  # y = σ(WX + b)  self.a\_input = feature  self.a\_output = self.activate\_function(np.dot(feature,self.W) + self.b)  return self.a\_output  def backward(self,output\_error):  z = output\_error \* self.derivative\_function(self.a\_output) # calc this layer new error  self.dW = np.dot(self.a\_input.T , z)  self.db = np.sum(z, axis=0, keepdims=True)  return np.dot(z, self.W.T)  def update(self,learning\_rate):  if (self.weight\_optimizer != None and self.bias\_optimizer!= None ):  self.W -= self.weight\_optimizer.calc(learning\_rate \* self.dW)  self.b -= self.bias\_optimizer.calc(learning\_rate \* self.db)  else:  self.W -= learning\_rate \* self.dW  self.b -= learning\_rate \* self.db |

為了更好的控制我們的神經網路模型，設計了一個名為model的Class，用來維護整個Neural Network，並且實作training與testing的介面，減少直接對list裡面的nn實例做使用，並且也可以透過這個model的Class額外設計如設計優化器(Optimizer)、繪圖(learning curve)等功能。

**C. Backpropagation**

Backpropagation在執行完forward時做backward計算所有neural中的模型參數weight與bias的梯度，接著搭配learning rate對所有的模型參數做修正。在此會以一個簡單的網路做例子(2,2,2,1)，如圖2.6所示。

一張含有 行, 圖表, 字型, 繪圖 的圖片

自動產生的描述

圖2.6、擁有2個參數輸入與1個參數輸出和兩個Hidden Layer的NN，其中沒有bias的偏量只有權重矩陣

已知微分可以求在某個曲線上的切線斜率，而透過這個求斜率的過程可以去找這個函數中的最大、最小值，但是是區域最大、區域最小值，而如果針對最終Loss function出來的loss做對要修正的權重做偏微分也能求出往區域最佳解的修正，如下方式子：

其中可以是模型中任何的要訓練的參數，如每個神經元的偏量、矩陣權重。有了這個概念之後就要一一對這些參數做偏微分求出各自的梯度，而在算梯度時會需要帶入到微積分的連鎖律，由圖2.7可知，計算的偏微需要經過兩條路才能到達最終的Loss：

一張含有 行, 圖表, 字型 的圖片

自動產生的描述

圖2.7、計算時的路徑

把他展開後得到：

其中中間的Sigma項可以解釋成 走上下兩條路，而則是L針對的偏微分，已知L是一個Loss function計算出來的結果，以MSE為例的話則是 ，而在最後一曾並沒有設計一個activation function因此 ，也就是可以將式子化作 得到 ，依此類推去推算。

不過由前面開始算太複雜了，如果由後往前看會發現他是有規律的，如圖2.8所示：

一張含有 筆跡, 行, 圖表 的圖片

自動產生的描述

圖2.8、由後往前推算

同樣帶出求修正率和梯度的算法： ，上述提到 的算法，而，因此可以變成 ，這就是對的修改，那如果要繼續往下算的話，如圖2.9所示

*一張含有 圖表, 行, 字型, 折紙 的圖片

自動產生的描述*

圖2.9、由後往前推算

同樣的展開後得到 會發現前面的值都一樣，但會有一個激活函數需要去求微分，後續推額外的參數都是利用相似的方式，接著回到第一層：

拆開來表示則是，而在實作上以矩陣乘法(np.dot)的方式可以直接由後往前算，在計算上還有一些小技巧，原先的因為loss function挑選MSE的關係因此微分後會帶負號，所以在update值時會成加號，如下方式子：

此外在程式的實作上最外層的 2 這個參數可以省略，會被 learning rate 吸收，因此可以省略掉。

最後由上方的算式中求出

已知 並且 ，等同於 ，因此回到一開始Sigmoid function的微分設計，適用於此直接針對 上一層的輸入來算下一層輸出的微分。

回頭來看，微分的Sigmoid function與 backward的function，Python 程式設計如下

|  |
| --- |
| def derivative\_sigmoid(x):  return np.multiply(x,1.0 - x)  def backward(self,output\_error):  z = output\_error \* self.derivative\_function(self.a\_output) # calc this layer new error  self.dW = np.dot(self.a\_input.T , z)  self.db = np.sum(z, axis=0, keepdims=True)  return np.dot(z, self.W.T) |

以為例

根據上一層計算完的結果：

乘上微分的Sigmoid function： =

再乘上原先的輸入：

3. Results of testing

**A. Screenshot and comparison figure**

*測試一*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 | 一張含有 文字, 螢幕擷取畫面, 鮮豔, 圖表 的圖片  自動產生的描述 |
| Dataset: easy\_XOR  n: 21 |  |

*測試二*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=256 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=256)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試三*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.001,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試四*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-8-8-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,8,sigmoid,derivative\_sigmoid)  m.add\_layer(8,8,sigmoid,derivative\_sigmoid)  m.add\_layer(8,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試五*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數改成兩個隱藏層ReLU、輸出層Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,relu,derivative\_relu)  m.add\_layer(4,4,relu,derivative\_relu)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試六*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 無激活函數 * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4)  m.add\_layer(4,4)  m.add\_layer(4,1)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試七*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 無激活函數 * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4)  m.add\_layer(4,4)  m.add\_layer(4,1)  m.training(10000,x,y,0.00001,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試八*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-40-40-1 * All Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,40,sigmoid,derivative\_sigmoid)  m.add\_layer(40,40,sigmoid,derivative\_sigmoid)  m.add\_layer(40,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

**B. Show the accuracy of your prediction**

*測試一*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 | 一張含有 文字, 螢幕擷取畫面, 功能表, 圖書 的圖片  自動產生的描述 |

*測試二*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=256 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=256)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試三*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.001,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試四*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-8-8-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,8,sigmoid,derivative\_sigmoid)  m.add\_layer(8,8,sigmoid,derivative\_sigmoid)  m.add\_layer(8,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試五*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數改成兩個隱藏層ReLU、輸出層Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,relu,derivative\_relu)  m.add\_layer(4,4,relu,derivative\_relu)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試六*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 無激活函數 * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4)  m.add\_layer(4,4)  m.add\_layer(4,1)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試七*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 無激活函數 * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4)  m.add\_layer(4,4)  m.add\_layer(4,1)  m.training(10000,x,y,0.00001,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試八*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-40-40-1 * All Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,40,sigmoid,derivative\_sigmoid)  m.add\_layer(40,40,sigmoid,derivative\_sigmoid)  m.add\_layer(40,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

**C. Learning curve (loss, epoch curve)**

*測試一*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 | 一張含有 文字, 螢幕擷取畫面, 繪圖, 行 的圖片  自動產生的描述 |
| Dataset: easy\_XOR  n: 21 | 一張含有 文字, 行, 繪圖, 圖表 的圖片  自動產生的描述 |

*測試二*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=256 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=256)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試三*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,4,sigmoid,derivative\_sigmoid)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.001,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試四*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-8-8-1 * 激活函數皆為Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,8,sigmoid,derivative\_sigmoid)  m.add\_layer(8,8,sigmoid,derivative\_sigmoid)  m.add\_layer(8,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試五*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 激活函數改成兩個隱藏層ReLU、輸出層Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4,relu,derivative\_relu)  m.add\_layer(4,4,relu,derivative\_relu)  m.add\_layer(4,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試六*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 無激活函數 * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4)  m.add\_layer(4,4)  m.add\_layer(4,1)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試七*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-4-4-1 * 無激活函數 * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,4)  m.add\_layer(4,4)  m.add\_layer(4,1)  m.training(10000,x,y,0.00001,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

*測試八*

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 四層網路，兩層隱藏層，2-40-40-1 * All Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 10,000 * Loss function: MSE * 每20場記錄一次loss | 訓練的程式  m = model(memory\_epoch=20)  m.clear()  m.add\_layer(2,40,sigmoid,derivative\_sigmoid)  m.add\_layer(40,40,sigmoid,derivative\_sigmoid)  m.add\_layer(40,1,sigmoid,derivative\_sigmoid)  m.training(10000,x,y,0.1,1000,batch=32)  pred\_y = m.testing(x,y,show\_info=True)  show\_result(x,y,(pred\_y > 0.5))  m.show\_learning\_curve\_loss()  m.show\_learning\_curve\_accuracy() |

實驗結果

|  |  |
| --- | --- |
| Dataset: Linear  n: 100 |  |
| Dataset: easy\_XOR  n: 21 |  |

**D. Anything you want to present**

此部分會提出四個不同圖形訓練的結果，和線性時n=10k時的訓練結果，並且測試ReLU針對XOR甚至其他資料集有noise時的訓練情況。但是此部分就不會特別顯示dataset中每一個data的跌代結果，而是以準確度/Loss作比較。

|  |  |
| --- | --- |
| From sklearn.datasets  Dataset: circles  n: 200  noise: 0.1  factor: 0.5 |  |
| From sklearn.datasets  Dataset: moon  n: 200  noise: 0.1 |  |
| Dataset: FULL\_XOR  def generate\_XOR\_hard():  import numpy as np  inputs = []  labels = []  for i in range(101):  for j in range(101):  x1 = 0.01 \* i  x2 = 0.01 \* j  inputs.append([x1, x2])  labels.append(int((x1 > 0.5) != (x2 > 0.5)))  return np.array(inputs), np.array(labels).reshape(-1, 1) |  |
| Dataset: NOISE\_XOR  def generate\_XOR\_hard\_2(noise\_level=0.1):  inputs = []  labels = []  for i in range(101):  for j in range(101):  x1 = 0.01 \* i  x2 = 0.01 \* j  # 引入隨機噪音  x1\_noisy =x1+np.random.uniform(-noise\_level, noise\_level)  x2\_noisy=x2+np.random.uniform(-noise\_level, noise\_level)  inputs.append([x1\_noisy, x2\_noisy])  labels.append(int((x1 > 0.5) != (x2 > 0.5)))  return np.array(inputs), np.array(labels).reshape(-1, 1) |  |

實驗結果 Dataset: Linear (n=10000)

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 2-4-4-1 * All Sigmoid * Optimizer: SGD ,batch=32 * Learning rate: * Epoch = 1000 * Loss function: MSE * 每100場記錄一次loss | loss=0.1362 accuracy=99.80% |
| 訓練的參數、架構為   * 2-4-4-1 * All Sigmoid * Optimizer: MGD ,batch=32 * MGD: m=0.9 * Learning rate: * Epoch = 1000 * Loss function: MSE   每100場記錄一次loss | loss=0.0041 accuracy=99.88% |
| 訓練的參數、架構為   * 2-4-4-1 * ReLU、ReLU、Sigmoid * Optimizer: MGD ,batch=32 * MGD: m=0.9 * Learning rate: * Epoch = 1000 * Loss function: MSE   每100場記錄一次loss | loss=0.0011 accuracy=99.96% |

實驗結果 Dataset: circles

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 2-4-4-1 * All Sigmoid * Optimizer: MGD ,batch=32 * MGD: m=0.9 * Learning rate: * Epoch = 1000 * Loss function: MSE * 每100場記錄一次loss | loss=0.2446 accuracy=50.00% |
| 訓練的參數、架構為   * 2-4-4-1 * ReLU、ReLU、Sigmoid * Optimizer: SGD,batch=32 * Learning rate: * Epoch = 1000 * Loss function: MSE   每100場記錄一次loss | loss=0.0222 accuracy=99.00% |

實驗結果 Dataset: moon

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 2-4-4-1 * ReLU、ReLU、Sigmoid * Optimizer: SGD,batch=32 * Learning rate: * Epoch = 10000 * Loss function: MSE   每100場記錄一次loss | loss=0.0755 accuracy=89.50% |
| 訓練的參數、架構為   * 2-4-4-1 * ReLU、ReLU、Sigmoid * Optimizer: SGD,batch=32 * Learning rate: * Epoch = 2000 * Loss function: MSE   每100場記錄一次loss | loss=0.0015 accuracy=100.00% |

實驗結果 Dataset: FULL\_XOR

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 2-4-4-1 * ReLU、ReLU、Sigmoid * Optimizer: SGD,batch=32 * Learning rate: * Epoch = 2000 * Loss function: MSE * 每100場記錄一次loss | loss=0.0122 accuracy=98.73% |
| 訓練的參數、架構為   * 2-4-4-1 * All Sigmoid * Optimizer: SGD,batch=32 * Learning rate: * Epoch = 2000 * Loss function: MSE   每100場記錄一次loss | loss=0.0015 accuracy=99.90% |

實驗結果 Dataset: NOISE\_XOR

|  |  |
| --- | --- |
| 訓練的參數、架構為   * 2-4-4-1 * ReLU、ReLU、Sigmoid * Optimizer: MGD ,batch=32 * MGD: m=0.9 * Learning rate: * Epoch = 2000 * Loss function: MSE   每100場記錄一次loss | loss=0.0644 accuracy=90.39% |
| 訓練的參數、架構為   * 2-4-4-1 * All Sigmoid * Optimizer: MGD ,batch=32 * MGD: m=0.9 * Learning rate: * Epoch = 2000 * Loss function: MSE   每100場記錄一次loss | loss=0.0631 accuracy=90.48% |

4. Discussion

**A. Try different learning rates**

在上方的實驗中，learning rate的影響，最基本的是影響訓練的收斂速度圖，和收斂於區域最佳解，如圖4.1下方四張(a),(b),(c),(d)圖，這四張圖來自於第三章提到的*測試一*與*測試三*的Loss曲線圖，由曲線圖中可以看到xor的曲線(b)、(d)都已經收斂了，但是(b)收斂在區域最佳解，沒有到非常好的結果，而由(a)、(b)兩張linear的Loss曲線可以看出 (a) 因為學習率為0.001的關係，因此尚未收斂，而(b) 學習率為0.1 的嘗試大概小於一千次就收斂了，這是其中一個可以看到調整Learning rate會得到的結果。

|  |  |
| --- | --- |
| 1. linear      1. non-linear | 一張含有 文字, 螢幕擷取畫面, 繪圖, 行 的圖片  自動產生的描述   1. linear   一張含有 文字, 行, 繪圖, 圖表 的圖片  自動產生的描述   1. non-linear |

圖4.1、(a) 與 (b) 皆為學習率0.001 於(測試三)的loss 曲線，分別是 (a) linear (b) xor；(c) 與 (d) 皆為學習率0.1於(測試一)的loss曲線

接著要提到另一個學習率會影響的結果，於*測試六*與*測試七*時，由於一次的移動步伐太大，或者矩陣運算時溢位了(overflow)，導致測試六無法去做訓練，但是調整學習率，調整到非常低時就可以看到學習的效果，具體的學習效果會在C. Try without activation functions時說明。

**B. Try different numbers of hidden units**

藉由測試一與測試四，在同樣的參數下，同樣的訓練下，不同的網路模型，多個unit會給予更好的訓練結果，但是過多的unit反倒會導致訓練收斂變慢，一次訓練時間變長，如*測試八*或圖4.2。

|  |  |
| --- | --- |
| 1. 測試一 線性 訓練時 已收斂 | 1. 測試八 線性 訓練時收斂慢 |

圖4.2、測試一與測試八Neural network中不同 Unit 數量收斂的差異

**C. Try without activation functions**

在A小節時有提到，不使用激活函數的情況下可能會導致做矩陣乘法時溢位的問題，而調整學習率可以改善矩陣運算溢位的問題，但是學習率越小又會導致區域最佳解的行程，此外從*測試七*的圖中可以看到沒有帶入激活函數是很難達到解非線性問題的。

**D. Anything you want to share**

由多個測試中，使用ReLU來當作激活函數通常來講要比Sigmoid還要來的好(於3. D. Anything you want to present中的Linear (n=10000) 和 circles的dataset)中都可以看到，將中間隱藏層的激活函數改成ReLU，效果都會比直接使用 Sigmoid還要來的好，而且Sigmoid可能會遇到梯度消失的情況(於Extra中有提到)，而ReLU並不會有梯度消失的問題。

此外本研究報告還有實作SGD與MGD，由下方圖4.3中可以看到，同樣是使用Sigmoid在相同的學習率下，MGD的訓練收斂表現比SGD還要來的快。

|  |  |
| --- | --- |
| (a)學習率低時利用SGD的收斂狀況 | (b)學習率低時利用MGD的收斂狀況 |

圖4.3、相同網路架構，學習率 MGD與SGD的訓練收斂比較圖

5. Extra

**A. Implement different optimizers**

在本實驗中以SGD(Stochastic Gradient Descent)作為基礎的Optimizer，他會將輸出的training data分成多個指定的大小(batch)，然後分批去做訓練，但是在分的過程中會去打亂分布，確保資料不會一值相同的訓練。在實作上則是以下方的程式碼為例

|  |
| --- |
| def training(self,n,training\_data,ground\_truth,learning\_rate=0.1,show\_info=5000,batch=32):  training\_set = np.hstack((training\_data,ground\_truth))  tot\_dataset = len(training\_data)  for epoch in range(n):  np.random.shuffle(training\_set)  batch\_training\_data = training\_set[:,:-1]  batch\_prediction = training\_set[:,-1:]  for i in range(int(math.ceil(tot\_dataset / batch))):  st = batch \* i  ed = min(batch \* (i+1),tot\_dataset)    prediction = self.forward(batch\_training\_data[st:ed])  # backward pass compute \delta weights  self.backward(batch\_prediction[st:ed],prediction)  # update  self.update(learning\_rate) |

先把training\_data與ground\_truth堆疊起來，確保在shuffle時不會失去對應的值，然後接著根據 batch的大小去做分配，每一次batch的訓練完畢才會去做 backward的動作和update參數。

這樣的優點在於當資料集過大時，分批可以更好的去訓練模型，比起一般的GD一次性全部的資料去訓練，準確度大幅提升。

本實驗報告也嘗試了另一種的優化器，Momentum-based Gradient Descent，MGD帶出了以下式子：

一張含有 字型, 白色, 印刷術, 文字 的圖片

自動產生的描述

一張含有 圓形, 設計 的圖片

自動產生的描述

MGD的原理是控制在更新時的移動步數，如果跟上一次更新同向則會加倍反之削減，在每一個neural中的參數都需要維護一個MGD的Optimizer，MGD有一個參數用來調整根據上一次移動的大小偏差，如果越高，那往同一方向的修正時移動量會增加。下方程式碼為MGD的實例：

|  |
| --- |
| class Momentum(Optimizer):  m = 0.9  # v^t = [ learning\_rate \* gradient t=0  # [ mv(t-1) + learning\_rate \* gradient t>=1  #  # theta = theta - v(t)  last\_v = None  def \_\_init\_\_(self,m=0.9):  self.m = m    # calc v^(t)  def calc(self,gradient):  if (self.last\_v is None):  ret = gradient  else:  ret = self.last\_v \* self.m + gradient  self.last\_v = ret  return ret |

相對的在neural network update時也會判斷是否有優化器(optimizer)，有的話則使用該優化器。

|  |
| --- |
| def update(self,learning\_rate):  if (self.weight\_optimizer != None and self.bias\_optimizer!= None ):  self.W -= self.weight\_optimizer.calc(learning\_rate \* self.dW)  self.b -= self.bias\_optimizer.calc(learning\_rate \* self.db)  else:  self.W -= learning\_rate \* self.dW  self.b -= learning\_rate \* self.db |

**B. Implement different activation functions**

Sigmoid存在當輸入值很大時1，或很小時，這兩者的微分 都會偏向於 0，導至梯度消失(Vanishing Gradient)的現象。會導致權重的更新變得緩慢，甚至直接收斂。

而為了避免梯度消失的情況，本實驗報告有額外設計一個激活函數ReLU(Rectified Linear Unit)，函數圖形如圖5.1所示

一張含有 行, 繪圖, 圖表, 螢幕擷取畫面 的圖片

自動產生的描述

圖5.1、ReLU函數圖

ReLU的數學式子如下：

與Sigmoid的導數相比，ReLU本身的導數更容易計算，計算速度上也提升，並且也能防止上述提到的梯度消失的問題，並且ReLU比Sigmoid更利於模型學習稀疏特徵，ReLU的導數式子如下：

|  |
| --- |
| def relu(x):  return x \* (x > 0)  def derivative\_relu(x):  return 1. \* (x > 0) |