HW#2 Logistic Regression Report

B03901027 徐彥旻

1.Logistic regression function by Gradient Descent.

```
def train(X, Y, Epoch):
    W = array([random.random() for i in range(len(X[0]))])
    B = random.random()
    Loss = []
    L = computeLoss(W,B,X,Y)
    Loss.append( L / len(X) )
    DeltaW = zeros(len(X[0]))
    DeltaB = 0
    LR = 0.00000047 # learning rate
    for ii in range(Epoch):
        #   compute Delta and change parameters
        pL = partialLoss(W,B,X,Y)
        DeltaW = - LR * pL[0] # pL[0] is partial Loss partial W
        W = W + DeltaW
        DeltaB = - LR * pL[1] # pL[1] is partial Loss partial B
        B = B + DeltaB
        L = computeLoss(W,B,X,Y)
        Loss.append(L / len(X))
        if ii>100 and Loss[-1]>Loss[-2]:
            break
    return (W,B,Loss)
```

上方的程式碼是整個程式架構中的核心函數，處理參數更新，並且記錄整個過程的 Loss 變化。 X 選用與解答的相關係數的絕對值前九高的特徵。 Y 為解答（0, 1）Epoch 為將整個 training data 跑過的次數。

標示紅色的程式碼，其目的是為了讓 model 在 Loss 為最低時停下（附圖一為出現 Loss 在一段時間後反而升高的現象）。

2.(1%) Describe your another method, and which one is the best.

```
class Config:
    def __init__(self, LR, nEpoch, ep, RC, batch = 0):
        self.LearningRate = LR
        self.Epoch = nEpoch
        self.Epsilon = ep
        self.RegularizationConstant = RC
        self.Batch = batch

class Activation:
    def __call__(self,z):
        raise NotImplementedError("Subclass must implement abstract method"
)
    def backprop(self,a):
        raise NotImplementedError("Subclass must implement abstract method"
)

class RelU(Activation):
    def __call__(self,z):
        a = matrix(zeros(len(z))))
        for i in range(len(z)):
            if z[0,i] > 0:
                a[0,i] = z[0,i]
        return a
    def backprop(self,a):
        partial = matrix(zeros(len(a)))
        for i in range(len(a)):
            if a[0,i] > 0:
                partial[0,i] = 1
        return partial

class Sigmoid(Activation):
    def __call__(self,z):
        a = matrix(zeros(len(z)))
        for i in range(len(z)):
            if z[0,i] > -700:
                a[0,i]= 1/(1 + exp(-z[0,i]))
            else:
                a[0,i]= 0
        return a
    def backprop(self,a):
        return a * (1 - a)

Act = {'sigmoid': Sigmoid(), 'relu':RelU()}
```

```
class NeuronLayer:
    def __init__(self, nNeuron, dimension, activation='relu'):
        self.W = matrix(random.rand(nNeuron, dimension))
        self.B = matrix(random.rand(nNeuron))
        self.act = Act[activation]
        self.a = matrix(zeros(nNeuron))
    def __call__(self, a_prev):
        z = self.W * a_prev + self.B
        self.a = self.act(z)
        return self.a
    def backprop(self):
        return self.act.backprop(self.a)
```

2.1 已完成部分

1. 於 lib/DNN.py 中將訓練的相關參數包成 class Config
2. 將 Activation function 實作成 functional object ，繼承架構為 class Sigmoid 與 class RelU 繼承 class Activation ，並以 __call__ 實作函數，並有處理偏微分的 member function 。使用上是以 dictionary Act 將各一個 actionvation function 包起來，供 class NeuronLayer 使用。
3. 實作 class NeuronLayer ， 儲存 W, B, activation function (與第二點中的 Act 的 value 做 binding) ， 實作 propagation 與 backpropagation 的函數。

```
class DNN:
    def __init__(self, inputDimension):
    def add(self, nNeuron, activation='relu'):
    def cleanGradientC(self):
    def propagate(self, x): #TODO compute the self.a in each layer
```

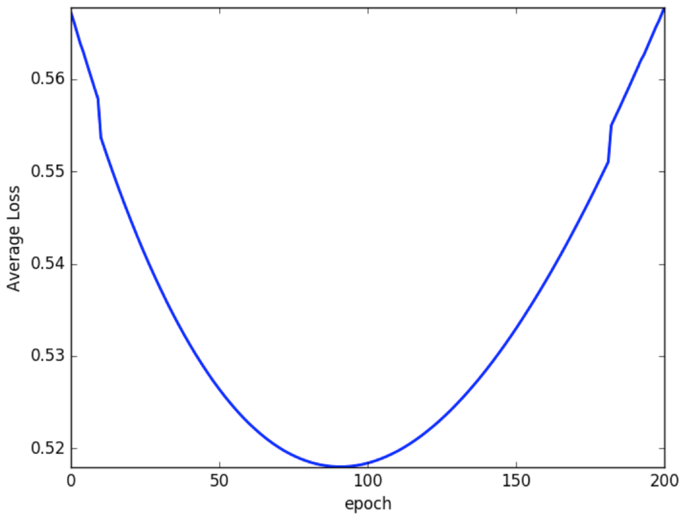
2.2 未完成部分

1. 上方是在未完成的 class DNN 已經實作的函數，其中 cleanGradientC 是在將 Loss 對所有的參數的偏微分重設為零，在每個新的 batch 開始前會呼叫。

2. batch、整個網路的 backpropagation 、 對應 Keras 的 compile 尚未實作。

3.(2%) Other Discussion

附表一是對不同的參數（維度）的 logistic 做 N-fold cross validation ，解果顯示九個參數的表現最為穩定。



附圖一

	9 個參數	8 個參數	7 個參數	AdaDelta 9 個參數
a	80.97%	78.27%	75.43%	77.83%
b	78.77%	80.87%	80.95%	70.89%
c	79.97%	80.87%	75.24%	80.42

附表一