ML_hw2 Report

電機四 B02504086 陳品融

1. Logistic regression function.

(1) method:

(i) data & feature:

將 data vector 全部 58 維當作 feature，而在 training 前會先隨機切 10%的 data 作 validation，再將剩餘的 data 用 full batch 的方式丟下去 train。

(ii) foward structure:

將 input data 與 58 維的 weight vector 內積加 bias 得 z1，再將 z1 通過 sigmoid function 算出機率，最後 predict 時如果機率大於 0.5 輸出 1，機率小於 0.5 則輸出 0。

(iii) loss:

使用 cross_entropy 作爲 loss，其式子如下：

np.sum(-(y_ * ln(y) + (1 - y_) * ln(1 - y))) / len(y)

需注意的是當 predict 出來的 y 爲 0 或 1 時，ln(y)或 ln(1 – y)會變負無限大，因此要加個微小的 epsilon 值來避免這種情況發生。我的 epsilon 是設爲 1e-20。

(iv) optimizer:

我的 optimizer 選用 adam，adam 同時計算了微分與微分的平方，運用動量的原理，使其相較 sgd，adagrad 等 optimizer 更有可能跳脫 local minima。其式子如下：

$m_t=\beta_1 m_{t-1}+(1-\beta_1)g_t$，$v_t=\beta_2 v_{t-1}+(1-\beta_2)g_t^2$ ，$\acute{m}_t=m_t \,/\, (1-\beta_1^t)$，$\tilde{v}_t=v_t \,/\, (1-\beta_1^t)$

$\theta_{t+1}=\theta_t-\eta\acute{m}_t \,/\, (\tilde{v}_t^{\,1/2}+\epsilon)$

其中 g_t 爲 gradient，$\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e\text{-}8$, t 爲第幾次 epoch。根據實際 training 過程的觀察，adam 不僅能收斂在較好的值，且收斂速度比 sgd，adagrad 等快上許多。

(v) save_best:

利用 save_best 的技巧將最好的 weight 存起來，讓我可以調大一點的 epoch 數也不用怕 overfit。

(2) code:

(i) split_validation:

```
def split_validation(x, y, split):
    indices = np.arange(len(x))
    np.random.shuffle(indices)
    x_org = x[indices]
    y_org = y[indices]
    x_val = x_org[:int(len(x)*split)]
    y_val = y_org[:int(len(y)*split)]
    x = x_org[int(len(x)*split):]
    y = y_org[int(len(y)*split):]
    return x, x_val, y, y_val
```

(ii) forward & loss:

```
def calculate_loss(x, y_, w, b):
    y = foward(x, w, b)
    e = 1e-20
    return np.sum(-(y_ * np.log(y + e) + (1 - y_) * np.log(1 - y + e))) / len(y)

def foward(x, w, b):
    z1 = np.dot(x, w) + b
    y = sigmoid(z1)
    return y
```

(iii) adam & save_best

```
#adam
t += 1
dw = (np.dot(x.T, y - y_) + lambda_ * w) / len(X)
db = np.sum(y - y_) / len(x)
m_w = beta1 * m_w + (1.0 - beta1) * dw
m_b = beta1 * m_b + (1.0 - beta1) * db
m_w_hat = m_w / (1.0 - beta1**t)
m_b_hat = m_b / (1.0 - beta1**t)
v_w = beta2 * v_w + (1.0 - beta2) * np.square(dw)
v_b = beta2 * v_b + (1.0 - beta2) * np.square(db)
v_w_hat = v_w / (1.0 - beta2**t)
v_b_hat = v_b / (1.0 - beta2**t)
w -= lr * m_w_hat / (np.sqrt(v_w_hat) + epsilon)
b -= lr * m_b_hat / (np.sqrt(v_b_hat) + epsilon)

train_loss = calculate_loss(x, y_, w, b)
val_loss = calculate_loss(x_val, y_val, w, b)
if i % 1000 == 0:
   print 'Training(Validation) loss after %i epoch: %f(%f)' %(i, train_loss, val_loss)
if val_loss < loss_best:
   loss_best = val_loss
   w_best = np.copy(w)
   b_best = b
```

2. DNN function

(1) method:

(i) data & feature:

與 logistic_regression 相同，我的 dnn 同樣將 data vector 全部 58 維當作 feature，而在 training 前也會先隨機切 10%的 data 作 validation，不同的是 train 時是用 batch_size = 600 丟下去 train，每個 batch 都是 random shuffle 過的 data。

(ii) foward structure:

我是用一個兩層 hidden layer 的 dnn model，第一層的參數 W1 爲 58*69 的矩陣，第二層則爲 69*32，其 activation function 皆爲 relu，最後 output layer 通過 sigmoid 輸出一個機率。hidden layer 的層數及每層的 size 是經過不斷地 trial-and-error 試出來的結果。

(iii)gradient

使用 backpropagation 以達到較高的效率。

(iv) loss:

與 logistic_regression 相同，皆是用 cross_entropy 作爲 loss。

(v) optimizer:

與 logistic_regression 相同，皆是用 adam 作爲 optimizer。

(vi) save_best:

與 logistic_regression 不同，我 dnn save_best 衡量的 metrics 爲 val_accuracy，而不是 val_loss，因爲我覺得 val_acc 似乎是一個更爲合理的指標。

(2) code:(與 logistic_regression 相似的 code 不再重複貼上)

(i) shuffle:

```
def shuffle(X, y_, batch_size):
   indices = np.arange(len(X))
   np.random.shuffle(indices)
   X = X[indices]
   y_ = y_[indices]
   for idx in xrange(0, len(X) / batch_size, batch_size):
      yield X[idx : idx + batch_size], y_[idx : idx + batch_size]
```

值得注意的是，使用 yield 可以減少記憶體存取的空間

(ii) forward & backward propagation

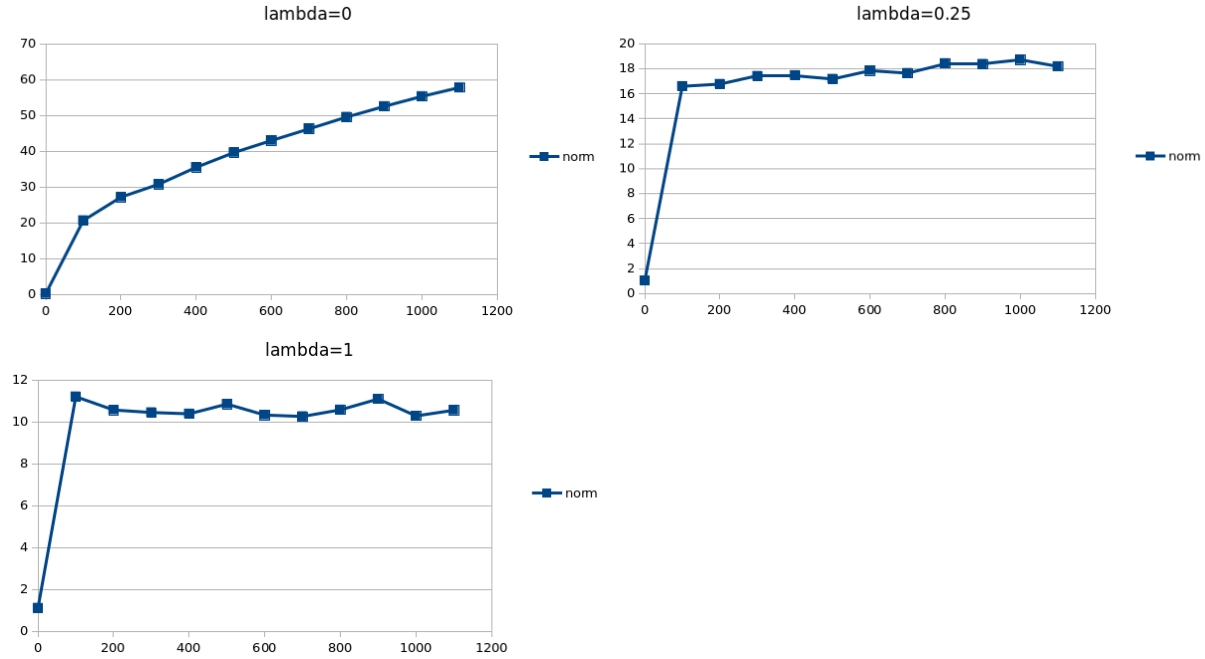
```
for i in xrange(0, num_epoch):
   for X_sh, y_sh in shuffle(X, y_, batch_size):
      # Forward propagation
      z1 = X_sh.dot(W1) + b1
      a1 = np.maximum(z1, 0)
      z2 = a1.dot(W2) + b2
      a2 = np.maximum(z2, 0)
      z3 = a2.dot(W3) + b3
      y = sigmoid(z3)

      # Backpropagation
      delta4 = y * (1 - y) * (-1) * (y_sh / (y + e) - (1 - y_sh) / (1 - y + e))
      dW3 = (a2.T).dot(delta4)
      db3 = np.sum(delta4, axis=0, keepdims=True)
      delta3 = delta4.dot(W3.T) * (np.greater(a2, 0).astype(float))
      dW2 = (a1.T).dot(delta3)
      db2 = np.sum(delta3, axis=0, keepdims=True)
      delta2 = delta3.dot(W2.T) * (np.greater(a1, 0).astype(float))
      dW1 = np.dot(X_sh.T, delta2)
      db1 = np.sum(delta2, axis=0, keepdims=True)
```

比較 logistic_regression 與 dnn：
根據 kaggle 上 public 及 private 的 score，logistic_regression 最高分別可達 0.92667 及 0.92000，至於 dnn 則是 0.96333 及 0.95333，所以很明顯的 dnn 的 performance 比 logistic_regression 好很多。分析原因如下：logistic_regression 爲線性的，因此只要待 predict 的兩個 class 爲線性不可分的話，則 logistic_regression 便有其極限；至於 dnn 的話，由於其參數之間是非線性複雜相連的，因此即便是非線性的 task 也有機會 predict 出好的結果，且通常越 deep 的 nn 能用較少參數得到較好的 performance。

3. Other discussion and detail.

(1) dnn regularization

lambda	0	0.1	0.2	0.25	0.5	0.7	1
private score	0.93000	0.93000	0.94667	0.94667	0.92000	0.92333	0.91000



由圖可知，lambda 越大則 weight 的 norm 確實會比較小且會被限制在某個值，若完全不加 lambda，則 weight 的 norm 會線性地增加。至於 score 的部分，實驗的結果是 lambda 從 0 慢慢加

到 0.25 時，private score 會漸漸上升，之後便開始往下掉，所以 lambda 大約要選擇在 0.2~0.25 左右的值。

(2) threshold
此次 training 的過程，如果 sigmoid output 出來的機率大於 0.5 輸出 1，小於 0.5 則輸出 0。然而在稍微觀察了 dataset 之後我發現，label 爲 0 的 data 似乎比較多，大約佔了 6 成，所以是否把 threshold 值更改爲 0.6 所得到的 score 會比較好呢？實驗的結果爲幾乎沒有差別，我的推測是 dnn 最後 output 的機率都是接近 0 或 1，因此沒有影響。

(3) save_best
一開始使用 save_best 是爲了省去處理 overfit 的步驟，而後來我發現，常常 save_best 得到的 val_acc 很高，但丟上去 kaggle 後的 public score 卻不見得理想。因此我開始嘗試一些其他的 metrics 比如 val_acc + 0.1/val_loss，然而實際測試的結果發現效果不彰。分析的原因是有可能 training 的過程中不太穩定，導致雖然已經 overfit 了然而某個 epoch 的 val_acc 卻突然衝很高。

(4) learning rate
我一開始在算 loss 對 weight 的微分時，最後都有除以 batch_size 作取平均的動作，然而造成的結果是 score 一直卡在某個瓶頸上不去，後來拿掉之後，score 便有所進步，收斂速度也比較快。而稍微分析一下可發現，不取平均對應的其實就是 learning rate 往上調，由此可知 learning rate 對 training 的過程有顯著的影響。