0516003 李智嘉 DL Lab1 report

1. Introduction

現在既有的 Deep Learning 套件都已經內建自動 back propagation 的功能，直接使用的話很容易會忘記 DL 最根本的技巧。

這次 Lab 希望我們只用 numpy 或其他套件來實做出 back propagation，目的是熟悉其原理，不至於只會套現成套件的 function，而是懂內部實際如何運作的。

這次的作業我將其包成一個 class，可以建制任意層且任意維度的 neural network，activation function 為 sigmoid function，input 是多個二維的點，output 為這些點的類別，只有紅和藍兩類。

比較特別的是，為符合這次作業要求，在 training 階段達到所有點都分類完成之後，就會中斷 training，以此節省使用者時間。

2. Experiment setups

A. Sigmoid functions

$$y = \mathrm{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\frac{dy}{dx} = -(1+e^{-x})^{-2} \cdot (-e^{-x}) = \frac{e^{-x}}{1+e^{-x}} \cdot \frac{1}{1+e^{-x}} = (1-\sigma(x))\,\sigma(x)$$

sigmoid 與其 derivative 推導如右圖

而因為 derivative_sigmoid 傳入的參數為已經經過 sigmoid 的值，所以就將 sigmoid(x)代換成 x 了

```python
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))
def derivative_sigmoid(x):
    return np.multiply(x, 1.0-x)
```

B. Neural network

整個 neural network 分成三個部份，initial, fit data 和 predict data

```python
57 class Sigmoid_Network():
58     def __init__(self, layers):
59         self.activation = sigmoid
60         self.der_act = derivative_sigmoid
61         self.loss = loss
62         self.der_loss = der_loss
63         self.weights = []
64 
65         l = len(layers)
66         for i in range(1, l-1):
67             self.weights.append(np.random.uniform(0, 1, (layers[i-1]+1, layers[i]+1)))  # plus bias term
68         self.weights.append(np.random.uniform(0, 1, (layers[-2]+1, layers[-1])))
```

class 的 initial 先建制好 activation function, derivative activation function, loss function, derivative loss function 以及 weights，而 weights 可根據輸入的 layers 數來做調整。我還再多加上 bias term 到 output layer 以外的每一層，可以參考 66~68 行

這邊我 default 的架構是 [2, 4, 4, 1]，也就是 input 2 維，output 1 維，中間兩層 4 維的

hidden layer，主程式如右圖

```python
if __name__ == '__main__':
    x, y_true = generate_linear(500)
    #x, y_true = generate_XOR_easy()
    nn = Sigmoid_Network([2,4,4,1])
    nn.fit(x, y_true)
    y, _ = nn.predict(x, y_true)
    print(y)
    show_result(x, y_true, y)
```

C. Backpropagation

為求邏輯通順，這邊將 fit data 的三個步驟，forward passing, backpropagation 以及 update parameter 一起說明

```python
    def fit(self, X, y, Lr=0.1, epochs=100000, print_per_epoch=5000):
        ones = np.ones(len(X)).reshape(-1, 1)
        X = np.concatenate((X, ones), axis=1)

        for time in range(epochs):      # for every epochs
            for x,y_true in zip(X,y):         # for every single data
                # forward
                output = [x]
                for l_idx in range(len(self.weights)):
                    dot = output[l_idx].dot(self.weights[l_idx])
                    act = self.activation(dot)
                    output.append(act)
                output = np.array(output)

                # backpropagation
                deltas = []     # store deltas for weights update
                prev = []       # store previous gradient for backpropagation
                product = self.der_loss(output[-1], y_true)*self.der_act(output[-1])
                prev.append(product)
                delt = output[-2].reshape(-1, 1).dot(product.reshape(1, -1))
                deltas.append(delt)
                for i in range(len(output)-2, 0, -1):
                    product = self.weights[i].dot(prev[-1])*self.der_act(output[i])
                    prev.append(product)
                    delt = output[i-1].reshape(-1, 1).dot(product.reshape(1, -1))
                    deltas.append(delt)
                deltas.reverse()
                deltas = np.array(deltas)

                # update
                for i in range(len(self.weights)):
                    self.weights[i] -= Lr * deltas[i]
```

$$X = \begin{bmatrix} x_{11} & x_{12} & 1 \\ x_{21} & x_{22} & 1 \\ \vdots & \vdots & \vdots \end{bmatrix} \qquad x = [x_{11} \;\; x_{12} \;\; 1], \; y\_true = ground\_truth$$

forward:

前一層 output vector 與 該層 weight 作矩陣乘，

通過 activation function，並更新 output

backpropagation:

output layer delta:

$$a^{(2)} W^{(3)} = z^{(3)}$$

$$\Rightarrow \sigma(z^{(3)}) = y$$

$$\Rightarrow loss = y - y\_truth$$

$$\frac{d\,loss}{dy} \frac{dy}{dz^{(3)}} \frac{dz^{(3)}}{dw^{(3)}} = \text{self.der\_loss}(\text{output}[-1],\ y\_true) \cdot \text{self.der\_activation}(\text{output}[-1]) \cdot a^{(2)}$$

$$= \text{product} \cdot a^{(2)} = \text{delta}$$

(為使 delta 維度與 weight 對應，將會是 [prev][current])

hidden layer delta:

$$\underbrace{\frac{d\,loss}{dy} \frac{dy}{dz^{(3)}}}\ \underbrace{\frac{dz^{(3)}}{da^{(2)}}}\ \underbrace{\frac{da^{(2)}}{dz^{(2)}}}\ \frac{dz^{(2)}}{dw^{(2)}}$$

$$= \underbrace{\text{previous\_product} \cdot W^{(3)} \cdot \text{self.der\_activation(output)}} \cdot a^{(1)}$$

$$= \text{current\_product} \cdot a^{(1)}$$

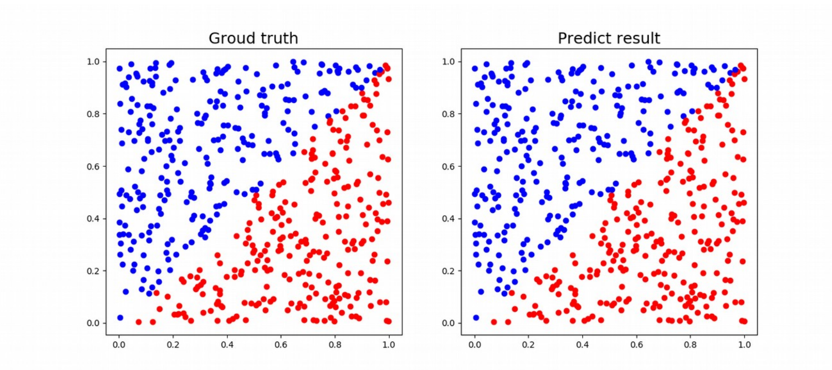
$$= \text{delta}$$

update:

weight − LearningRate · delta

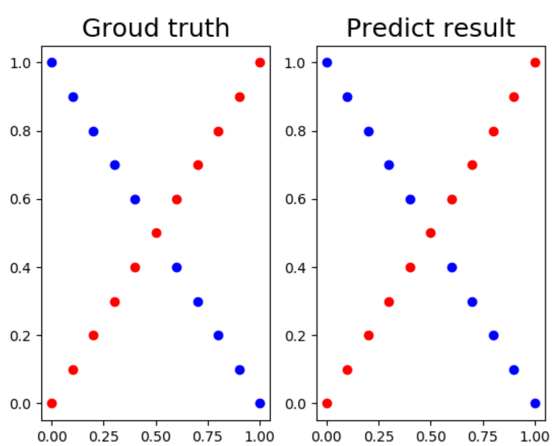
3. Results of testing

Classify 500 Linear Dots with 0.1 learning rate :




```
(kk) karljackab@pc3421:~/DL$ python3 lab1_0516003.py
epoch 0 loss : 0.12512503820555415
epoch 5000 loss : 0.0036539688296786544
epoch 10000 loss : 0.002174803590383311
epoch 15000 loss : 0.0012198091230990553
epoch 20000 loss : 1.6743046774241587e-05
All points are classified correctly, break training
[[6.25484401e-09]
 [4.42123012e-08]
 [6.79123190e-09]
 [9.99999980e-01]
 [6.94010508e-09]
 [1.02468923e-08]
 [9.99998227e-01]
 [9.99999993e-01]
 [1.81402729e-08]
 [6.27122341e-09]
 [6.51040058e-09]
 [6.86377691e-09]
 [9.99999983e-01]
 [9.99999993e-01]
 [6.37462740e-09]
 [2.62084281e-08]
 [6.25546429e-09]
 [6.25726515e-09]
 [9.99999980e-01]
 [6.25319290e-09]
 [3.86374943e-05]
 [9.99999991e-01]
 [9.99999993e-01]
 [6.49515738e-09]
 [9.99999992e-01]
```

Classify XOR Dots with 0.1 learning rate :




```
(kk) karljackab@pc3421:~/DL$ python3 lab1_0516003.py
epoch 0 loss : 0.21618295052564745
epoch 5000 loss : 0.12451621825218709
epoch 10000 loss : 0.021769653682732256
epoch 15000 loss : 0.0017340014850575364
All points are classified correctly, break training
[[1.58308128e-05]
 [9.98788476e-01]
 [9.23948700e-06]
 [9.99352448e-01]
 [7.91471596e-06]
 [9.99480285e-01]
 [2.23982477e-05]
 [9.98576563e-01]
 [1.36020623e-02]
 [8.67147954e-01]
 [1.89693790e-01]
 [1.14929971e-02]
 [8.62637862e-01]
 [5.57586239e-04]
 [9.97867809e-01]
 [8.90073921e-05]
 [9.99675932e-01]
 [3.64705632e-05]
 [9.99833444e-01]
 [2.45873974e-05]
 [9.99872650e-01]]
```

Compare Different Learning Rate

Linear Dots

```
(kk) karljackab@pc3421:~/DL$ python3 lab1_0516003.py
epoch 0 loss : 0.12793071203767317
epoch 5000 loss : 0.00016770773021384519
All points are classified correctly, break training
[[3.39510404e-06]
 [9.99997336e-01]
 [3.26624107e-06]
 [3.22986591e-06]
 [3.36996934e-06]
 [2.98994816e-06]
 [9.99998921e-01]
 [9.99998983e-01]
 [9.99999106e-01]
 [9.99998447e-01]
 [9.99997848e-01]
 [8.73861069e-06]
 [9.99993010e-01]
 [9.99999102e-01]
 [3.06830625e-06]
 [9.99999107e-01]
 [9.99998848e-01]
 [3.53645065e-06]
 [9.99999058e-01]
 [9.99998641e-01]
 [9.99999108e-01]
 [9.99974602e-01]
 [2.47782999e-02]
 [3.44848033e-06]
 [9.99999038e-01]
 [9.99998311e-01]
 [9.99999107e-01]
 [9.99998571e-01]
 [9.94624041e-01]
 [9.99999102e-01]
```

```
(kk) karljackab@pc3421:~/DL$ python3 lab1_0516003.py
epoch 0 loss : 0.19667583185759935
epoch 5000 loss : 0.0027708225502172923
All points are classified correctly, break training
[[9.99243219e-01]
 [2.06119643e-04]
 [4.56185968e-04]
 [9.99939029e-01]
 [9.99973003e-01]
 [1.68849791e-04]
 [9.99968245e-01]
 [9.99972041e-01]
 [3.41744065e-04]
 [9.99963472e-01]
 [1.41478937e-02]
 [9.97805565e-01]
 [9.99974949e-01]
 [9.99632080e-01]
 [2.31848577e-02]
 [9.99803585e-01]
 [9.99965192e-01]
 [4.45159854e-04]
 [3.20086233e-03]
 [9.99647896e-01]
 [1.68469642e-04]
 [9.98486919e-01]
 [9.98797999e-01]
 [9.99763925e-01]
 [2.49375971e-04]
 [9.98000387e-01]
 [1.72013349e-04]
 [2.01091203e-04]
 [1.89866957e-03]
```

XOR Dots

```
(kk) karljackab@pc3421:~/DL$ python3 lab1_0516003.py
epoch 0 loss : 0.12512503820555415
epoch 5000 loss : 0.0036539688296786544
epoch 10000 loss : 0.002174803590383311
epoch 15000 loss : 0.0012198091230990553
epoch 20000 loss : 1.6743046774241587e-05
All points are classified correctly, break training
[[6.25484401e-09]
 [4.42123012e-08]
 [6.79123190e-09]
 [9.99999980e-01]
 [6.94010508e-09]
 [1.02468923e-08]
 [9.99998227e-01]
 [9.99999993e-01]
 [1.81402729e-08]
 [6.27122341e-09]
 [6.51040058e-09]
 [6.86377691e-09]
 [9.99999983e-01]
 [9.99999993e-01]
 [6.37462740e-09]
 [2.62084281e-08]
 [6.25546429e-09]
 [6.25726515e-09]
 [9.99999980e-01]
 [6.25319290e-09]
 [3.86374943e-05]
 [9.99999991e-01]
 [9.99999993e-01]
 [6.49515738e-09]
 [9.99999992e-01]
```

```
(kk) karljackab@pc3421:~/DL$ python3 lab1_0516003.py
epoch 0 loss : 0.24648443717366328
epoch 5000 loss : 0.12472589571375879
epoch 10000 loss : 0.12471419029048858
epoch 15000 loss : 0.12469310234236299
epoch 20000 loss : 0.1246417510111126
epoch 25000 loss : 0.12445272602036025
epoch 30000 loss : 0.1219839057719225
epoch 35000 loss : 0.02315049506265124
All points are classified correctly, break training
[[0.00935673]
 [0.97983552]
 [0.01586393]
 [0.97343478]
 [0.04797573]
 [0.94891713]
 [0.164782  ]
 [0.83362858]
 [0.3138513 ]
 [0.5133242 ]
 [0.3489438 ]
 [0.29380649]
 [0.50805816]
 [0.21586688]
 [0.81522029]
 [0.15151652]
 [0.94381504]
 [0.107615  ]
 [0.97461656]
 [0.07949595]
 [0.98333192]]
```

上方左圖皆為 0.1 learning rate，右圖為 0.02 learning rate

可以看到在 Linear mode 時，完成所需的 epoch 差別不大，但是 XOR mode 時就有滿大的差別。可能是因為 Linear 相對 XOR 來說是簡單很多的，就算 learning rate 很小，也可以很快就收斂。

4. Discussion

其實之前看了許多 DL 相關的 paper，都把直接 back propagation 當作習以為常的東西，這次實際的手刻後才發現實做起來這麼複雜。

除了其中數學的推導，在 back propagate 時的維度也花了許多時間判斷。最後還是先把第一版寫死的 code 刪掉之後，重寫包成 class 的版本，才用比較架構的方式寫完。

很高興有這個機會可以練習，期待之後的 lab。