layer.py

```
import numpy as np

class Dense(object):
    def __init__(self,
    N_inputs,
    N_outputs,
    activation
    ):
        """Define the number of input and output NODES for a particular layer. A layer could then be made by calling
        layer= Dense(N_inputs,N_outputs). For example, a single layer NN with no hidden layers could be done with
        model_1=[Dense(N_inputs,N_outputs)]
        all these things are first initialized randomly here, and then they pass to the later functions of this class

        activation is activation class (such as tanh) that well be specified in run_framework.py
        """
        self.N_inputs = int(N_inputs)
        self.N_outputs = int(N_outputs)
        self.activation=activation
        self.learning_rate=int(1e-1)



        #the size of the weights is a matrix of ( N_inputs + 1) X (N_outputs) (and the inputs) is w[N_inputs]+b which
is the rows, and the outputs will have [N_outputs]
        rows = self.N_inputs+1#the +1 because there will be a bias vector
        columns = self.N_outputs
        self.weights = np.random.sample(size=(rows,columns) )
        #random sample returns a random unifrom between0 and 1
        self.w_grad = np.zeros((self.N_inputs+1, self.N_outputs))

        #Define set of inputs coming in to the network
        self.x = np.zeros((1,self.N_inputs+1))
        self.y=np.zeros((1,self.N_outputs))

    def forward_propagate_layer(self, inputs):
        """propagate the inputs forward through the NN

        Args:
            inputs : (INPUT TO THE LAYER) vector of values of size [1,N_input]


        Returns:
            y : of size [1, N_out]
        """
```

```
        #inputs are the inputs to the layer
        #make a 1X1 bias matrix of ones
        bias=np.ones((1,1))
        #stack the bias on top of the inputs by adding it as a new column (axis 1)
        self.x = np.concatenate((inputs, bias), axis=1)
        #matrix-multiply the selt of augmented inputs x with the weights
        self.y_intermediate = self.x @ self.weights
        #the shapes of what being multiplied is the following:
        #[1, N_in +1] X [N_in +1, N_out] = [1, N_out]

        #Perform activation on the output for the final output
        self.y = self.activation.calc(self.y_intermediate)
        #here y is really yprime in the back_propagate_layer() layer , ie yprime=activation_function(y)
        return self.y

    def back_propagate_layer(self, dLoss_dy):
        """
        Args:
            dLoss_dy ([type]): the derivative of loss wrt y for ONE LAYER.
            dL/dx = dL/dy * dy/dx
            EG if its a linear layer, y=mx+b and and L=(y-x)^2 with no activation function then:
            dL/dx = dL/dy * m (and we dont need to simplify since dL/dy is an input)

        If there is an activation function f such that y = f(y') then
        dLoss/dx = dLoss/dy dy/dy' dy'/dx
                        = dLoss/dy d [f(y')]/dy' dy'/dx

        Returns:
            dLoss/dx of the current layer
        """
        # $y' = \vec{x} \cdot \vec{w}^T $
        # i.e. yprime = self.x @ self.weights.transpose()
        # so $dy'/dx = x$ i.e. dyprime_dx = self.weights
        # and $dyprime/dw = \vec{w}^T$ ie dyprime_dw=self.weights.transpose()

        #dy/dy'=d [f(y')]/dy'
        dy_dyprime = self.activation.calc_deriv(self.y)

        #y = f(y') = f(x @ weights) so
        # dy/dw = dy/dy' dy'/dw  = dy/dy' * x
        dy_dw = self.x.transpose() @  dy_dyprime

        #dL/dw = dL/dy * dy/dw

        dLoss_dw = dLoss_dy * dy_dw

        #update the weights by subtracting the gradient
        self.weights = self.weights - ( dLoss_dw * self.learning_rate)
```

```
        # $L = (y-x)^2 = ( f(y') - x)^2 $ so
        # dL/dx = dL/d[f(y')] d [f(y')]/dy' dy'/dx
        dLoss_dx = (dLoss_dy * dy_dyprime) @ self.weights.transpose()

        #return everything except the last column, which is the bias term, which is always 1 (const) and not backpropagated
        return dLoss_dx[:, :-1]
```