Performing Numerical Case Studies on an Inhouse Developed Feedforward Neural Network

Bryan Lietz, Dharanidharan Arumugam, Samyukthalakshmi Saravanan

1 Introduction

Feedforward networks (FNN) are a type of artificial neural network that is composed of fully connected layers of interconnected nodes, with each layer processing and transforming the input data before passing it on to the next layer [1].

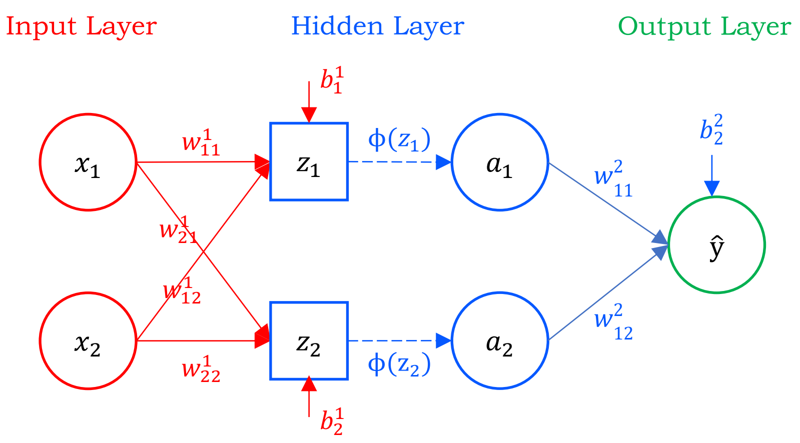

Fig.1: Schematic showing the basic elements of a feedforward neural network.

Fig. 1 shows an architecture of a simple single-layer feedforward network that consists of an input layer, a hidden layer, and an output layer. The layers are connected using weights (w) and biases (b), and each node in the network performs a linear transformation followed by a nonlinear transformation through an activation function. This is called a feedforward operation. Another important operation that is involved in FNN is backpropagation. Backpropagation is important for fine-tuning network parameters (weights (w) and biases (b)) to obtain desired prediction values. These operations are defined on a layer level. So, in a multilayer neural network, these operations are repeated over each fully connected layer.
The feedforward operation of a fully connected layer is mathematically expressed as:

$$Z = \mathrm{W}\mathrm{X} + b \qquad \dots 1a$$

$$Y = \phi(Z) \qquad \dots 1b$$

Where X is the input matrix of size m by n, m is the number of input features or number of neurons in the previous layers and n is the number of instances, W and b are the weight matrix and bias vector, respectively and ϕ is the activation function. Activation functions are important to impart non-linearity in the FNN networks.
The backpropagation operation of a fully connected layer using a minibatch gradient involves backpropagating the error ($\partial E/\partial Y$) it receives from the layer after to the layer before ($\partial E/\partial X$) which is mathematically expressed as:

$$\frac{\partial E}{\partial X} = \mathrm{W}^{\mathrm{T}} \frac{\partial E}{\partial Y} \odot \phi'(Z) \qquad \ldots 2$$

The weights and biases of each layer are updated in the following way:

$$W_{new} = \mathrm{W}_{\mathrm{old}} - \frac{\eta}{n}\frac{\partial E}{\partial Y} X^{\mathrm{T}}$$
$$b_{new} = b_{\mathrm{old}} - \frac{\eta}{n}\sum \frac{\partial E}{\partial Y} \qquad \ldots 3$$

For the last layer error is defined using mean squared error (for regression problems) averaged over the number of instances in a minibatch (n) as follows:

$$E = \frac{1}{2}\sum_{i=1}^{n}\left(y_{pred} - y_{true}\right)^2 \qquad \ldots 4a$$

Thus, the output for the final regression becomes,

$$\frac{\partial E}{\partial Y} = y_{pred} - y_{true} \qquad \ldots 4b$$

In the case of classification, instead of a mean square error, a cross-entropy error function is used.

In this project, a multi-layer feedforward network based on the above mathematical operations is developed using the NumPy Python library [2]. The rest of this report is organized as follows. Section 2 describes the details of the developed program, Section 3, 4, 5, and 6 discusses the numerical case studies carried out in the study and finally Section 7 describes the conclusions drawn from the study and discusses the scope for further investigations.

2 Details of the program developed

The program developed consists of two major class objects: 1. Network and 2. Layer. The network class objects include methods like ‘add’ for adding layers. ‘fit’ for fitting the training data and so on. The layer class object involves important methods that include feedforward and backpropagation. The developed program is presented in the Appendix section. Building a neural network model from the developed program involves the following steps:

- Defining input data
- Creating a network class object
- Adding hidden layers
- Adding the output layer (a regression or a classification layer)
- Training the model
- Make predictions using the trained model

A code snippet to build and train a multilayer fully connected neural network model is given below:

```
import numpy as np
from neural_network import Network
from layers import fcLayer, classLayer, regLayer

x_train = np.expand_dims(np.linspace(-1,1,1000),axis=0)
y_train = np.exp(x_train) + 0.02*np.random.randn(1,x_train.shape[1])
fcnet = Network()
fcnet.add(fcLayer(1, 10,'sigmoid')
fcnet.add(fcLayer(10, 10,'sigmoid'))
fcnet.add(regLayer(10, 1,'linear'))
fcnet.optimizer(eta=0.2, lamda=10, regularizer ='none')
[train_loss, test_loss] = fcnet.fit(x_train, y_train, x_train, y_train, batch_size
= 8, epochs=2500)
predictions = fcnet.predict(x_train)
```

3. Fitting the single-layer (hidden) network to a linear function

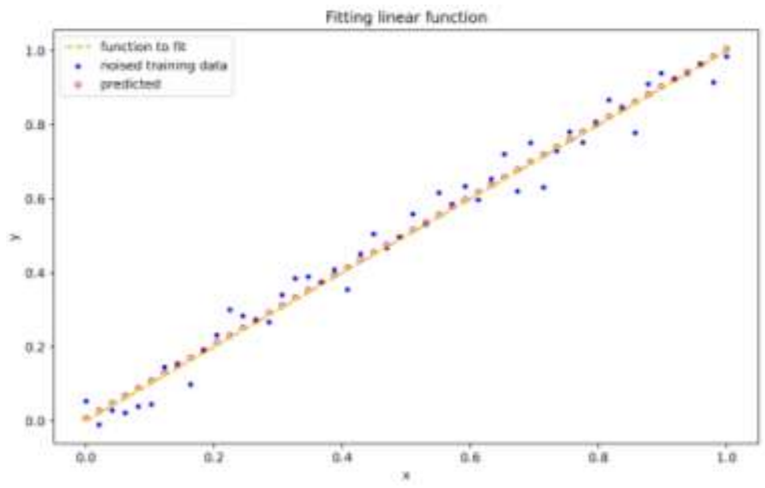

Fig. 2: Training the single hidden layer neural network to fit the noised linear function data.

One of the elemental operations in the fully connected neural network is the affine transformation, hence a single-layer neural network can easily fit a linear function. To verify this fact, linear data added with Gaussian noise is generated as given in Eq. 5

$$y = x + \varepsilon \text{ with } \varepsilon \sim N(0,0.04) \qquad \dots 5$$

A smaller dataset of size 50 is used. The generated data along with the actual function is plotted in Fig. 2. To fit the data, a singly hidden layer neural network with mere 2 neurons is used. Both the hidden layer and the output layer use the 'linear' activation function. With a learning rate of 0.01 and a batch size of 1, the network is run for 500 epochs. The predictions made by the network for the trained data are shown in Fig. 2. Two things we can observe in the figure, one is the network ignores the noise and second the predictions almost fall on the actual function line. The mean squared error of the fitted model is 0.04.

4. Fitting the single layer (hidden) network to non-linear functions.

We demonstrated above the ability of a single hidden layer neural network to capture a linear relationship between the input feature and a target variable. The real test of the single-layer network is to fit non-linear functions. Thus, in this study, two highly non-linear functions: exponential and sinusoidal functions are used to understand the influence of the hyperparameters of the network in its predictive performance. For all the numerical analyses except when the influence of the activation function is studied, the 'sigmoid' activation function is used for the exponential function, and the 'tanh' activation function is used for the sinusoidal function. In both cases, a 'linear' activation function is used for the output layers since the analysis of regression type. In all the numerical analyses conducted on these nonlinear functions, a batch size of 10 is used for the gradient backpropagation.

4.1. Generating data for non-linear functions.

The synthetic datasets for the exponential and sinusoidal functions are created by adding Gaussian noise with an amplitude of 0.02 (see Eq. 6a and Eq.6b). We generated 1000 instances in each case for the training of the network.

$$y = \exp(x) + \varepsilon \text{ with } \varepsilon \sim N(0,0.02) \qquad \dots 6a$$

$$y = \sin(x) + \varepsilon \text{ with } \varepsilon \sim N(0,0.02) \qquad \dots 6b$$

The generated noised data for the exponential and sinusoidal functions are plotted in Fig. 3a. and Fig. 3b, respectively.

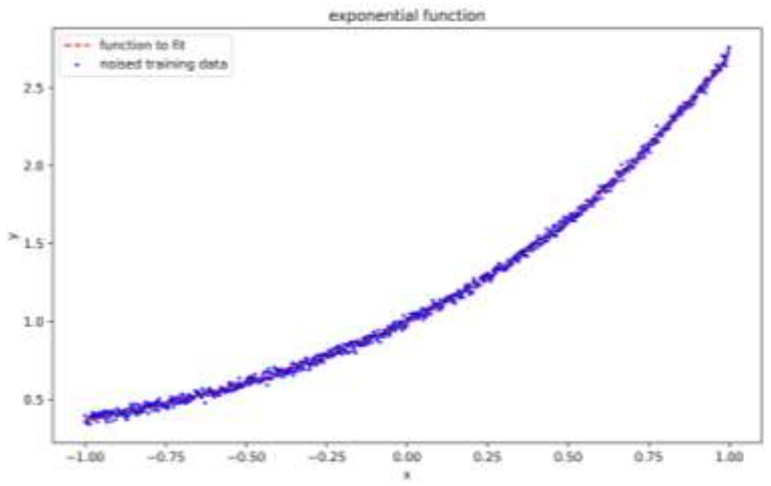

Fig. 3a: Plot of exponential function data added with Gaussian noise.

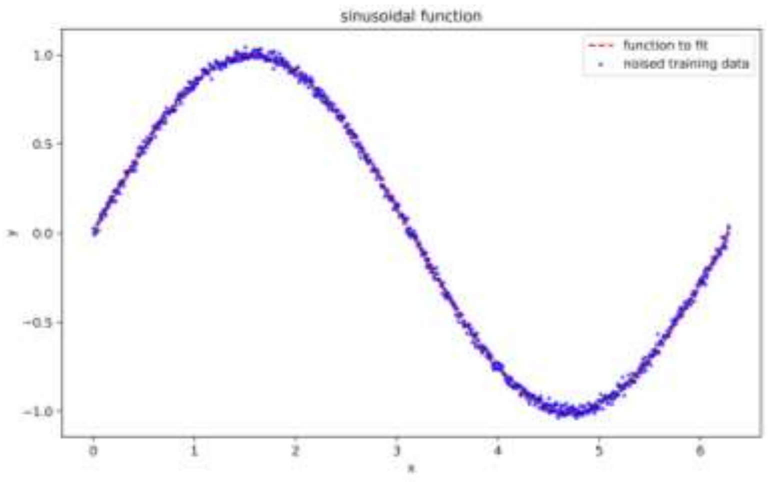

Fig. 3b: Plot of sinusoidal function data added with Gaussian noise.

4.2. Varying hidden layer size.

The first analysis of these non-linear functions involves investigating the ability of the single-layer network to fit the non-linear data for different hidden layer sizes. Hidden layers with 2, 5, 10, and 15 neurons were used in the analysis. The learning rate and epochs used in the analysis are 0.2 and 2500, respectively. Fig. 4a and Fig. 4b show predictive curves of the network for the different layer sizes. Two neurons are sufficient to accurately fit the exponential function whereas the sinusoidal function requires 10 neurons. The best mean squared error obtained by the models for exponential and sinusoidal functions are 0.195 and 0.265, respectively.

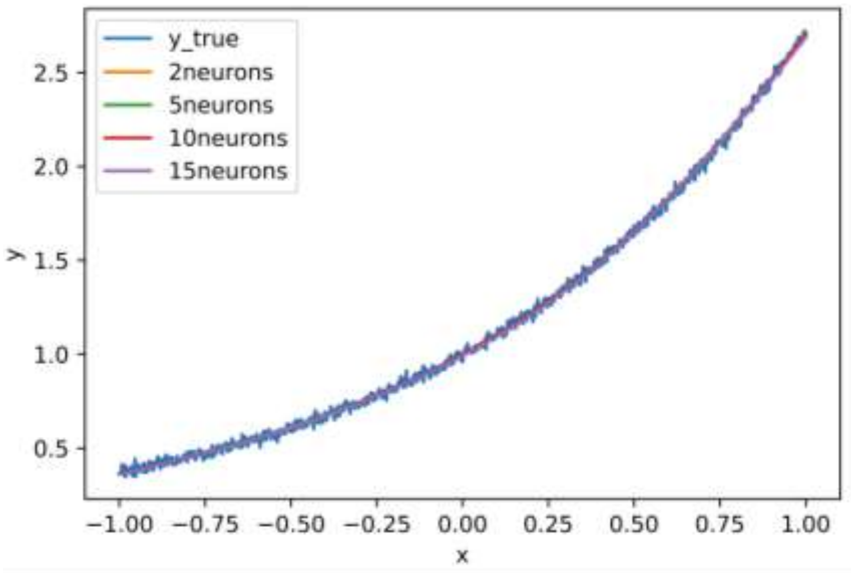

Fig. 4a: Curves fitted by the NN models of different hidden layer sizes to the exponential function.

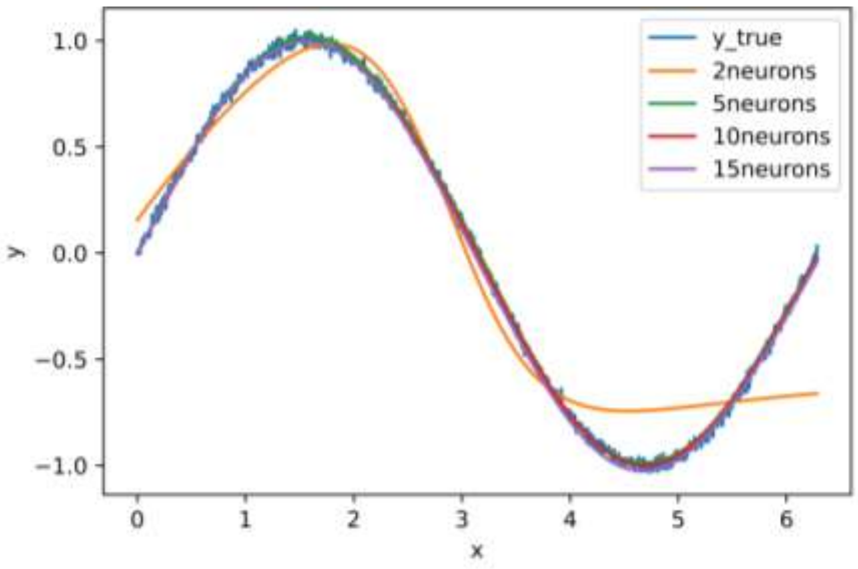

Fig. 4b: Curves fitted by the NN models of different hidden layer sizes to the sinusoidal function.

4.3. Varying epochs.

An epoch constitutes fine-tuning of the network parameters using all the training data instances in one cycle. Thus, the higher the number of epochs the better the fit obtained from the model to the training data, however, increasing the number of epochs will also increase the computational time. Also, once the convergence is reached increasing the epochs will not gain any increase in accuracy. Usually, the optimal epoch is chosen by looking at the plot between the training loss and epochs. Even in our study, we chose the optimal epoch in that way, yet to present how the model performance evolves over the epoch, the curves fitted by the model for different epochs, which included 100, 500, 1000, and 2500, are plotted. The hidden layer of the networks used in the analysis consists of 10 neurons.

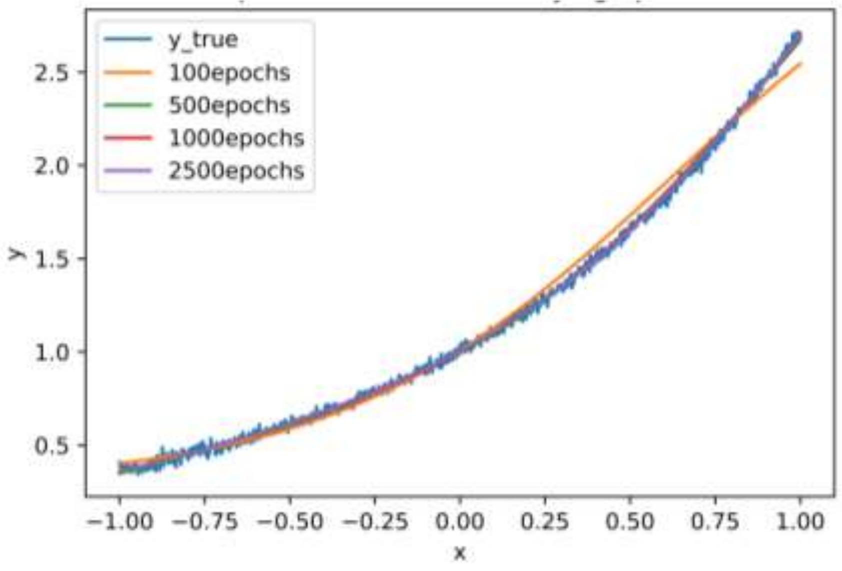

Fig. 5a: Curves fitted by the NN models with different epochs to the exponential function.

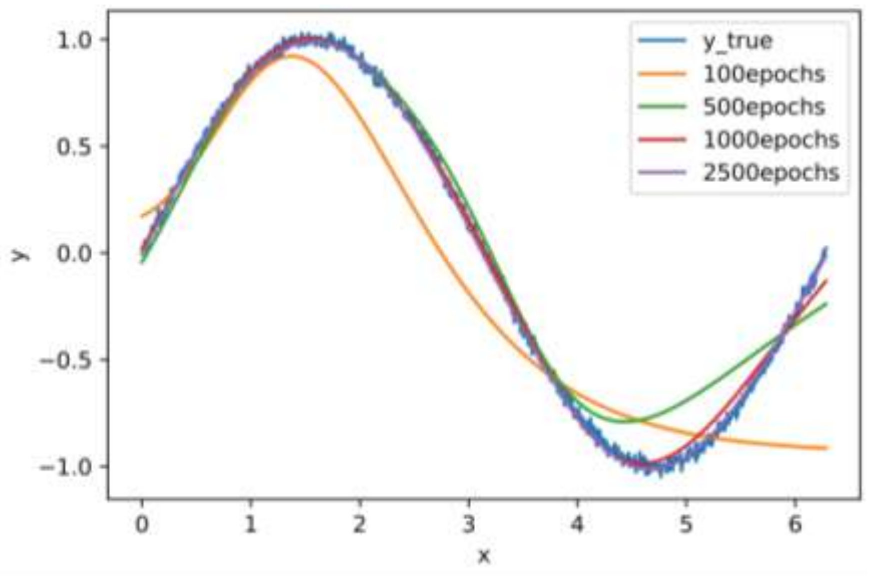

Fig. 5b: Curves fitted by the NN models with different epochs to the sinusoidal function.

A 0.2 learning is used for the mini-batch gradient descent. Fig. 5a and Fig. 5b show the predictive curves of the network for the different epochs. These plots show that exponential functions achieve faster convergence than the sinusoidal function. This may be because the exponential function is monotonic whereas the sinusoidal function has curvature reversal. The best mean squared error obtained by the models for exponential and sinusoidal functions occur at 2500 epochs and are 0.215 and 0.280, respectively.

4.4. Varying learning rates.

One of the important tasks in neural network training is the fine-tuning of the hyperparameters of the optimizer, in our case, it is just the learning rate. Smaller learning rates may lead to slower convergence and higher learning rates may not achieve convergence at all. The optimal learning rates need to be found by looking at the training loss over the number of epochs. In practice, the optimal parameters of the optimizer are usually fixed through automation. In our case, to learn how the learning rates influence the convergence, four different learning rates (0.05,0.1,0.2 and 0.3) are selected and the training loss vs epochs plots corresponding to each learning rate are constructed. Fig. 6.a and 6. b show the plots of training loss versus epochs for different learning rates for exponential and sinusoidal functions, respectively. In the case of the exponential function, models with all four learning rates achieve a good convergence at epoch 500. However, we usually prefer a learning rate with a sharper turn (which means rapid convergence), in that regard, learning rates 0.2 and 0.3 are preferred. In the case of sinusoidal models, all the selected learning rates achieve good convergence near the 1200 epoch. However, unlike the case of the exponent function, it is not clear enough to choose the optimal learning parameter by looking at the convergence trends. Nevertheless, with an epoch of 2500, any one of the learning parameters analyzed can be selected as the optimal learning parameter.

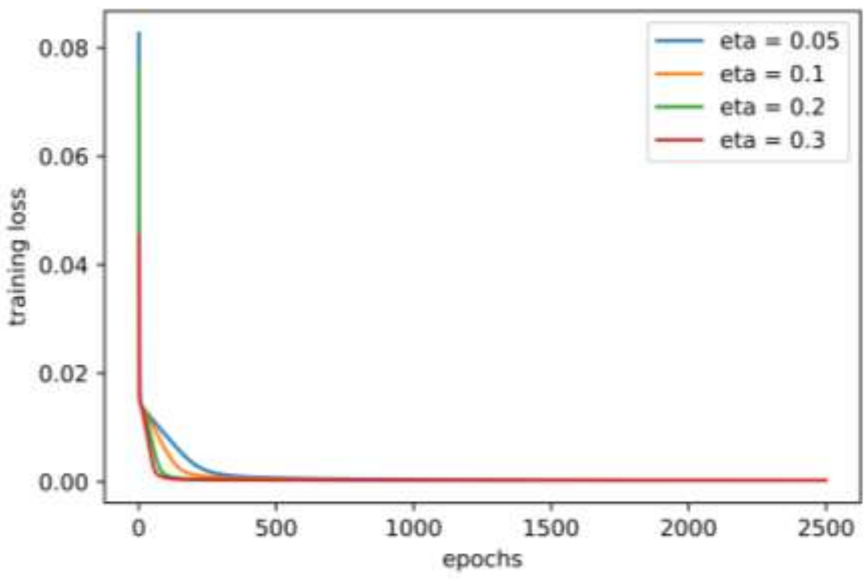

Fig. 6a: Curves fitted by the NN models with different learning rates to the exponential function.

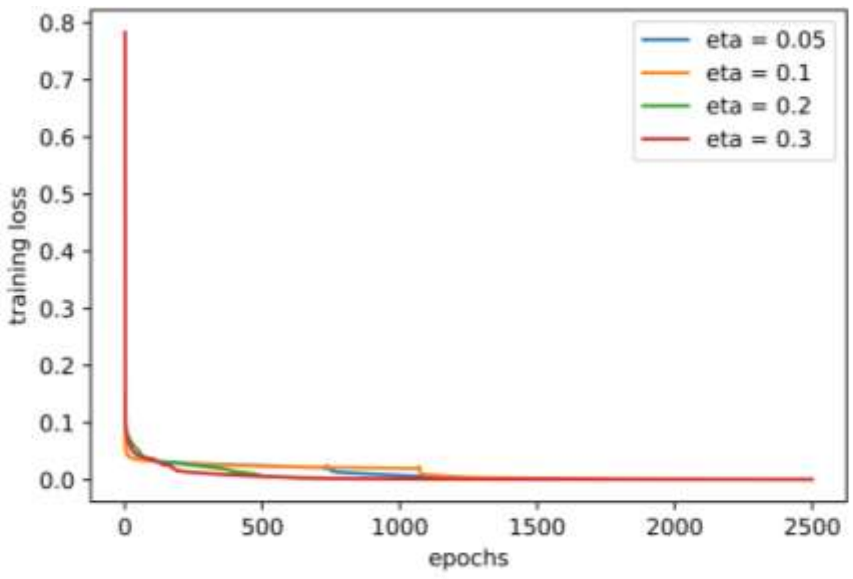

Fig. 6b: Curves fitted by the NN models of different learning rates to the sinusoidal function.

4.5. Changing activation functions

Activation functions are critical for imparting non-linearity in fully connected neural network models. 'Sigmoid', 'tanh', 'SoftMax', 'ReLU', and 'eLU' are some of the popularly used activation functions [3]. In our study, we used sigmoid, tanh, and ReLU to understand their influence on the fitting ability of the neural network models. Only the hidden layer activation functions are changed and for the output layer linear activation is used since the task is regression in nature. A learning rate of 0.2 and an epoch of 2500 is used in the analysis. Fig. 7a shows the fitted curves of the NN models with different activation functions to the exponential function. Both 'Sigmoid' and 'tanh' achieve a very good fit to the exponential function. 'ReLU' is slightly off for the upper half of the curve. Fig. 7b shows the fitted curves of the NN models with different activation functions to the sinusoidal function. In this case, only 'tanh' achieves a very good fit. Though 'sigmoid' is marginally off from the actual curve, still it can capture the trend of the sine function. Among the three 'ReLU' performed very poorly, it resulted in a poor straight-line fit for a sinusoidal function. 'ReLU' may work better with additional layers and larger layer sizes.

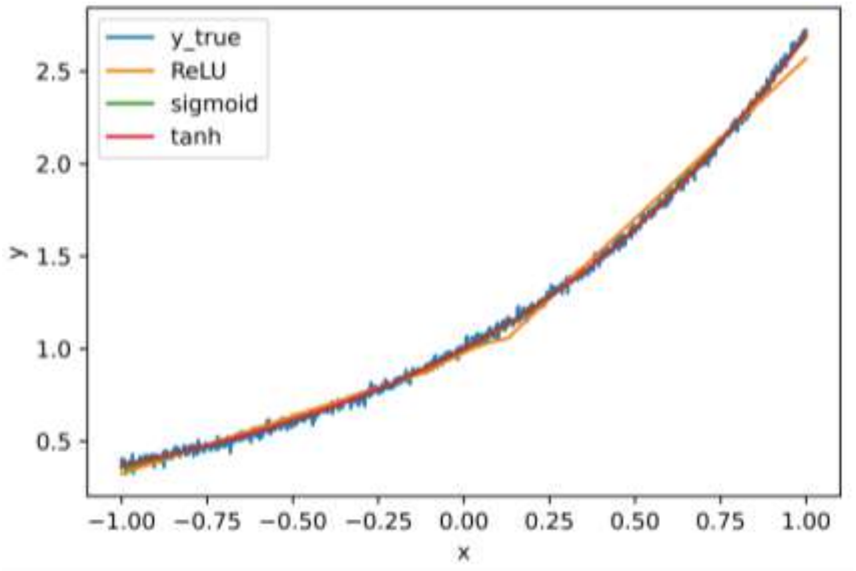

Fig. 7a: Curves fitted by the NN models with different activation functions to the exponential function.

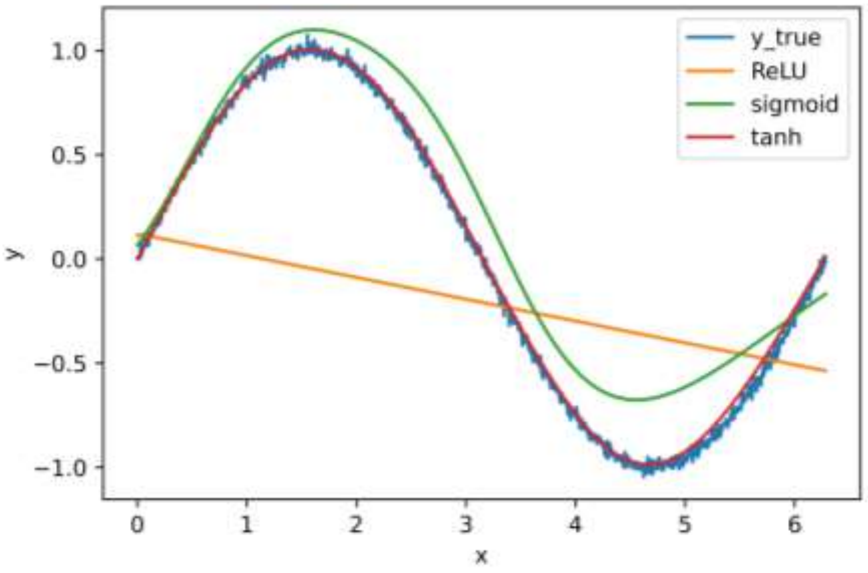

Fig. 7b: Curves fitted by the NN models of different activation functions to the sinusoidal function.

5. Interpolation and extrapolation

So far, we have only investigated the ability of a single-layer neural network to fit the right curve to the training data. With the right combination of the number of layers, layer size, activation function, learning parameters, and network hyperparameters, we can practically achieve a good fit for any complex function. The challenge of neural networks lies in making the right prediction for unknown, unexposed data.

5.1. Interpolation

We first start with investigating the predictive performance of the single-layer neural network or the unknown data which falls within the range of the input data, in other words, we are testing the ability of the single-layer NN models to interpolate. Once again, we use the same exponential and sinusoidal functions for the analysis. We take a total of 100 instances in each case and partition them into training and testing data with a data split ratio of 75% to 25%. The learning rate of 0.2, epoch of 2500, and layer size of 10 are found to be optimal and used in the analysis. The batch size is reduced to 4 since the data size is small. The amplitude of the Gaussian noise is increased to 0.05 so that the neural net generalizes well. The prediction made at the unknown locations by the optimal single-layer NN model for the exponential function data is shown in Fig. 8a. As you can see in the figure, the trained NN model interpolates the unknown data well. But in the case of the sinusoidal function (refer to Fig. 8b), the NN models struggle to achieve good predictions for the unexposed input values inside the exposed range.

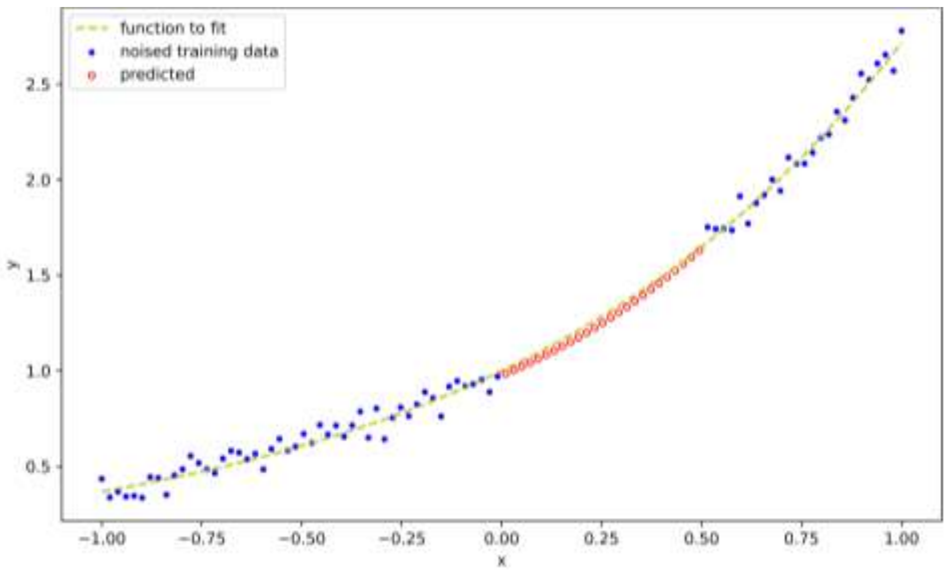

Fig. 8a: Predicting functional values of the exponential function for the unexposed data that are inside the exposed input range.

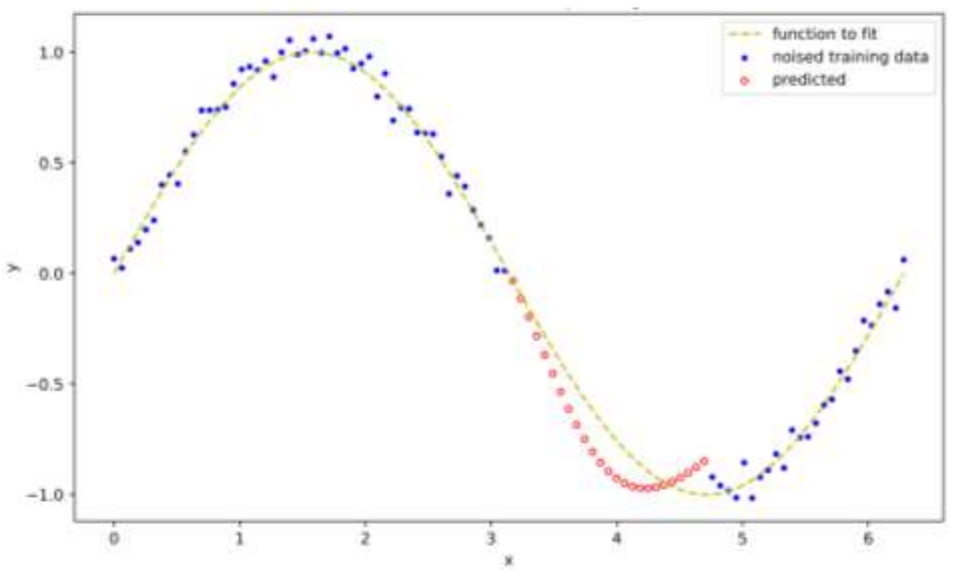

Fig. 8b: Predicting functional values of the exponential function for the unexposed data that are inside the exposed input range.

5.2. Extrapolation

Next, we try to understand the ability of the single-layer NN model to predict the data outside the input range it is exposed to, in other words, we are testing the ability of the single-layer NN models to interpolate. We again take a total of 100 instances in each case and partition them into training and testing data with a data split ratio of 75% to 25%. The network hyperparameters are the same as the parameters used in the interpolation task (refer to section 5.1).

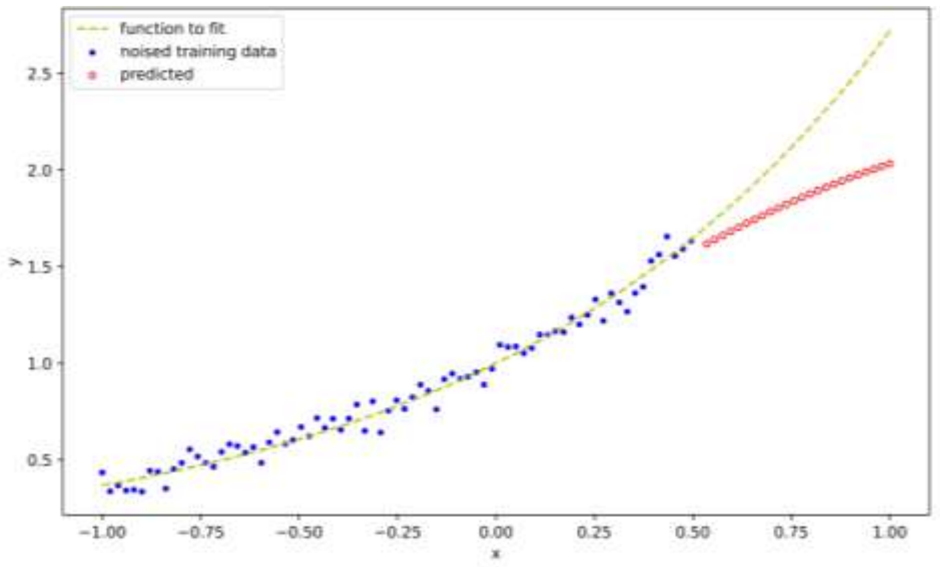

Fig. 9a: Predicting functional values of the exponential function for the unexposed data that are outside the exposed input range.

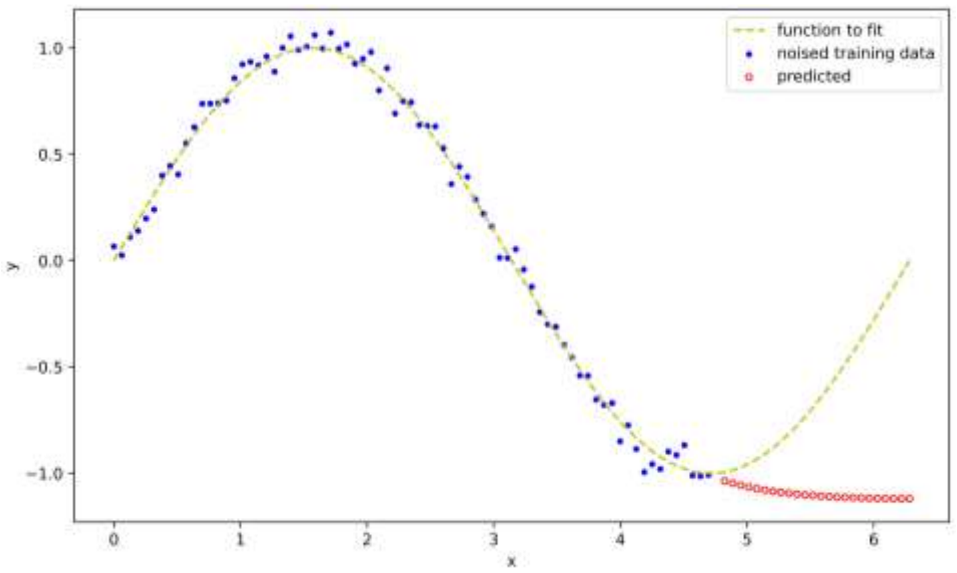

Fig. 9b: Predicting functional values of the exponential function for the unexposed data that are outside the exposed input range.

Fig. 9a and Fig. 9b show the predictions made by the optimized single-layer NN model for the test data that falls outside the training input data range. As observed in these figures, the NN models perform poorly in the case of extrapolation for both exponential and sinusoidal function data.

6. Regularization

A major limitation of the neural network models, especially the fully connected feedforward networks, is the tendency to overfit the data. Overfit models do not generalize well which means they don't extend their good predictive performance demonstrated for the training data to the unexposed, unlearnt data. Thus, overfit models are said to have a low bias (higher accuracy) and high variance (sensitivity to noise). In our program, we incorporated 'L1' and 'L2' regularization techniques to overcome this limitation. The objective of regularization is to achieve low variance without compromising accuracy [4]. Thus, proper regularization of feedforward neural networks can result in a low bias and low variance model. L1 and L2 regularization involves adding penalty terms to the cost function (refer to Eq.7a and Eq.7b) so that it prevents the uninhibited growth of weights.

The penalty term for L1 regularization is given as,

$$\lambda \sum_{i=1}^{m} |w_i| \qquad \dots 7a$$

The penalty term for L2 regularization is given by,

$$\lambda \sum_{i=1}^{m} w_i^2 \qquad \dots 7b$$

Here, λ is a regularization parameter, m is the number of weight parameters and w_i is i^{th} weight term.

To understand the effect of regularization on the predictive performance of a single-layer neural network, a linear model dataset is created with just 20 instances. The layer size is increased to 100 so that the NN model can overfit the data. Then the influence of the 'L1' and 'L2' regularization is investigated. Fig. 10 shows the curves that fit my NN models with no regularization, 'L1' and 'L2' regularization to a linear dataset. As you can see, the NN model with no regularization tends to learn the noises and resulting in fitting a nonlinear polynomial curve to a linear function. 'L2' regularization achieves the correct linear fit of the data. 'L1' regularization results in the underfitting of the model. Here, the, λ is taken as 0.1.

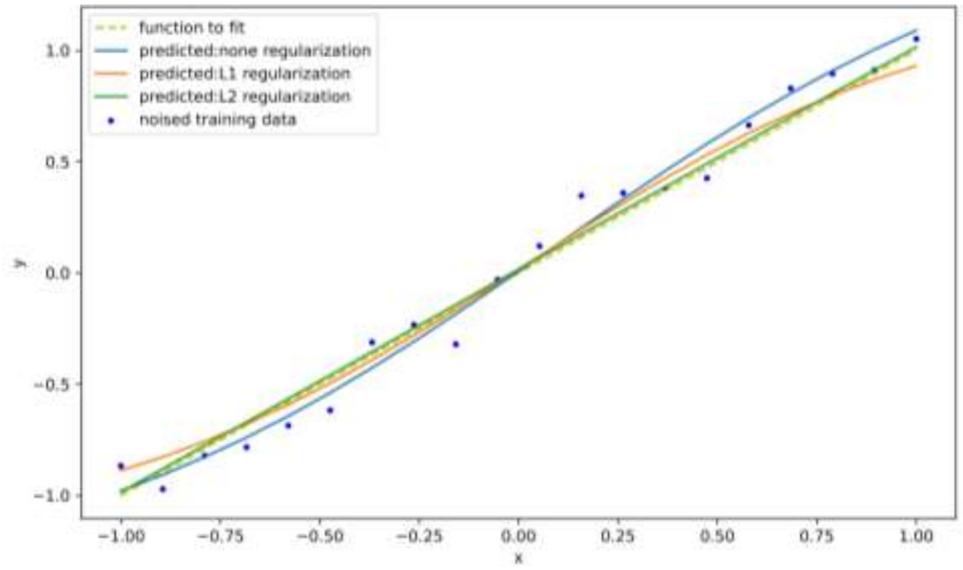

Fig. 10: Influence of regularization on an overfit NN model

The regularization parameter λ needs to be fine-tuned to obtain a desired low variance and low bias model. Larger λ values result in higher bias and smaller λ values result in higher variance. Hence, to understand the effect of λ on the model fitting of the single-layer feedforward network, the NN models are regularized with four different λ values : 0, 0.1, 1, and 5. Fig. 11 shows the influence of λ of models with 'L1' regularization on the fitted curves to the data generated with a linear function. λ value of 0 indicates that the model is without any regularization. As seen in the figure, higher λ values result in a low variance linear model but with a high bias. When $\lambda = 5$, most of the weight parameter values are zeroed resulting in a constant model. The optimal value of λ may lie in the unexplored range of 0.1 to 1. The same exercise is repeated for L2 regularization and the resulting model curves are plotted in Fig. 12. The plots again show the similar model behavior observed in the L1 regularization analysis. However, we can achieve the desired low variance low bias linear model for the λ value of 0.1.

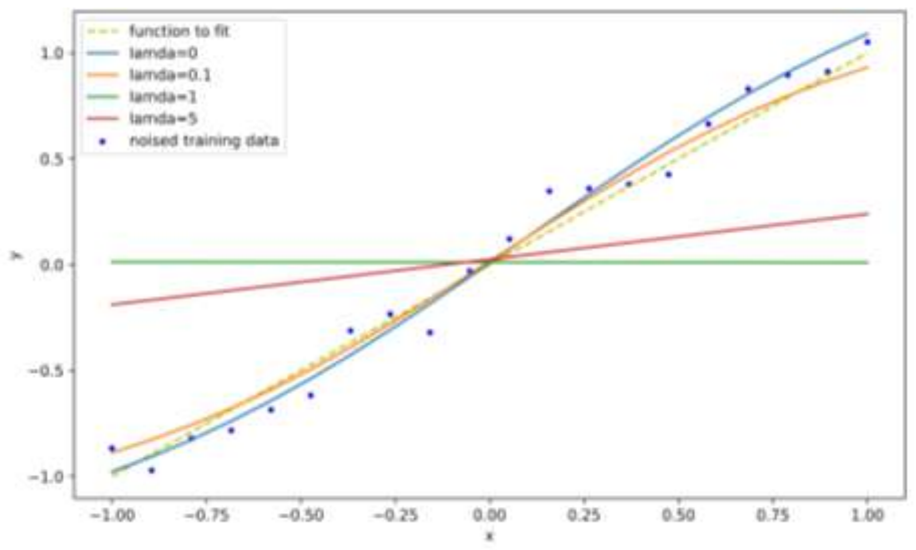

Fig. 11: Influence of the regularization parameter on an L1 regularized NN models

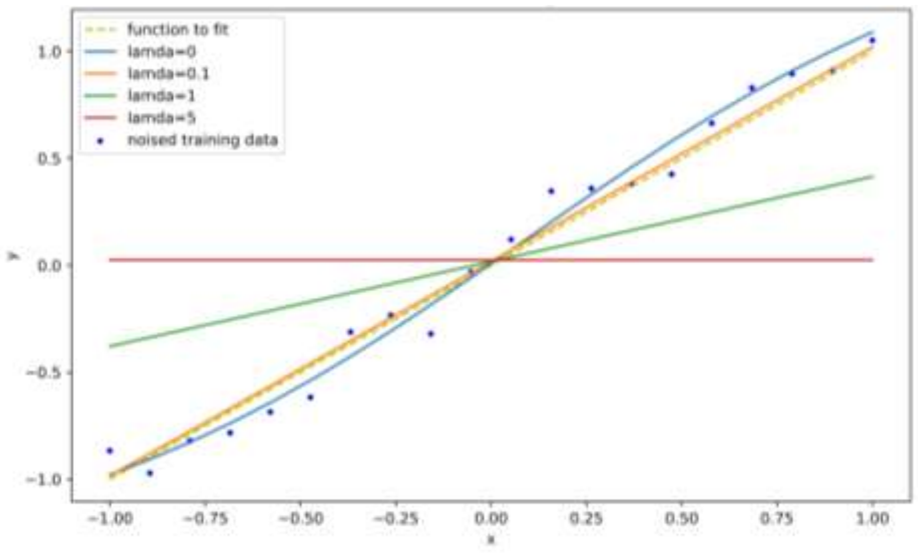

Fig. 12: Influence of the regularization parameter on an L1 regularized NN models

7. Concluding remarks and discussion

In this project, we developed a Python program for a multi-layer feedforward network that can perform regression and classification tasks. We also conducted several numerical case studies to understand the influence of network hyperparameters on the predictive performance of single-layer networks. One of the case studies shows that the NN models are poor in extrapolating the functional values. We also investigated the use of L1 and L2 regularization to achieve a low variance low bias model in the case of model overfitting.

Though we conducted an extensive numerical case study, there are still many investigations that can be done with the developed program to further our understanding of the feedforward networks.

One of the important areas to explore is to investigate the influence of hyperparameters on classification tasks. Specifically, understanding the influence of activations functions in constructing the decision boundaries for classification. Also, in our study, we only used the minibatch gradient descent method for optimizing the network parameters. But there are several optimization techniques available that perform better than the minibatch gradient descent method. Comparing the convergence characteristics of the optimizing techniques will also make an interesting study.

References

1. Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, *39*(1), 43-62.
2. Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster.
3. Sharma, S., Sharma, S., & Athaiya, A. (2017). Activation functions in neural networks. *Towards Data Sci*, *6*(12), 310-316.
4. Girosi, F., Jones, M., & Poggio, T. (1995). Regularization theory and neural network architectures. *Neural Computation*, *7*(2), 219-269.

Appendix

1. Neural network.py

```python
import numpy as np

class Network:
    def __init__(self):
        self.layers = []
        self.eta = 0.01
        self.lamda = 2
        self.regularizer = 'none'

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)

    # predict output for given input
    def predict(self, input_data):
        output = input_data
        for layer in self.layers:
            output = layer.feedforward(output)
        return output

    def optimizer(self,eta,lamda,regularizer):
        self.eta = eta
        self.lamda = lamda
        self.regularizer = regularizer

    # train the network
    def fit(self, x_train, y_train, x_test, y_test, batch_size, epochs):

        no_of_instances = x_train.shape[1]
        no_of_batches = np.ceil(no_of_instances/batch_size).astype('int')
        permuted_indices = np.random.permutation(np.arange(0,no_of_instances))
        x_train = x_train[:,permuted_indices]
        y_train = y_train[:,permuted_indices]

        idx = np.arange(0,no_of_instances,batch_size)
        if no_of_batches != len(idx)-1:
            idx = np.append(idx,no_of_instances)

        train_loss = np.zeros((epochs,1))
        test_loss = np.zeros((epochs,1))
        # training loop
        for i in range(epochs):
            loss_tr = 0
            for j in range(no_of_batches):
                x = x_train[:,idx[j]:idx[j+1]]
                y = y_train[:,idx[j]:idx[j+1]]
                # forward propagation
                output = x
                for layer in self.layers:
                    output = layer.feedforward(output)

                # compute train loss
                [loss, error]= self.layers[-1].estimate_loss(y, output)
                loss_tr += loss

                for layer in reversed(self.layers):
                    error = layer.backpropogate(error, self.eta, self.lamda,
self.regularizer)

            #compute test loss
            output = x_test
            for layer in self.layers:
                output = layer.feedforward(output)

            [loss_ts, error]= self.layers[-1].estimate_loss(y_test, output)

            # calculate average error on all samples
            loss_tr /= no_of_instances
            loss_ts /= no_of_instances
            train_loss[i,0] = loss_tr
            test_loss[i,0] = loss_ts
            print('epoch %d/%d   train_loss=%f   test_loss=%f' % (i+1, epochs,
loss_tr, loss_ts), end='\x1b\r')

        return train_loss, test_loss
```

2. layers.py

```python
import numpy as np
from activation_functions import activation, derivative

class fcLayer:
    # input_size = number of input neurons
    # output_size = number of output neurons
    def __init__(self, input_size, output_size,act_fn):
        self.weights =
np.random.rand(output_size,input_size)*np.sqrt(2/input_size) #he's initialization
        self.bias = np.zeros((output_size,1))
        self.act_fn = act_fn

    # feedword
    def feedforward(self, input_data):
        self.input = input_data
        self.netinput = np.matmul(self.weights,self.input) + self.bias
        self.output = activation(self.netinput,self.act_fn)
        return self.output

    # backpropogation
    def backpropogate(self, output_error, eta, lamda, regularizer):
        n = output_error.shape[1]
        output_error = derivative(self.netinput,self.act_fn)*output_error
        input_error = np.matmul(self.weights.T, output_error)
        errW = np.matmul(output_error,self.input.T)
        errB = np.expand_dims(np.sum(output_error,axis=1)/n,axis=1)

        dB = eta*(errB/n)
        if regularizer == 'none':
            dW = eta*(errW/n)
        elif regularizer == 'L1':
            dW = eta*(errW/n) + eta*(lamda/n)*np.sign(self.weights)
        elif regularizer == 'L2':
            dW = eta*(errW/n) + eta*(lamda/n)*self.weights

        self.weights -= dW
        self.bias -= dB
        return input_error

class classLayer:
    # input_size = number of input neurons
    # output_size = number of output neurons
    def __init__(self, input_size, output_size):
        self.weights = 
np.random.rand(output_size,input_size)*np.sqrt(1/input_size)
        self.bias = np.zeros((output_size,1))

    # predict output for given input
    def estimate_loss(self,y_true,y_pred):
        size  = y_true.shape[0]
        if size == 1:
            loss  = - np.sum(np.log(y_pred)*y_true + np.log(1-y_pred)*(1-y_true))
        else:
            loss  = - np.reduce_sum(np.log(y_pred)*y_true)
        error = y_pred-true
        return (loss, error)

    # feedword
    def feedforward(self, input_data):
        self.input = input_data
        self.netinput = np.matmul(self.input, self.weights) + self.bias
        self.output = np.exp(self.netinput)
        self.output = self.output/np.sum(self.output,axis=0)
        return self.output

    # backpropogation
    def backpropogate(self, output_error, eta, lamda, regularizer):
        n = output_error.shape[1]
        input_error = np.matmul(self.weights.T, output_error)
        errW = np.matmul(output_error,self.input.T)
        errB = np.expand_dims(np.sum(output_error,axis=1)/n,axis=1)

        # update parameters
        dB = eta*(errB/n)
        if regularizer == 'none':
            dW = eta*(errW/n)
        elif regularizer == 'L1':
            dW = eta*(errW/n) + eta*(lamda/n)*np.sign(self.weights)
        elif regularizer == 'L2':
            dW = eta*(errW/n) + eta*(lamda/n)*self.weights

        self.weights -= dW
        self.bias -= dB
        return input_error

class regLayer(fcLayer):
    def estimate_loss(self,y_true,y_pred):
        loss  = 0.5*np.sum(np.mean((y_pred-y_true)**2,axis=0));
        error = y_pred-y_true
        return (loss, error)
```

3. activations.py

```
import numpy as np

def activation(x, act_fn):
    if act_fn == 'sigmoid':
        z = 1/(1+np.exp(-x))
    elif act_fn == 'tanh':
        z = np.tanh(x)
    elif act_fn == 'linear':
        z = np.copy(x)
    elif act_fn == 'softmax':
        z = np.exp(x)
        z = z/np.sum(z, axis=0)
    elif act_fn == 'ReLU':
        z = x
        z[x < 0] = 0
    elif act_fn == 'eLU':
        z = 0.3*(np.exp(x)-1)
        z[x >= 0] = x[x >= 0]
    else:
        z = np.copy(x)
        print("invalid function so Linear function used")
    return z

def derivative(x, act_fn):
    if act_fn == 'sigmoid':
        z = 1/(1+np.exp(-x))
        dz = z*(1-z)
    elif act_fn == 'tanh':
        dz = 1-np.tanh(x)**2
    elif act_fn == 'linear':
        dz = np.ones_like(x)
    elif act_fn == 'ReLU':
        dz = np.ones_like(x)
        dz[x < 0] = 0
    elif act_fn == 'eLU':
        a = 0.3
        dz = a + np.exp(x)
        dz[x >= 0] = 1
    else:
```

```python
        dz = np.ones_like(x)
        print("invalid function so Linear function used")
    return dz

```