陽明 大一大二不分系二年級 10612012 張凱博

(一) Introduction

Backpropagation在Artificial Neural Networks上屬於supervised learning。Feed-forward neural networks的構想起源於人類神經網絡系統，而Backpropagation正是建立在Feed-forward neural networks之上的，當有一個define好的network structure，backpropogation藉由計算loss function的gradient，train出在fully connected multilayer feed-forward neural network裡的weight，得到趨近於convergence的weight (gradient descent optimization algorithm)。

其原理是當Neural Network有Output的時候，計算它和Output期望值的Error，再把error和前一層layer的activation function(output)的微分值相乘，然後在依序和每個neuron連接的weight相乘算總和，乘上再前一層以output為argument的layer的actication function，照這樣一層層由後往前累加計算，利用程式計算backpropagation過程當中的errors記起來後，update每一層中所有的weight，重複累積上述過程，就完成backpropagation。

(二) Experment setups

A. Sigmoid functions

設sigmoid(activation) = 1.0 / (1.0 + exp(-activation))，activation是在做forward propagation時neuron輸出的output，是那一顆neuron前一層layer的input和weight的summation。

```
In [5]: def sigmoid(activation):
            return 1.0 / (1.0 + exp(-activation))
```

```
In [7]: def derivative_sigmoid(output):
            return output * (1.0 - output)
```

B. Neuron Network

創造出一個Network神經網絡，設定好裡面hidden layer的層數，裡面的內容物是一個 fully connected weight的dictionary，之後會再藉由輸入input，就可以和Network裡面的weights相乘計算summation，進而完成forward propagation得到outputs。

```
In [11]: def initialize_network(n_inputs, n_hidden1, n_hidden2, n_outputs):
             network = list()
             hidden_layer1 = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden1)]
             network.append(hidden_layer1)
             hidden_layer2 = [{'weights':[random() for i in range(n_hidden1 + 1)]} for i in range(n_hidden2)]
             network.append(hidden_layer2)
             output_layer = [{'weights':[random() for i in range(n_hidden1 + 1)]} for i in range(n_outputs)]
             network.append(output_layer)
             return network
```

C. Backpropagation

若要完成backpropagation，首先要先完成forward propagation，再利用迴圈計算多次backpropagation後重複更新每一層的weight。

```
In [13]: def back_propagation(dataset, n_inputs, n_hidden1, n_hidden2, n_outputs, l_rate, n_epoch):
             network = initialize_network(n_inputs, n_hidden1, n_hidden2, n_outputs)
             train_network(network, dataset, l_rate, n_epoch, n_outputs)
```

```python
In [10]: # Train a network for a fixed number of epochs
def train_network(network, dataset, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        sum_error = 0
        outputs_copy = []
        for row in dataset:
            outputs = forward_propagate(network, row)
            outputs_copy = outputs.copy()
            #outputlayer有2個neuron，若dataset中ground truth為1的標註為[1,0]，0的標註[0, 1]
            expected = [0 for i in range(n_outputs)]
            expected[int(row[-1])] = 1
            sum_error += sum([(expected[i]-outputs[i]) for i in range(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        if epoch%1000 == 0:
            print('epoch'+ str(epoch) + 'loss : '+ str(sum_error))
            print(outputs_copy)
```

(三) Results of my testing

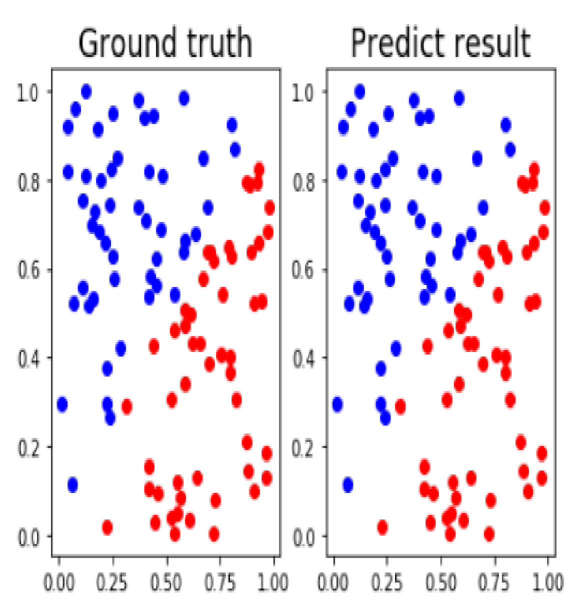
A. Screenshot and comparison figure

1. generate_linear

```
 epoch 0 loss : -25.860381035674262
[0.6227626003785245, 0.3847236059170465]
 epoch 1000 loss : 0.024716568068571136
[0.0004886881395179133, 0.9995764368285838]
 epoch 2000 loss : 0.0026192536373801987
[0.0004095959584610438, 0.9996650921219874]
 epoch 3000 loss : 0.001355726722737698
[0.0002939575416930025, 0.9997605800181882]
 epoch 4000 loss : 0.0009626673260881682
[0.0002343004836374635, 0.9998092495964587]
 epoch 5000 loss : 0.0007538276059417871
[0.00019737152957477184, 0.9998393060386144]
 epoch 6000 loss : 0.0006222696680805406
[0.00017194794816817018, 0.9998599812854514]
 epoch 7000 loss : 0.0005313154312109165
[0.00015322609729700128, 0.9998752018301342]
 epoch 8000 loss : 0.0004645024152529901
[0.0001387825938269713, 0.9998869431669987]
 epoch 9000 loss : 0.0004132609301088283
[0.00012725262706159862, 0.9998963162580417]
 epoch 10000 loss : 0.0003726689322239384
[0.00011780452299934697, 0.9999039975209338]
 epoch 11000 loss : 0.0003396906742203424
[0.00010990048208590478, 0.9999104241358866]
 epoch 12000 loss : 0.00031234913019500144
[0.00010317621046531534, 0.9999158921463114]
 epoch 13000 loss : 0.00028930009753718627
[9.737555164478182e-05, 0.9999206096781095]
 epoch 14000 loss : 0.00026959705700809605
[9.23127658691123e-05, 0.9999247276274027]
 epoch 15000 loss : 0.0002525538467572889
[8.784966064570236e-05, 0.9999283582564834]
 epoch 16000 loss : 0.0002376606768487885
[8.38811437467768e-05, 0.999931586936815]
```


```
 epoch 16000 loss : 0.0002376606768487885
[8.38811437467768e-05, 0.999931586936815]
```


```
 epoch 17000 loss : 0.00022453074301599229
[8.032577430418899e-05, 0.9999344798273633]
 epoch 18000 loss : 0.00021286515962473781
[7.71193961640992e-05, 0.9999370890472352]
 epoch 19000 loss : 0.0002024292639008746
[7.421073592830976e-05, 0.9999394562515965]
```

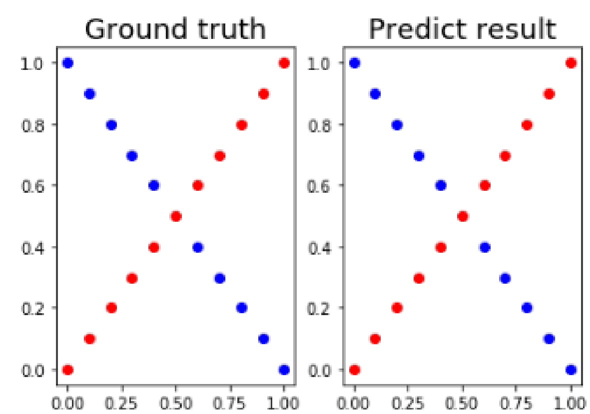


2. generate_XOR_easy

```
 epoch 0 loss : -14.40875497552383
[0.7637095713849967, 0.768760276275519]
 epoch 1000 loss : 0.0019657768949963295
[0.5301747475406169, 0.4706030831562733]
 epoch 2000 loss : 0.01120451769838627
[0.4437409217323699, 0.5574586143439919]
 epoch 3000 loss : 0.0009239132644730703
[0.03937292409906541, 0.9606813651387229]
 epoch 4000 loss : -0.006883922080516814
[0.01178606359606466, 0.9884296841929824]
 epoch 5000 loss : -0.0014392229490558784
[0.005946925811866545, 0.9940443783290669]
 epoch 6000 loss : -0.000840343121854137
[0.004490611603310708, 0.9954830224357116]
 epoch 7000 loss : -0.0006048130546613829
[0.0037714115185233854, 0.9961978078178312]
 epoch 8000 loss : -0.00047713811739026244
[0.003324322668612606, 0.9966436541967325]
 epoch 9000 loss : -0.0003964253131915566
[0.003012087929571502, 0.9969557365327482]
 epoch 10000 loss : -0.0003405147112427576
[0.0027780808621902385, 0.9971900302823615]
 epoch 11000 loss : -0.00029935348422419707
[0.0025941796491698956, 0.9973744057102358]
 epoch 12000 loss : -0.00026770273611333655
[0.0024446430955425667, 0.9975244937825221]
 epoch 13000 loss : -0.00024255646352768732
[0.002319886543073549, 0.997649826990245]
 epoch 14000 loss : -0.00022206291170730994
[0.0022136970396965435, 0.9977565925270605]
```


```
[0.0024446430955425667, 0.9975244937825221]
 epoch 13000 loss : -0.00024255646352768732
[0.002319886543073549, 0.997649826990245]
 epoch 14000 loss : -0.00022206291170730994
[0.0022136970396965435, 0.9977565925270605]
```


```
 epoch 15000 loss : -0.00020501723109850437
[0.002121847284920717, 0.9978490048400503]
 epoch 16000 loss : -0.00019060064183282232
[0.0020413476483939355, 0.9979300473390378]
 epoch 17000 loss : -0.00017823665890738484
[0.0019700159722881023, 0.9980018994870855]
 epoch 18000 loss : -0.00016750726529617008
[0.001906217051983345, 0.9980661957025685]
 epoch 19000 loss : -0.0001581017219621512
[0.0018486979972072463, 0.9981241891238845]
```



B. Caculate accuracy

1. generate_linear

```
In [50]: Round=0
         count_truth=0
         ground_truth = [int(raw[-1])for raw in dataset1]
         for i in predictions:
             if i == ground_truth[Round]:
                 count_truth += 1
             Round += 1
         print(count_truth/len(predictions))
```

```
1.0
```

2. generate_XOR_easy

```
In [54]: Round=0
         count_truth=0
         ground_truth = [int(raw[-1])for raw in dataset2]
         for i in predictions:
             if i == ground_truth[Round]:
                 count_truth += 1
             Round += 1
         print(count_truth/len(predictions))

1.0
```

<Conclusion>

無論是generate_linear或是generate_XOR_easy，正確率皆為100%。

(四) Discussion

這次實作對我來說，最難的是將backpropagation的數學表達式具體了解，並且轉換為程式碼，特別是對完全沒有修過線代、機率和機器學習的我，完全是從零開始，但是我還是花了約30個小時的時間，上網找資料，一行行閱讀別人之前寫有關backpropagation的程式碼還有數學式的一部部推導，雖然花了很久，也debug了無數多次，可是我真的學到了很多，下列附圖是在學習backpropagation所作的筆記。