

Assignment-1 Report

109503517 通訊三 蔡雅各

1.編譯結果

```
Jacob@jacob-VirtualBox: ~/桌面/hw1
jacob@jacob-VirtualBox:~$ cd /home/jacob/桌面/hw1
jacob@jacob-VirtualBox:~/桌面/hw1$ gcc -o main main.c layer.c neuron.c -ln
jacob@jacob-VirtualBox:~/桌面/hw1$ ls
c.out  iteration_times.txt  layer.h  main  neuron.c  README.md
backprop.h  layer.c  loss.n  main.c  neuron.h  total_cost.txt
jacob@jacob-VirtualBox:~/桌面/hw1$
```

使用 gcc 此命令 compile，並以 ls 確認是否 compile

2.執行結果(目標為 4bitXOR 運算，2 層 hidden layers 及每層各 4 個 neurons)

```
Jacob@jacob-VirtualBox:~/桌面/hw1$ ls
c.out  iteration_times.txt  layer.h  main  neuron.c  README.md
backprop.h  layer.c  loss.n  main.c  neuron.h  total_cost.txt
jacob@jacob-VirtualBox:~/桌面/hw1$ ./main
Enter the number of Layers in Neural Network(including input layer and output layer):
4
Enter number of neurons in Layer[1]:
4
Enter number of neurons in Layer[2]:
4
Enter number of neurons in Layer[3]:
4
Enter number of neurons in Layer[4]:
4

Created Layer: 1
Number of Neurons in Layer 1: 4
Neuron 1 in Layer 1 created
Neuron 2 in Layer 1 created
Neuron 3 in Layer 1 created
Neuron 4 in Layer 1 created

Created Layer: 2
Number of Neurons in Layer 2: 4
Neuron 1 in Layer 2 created
Neuron 2 in Layer 2 created
Neuron 3 in Layer 2 created
Neuron 4 in Layer 2 created

Created Layer: 3
Number of Neurons in Layer 3: 4
Neuron 1 in Layer 3 created
Neuron 2 in Layer 3 created
Neuron 3 in Layer 3 created
Neuron 4 in Layer 3 created

Created Layer: 4
Number of Neurons in Layer 4: 1
Neuron 1 in Layer 4 created

Initializing weights...
0w[0][0]: 0.774985
1w[0][0]: 0.316282
2w[0][0]: 0.823785
3w[0][0]: 0.449451
0w[0][1]: 0.980278
1w[0][1]: 0.351990
2w[0][1]: 0.956818
3w[0][1]: 0.962215
0w[0][2]: 0.210026
1w[0][2]: 0.796179
2w[0][2]: 0.611980
3w[0][2]: 0.405809
0w[0][3]: 0.272334
1w[0][3]: 0.722168
2w[0][3]: 0.473719
3w[0][3]: 0.893649
0w[1][0]: 0.695811
1w[1][0]: 0.244289
2w[1][0]: 0.801819
3w[1][0]: 0.721326
0w[1][1]: 0.791526
1w[1][1]: 0.172162
2w[1][1]: 0.387369
3w[1][1]: 0.238218
0w[1][2]: 0.950672
1w[1][2]: 0.546626
2w[1][2]: 0.610634
3w[1][2]: 0.568298
0w[1][3]: 0.385539
1w[1][3]: 0.884560
2w[1][3]: 0.395939
3w[1][3]: 0.834994
0w[2][0]: 0.747939
1w[2][0]: 0.827054
0w[2][1]: 0.580006
1w[2][1]: 0.563776
Neural Network Created Successfully...
Enter the learning rate (Usually 0.15):
0.15
Enter the number of training examples:
16
```

./main 啟動並輸入 layer 跟 neuron 的數量

```
Jacob@jacob-VirtualBox:~/桌面/hw1$ ./main
Enter the number of Layers in Neural Network(including input layer and output layer):
4
Enter number of neurons in Layer[1]:
4
Enter number of neurons in Layer[2]:
4
Enter number of neurons in Layer[3]:
4
Enter number of neurons in Layer[4]:
4

Created Layer: 1
Number of Neurons in Layer 1: 4
Neuron 1 in Layer 1 created
Neuron 2 in Layer 1 created
Neuron 3 in Layer 1 created
Neuron 4 in Layer 1 created

Created Layer: 2
Number of Neurons in Layer 2: 4
Neuron 1 in Layer 2 created
Neuron 2 in Layer 2 created
Neuron 3 in Layer 2 created
Neuron 4 in Layer 2 created

Created Layer: 3
Number of Neurons in Layer 3: 4
Neuron 1 in Layer 3 created
Neuron 2 in Layer 3 created
Neuron 3 in Layer 3 created
Neuron 4 in Layer 3 created

Created Layer: 4
Number of Neurons in Layer 4: 1
Neuron 1 in Layer 4 created

Initializing weights...
0w[0][0]: 0.774985
1w[0][0]: 0.316282
2w[0][0]: 0.823785
3w[0][0]: 0.449451
0w[0][1]: 0.980278
1w[0][1]: 0.351990
2w[0][1]: 0.956818
3w[0][1]: 0.962215
0w[0][2]: 0.210026
1w[0][2]: 0.796179
2w[0][2]: 0.611980
3w[0][2]: 0.405809
0w[0][3]: 0.272334
1w[0][3]: 0.722168
2w[0][3]: 0.473719
3w[0][3]: 0.893649
0w[1][0]: 0.695811
1w[1][0]: 0.244289
2w[1][0]: 0.801819
3w[1][0]: 0.721326
0w[1][1]: 0.791526
1w[1][1]: 0.172162
2w[1][1]: 0.387369
3w[1][1]: 0.238218
0w[1][2]: 0.950672
1w[1][2]: 0.546626
2w[1][2]: 0.610634
3w[1][2]: 0.568298
0w[1][3]: 0.385539
1w[1][3]: 0.884560
2w[1][3]: 0.395939
3w[1][3]: 0.834994
0w[2][0]: 0.747939
1w[2][0]: 0.827054
0w[2][1]: 0.580006
1w[2][1]: 0.563776
Neural Network Created Successfully...
Enter the learning rate (Usually 0.15):
0.15
Enter the number of training examples:
16
```

輸入 learning rate 及 training data 的數量

```
Jacob@jacob-VirtualBox:~/桌面/hw1$ ./main
Enter the number of Layers in Neural Network(including input layer and output layer):
4
Enter number of neurons in Layer[1]:
4
Enter number of neurons in Layer[2]:
4
Enter number of neurons in Layer[3]:
4
Enter number of neurons in Layer[4]:
4

Created Layer: 1
Number of Neurons in Layer 1: 4
Neuron 1 in Layer 1 created
Neuron 2 in Layer 1 created
Neuron 3 in Layer 1 created
Neuron 4 in Layer 1 created

Created Layer: 2
Number of Neurons in Layer 2: 4
Neuron 1 in Layer 2 created
Neuron 2 in Layer 2 created
Neuron 3 in Layer 2 created
Neuron 4 in Layer 2 created

Created Layer: 3
Number of Neurons in Layer 3: 4
Neuron 1 in Layer 3 created
Neuron 2 in Layer 3 created
Neuron 3 in Layer 3 created
Neuron 4 in Layer 3 created

Created Layer: 4
Number of Neurons in Layer 4: 1
Neuron 1 in Layer 4 created

Initializing weights...
0w[0][0]: 0.774985
1w[0][0]: 0.316282
2w[0][0]: 0.823785
3w[0][0]: 0.449451
0w[0][1]: 0.980278
1w[0][1]: 0.351990
2w[0][1]: 0.956818
3w[0][1]: 0.962215
0w[0][2]: 0.210026
1w[0][2]: 0.796179
2w[0][2]: 0.611980
3w[0][2]: 0.405809
0w[0][3]: 0.272334
1w[0][3]: 0.722168
2w[0][3]: 0.473719
3w[0][3]: 0.893649
0w[1][0]: 0.695811
1w[1][0]: 0.244289
2w[1][0]: 0.801819
3w[1][0]: 0.721326
0w[1][1]: 0.791526
1w[1][1]: 0.172162
2w[1][1]: 0.387369
3w[1][1]: 0.238218
0w[1][2]: 0.950672
1w[1][2]: 0.546626
2w[1][2]: 0.610634
3w[1][2]: 0.568298
0w[1][3]: 0.385539
1w[1][3]: 0.884560
2w[1][3]: 0.395939
3w[1][3]: 0.834994
0w[2][0]: 0.747939
1w[2][0]: 0.827054
0w[2][1]: 0.580006
1w[2][1]: 0.563776
Neural Network Created Successfully...
Enter the learning rate (Usually 0.15):
0.15
Enter the number of training examples:
16
Enter the Inputs for training example[1]:
0 0 0
Enter the Inputs for training example[2]:
0 0 1
Enter the Inputs for training example[3]:
0 0 1 1
Enter the Inputs for training example[4]:
0 1 0 0
Enter the Inputs for training example[5]:
0 1 0 1
Enter the Inputs for training example[6]:
0 1 1 0
Enter the Inputs for training example[7]:
0 1 1 1
Enter the Inputs for training example[8]:
1 0 0 0
Enter the Inputs for training example[9]:
1 0 0 1
Enter the Inputs for training example[10]:
1 0 1 0
Enter the Inputs for training example[11]:
1 0 1 1
Enter the Inputs for training example[12]:
1 1 0 0
Enter the Inputs for training example[13]:
1 1 0 1
Enter the Inputs for training example[14]:
1 1 1 0
Enter the Inputs for training example[15]:
1 1 1 1
```

輸入 input，根據[x]輸入其二進制之 x 值##

```

Jacob@jacob-VirtualBox: ~/桌面/hw1
Enter the Desired Outputs (Labels) for training example[1]:
0
Enter the Desired Outputs (Labels) for training example[2]:
1
Enter the Desired Outputs (Labels) for training example[3]:
0
Enter the Desired Outputs (Labels) for training example[4]:
1
Enter the Desired Outputs (Labels) for training example[5]:
0
Enter the Desired Outputs (Labels) for training example[6]:
0
Enter the Desired Outputs (Labels) for training example[7]:
1
Enter the Desired Outputs (Labels) for training example[8]:
1
Enter the Desired Outputs (Labels) for training example[9]:
0
Enter the Desired Outputs (Labels) for training example[10]:
0
Enter the Desired Outputs (Labels) for training example[11]:
1
Enter the Desired Outputs (Labels) for training example[12]:
0
Enter the Desired Outputs (Labels) for training example[13]:
1
Enter the Desired Outputs (Labels) for training example[14]:
1
Enter the Desired Outputs (Labels) for training example[15]:
0

```

輸入其對應的輸出值

```

Jacob@jacob-VirtualBox: ~/桌面/hw1
-----
Input: 1.000000
Input: 0.000000
Output: 1

Input: 1.000000
Input: 1.000000
Input: 1.000000
Input: 1.000000
Output: 1

Input: 0.000000
Input: 0.000000
Input: 0.000000
Input: 0.000000
Output: 0

Input: 0.000000
Input: 0.000000
Input: 0.000000
Input: 1.000000
Output: 1

Input: 0.000000
Input: 0.000000
Input: 1.000000
Input: 0.000000
Output: 1

Input: 0.000000
Input: 0.000000
Input: 1.000000
Input: 1.000000
Output: 0

Input: 0.000000
Input: 1.000000
Input: 0.000000
Input: 0.000000
Output: 1

Input: 0.000000
Input: 1.000000
Input: 0.000000
Input: 1.000000
Output: 0

```

訓練過程，設定為 10000 次 epochs

```

Jacob@jacob-VirtualBox: ~/桌面/hw1
Enter input to test:
0 1 0 0
Output: 1

Enter input to test:
0 1 0 1
Output: 0

Enter input to test:
0 1 1 0
Output: 0

Enter input to test:
0 1 1 1
Output: 1

Enter input to test:
1 0 0 0
Output: 1

Enter input to test:
1 0 0 1
Output: 0

Enter input to test:
1 0 1 0
Output: 0

Enter input to test:
1 0 1 1
Output: 1

Enter input to test:
1 1 0 0
Output: 0

Enter input to test:
1 1 0 1
Output: 1

Enter input to test:
1 1 1 0
Output: 1

Enter input to test:
1 1 1 1

```

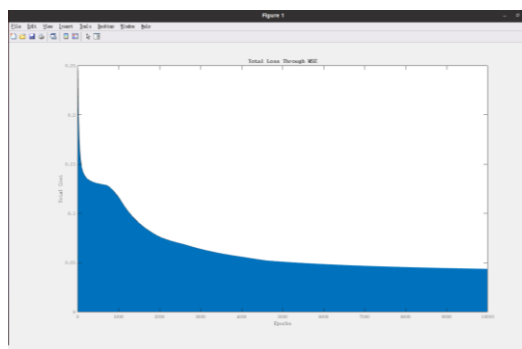
測試訓練結果，正確率來到 94%(15/16)

##由於此開源是將所有可能值當成 training data，因此 4bit 的 input data 為 15 個

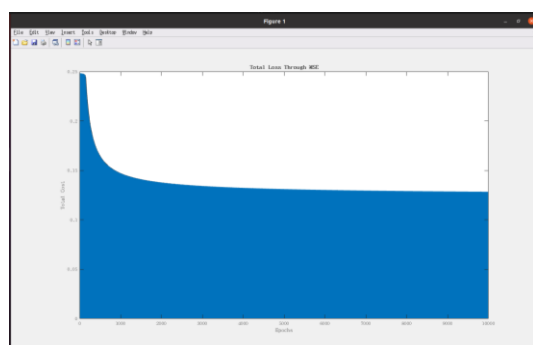
3.分析

底下的表格是我用 4bit 運算 XOR 訓練後的結果，很巧的是 run 三次竟然都一樣，旁邊的圖是我寫檔紀錄 total cost 在 epoch 數的變化，並以 matlab 作圖呈現。三次的 run 都是參考開源的訓練方式(2 hidden layers, 4 neurons per hidden layer, lr=0.15, 10000 epochs)

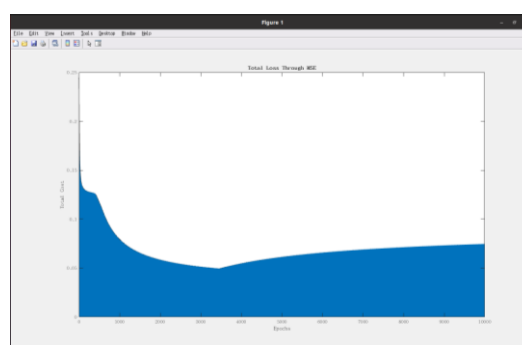
1	4-bit XOR operation with 2 hidden layers, 4 neurons per hidden layer, lr=0.15, 10000 epochs					
2	input	XOR	1st test	2nd test	3rd test	
3	0	0	0	0	0	
4	1	1	1	1	1	
5	2	1	1	1	1	
6	3	0	0	0	0	
7	4	1	1	1	1	
8	5	0	0	0	0	
9	6	0	0	0	0	
10	7	1	1	1	1	
11	8	1	1	1	1	
12	9	0	0	0	0	
13	10	0	0	0	0	
14	11	1	1	1	1	
15	12	0	0	0	0	
16	13	1	1	1	1	
17	14	1	1	1	1	
18	15	0	1	1	1	
19	Accuracy(%)	X	93.75	93.75	93.75	



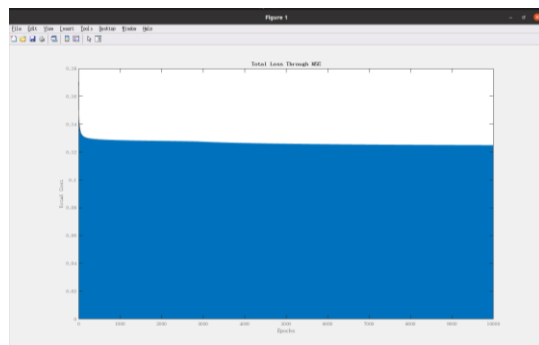
上圖為對照組，底下為實驗組，六張 **loss function** 作圖是改變不同條件所做的 **total cost** 的圖，第一組為 **learning rate** 改變，第二組為 **hidden layer** 的 **neuron** 數比較，第三組為 **hidden layer** 的層數改變，具體改變均標示在圖下方。



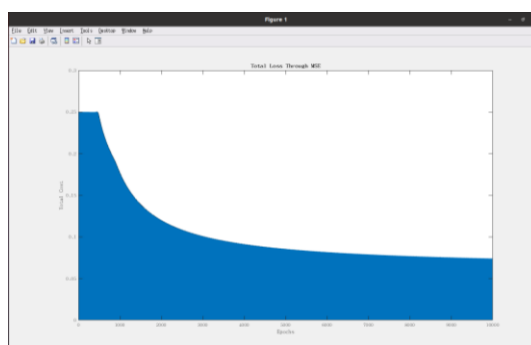
Learning rate = 0.01



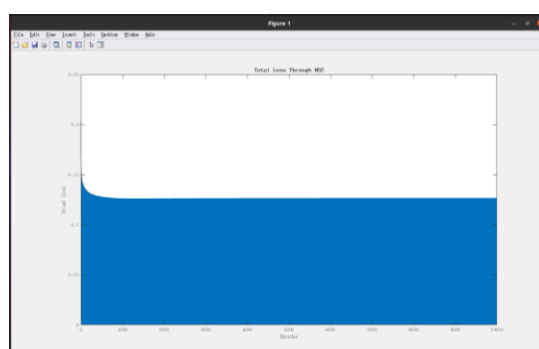
Learning rate = 0.3



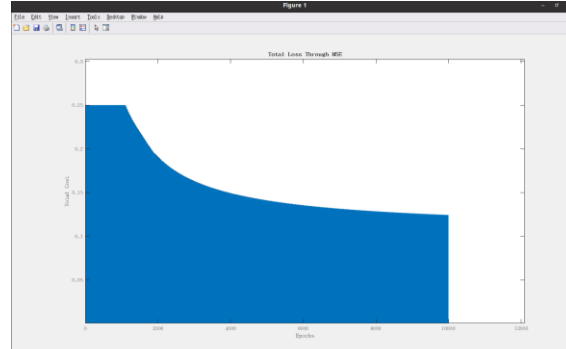
Number of neuron = 2 per hidden layer



Number of neuron = 8 per hidden layer



1 hidden layer



4 hidden layers

透過上方的實驗組，驗證了我所學到的 **deep learning** 的知識，像是適當的 **learning rate** 是很重要的，不然 **loss** 反而會更大。又像是 **neuron** 數越多會讓降低 **loss** 的效率更好。

4.問題探討

(1)Linux 指令的運用

平常不常使用 Linux 的環境，等於是重新複習終端機的使用。對於指令的使用也爬了很多文，至少 10 幾個分頁是必須的(install gcc, vim, matlab, compiling .c ... and more)。

(2)學習程式的架構

這次的程式碼是來自 open source，看懂程式碼就算是把程式語法抓一點感覺回來，而且比較特別的是因為是做 neural network，剛好跟我專題在看的東西幾乎一樣，就憑著 deep learning 的知識去拆解他的函式，搭配網站的解說分析，讓我能快速了解程式的運行並給予註解，甚至我修復了一個他的 cost function 的 bug，他使用 Mean Squared Error 來算 total cost，但他在疊代的過程忘了把前面的 mean 消掉，如下圖第 315 行。這讓我覺得蠻有成就感的，感覺學到的東西派上用場了。

```
304 // Compute Total Cost
305 void compute_cost(int i)
306 {
307     int j;
308     float tmpcost=0;
309     float tcost=0;
310
311     for(j=0;j<num_neurons[num_layers-1];j++)
312     {
313         tmpcost = desired_outputs[i][j] - lay[num_layers-1].neu[j].activ;
314         cost[j] = (tmpcost * tmpcost)/2;
315         tcost = tcost + cost[j];
316     }
317
318     full_cost = (full_cost + tcost)/n;
319     n++;
320     // printf("Full Cost: %f\n",full_cost);
321 }
322
```

(3)訓練的不穩定性

從我的分析部分中可以看到使用開源的範例的運算效率是最好的，大部分測試 MSE 都能壓在 0.05 以內。但浮動的狀況仍然會有至 0.1 附近就開始收斂，這我並不確定式甚麼原因造成的，但根據我微薄的知識推測是在 gradient descent 遇到 local minimum 或 saddle point 而跑不出來嗎?但收斂區只要根據原本開源的 instruction 就都能在 0.15 以下收斂，算是還可以接受的結果吧。