Deep Learning
Spring-2025
Assignment 1
Implementation of Linear and Logistics Regression

Submission:

Submit all of your codes and results in a single zip file with name FirstName_RollNumber_01.zip

- Submit single zip file containing
  - codes (b) report.pdf (c) Saved Models (d) Readme.txt
- There should be a Report.pdf detailing your experience and highlighting any interesting results. Kindly don't explain your code in the report, just analyze the results. Your report should include your comments on the results of all the steps, with images, for example what happened when you changed the learning rate etc.
- Readme.txt should explain how to run your code, preferably it should accept the command line arguments e.g dataset path used for training the model.
- The assignment is only acceptable in .py files. No Jupyter notebooks i.e. ipynb files will be accepted.
- In the root directory, there should be a python file, a report and a folder containing saved models.
- Root directory should be named as **FirstName_RollNumber_01**
- Follow all the naming conventions.
- Each convention/rule violation, there will be a 2% penalty.
- Email instructor and TA's if there are any questions. You cannot look at others code or use others code, however you can discuss with each other. Plagiarism will lead to a straight zero with additional consequences as well.
- 10% (of obtained marks) deduction per day for late submission.
- Screenshots will not be accepted. you have to attach clear images of your plots in your report.
- You should submit both working code and a report. If your code is not working, then your report will also not be accepted. Similarly, only submitting a report with no accompanying code will also not be accepted.

Due Date: 11:59 PM on Wednesday, 6th March 2025

PS: Late submission with 10% penalty per day will only be accepted till 16th March 2025 11:59 PM (Sunday) after that your assignment 01 will be marked ZERO.

Note: For this assignment (and for others in general) you are not allowed to search online for any kind of implementation. Do not share code or look at the other's code. You should not be in possession of any implementation related to the assignment, other than your own. In case of any confusion please reach out to the TA's and instructor (email them or visit them).

Objectives: In this task you will write the code for implementing the linear regression using Numpy arrays. The goals of this assignment are as follows.

- Hand-on experience on Numpy arrays
- Understand how to design and implement an efficient architecture for linear regression
- For each layer:
    - Initialize the number of features and learnabe parameters
    - Implement feedforward
    - Keep track of the gradients
    - Understand the mechanism of optimization
- Compare the performance of the model on different hyper-parameters i.e. batch size, learning rate, array initialization
- Students will learn how to make a data-loader for the provided dataset.

NOTE: You can only use Numpy for code implementations. It's recommended that you use VS Code for debugging.

Report: You have to write a report explaining, what is your implementation logic? How you came up with the optimized weights (how you find learning rate?). Which numpy array initialization performs best on the given dataset? Finally, your comment?

Task 1: Implement linear regression using Stochastic Gradient Descent via Numpy arrays

1.1 Linear Regression using Stochastic Gradient Descent:

Linear regression is an algorithm that provides a linear relationship between independent variables and a dependent variable to predict the outcome of future events.

1.1.1 Feed Forward: Mathematically, we can predict the outcome of the future events through the following equation

$$\hat{y} = X^T w$$

You can calculate a prediction of the outcome $\hat{y}$, using the above formula. X is the input data variable, where θ is the learnable parameter. The dimensions of the above variables should be:

- $X \in R^{d \times N}$: is a matrix of input data. d is the features size and N is the number of samples.
- $w \in R^{d \times 1}$: weights/coefficient of linear function.
- $\hat{y} \in RN \times 1$: predicted value, one value for each sample.

1.1.2. Compute Loss: A loss will be computed between ground truth targets (y) and predicted targets $\hat{y}$.

$$L(y, \hat{y}) = \frac{1}{2N} \sum_{i=1}^{N} \left(y_i - \hat{y}_i \right)^2$$

1.1.3. Gradient: The gradient of the equation (2) can be computed by the following equation:

$$grad = \frac{\delta L}{\delta w} = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \hat{y}_i \right) x_i$$

1.1.4. Update the Parameter:

The learnable parameter w can be updated by the following equation:

$$w = w - learning_{rate} * grad$$

1.2 Dataset

For this task we will be using the California Housing Dataset. The dataset csv files are attached. This dataset comes as part of sklearn library. You have to use the dataset of testing and training as provided in df_train and df_test csv files. There are 20640 samples, each sample consists of 8 features and target value, i.e. median house price. Our objective is to design a linear for predicting median house price. The target column values lie between the range 0.15 - 5. To load the dataset, you can use the following lines:

```
import pandas as pd
df_train =    pd.read_csv("./path_to_csv.csv")
df_test  =    pd.read_csv("./path_to_csv.csv")
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	target
14196	3.2596	33.0	5.017657	1.006421	2300.0	3.691814	32.71	-117.03	1.030
8267	3.8125	49.0	4.473545	1.041005	1314.0	1.738095	33.77	-118.16	3.821
17445	4.1563	4.0	5.645833	0.985119	915.0	2.723214	34.66	-120.48	1.726
14265	1.9425	36.0	4.002817	1.033803	1418.0	3.994366	32.69	-117.11	0.934
2271	3.5542	43.0	6.268421	1.134211	874.0	2.300000	36.78	-119.80	0.965
...	...	...	...	...	...	...	...	...	...
11284	6.3700	35.0	6.129032	0.926267	658.0	3.032258	33.78	-117.96	2.292
11964	3.0500	33.0	6.868597	1.269488	1753.0	3.904232	34.02	-117.43	0.978
5390	2.9344	36.0	3.986717	1.079696	1756.0	3.332068	34.03	-118.38	2.221
860	5.7192	15.0	6.395349	1.067979	1777.0	3.178891	37.58	-121.96	2.835
15795	2.5755	52.0	3.402576	1.058776	2619.0	2.108696	37.77	-122.42	3.250

16512 rows × 9 columns

Figure 1: Training Dataframe output example.

The details of each feature are as follows:

Table 1: California House Dataset Information

feature_names	MedInc, HouseAge, AveRooms, AveBedrms, Population, AveOccup, Latitude, Longitude
Attribute Information:	• MedInc : Median income in block group • HouseAge : Median house age in block group • AveRooms : Average number of rooms per household • AveBedrms : Average number of bedrooms per household • Population : Block group population • AveOccup : Average number of household members

	• Latitude : Block group latitude • Longitude : Block group longitude

The above code read training and testing dataset.

I. **Target** is the value you have to predict, and which will model learn via loss function.
II. **Rest** of the columns is the features.
III. **Testing set** used for evaluation purpose only, you'll have to produce results in evaluation.

1.3 Normalization:

Normalization allows us to convert all the features to the same scale. For each feature we will compute its mean and standard deviation. Then we will subtract the mean from each observation and divide it by standard deviation to get the normalized values.

$$x_{ij} = \frac{(x_{ij} - \mu_j)}{\sigma_j}$$

Calculate a normalized value (Z), using the above formula. The symbols are:

- x_{ij} : is the j^{th} feature of the ith sample.
- μ_j, σ_j : mean and standard deviation of the jth feature

NOTE:

- You must only compute mean and variance of the training data.
- Same mean and standard deviation should be used when inferring over the validation or training data.
- Mean and standard deviation should be saved as part of the model.

1.4 Stochastic Gradient Descent(SGD)

SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

It is also common to sample a small number of data points instead of just one point at each step and that is called "mini-batch" gradient descent. Mini-batch tries to strike a balance between the goodness of gradient descent and speed of SGD.

1.5 Implementation Details

1.5.1. Data Splitting

After reading the dataset (both samples and their ground-truth), you should divide the training set into two subsets, training, and validation. DO NOT USE BUILT-IN LIBRARY.

```
df_train, df_val = split_df(df_train)
```

```
If you faced any issues, use X for features and Y for labels and then split these.
```

Where $X \in R^{d \times N}$, $Y^{\epsilon} R^{N \times 1}$.

train_X, traind_Y or df_train: assign first 0.85*N samples to the training set.

val_X, val_Y or df_val: assign next 0.15*N samples to the validation set.

Note: these are numpy arrays or dataframes, use the method in which you are comfortable.

1.5.2. Data preprocessing

Compute mean and standard deviation, and then pass the input X to the normalize() function.

```
Me = set_mean(X)
St = set_standardDeviation(X)
[X] = normalize(X, Me, St)
```

The argument of normalize() function is X where $X \in R^{d \times N}$, normalize() function will return the numpy array of shape $(d\ x\ N)$. Kindly refer to section 1.3. related to the in-depth details for the implementation of this function. Don't forget to save Me and St in the model.

1.5.3. Data Loader

Prepare a DataLoader class using pytorch. In this you can use the pytorch library to create a dataloader. In the initializer method of dataset class, you have to pass the dataframe or input and output arrays and in results a processed data in the batch form return.

Three main methods utilized while creating dataloader:

- __init__ □ Initializer method, class variables should be defined here.
- __getitem__ □ Get Item will return a batch of processed data based on the provided id's.
- __len__ □ this methods will return the overall size of the data.

1.5.4. Initialize Network

There should be two network you have to define. One with 8 neurons in hidden layer and 2nd network must use 2 neurons in the hidden layer. The network should contain only 3 layers (input, hidden and output).

```
net = linear_regression_network(N)
```

For example if you pass following parameters to this function:

```
net = linear_regression_network(8)
```

- Use `HI Initialization` to initialize the weights. You can use any built-in library for this task only.
- It should return you the network architecture with parameters initialized:

```
self.theta = dx1
```

1.5.5. Training

Create two variables of dataframe in as X and Y. X is used for data as input and Y for the output. You'll have to separate the features and output labels in train_X and train_Y for df_train dataframe.

```
[model, loss_epoch_tr, loss_epoch_val] = train(net, train_X, train_Y, val_X, val_Y, batch_size, n_epochs, lr)
```

- This function returns a trained model
- `net` network architecture
- `train_X` and `train_Y` will be used for training
- `val_X` and `val_Y` will be used for validation
- `lr is the` learning rate
- `batch_size` tells how many examples to pick for each iteration.
- `n_epochs` how many training epochs to run

Note: *Please make sure your code is modular. You can divide your training process into following functions*

Implement the mini-batch gradient descent algorithm.

- Apply one for loop to iterate over the number of epochs, then use a nested for loop to iterate over the number of the samples that you set in the batch. Then, implement the following functions inside the nested for loop.

1.5.6. y_hat = feed_forward(X_b)

This function will forward through your input examples. Kindly refer to section 1.5.2. for the dimensions of X_b**.** Kindly refer to Section 1.1.1 for the details of the implementation of feed forward function. The feed_forward() function will return the predicted output. Note: size of X_b during training will be determined by the batch size.

1.5.7. loss = l2_loss(groundTruth, y_hat)

This function will tell how far are your predictions from the ground-truth. Apply L2 loss as a cost function (KIndly refer to the section 1.1.2 for the details). Kindly refer to section 1.5.2. for the dimensions of train_Y**.** Refer section 1.5.5. for the details of y_hat. Note during training groundTruth
= train_Y and during testing it will be test_Y.

1.5.8. grad = compute_gradient(train_X, train_Y, y_hat)

This function will find the gradient. Kindly refer to section 1.1.3 related to the in-depth details for the implementation of this function. Refer section 1.5.2 for finding the details of train_X, train_Y. Refer section 1.5.5. for finding the details of y_hat.

1.5.9. self.theta = optimization(lr, grad, theta)

This function will multiply the learning rate with gradient and subtract it from the *theta* to update the *theta*. Kindly refer to section 1.1.4 related to the in-depth details for the implementation of this function. Refer section 1.5.4 for finding the details of lr. Refer section 1.5.7 for finding the details of grad.

$$self.theta = self.theta - lr * gradient$$

1.5.10. train_model = pickle.dump(model, open('model.pkl', 'wb'))

After the training, save the trained model via pickle library or some other library. To save the ML model using Pickle all we need to do is pass the model object into the dump() function of Pickle. This will serialize the object and convert it into a “byte stream” that we can save as a file called model.pkl.

1.5.11. test_model = test_function(train_model, test_X, test_Y)

After the training, do inference on the test dataset. **First load the trained model using pickle.load('model.pkl', 'rb').** Refer section 1.5.2 for finding the details of test_X, and test_Y. Pass the test_X to feed_forward() function to get the predictions. Then, pass the predictions and test_Y to the loss() function to get the loss of the test dataset.

Merging all functions together:

You will now see how the overall model is structured by putting together all the building blocks (functions that you implemented in the previous parts) together, in the right order. Now implement a main function in which you have to call all the above functions in the correct order to train and test your network.

In metrics, you'll have to computer $R - score$, also create a graph of the score.

Note: You are not restricted to implement the assignment in a way that is explained above, you can break down or merge the several functions, but you are required to implement in a modular way.

Task 2: Binary Classification with Logistic Regression Using Pytorch

2. Binary Classification with Logistic Regression

Binary classification with logistic regression involves using the logistic function to model the probability of an observation belonging to one of two classes, typically represented as 0 or 1.

2.1 Titanic Dataset:

The dataset is attached with the assignment. This dataset contains information on passengers aboard the Titanic, including attributes such as age, sex, ticket class, and whether or not they survived the disaster. There are 891 samples in the training set and 418 samples in the test set. Each sample contains several features. The task is to predict whether a passenger survived or not (binary classification).

Please note that while random guessing would yield approximately 50% accuracy due to the binary nature of the survival outcome, any model performing significantly below this threshold (e.g., below 60%) is likely not learning effectively from the data. If your final results are less than 60% in terms of accuracy, your solution(s) will not be graded.

2.2 Data loading and normalization

Implement a loadDataset() function to load the provided Titanic dataset. The loadDataset() function should accept the path of the dataset, size of training, validation, and testing data, and batch size. (NOTE: Don't hard code batch size, it should be changeable at the time of testing and inference). It should load the dataset and split it into training, validation, and testing subsets according to the specified sizes. The function should return the loaded training, validation, and testing data.

Additionally, normalize the data by subtracting the mean and dividing by the standard deviation. Ensure zero mean per batch and 0.5 variance.

2.3 Model Definition

Now we will define the logistic regression model in detail.

Logistic regression is a type of regression analysis used for predicting the probability of a binary outcome. In the case of binary classification, it predicts the probability that an instance belongs to a particular class.

The logistic regression model can be mathematically represented as follows:

Given input features x, the model predicts the probability $P(y = 1|x)$ that the output y is equal to 1.

The logistic regression model applies a linear transformation to the input features followed by a logistic (sigmoid) activation function to produce the output probability.

A. Linear Transformation

The linear transformation is defined as

$$z = w^T x + b$$

Where

- w is the weight vector of size $n \times 1$ (where n is the number of features),
- x is the input feature vector of size $n \times 1$,
- b is the bias term.

The linear transformation computes the weighted sum of the input features along with the bias term.

B. Logistic (Sigmoid) Activation Function

The output of the linear transformation is then passed through a sigmoid function to squash the output into the range [0, 1]. The sigmoid function is defined as

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

Where

- σ represents the sigmoid function,
- z is the input to the sigmoid function (the result of the linear transformation).

The output of the sigmoid function represents the probability that the output y is equal to 1, given the input features x.

Combining the linear transformation and the sigmoid activation function, the output of the logistic regression model is

$$\hat{y} = P(y = 1|x) = \sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Where

- $\hat{y}$ is the predicted probability that the output y is equal to 1,
- $w^T x + b$ is the linear transformation,
- $\sigma(\cdot)$ is the sigmoid activation function.

This model is then trained using optimization algorithms such as gradient descent or stochastic gradient descent to find the optimal values of weights w and bias b that minimize a loss function, typically the cross-entropy loss, over the training data.

2.4 Loss Function

Certainly! The cross-entropy loss function, also known as log loss, is commonly used for classification tasks, including logistic regression. It measures the difference between the predicted probability distribution and the actual probability distribution of the classes.

Let's denote

- y as the true class label (0 or 1 for binary classification)
- $\hat{y}$ as the predicted probability of the positive class (class 1)
- $1 - \hat{y}$ predicted probability of the negative class (class 0) For

binary classification, the cross-entropy loss function L is defined as

$$L(y, \hat{y}) = -\left(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})\right)$$

This function penalizes the model proportionally to the difference between the true label and the predicted probability. Specifically

- If $y = 1$, the loss term $-\ y\ .\ log(\hat{y})$ penalizes the model when the predicted probability $\hat{y}$ is low (far from 1).
- If y = 0 , the loss term $-\ (1\ -\ y).\ log(1\ -\ \hat{y})$ penalizes the model when the predicted probability is $\hat{y}$ high (far from 0).

The overall loss is the sum of the individual losses over all examples in the dataset, divided by the number of examples N, to obtain the average loss:

$$L = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \hat{y})$$

Where

- N is the number of examples in the dataset
- $y_i and \hat{y}$ are the true class label and predicted probability for example i , respectively.

This loss function encourages the predicted probabilities to match the true labels, as minimizing the loss will lead to a higher likelihood of correct classification. It is a key component in training logistic regression models and other classifiers for classification tasks.

2.5 Gradient Descent Optimization

Sure, let's implement gradient descent and stochastic gradient descent (SGD) for training the logistic regression model.

i. Gradient Descent (GD)

Gradient descent is an optimization algorithm used to minimize the loss function by iteratively updating the model parameters in the direction of the negative gradient of the loss function with respect to the parameters.

The parameter update rule for GD is given by

$$\theta = \theta - \alpha \cdot \nabla_\theta L$$

Where

- θ are the model parameters (weights and biases)
- α is the learning rate (step size)
- $\nabla_\theta L$ is the gradient of the loss function with respect to the parameters

The gradient $\nabla_\theta L$ is calculated using the chain rule of calculus and backpropagation.

ii. Stochastic Gradient Descent (SGD)

Stochastic gradient descent is a variant of gradient descent where the parameters are updated using the gradient computed from a single randomly chosen data point (or a small batch of data points) at each iteration.

The parameter update rule for SGD is similar to GD, but the gradient is computed using only a single data point (or a small batch)

$$\theta = \theta - \alpha \cdot \nabla_\theta L_i$$

Where

- L_i is the loss for the i-th data point in the dataset

SGD is computationally more efficient and can converge faster than GD, especially for large datasets.

- **Training Loop**
  - Iterate over the training data in batches.
  - Compute the gradients of the loss function with respect to the model parameters.
  - Update the model parameters using the gradients and the chosen optimization algorithm.
  - Optionally, track the training and validation accuracy to monitor the model's performance.
- **Model Evaluation**
  - Evaluate the trained model on the test set to measure its performance in terms of accuracy or other relevant metrics.

- **Save the Model**
    - Save the trained model parameters to disk for future use.

2.6 Implementation Details

2.6.1 loadDataset() function

Implement a loadDataset() function to load the provided Titanic dataset. The loadDataset() function should accept the path of the dataset, size of training, validation, and testing data, and batch size. Please refer to section 2.2.

```
def loadDataset(dataset_path, train_size, val_size, test_size, batch_size):
…
return X_train, y_train, X_val, y_val, X_test, y_test
```

2.6.2 Data Splitting

After reading the dataset (both samples and their ground-truth), divide it into three subsets: training, validation, and testing.

```
train_X, train_Y, val_X, val_Y, test_X, test_Y = data_split(X, Y)
```

2.6.3 Data Preprocessing

Compute mean and standard deviation, then normalize the input data.

```
mean = compute_mean(X)
std_dev = compute_standard_deviation(X)
X_normalized = normalize(X, mean, std_dev)
```

2.6.4 Initialize Network

Initialize the logistic regression network.

```
net = logistic_regression_network(input_size)
```

2.6.5 Training

```
[model, loss_epoch_tr, loss_epoch_val] = train(net, train_X, train_Y, val_X,
val_Y, batch_size, n_epochs, lr)
```

This function returns

- A trained model
- Network architecture
- Training and validation loss per epoch

2.6.6 Testing

After training, do inference on the test dataset.

```
test_model = test_function(model, test_X, test_Y)
```

2.6.7 Feed Forward

This function will forward propagate through the network.

```
y_hat = feed_forward(X_b)
```

2.6.8 Loss Calculation

Calculate loss using the logistic loss function.

```
loss = logistic_loss(ground_truth, y_hat)
```

2.6.9 Gradient Computation

Calculate gradient for backpropagation.

```
grad = compute_gradient(train_X, train_Y, y_hat)
```

2.6.10 Gradient Descent Optimization

Update weights using gradient descent.

```
net.theta = optimization(lr, grad, net.theta)
```

2.6.11 Saving Trained Model

```
train_model = pickle.dump(model, open('model.pkl', 'wb'))
```

2.6.12 Testing Trained Model

```
test_model = test_function(train_model, test_X, test_Y)
```

2.6.13 Visualize Results **[10 points]**

Write a function that plots loss and accuracy curves and sample images and predictions made by model on them. Function should also plot confusion matrix, f1_score, and accuracy on test data. Review `sklearn.metrics` for getting different metrics of predictions.

2.6.14 Main Function

Put all the building blocks together in the right order to train and test the network

Ensure that you have implemented the logistic regression functions for feed forward, loss calculation, gradient computation, and optimization. Additionally, adapt any other necessary functions accordingly for logistic regression.

Note: You are not restricted to implement the assignment in a way that is explained above, you can break down or merge the several functions, but you are required to implement in a modular way.

Bonus (5%):

Dropout Layer implementation:

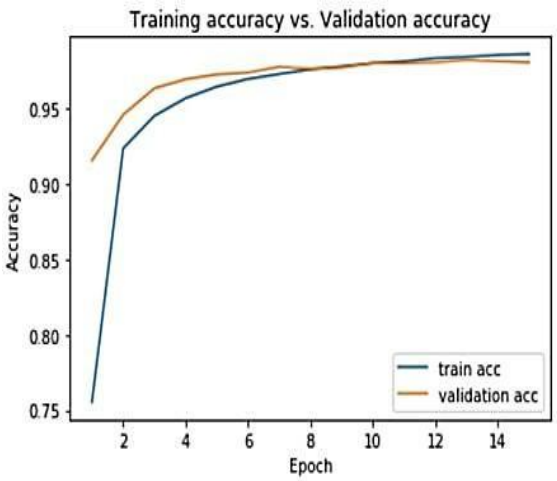
- This step is **not compulsory**, however if your marks in any section is deducted, these bonus points will cover it up.
- In this, you have to implement the dropout layer, after hidden layer and before the hidden layer and analyze the results.
- Dropout rate vary in each experiment from 10% to 40%, not below/more than this.

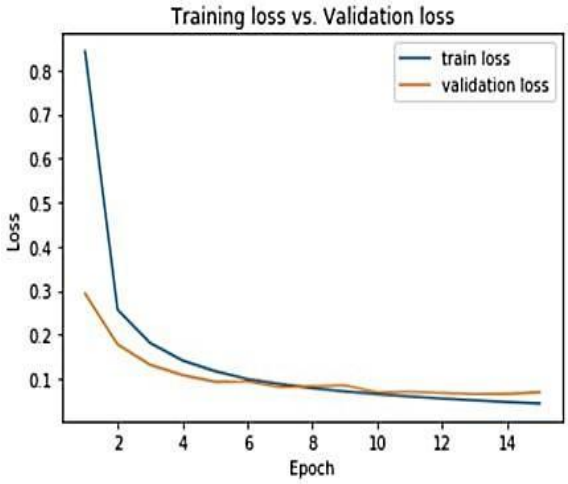
Report

For this assignment, and all other assignments and projects, you must write a report. In the report you will describe the critical decisions you made, important things you learned, or any decisions you made to write your algorithm a particular way. For each experiment, you are required to provide analysis of various hyper parameters.

1. Train model 20 epochs.
2. Training Plots: Plot training, validation and testing loss for both training and testing data with normalization and without normalization.
   - After every epoch, compute the loss for the whole training dataset and report it. NOTE: for the logistic regression you will use the cross-entropy loss and for the linear regression you will use the L2 norm. Compute the same for the validation set and testing set.
   - Plot the training loss, testing loss and validation loss curves for the 100 epochs. Analyze and write down your findings.
3. Plot the loss curve by changing hyper-parameters.
   - learning_rate : 0.01, 0.00001, 0.9
   - number of epoch : 100, 150
   - initialization: all three options

   Make a table indication mean loss over validation data for each. Pick best hyper-parameters by looking at the validation set, train the model using those and report results on the Testing Dataset.
4. Your report should be generated from your own code that you have submitted.
5. your plots should look similar as follow:





Marks Division [100 points]

The marks division is as below:

- **Working code [70 points]**
  - i. Task 1 [30 points]
    1. Feedforward [05 points]
    2. Optimization [05 points]
    3. Normalization [05 points]
    4. Data Loader [05 points]
    5. Loss [05 points]
    6. Training Loop [05 points]
    7. Code commenting [05 points]
  - ii. Task 2 [30 points]
    1. Feedforward [05 points]
    2. Optimization [05 points]
    3. Normalization [05 points]
    4. Data Loader [05 points]
    5. Loss [05 points]
    6. Training Loop [05 points]
    7. Code commenting [05 points]
- **Report [20 points]**
  - i. Task 1 [10 points]
    1. Loss curve , R-squared score curve, & Prediction Curve [05 points]
    2. Analysis (in different experiments) [05 points]
  - ii. Task 2 [10 points]
    1. Loss graph , Accuracy, f1-score, confusion matrix, & classification score [05 points]
    2. Analysis (in different experiments) [05 points]
- **Viva [10 points]**