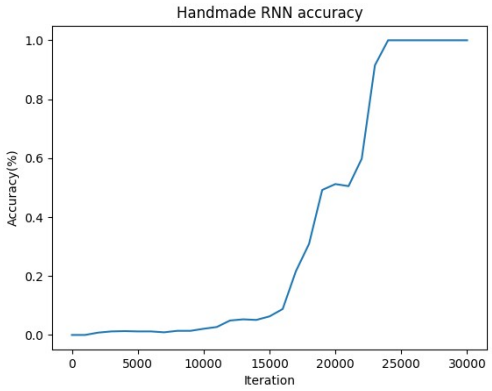
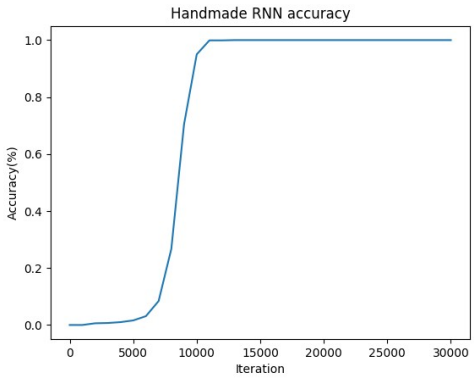
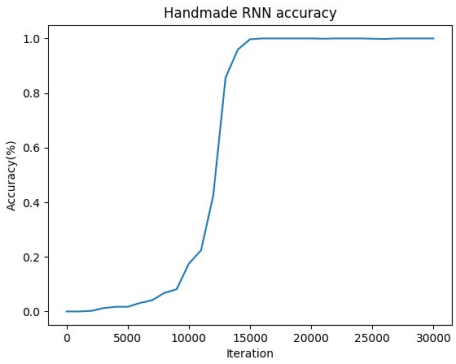
0516003 李智嘉 lab4 report

1. Plot

這邊我跑了三次，可以發現開始收斂時大部分都是急速的收斂，但開始收斂的時間都不太一樣，應該跟 data 每次得到的順序不一樣有關係







2. How to generate data

首先我先分別對 x 和 y 隨機取從 -128 ~ 127 的一個整數，再用 numpy 套件讓他變成 binary 來表示，而因為回傳值為一個 string 的 list，所以又把他的每個元素變成 integer。再把兩數包在一起就完成 data 了

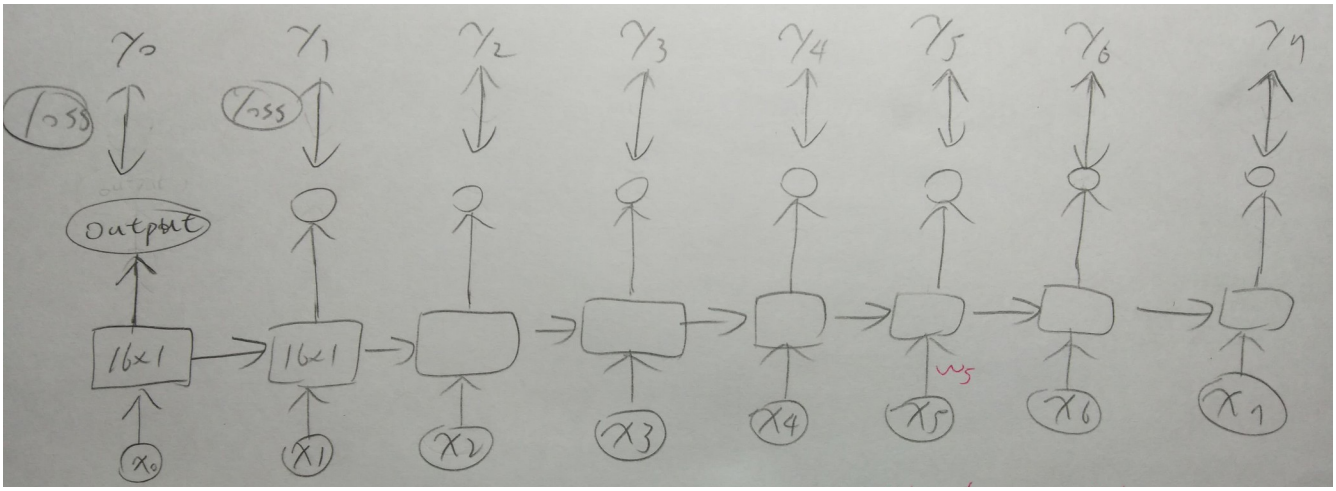
再來結果則是相加之後，再用一樣的方式轉換成 binary，但他有可能會溢位，這邊我的處理方式是直接取最低的 8 位

而由於後面有要比較 predict 結果以及 ground_truth 的關係，這邊我直接轉換成 numpy array 了

code 如下

```python
def get_data(bits=8):
    x = np.random.randint(-128, 128)
    x_num = np.binary_repr(x, width=bits)
    x_num = [int(i) for i in x_num]

    y = np.random.randint(-128, 128)
    y_num = np.binary_repr(y, width=bits)
    y_num = [int(i) for i in y_num]

    num = [[x, y] for x, y in zip(x_num, y_num)]

    res = x + y
    res_num = np.binary_repr(res, width=bits)
    res_num = np.array([int(i) for i in res_num][-1:-9:-1])[::-1]

    # num is a zip list, ex: x=1, y=2 => [[0, 0], [0, 0], ... [0, 1], [1, 0]]
    # res_num is a numpy array, ex: x=1, y=2 => res=3 => [0, 0, ... ,1, 1]
    return num, res_num
```

3. Mechanism of forward propagation



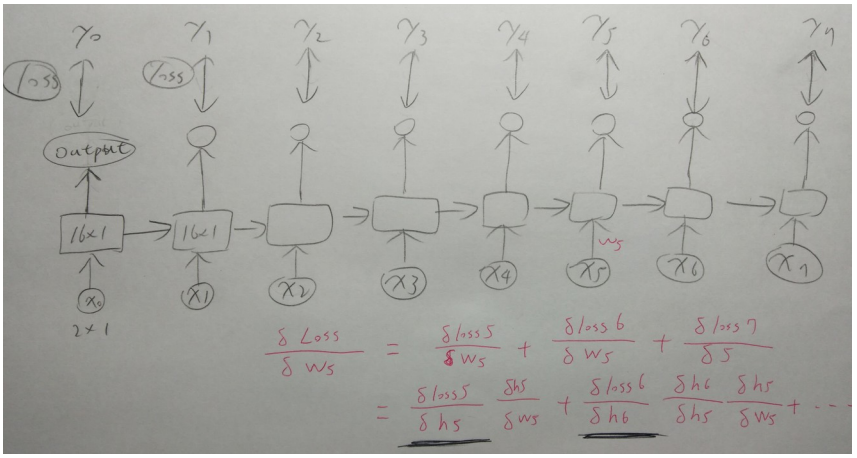
如上圖，每個 x 分開丟，每個 x 的維度在這邊是 2*1

再來變成 16*1 維的 latent vector，而這個 latent vector 還要再加上前一個 latent code 乘上某個 weight，因此後面的 output 某種程度上考慮了前面的 data，可使其運用在需要前後關係的資料上

而當前的 latent vector 會再經過轉換變成 output，再跟 y 比較算出 loss 整體公式如同 spec 上面附的，這邊截圖下來作參考

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)},$$
$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}),$$
$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)},$$

4. Mechanism of BPTT



如右圖，如果我想算 w5 的 gradient 的話，就需要將所有 loss 對其做偏微分

而有趣的是，output 中有 w5 參與的只有當前的 loss 以及之後的時間點的 loss，因此可以變成 loss5, loss6, loss7 對其的偏微分（第一個等號）

再更化簡之後，可以發現 loss6 和 loss7 的偏微分，在計算後面時間點的 w（如 w6 或 w7）gradient 時也會計算到，因此可以看作是從最後一個時間點開始做偏微分，一直做到前面的時間點，故稱作 Back Propagation Through Time

公式同樣如 spec 上附的，這邊就沒有再作推導了

5. How the code work

整體 code 架構如右圖，設定好 Learning rate, Iteration 以及多少次要計算一次 accuracy 之後，宣告已經定義好的 model

```
124  Lr = 0.1
125  Iteration = 30000
126  Record_cnt = 1000
127
128  model = RNN_binAddModel()
129
130  cor_cnt = 0
131  err_dig = 0
132  accs = []
133  for iter in range(Iteration):
134      x, y = get_data()
135      res = model.forward(x, y, Lr)
136      comp = 8 - (y==res).sum()
137      err_dig += comp
138      if comp == 0:
139          cor_cnt += 1
140
141      if iter % Record_cnt == 0:
142          print(f'{iter:5} Error: {err_dig/Record_cnt/8}')
143          print(f'{iter:5} Accuracy: {cor_cnt/Record_cnt}')
144          accs.append(cor_cnt/Record_cnt)
145          cor_cnt = 0
146          err_dig = 0
147  visualize(accs, Iteration, Record_cnt)
```

再來就是在每個 iteration 都先得到 x 與 y，再丟進去跑 model，而這個 model 再跑完之後會自己更新 weight，並回傳 predict 結果

之後比較 y 與 predict 值相同的位數，如果全部相同的話就將 correct count +1，而當 iteration 到達 record count 的倍數時，就紀錄並顯示 Accuracy 與 Error 為多少，最後跑完再輸出 accuracy 趨勢圖

get_data 函數在上面講解如何取得 data 時就已經解釋過了，這邊就不再說明

再來是處理進來的 data，首先因為考慮到加法是由低到高，因此將資料反轉來輸入，讓較高位數可以看到較低位數的資訊才算合理

```
38      def forward(self, x, y, Lr):
39          hidden = []
40          output = []
41          x, y = x[::-1], y[::-1]
42          for idx, x_unit in enumerate(x):
43              x[idx] = np.atleast_2d(x_unit).reshape(2, 1)
44          for idx, y_unit in enumerate(y):
45              y[idx] = np.atleast_2d(y_unit)
46          a = self.bx[0] + self.wx[0].dot(x[0])
47          hidden.append(np.tanh(a))
48          o = self.co[0] + self.wo[0].dot(hidden[0])
49          output.append(o)
50          for i in range(1, 8):
51              a = self.bx[i] + self.ww[i].dot(hidden[i-1]) + self.wx[i].dot(x[i])
52              hidden.append(np.tanh(a))
53              o = self.co[i] + self.wo[i].dot(hidden[i])
54              output.append(o)
55
56          self.backward(x, y, hidden, output, Lr)
57
58          res = []
59          for i in output:
60              if i[0][0] > 0.5:
61                  res.append(1)
62              else:
63                  res.append(0)
64          res = np.array(res)[::-1]
65          return res
```

接著讓 x 和 y 的每個元素都是至少 2D 的，使之後的 forward 和 backward 都可以順利進行

然後就是第 46~54 行的 forward 運算，而因為第一個 time slot 沒有前面的 hidden vector 可以運算，我這邊是單純的抽出來額外寫

再來進到 backward function 更新後，就可以利用 predict 結果算出 predict 的數值了，這邊我設定成大於 0.5 是 1，小於的話就是 0，最後因為前面先將 x 和 y 都反轉，這邊就反轉回來，回傳回去

再看到剛剛呼叫到的 backward，用來計算 gradient 以及更新 weights

在計算 gradient 可以看到 78~100 行，這邊是直接按照 spec 上的結果變成 code，就不再推導了

更新 weight 的部份是在 103~108 行，也是很單純的減掉 Learning rate 乘上先前計算的 gradient，這樣就完成了

```
67      def backward(self, x, y, hidden, output, Lr):
68          h_g = []
69          loss_g = []
70          wx_g = []
71          ww_g = []
72          wo_g = []
73          bx_g = []
74          co_g = []
75          H = []
76
77          # preprocess H and gradient of loss
78          for i in range(8):
79              H.append(np.diag(1-np.power(hidden[i].reshape(self.latent), 2)))
80              loss_g.append(output[i] - y[i])
81
82          # gradient of h
83          h_g.append(self.wo[-1].transpose().dot(loss_g[-1]))
84          for i in range(6, -1, -1):
85              temp = self.ww[i].transpose().dot(H[i+1]).dot(h_g[-1])
86              h_g.append(temp + self.wo[i].transpose().dot(loss_g[i]))
87          h_g = h_g[::-1]
88
89          # gradient of ww
90          ww_g.append(np.zeros((self.latent, self.latent)))
91          for i in range(1, 8):
92              temp = H[i].dot(h_g[i]).dot(hidden[i-1].transpose())
93              ww_g.append(temp)
94
95          # gradient of wx, wo and bx
96          for i in range(8):
97              wx_g.append(H[i].dot(h_g[i]).dot(x[i].transpose()))
98              wo_g.append(loss_g[i].dot(hidden[i].transpose()))
99              bx_g.append(H[i].dot(h_g[i]))
100             co_g.append(loss_g[i])
101
102         # update weight
103         for i in range(8):
104             self.wx[i] -= Lr*wx_g[i]
105             self.wo[i] -= Lr*wo_g[i]
106             self.ww[i] -= Lr*ww_g[i]
107             self.bx[i] -= Lr*bx_g[i]
108             self.co[i] -= Lr*co_g[i]
```

6. Discussion

這次作業利用 numpy 與 python 原生語法實做了簡單的 RNN，這邊我有做到正負數的加減。

全部較困難的部份應該是在 gradient 維度吧，但是因為已經有 graidnet 的結果出來了，實做其實也不算太難，稍微想一下前後關係以及 transpose，再嘗試一下就出來了。一部分應該也是因為前面有手刻 back propagation 的經驗了，這次的 BPTT 沒有太大障礙。

這次 lab 沒有用到 GPU 資源，也跑得很快，寫錯可以很快就知道哪邊有問題，也是這次 lab 可以做的比較快的原因之一，下個 lab 要做的是 CVAE，感覺就很有挑戰性，應該要早點開始做了。