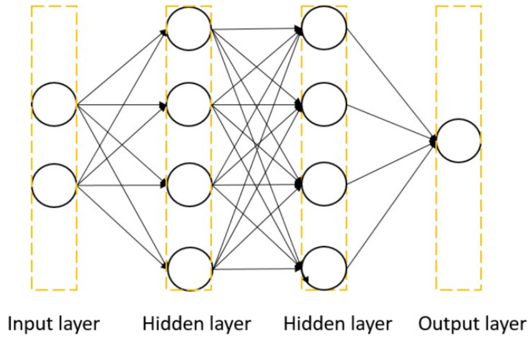
1.Introduction



這次作業需要實做出一個雙層的 neural network，包括輸入層、兩層隱藏層與輸出層。其中由於訓練資料是二元分類問題，輸入層的維度為 2，輸出層的維度為 1。

當我們對神經元進行輸入後，經過內部迴歸模型對輸入的權重加乘，再經過 active function，便完成了該節點的輸出。該輸出會再傳給下一個神經元，作為該神經元的輸入值，如此一層層傳遞下去，直到最後一層的輸出層，產生預測結果，此過程稱為 Forward-Propagation。

neural network 會反覆由預測結果和真實結果之間的差距，對整個神經網路，由後面神經元至前層神經元的更新，此過程稱為 Back-Propagation。

透過這次作業的實做，能夠更清楚了解 Forward-Propagation 和 Back-Propagation 的工作原理。

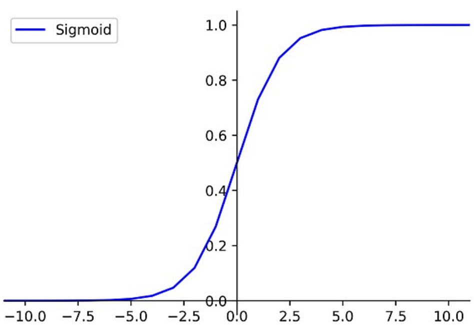
2.Experiment setups

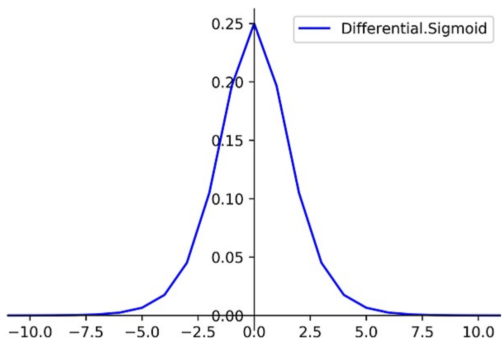
A. Sigmoid functions

在 neural network 中，若不使用 active function，neural network 即是以線性的方式組合運算，使用 active function，主要是利用非線性方程式，解決非線性問題。Sigmoid 函數是深度學習領域開始時使用頻率最高的 activation function，它是易於於求導數之平滑函數，其函數和導函數圖形如下。並使用以下程式碼來實現。

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma'(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = (-1)\cdot\frac{1}{(1+e^{-x})^2}\cdot(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2}$$



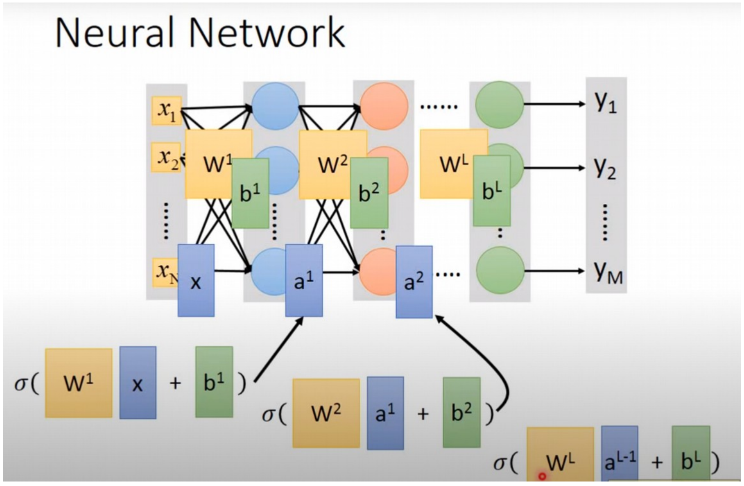



```
if self.activation_func == 'sigmoid':
    return 1.0/(1.0+np.exp(-x))
```

```
if self.activation_func == 'sigmoid':
    return np.multiply(x,1.0-x)
```

B. Neural network

假設上一層結點 i,j,k,…等一些結點與本層的結點 w 有連接，那麼結點 w 的值就是通過上一層的 i,j,k 等結點以及對應的連接權值進行加權和運算，最後通過一個非線性函數（sigmoid 等函數），最後得到的結果就是本層結點 w 的輸出。最終不斷的通過這種方法一層層的運算，得到輸出層結果。

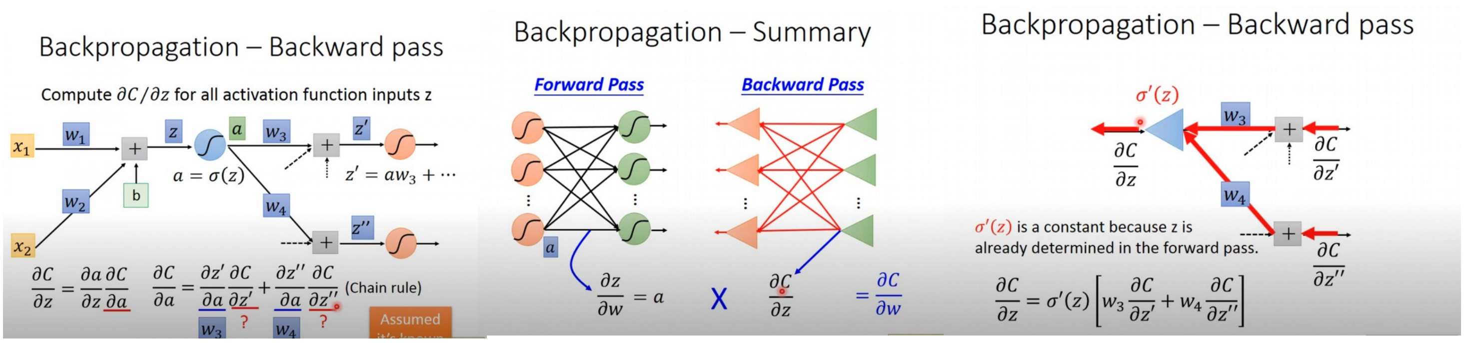


程式實做（我沒有寫 bias 項）

```python
def forward(self,x,W):
    Z = list([])
    A = list([])
    for l in range(self.layers+1):
        if not l==0:x = A[-1]
        Z.append( np.dot(x,W[l]) )
        A.append(np.array([self.activation(item) for item in Z[-1] ],dtype=np.float128))

    pred_y = A[-1]
    return Z,A,pred_y
```

C. Backpropagation

Backpropagation 是誤差反向傳播的簡稱，是一種與最佳化方法（如梯度下降法）結合使用的方法，透過微積分的連鎖律，我們可以計算出損失函數對每一個節點的梯度，再進一步算出對每個權重值的梯度。



程式實做

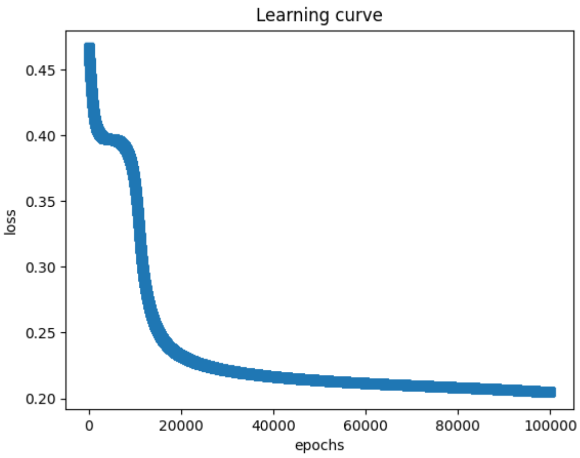
```
def back(self,W,A,pred_y,x,y,num_of_nodes):
    dW = init_parameters_zeros(num_of_nodes)
    dZ = list([])

    for k in range(self.layers,-1,-1):
        if k == self.layers :
            tmp_dZlist = []
            for i in range(len(y)):
                tmp_dZlist.append((y[i]-pred_y[i])*self.derivative_activation(A[self.layers][i]))
            dZ.insert(0,np.array(tmp_dZlist,dtype=np.float128))
        else :
            tmp_dZlist = list([])
            for i in range(len(A[k])):
                dZtmp = 0
                for j in range(len(dZ[0])):
                    dZtmp += dZ[0][j] * W[k+1][i][j]
                dZtmp = dZtmp*self.derivative_activation(A[k][i])
                tmp_dZlist.append(dZtmp)
            dZ.insert(0,np.array(tmp_dZlist,dtype=np.float128))

    for i in range(self.layers+1):
        for j in range(len(W[i])):
            for k in range(len(W[i][j])):
                if not i==0:
                    dW[i][j][k] = dZ[i-1][j] * dZ[i][k]
                else:
                    dW[i][j][k] = x[j] * dZ[i][k]
    return dW
```

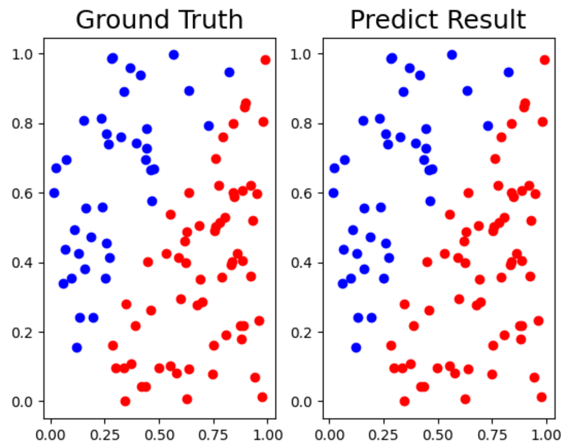
3.Results of your testing

(1) linear

程式執行方法：

python3 hw1.py --task linear --batch_size 100 --lr 0.5 --epochs 100000 --optimizers GD --N 100 --hidden_units 2 --activation_func sigmoid



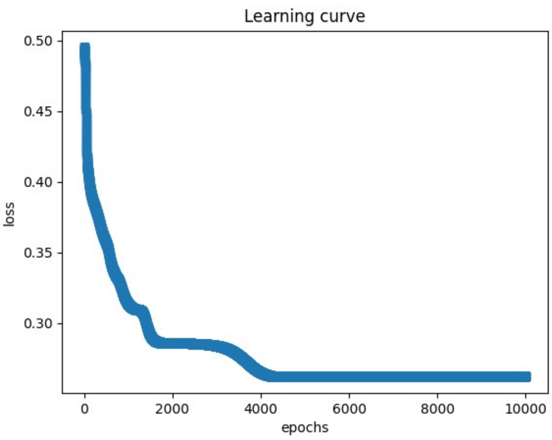


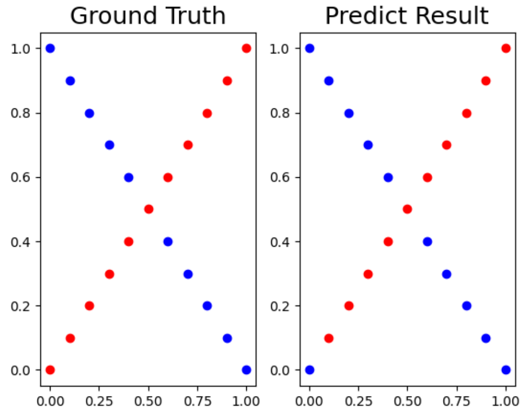

```
20:45:22  epoch= 85000 Loss= 0.20759575363409612489
20:45:26  epoch= 85500 Loss= 0.20751433578804313922
20:45:31  epoch= 86000 Loss= 0.2074324098918373797
20:45:35  epoch= 86500 Loss= 0.20734994077144711426
20:45:40  epoch= 87000 Loss= 0.20726689254108546592
20:45:45  epoch= 87500 Loss= 0.20718322857602706761
20:45:50  epoch= 88000 Loss= 0.20709891148381432316
20:45:55  epoch= 88500 Loss= 0.20701390307351310464
20:46:00  epoch= 89000 Loss= 0.20692816432270213292
20:46:05  epoch= 89500 Loss= 0.20684165534194422024
20:46:10  epoch= 90000 Loss= 0.20675433533660602527
20:46:14  epoch= 90500 Loss= 0.20666616256608448429
20:46:19  epoch= 91000 Loss= 0.20657709430078496037
20:46:23  epoch= 91500 Loss= 0.20648708677760472612
20:46:28  epoch= 92000 Loss= 0.2063960951552359438
20:46:33  epoch= 92500 Loss= 0.20630407347134864885
20:46:38  epoch= 93000 Loss= 0.2062109746046827275
20:46:43  epoch= 93500 Loss= 0.20611675024630559485
20:46:48  epoch= 94000 Loss= 0.20602135088581424022
20:46:53  epoch= 94500 Loss= 0.20592472582010517174
20:46:57  epoch= 95000 Loss= 0.20582682319451994184
20:47:02  epoch= 95500 Loss= 0.20572759008869347733
20:47:07  epoch= 96000 Loss= 0.20562697266225289524
20:47:11  epoch= 96500 Loss= 0.20552491637855832733
20:47:16  epoch= 97000 Loss= 0.20542136632780940217
20:47:21  epoch= 97500 Loss= 0.20531626767385346371
20:47:26  epoch= 98000 Loss= 0.20520956625162885572
20:47:30  epoch= 98500 Loss= 0.20510120934396494103
20:47:35  epoch= 99000 Loss= 0.20499114666694482534
```

```
0.49974551572716888035
0.49974497156744092995
0.10576699123627181962
0.001810226164572917604
6.668015379984051555e-05
0.49970775629922699687
0.00046363212100520100142
7.417679848051290241e-08
6.630279650598467335e-08
0.4992948417143853214
0.08890652394399669987
6.649099278488791028e-11
0.48076408631514147793
0.4996107690506225482
0.4997448624576181588
0.0023031342467738315717
0.49588122929799338326
2.5141962195083000392e-07
0.49969900970342090107
1.1987831969783921778e-08
0.49969643066571512097
0.49971527392058805403
0.21369239233349767478
0.0002791390605866182078
0.4994672244723581556
0.08910685118595268023
7.363775255069901435e-10
0.4997461894694492084
accuracy :  1.0
```

(2) xor

python3 hw1.py --task xor --batch_size 21 --lr 0.02 --epochs 10000 --optimizers Adam --hidden_units 30 --activation_func sigmoid





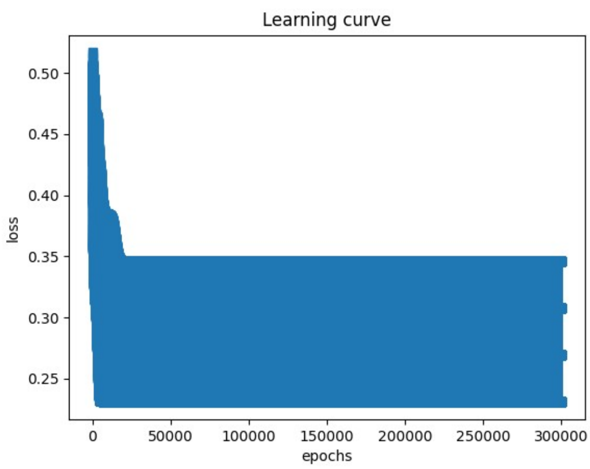

```
02:36:31  epoch= 0 Loss= 0.49549586737734632483
02:36:56  epoch= 500 Loss= 0.35675747615130063884
02:37:19  epoch= 1000 Loss= 0.31335750371031477102
02:37:42  epoch= 1500 Loss= 0.29135091206077406668
02:38:05  epoch= 2000 Loss= 0.28571249537389593445
02:38:29  epoch= 2500 Loss= 0.28539397559191799443
02:38:52  epoch= 3000 Loss= 0.28388759153271947333
02:39:16  epoch= 3500 Loss= 0.27708786579206244572
02:39:40  epoch= 4000 Loss= 0.26485753601026879177
02:40:05  epoch= 4500 Loss= 0.26193728070663100513
02:40:31  epoch= 5000 Loss= 0.26191328651295346643
02:40:59  epoch= 5500 Loss= 0.26191053666709045583
02:41:26  epoch= 6000 Loss= 0.26190936036489775204
02:41:50  epoch= 6500 Loss= 0.26190867740328521
02:42:14  epoch= 7000 Loss= 0.26190822031119797333
02:42:38  epoch= 7500 Loss= 0.26190788794600487337
02:43:02  epoch= 8000 Loss= 0.26190763274850152324
02:43:27  epoch= 8500 Loss= 0.26190742910257083003
02:43:51  epoch= 9000 Loss= 0.26190726185843124797
02:44:15  epoch= 9500 Loss= 0.26190712142016237338
```

```
0.5
0.5
4.703092029651021681e-05
0.5
6.292082860076770510 6e-454
0.5
8.882197833753359261 5e-512
0.5
3.365970033832660007 3e-514
0.49999999861078816426
2.669391179920134235 2e-514
2.643777236039217090 1e-514
0.49999999861326429 44
2.642717462902878913 3e-514
0.5
2.642673403198835436 7e-514
0.5
2.642671571062924017 5e-514
0.5
2.64267149487643784e-514
0.5
accuracy :  0.9523809523809523
```

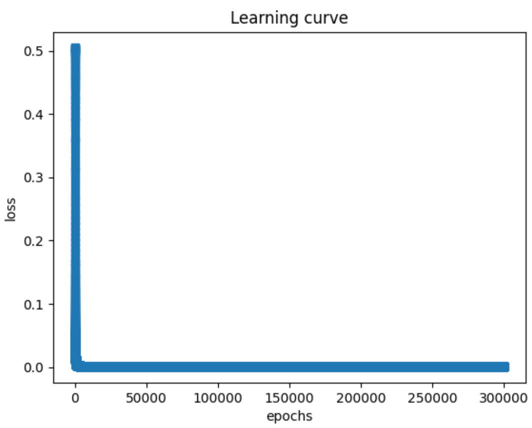
4.Discussion

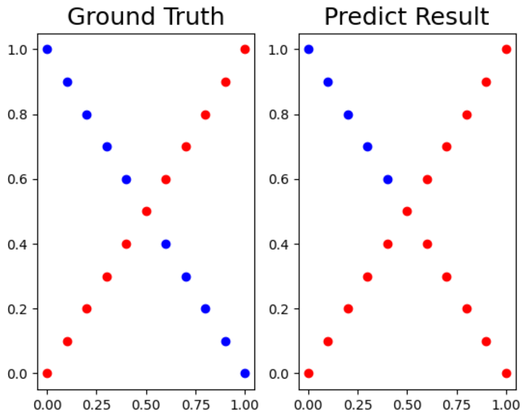
A.Try different learning rates

將 learning rates 調大，可以加快一開始 Loss 的收斂速度。但有時候 learning rates 過大，在某些地方會卡住，使 Loss 在兩個值間震盪下不去。



B.Try different numbers of hidden units
因為非線性的 xor 較難訓練。當我用一層只有 2 個 hidden units 時，雖然能夠將 Loss 降低到很小，但是它僅將預測正確的逼近 0 和 1，預測結果仍然偏線性，約 1/4 預測錯誤。使用多一些 hidden units 才能有較好預測結果。






```
05:58:00  epoch= 286500 Loss= 0.0005479273529396626 6955
05:58:01  epoch= 287000 Loss= 0.000547678541766418607 5
05:58:02  epoch= 287500 Loss= 0.0005474302816853392804 3
05:58:02  epoch= 288000 Loss= 0.000547182570513057482 6
05:58:03  epoch= 288500 Loss= 0.0005469354060794415740 7
05:58:04  epoch= 289000 Loss= 0.000546688786227353907 6
05:58:05  epoch= 289500 Loss= 0.0005464427088096718156 7
05:58:05  epoch= 290000 Loss= 0.000546197171693682009 1
05:58:06  epoch= 290500 Loss= 0.0005459521727570027331 5
05:58:07  epoch= 291000 Loss= 0.000545707709891700141 3
05:58:08  epoch= 291500 Loss= 0.0005454637809989109298 5
05:58:09  epoch= 292000 Loss= 0.000545220383993336480 5
05:58:09  epoch= 292500 Loss= 0.000544977516800635524 6
05:58:10  epoch= 293000 Loss= 0.0005447351773584902166 6
05:58:11  epoch= 293500 Loss= 0.0005444933636161106042 5
05:58:12  epoch= 294000 Loss= 0.0005442520735342120430 7
05:58:12  epoch= 294500 Loss= 0.0005440113050846412273 6
05:58:13  epoch= 295000 Loss= 0.000543771056250570786 1
05:58:14  epoch= 295500 Loss= 0.000543531325026335310 6
05:58:15  epoch= 296000 Loss= 0.000543292109417710940 4
05:58:15  epoch= 296500 Loss= 0.0005430534074409927767 4
05:58:16  epoch= 297000 Loss= 0.000542815217124708179 2
05:58:17  epoch= 297500 Loss= 0.0005425775365064737420 7
05:58:18  epoch= 298000 Loss= 0.0005423403636368257016 4
05:58:19  epoch= 298500 Loss= 0.0005421036965747877619 3
05:58:19  epoch= 299000 Loss= 0.000541867533391953059 8
05:58:20  epoch= 299500 Loss= 0.0005416318721685734282 4
```

```
0.005955368989042460159 7
1.0
4.365838857698739720 7e-93
1.0
4.365838857698739720 7e-93
1.0
4.365838857698739720 7e-93
1.0
4.365838857698739720 7e-93
1.0
4.365838857698739720 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
4.365583701525892944 7e-93
accuracy :  0.7619047619047619
```

C.Try without activation functions
計算過程中直接炸裂到無限大了。

5. Extra

A .Implement different optimizers

```
if self.optimizers == 'GD':
    new_W = init_parameters_zeros(self.num_of_nodes)
    for i in range(self.layers+1):
        for j in range(len(W[i])):
            for k in range(len(W[i][j])):
                new_W[i][j][k] = W[i][j][k] + self.lr * dW[i][j][k]
    return new_W
```

```
if self.optimizers == 'Momentum':
    global vt_last
    vt = init_parameters_zeros(self.num_of_nodes)
    new_W = init_parameters_zeros(self.num_of_nodes)
    beta = 0.9
    try : _ = vt_last
    except NameError: vt_last = init_parameters_zeros(self.num_of_nodes)

    for i in range(self.layers+1):
        for j in range(len(W[i])):
            for k in range(len(W[i][j])):
                vt[i][j][k] = beta * vt_last[i][j][k] + self.lr * dW[i][j][k]
                new_W[i][j][k] = W[i][j][k] + vt[i][j][k]
                vt_last[i][j][k] = vt[i][j][k]
    return new_W
```

```
if self.optimizers == 'AdaGrad':
    epsilon = 0.00000001
    n = 0
    for i in range(self.layers+1):
        for j in range(len(W[i])):
            for k in range(len(W[i][j])):
                n += dW[i][j][k] * dW[i][j][k]

    new_W = init_parameters_zeros(self.num_of_nodes)
    for i in range(self.layers+1):
        for j in range(len(W[i])):
            for k in range(len(W[i][j])):
                new_W[i][j][k] = W[i][j][k] + self.lr * dW[i][j][k] * (n+epsilon)**(-0.5)
    return new_W
```

```python
if self.optimizers == 'Adam':
    global vt_old
    global mt_old
    vt = init_parameters_zeros(self.num_of_nodes)
    mt = init_parameters_zeros(self.num_of_nodes)
    vt_hat = init_parameters_zeros(self.num_of_nodes)
    mt_hat = init_parameters_zeros(self.num_of_nodes)
    new_W = init_parameters_zeros(self.num_of_nodes)
    beta = 0.9
    epsilon = 0.00000001

    try : _,__ = vt_old,mt_old
    except NameError:
        vt_old = init_parameters_zeros(self.num_of_nodes)
        mt_old = init_parameters_zeros(self.num_of_nodes)
    for i in range(self.layers+1):
        for j in range(len(W[i])):
            for k in range(len(W[i][j])):
                vt[i][j][k] = beta * vt_old[i][j][k] + (1.0 - beta) * dW[i][j][k] * dW[i][j][k]
                mt[i][j][k] = beta * mt_old[i][j][k] + (1.0 - beta) * dW[i][j][k]

                vt_hat[i][j][k] = vt[i][j][k]/(1.0 - beta)
                mt_hat[i][j][k] = mt[i][j][k]/(1.0 - beta)

                new_W[i][j][k] = W[i][j][k] + self.lr * (mt_hat[i][j][k] / ((vt_hat[i][j][k])**0.5+epsilon)
    return new_W
```

B. Implement different activation functions

```python
def activation(self,x):
    if self.activation_func == 'sigmoid':
        return 1.0/(1.0+np.exp(-x))
    if self.activation_func == 'None':
        return 1.0 * x
    if self.activation_func == 'tanh':
        return np.tanh(x)
    if self.activation_func == "ReLU":
        return np.maximum(0, x)

def derivative_activation(self,x):
    if self.activation_func == 'sigmoid':
        return np.multiply(x,1.0-x)
    if self.activation_func == 'None':
        return 1.0
    if self.activation_func == 'tanh':
        return 1.0 - x ** 2
    if self.activation_func == "ReLU":
        return 1.0 * (x > 0)
```