framework.py

```
###Framework module
import numpy as np
import os
import matplotlib.pyplot as plt
plt.switch_backend('agg')#switch to interactive plot

class ANN(object):
    """


    Args:
        model:  a list of layers with the weights and input/output nodes initialized

    """
    def __init__(self,
    model,
    expected_input_range=(0,1),
    loss_func=None
    ):
        self.model= model
        #model is a LIST OF LAYERS
        # each self.layer is a member of the layers.Dense(N_in,N_out) class, so that you can call layer.forward_prop
        self.loss_func=loss_func#loss function class

        #it helps to set the numbers as integers so that we can call range on it later
        self.n_iter_train = int(1e2)#number of iterations to train
        self.n_iter_evaluate = int(1e1) #number of iterations to evaluate on
        self.expected_input_range=expected_input_range
        self.error_history = []# list of errors, at each iteration of training/evaluation, the error will be added to it
        self.viz_interval = int(1e1
        #obviously it should be that viz_interval > n_train_iter
        )#self.n_iter_train
        #average over the errors in the last n_bin_size iterations
        self.error_bin_size=int(1e3)
        # #min and max of report plot
        self.error_min=-3
        self.error_max=0
        self.report_path="training_reports"
        self.report_image="training_history.png"
        try:
            os.mkdir(self.report_path)
        except Exception:
            pass
```

```
def normalize(self, values):
    expected_range=self.expected_input_range
    expected_min, expected_max = expected_range
    scale_factor = expected_max - expected_min
    offset = expected_min
    scaled_values = (values - offset)/scale_factor - 0.5#this -0.5 is there so that the lowest value is -0.5
    return scaled_values

def normalize_IQN(self, values):
    expected_range=self.expected_input_range
    expected_min, expected_max = expected_range
    scale_factor = expected_max - expected_min
    offset = expected_min
    scaled_values = (values - offset)/scale_factor
    return scaled_values

def denormalize(self, normalized_values):
    expected_range=self.expected_input_range
    expected_min, expected_max = expected_range
    scale_factor = expected_max - expected_min
    offset = expected_min
    return (normalized_values + 0.5) * scale_factor + offset



def denormalize_IQN(self, normalized_values):
    expected_range=self.expected_input_range
    expected_min, expected_max = expected_range
    scale_factor = expected_max - expected_min
    offset = expected_min
    return normalized_values  * scale_factor + offset



def RMS(self, v):
    return (np.mean(v**2))**0.5

def back_propagate_data(self, dLoss_dy):
    """
    Args:
        dLoss_ly: derivative of the loss wrt y (the output of the WHOLE NN)
    """
    layers_in_reverse = self.model[::-1]
    for i_layer, layer in enumerate(layers_in_reverse):
        dLoss_dx = layer.back_propagate_layer(dLoss_dy)
        #the dLoss/dx becomes the dLoss/dy of the next layer
        # X layer1 - > layer_2 .... -> layer_n -> y
        #    <-...  dL/dx  <- BP dL/dy  <- dL/dx   <-BP  dL/dy
```

```
            dLoss_dy = dLoss_dx

    def train(self, training_set):
        """
        Args:
            training_set (GENERATOR): its a generator function for the data, so call it by doing next(training_set())
        """
        for iter in range(self.n_iter_train):
            x = next(training_set()).ravel()#ravel flattens it to 1-d array
            x=self.normalize(x)
            y=self.forward_propagate_data(x)
            loss = self.loss_func.calc(x,y)
            #calculate the (minimize) derivative of the loss wrt x, ie returns dLoss_dx
            loss_grad = self.loss_func.calc_gradient(x,y)
            #take the RMS of the loss (eg if you have a 2x2 error thats not useful!)
            # RMS_loss=(np.mean(loss**2))**0.5
            RMS_loss = self.RMS(loss)
            self.error_history.append(RMS_loss)

            #BACKPROPAGATION: back_propagate_data(dLoss_dx) = dLoss
            # self.back_propagate_data(loss_grad)
            if (iter + 1) % self.viz_interval==0:
                #generate a report every multiple of viz_interval (the 1 is just so that the 0th iteration isnt included)
                print(f'train y {y} \t loss \t {loss}')
                self.report_error()

    def evaluate(self, evaluation_set):
        for iter in range(self.n_iter_evaluate):
            x = next(evaluation_set()).ravel()
            x = self.normalize(x)
            y=self.forward_propagate_data(x)
            loss = self.loss_func.calc(x,y)
            #calculate the derivative of the loss wrt x, ie returns dLoss_dx
            loss_grad = self.loss_func.calc_gradient(x,y)
            #take the RMS of the loss (eg if you have a 2x2 error thats not useful!)
            RMS_loss= self.RMS(loss)

            self.error_history.append(RMS_loss)
            if (iter + 1) % self.viz_interval==0:
                    #generate a report every multiple of viz_interval (the 1 is just so that the 0th iteration isnt included)
                print(f'train y {y} \t loss \t {loss}')
                self.report_error()

    def forward_propagate_data(self, x):
        """Forward propagate the inputs to the entire NN (ie the data)

        The inputs x here are the inputs to the entire NN (the data)
```

```
        Since the layers.Dense(inputs, outputs) expects a 2d array for the inputs, we have to make our 1D input of shape (1,) into a 2D array of shape (1,N_inputs)"""
        y = x.ravel()[np.newaxis,:]
        #forward propagate through each layer
        for layer in self.model:
            y = layer.forward_propagate_layer(y)
        #remember that layer is a layers.Dense member
        return y.ravel()

    def forward_propagate_data_to_layer(self, x, i_layer):
        """
        Args:
            x ([type]): input data
            i_layer ([type]): the layer that you want to propagate to
        """
        y = x.ravel()[np.newaxis,:]
        for layer in self.layers[:i_layer]:
            #from teh beginning to layer i
            y = layer.forward_propagate_layer(y)
        return y.ravel()
    def forward_propagate_data_from_layer(self, x, i_layer):
        """
        Args:
            x ([type]): input data
            i_layer ([type]): the layer that you want to propagate from
        """
        y = x.ravel()[np.newaxis,:]
        for layer in self.layers[i_layer:]:
            #from layer i to the end
            y = layer.forward_propagate_layer(y)
        return y.ravel()

    def report_error(self):
        """history is a list of errors. Here we chop them up into bins """
        #take the full error history that is being appended in train() or evaluate
        history=self.error_history#get the global history
        n_bins=int(len(history)//self.error_bin_size)#chop up the history into bins (we dont want to specify the number of bins, instead specify the bin size)
        averaged_history=[]#average the errors over each bin
        for bin_i in range(n_bins):
            averaged_bin_start = bin_i * self.error_bin_size
            averaged_bin_end = (bin_i+1) * self.error_bin_size
            history_over_current_bin = history[averaged_bin_start:averaged_bin_end]
            # print('history_over_current_bin', history_over_current_bin)
            averaged_history.append(np.mean(history_over_current_bin))

        error_history = np.log10(np.array(averaged_history)+1e-10)
```

```python
    print('error_history[:5]', error_history[:5])
    #take the log of the averaged history because there we could really notice/see small differences that really matter. the small value at the end is so that if the argument of log is 0 it doesnt break
    report_min=np.minimum(self.error_min, error_history)
    report_max=np.maximum(self.error_max, error_history)
    fig=plt.figure()
    ax=plt.gca()#get the current axis object for this plot
    ax.plot(error_history)
    ax.set_xlabel(f"x {self.error_bin_size} iterations")
    ax.set_ylabel('log Loss')
    ax.grid()
    fig.savefig(os.path.join(self.report_path, self.report_image))
    plt.close()
```