INDIVIDUAL FINAL REPORT: MEDHASWETA SEN (TEAM NINE)

INTRODUCTION:

A neural network is a type of machine learning algorithm that is modelled after the structure of the human brain. Neurons are arranged in layers, with each layer connected to the previous and next layer. The input layer receives the input data, and the output layer produces the output. In between, there can be one or more hidden layers, where the computations are performed. Training a neural network involves adjusting the weights & biases of the neurons to minimize the difference between the predicted output and the actual output. This is typically done using backpropagation.

Backpropagation is an algorithm used to train such neural networks. It's a supervised learning method that adjusts the weights of the network to minimize the difference between the predicted output and the actual output. It works by first making a forward pass through the network to generate a prediction. The difference between the predicted output and the actual output is then calculated using a loss function like mean squared error. The research on faster algorithms falls roughly into two categories. The first category involves the development of heuristic techniques, which arise out of a study of the distinctive performance of the standard backpropagation algorithm. These heuristic techniques include such ideas as varying the learning rate, using momentum, and rescaling variables & standard numerical optimization techniques.

Our goal for the project was to build a neural network with a custom number of layers and then use backpropagation and the steepest descent algorithms to find the weights and biases for the same for any given dataset. We also planned on building the L-M Algorithm , Conjugate Gradient and the Momentum and the Variable learning rate algorithms for training of Backpropagation from scratch and testing them on a real life dataset as well a synthetic data for function approximation.

OUTLINE OF MY WORK :

1. Standardization of Datasets
2. PCA of the datasets
3. Conjugate gradient
4. Momentum and variable learning rate (attempted)
5. L-M Algorithm(Pseudo code implementation attempted)
6. Implementation of the algorithms on the datasets

DESCRIPTION AND RESULTS FOR MY WORK :

1. Standardization of Datasets:

   Our final goal was to reduce the dimension of the dataset and also deal with multicollinearity. In order to do that the best way was to perform PCA. The first step of PCA is standardization of the datasets. Standardization usually comes into the picture

when features of the input data set have large differences between their ranges, or simply when they are measured in different units (e.g., pounds, meters, miles, etc.). These differences in the ranges of initial features cause trouble for many machine learning models.

In order to standardize the datasets I have used the StandardScalar() function. For distance-based models, standardization is performed to prevent features with wider ranges from dominating the distance metric. But the reason we standardize data is not the same for all machine learning models, and differs from one model to another.

In principal component analysis, features with high variances or wide ranges get more weight than those with low variances, and consequently, they end up illegitimately dominating the first principal components (components with maximum variance). I used the word "illegitimately" here because the reason these features have high variances compared to the other ones is just because they were measured in different scales. Standardization can prevent this, by giving the same weightage to all features.

2. PCA implementation on the datasets :

Principal component analysis, or PCA, is a dimensionality reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

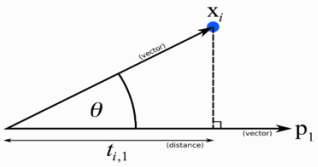
Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Because smaller data sets are easier to explore and visualize and make analysing data points much easier and faster for machine learning algorithms without extraneous variables to process.

So, to sum up, the idea of PCA is simple — reduce the number of variables of a data set, while preserving as much information as possible.

An explanation on how PCA works is as follows:

Geometrically, when finding the *best-fit line* for the swarm of points, our objective was to minimize the error, i.e. the residual distances from each point to the best-fit line is the smallest possible. This is also mathematically equivalent to maximizing the variance of the scores, $\mathbf{t}_a$.

We briefly review here what that means. Let $\mathbf{x}_i'$ be a row from our data, so $\mathbf{x}_i'$ is a $1 \times K$ vector. We defined the score value for this observation as the distance from the origin, along the direction vector, $\mathbf{p}_1$, to the point where we find the perpendicular projection onto $\mathbf{p}_1$. This is illustrated below, where the score value for observation $\mathbf{x}_i$ has a value of $t_{i,1}$.



Recall from geometry that the cosine of an angle in a right-angled triangle is the ratio of the adjacent side to the hypotenuse. But the cosine of an angle is also used in linear algebra to define the dot-product. Mathematically:

$$\cos\theta = \frac{\text{adjacent length}}{\text{hypotenuse}} = \frac{t_{i,1}}{\|\mathbf{x}_i\|} \qquad \text{and also} \qquad \cos\theta = \frac{\mathbf{x}_i'\mathbf{p}_1}{\|\mathbf{x}_i\|\|\mathbf{p}_1\|}$$

$$\frac{t_{i,1}}{\|\mathbf{x}_i\|} = \frac{\mathbf{x}_i'\mathbf{p}_1}{\|\mathbf{x}_i\|\|\mathbf{p}_1\|}$$

$$t_{i,1} = \mathbf{x}_i'\mathbf{p}_1$$

$$(1 \times 1) = (1 \times K)(K \times 1)$$

where $\|\cdot\|$ indicates the length of the enclosed vector, and the length of the direction vector, $\mathbf{p}_1$ is 1.0, by definition.

Note that $t_{i,1} = \mathbf{x}_i'\mathbf{p}_1$ represents a linear combination

$$t_{i,1} = x_{i,1}p_{1,1} + x_{i,2}p_{2,1} + \ldots + x_{i,k}p_{k,1} + \ldots + x_{i,K}p_{K,1}$$

So $t_{i,1}$ is the score value for the i^{th} observation along the first component, and is a linear combination of the i^{th} row of data, $\mathbf{x}_i$ and the direction vector $\mathbf{p}_1$. Notice that there are K terms in the linear combination: each of the K variables *contributes* to the overall score.

We can calculate the second score value for the i^{th} observation in a similar way:

$$t_{i,2} = x_{i,1}p_{1,2} + x_{i,2}p_{2,2} + \ldots + x_{i,k}p_{k,2} + \ldots + x_{i,K}p_{K,2}$$

And so on, for the third and subsequent components. We can compactly write in matrix form for the i^{th} observation that:

$$\mathbf{t}_i' = \mathbf{x}_i'\mathbf{P}$$

$$(1 \times A) = (1 \times K)(K \times A)$$

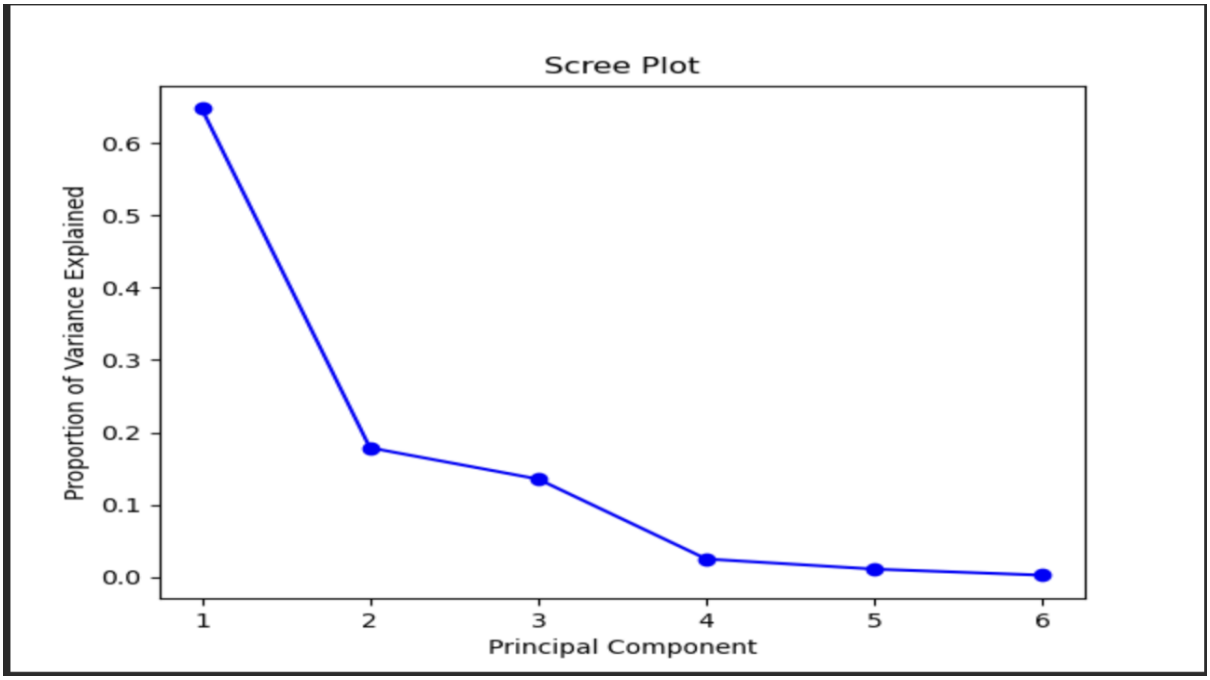
which calculates all A score values for that observation in one go. This is exactly what we derived earlier in the example with the 4 thermometers in the room.

Finally, for an entire matrix of data, $\mathbf{X}$, we can calculate all scores, for all observations:

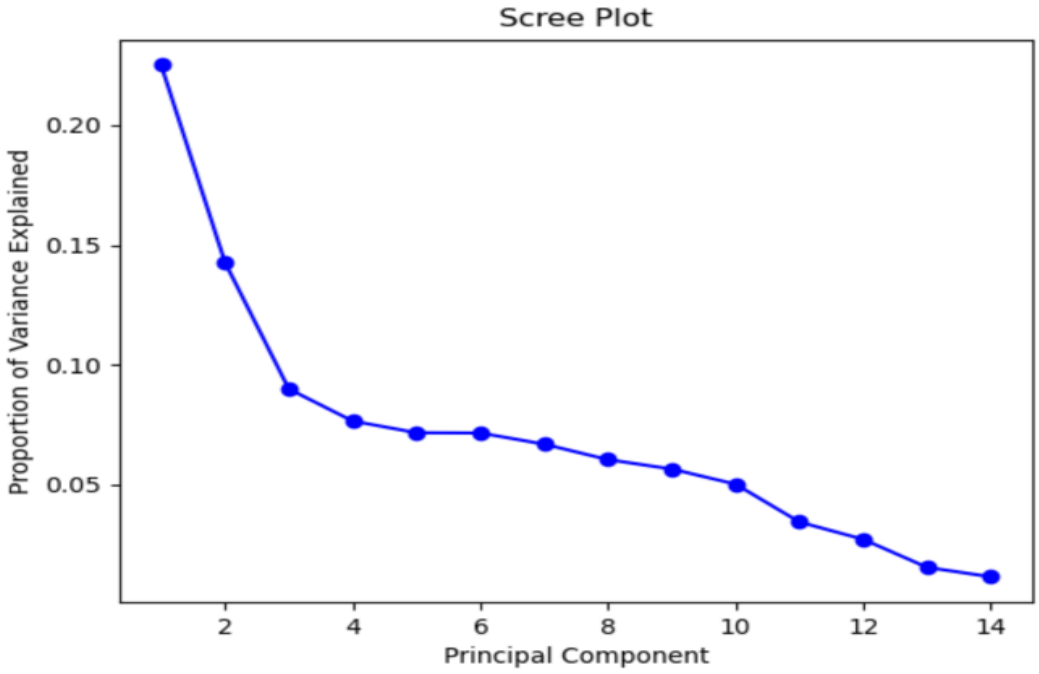
$$\mathbf{T} = \mathbf{XP} \qquad (1)\P$$

$$(N \times A) = (N \times K)(K \times A)$$

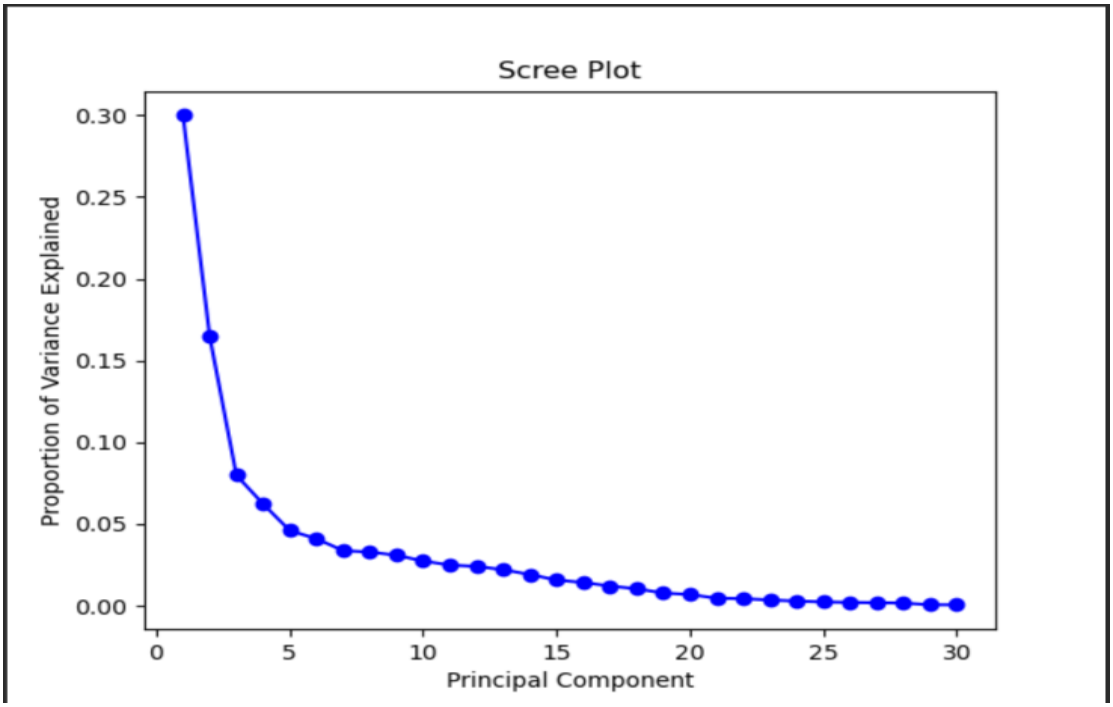
I have calculated PCA here using the python in built function and then plotted the scree plot of the PCs to see how well PCA works in approximating the total variability of the datasets. The scree plots are as follows:



Here is the scree plot for the small dataset. Here, the PC1 explains above 60 percent of the total variability of the data.



Here is the scree plot for the medium dataset. Here, the PC1 explains above 20 percent of the total variability of the data.



Here is the scree plot for the large dataset. Here, the PC1 explains about 30 percent of the total variability of the data.

3. Conjugate gradient implementation:

   Conjugate Gradient (CG) is an iterative method used to solve linear systems of equations of the form Ax=b, where A is a symmetric, positive-definite matrix. It is commonly used in numerical linear algebra and optimization problems.

   The conjugate gradient method seeks to minimize the quadratic form Q(x) = (1/2)x^T Ax - b^T x by iteratively finding a sequence of approximate solutions x_k that converge to the true solution x*. At each iteration, the method computes a direction p_k that is conjugate to the previous search direction, and then computes a step size alpha_k that minimizes Q(x_k + alpha_k p_k) along this direction. The resulting approximation x_{k+1} is then updated as x_{k+1} = x_k + alpha_k p_k.

   One of the key benefits of the CG method is that it requires only matrix-vector multiplications, which can be very efficient for large sparse matrices. Additionally, it has a fast convergence rate, requiring O(n) iterations to converge to the true solution, where n is the size of the system.

   However, the CG method can be sensitive to the conditioning of the matrix A, and may require preconditioning or other modifications for certain types of matrices.

   The Conjugate Gradient (CG) method can be used for optimizing the weights of a neural network using backpropagation. Backpropagation is an algorithm for computing

the gradient of the loss function with respect to the weights of a neural network, and the CG method can be used to efficiently minimize the loss function.

In the context of backpropagation, the weights of the neural network are treated as the variables in the linear system Ax=b, where A is the Hessian matrix of the loss function and b is the negative gradient of the loss function. The Hessian matrix is the matrix of second-order partial derivatives of the loss function with respect to the weights, and the negative gradient is the vector of first-order partial derivatives.

The CG method is then used to solve this linear system iteratively, updating the weights of the neural network in each iteration. At each iteration, the CG method computes a conjugate direction that is orthogonal to the previous search directions, and then computes a step size that minimizes the loss function along this direction.

The CG method can be particularly useful for training large neural networks with many weights, as it can efficiently compute the updates to the weights without requiring the computation of the full Hessian matrix, which can be prohibitively expensive for large networks. Additionally, the CG method can be combined with other optimization techniques, such as momentum or adaptive learning rate methods, to further improve the convergence speed and stability of the training process.

The pseudo code for Conjugate gradient is as follows:

1. Initialize the weights of the neural network randomly

2. Compute the initial gradient of the loss function with respect to the weights

3. Set the initial conjugate direction as the negative gradient

4. For k = 1 to max_iterations do:

5. Compute the step size along the conjugate direction using line search or other optimization method

6. Update the weights of the neural network using the step size and conjugate direction

7. Compute the new gradient of the loss function with respect to the weights

8. Compute the conjugate direction using the updated gradient and the previous conjugate direction

9. If convergence criteria are met, exit the loop and return the optimized weights

10. End for loop.

The convergence criteria can be based on various factors, such as reaching a desired level of accuracy or improvement in the loss function, or reaching a maximum number of iterations. Line search can be performed using techniques such as the Armijo rule or the Wolfe conditions to find an appropriate step size along the conjugate direction. The conjugate direction can be updated using the Polak-Ribière formula or the Fletcher-Reeves formula, which are common methods in the context of CG optimization.

Fletcher-Reeves Update (traincgf)

All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient) on the first iteration.

$$\mathbf{p}_0 = -\mathbf{g}_0$$

A line search is then performed to determine the optimal distance to move along the current search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

Then the next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

The various versions of conjugate gradient are distinguished by the manner in which the constant β_k is computed. For the Fletcher-Reeves update the procedure is

$$\beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient.

See [FlRe64] or [HDB96] for a discussion of the Fletcher-Reeves conjugate gradient algorithm.

In the following code, we reinitialize our previous network and retrain it using the Fletcher-Reeves version of the conjugate gradient algorithm. The training parameters for `traincgf` are `epochs`, `show`, `goal`, `time`, `min_grad`, `max_fail`, `srchFcn`, `scal_tol`, `alpha`, `beta`, `delta`, `gama`, `low_lim`, `up_lim`, `maxstep`, `minstep`, `bmax`. We have previously discussed the first six parameters. The parameter `srchFcn` is the name of the line search function. It can be any of the functions described later in this section (or a user-supplied function). The remaining parameters are associated with specific line search routines and are described later in this section. The default line search routine `srchcha` is used in this example. `traincgf` generally converges in fewer iterations than `trainrp` (although there is more computation required in each iteration).

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgf');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINCGF-srchcha, Epoch 0/300, MSE 2.15911/1e-05, Gradient
3.17681/1e-06
   TRAINCGF-srchcha, Epoch 5/300, MSE 0.111081/1e-05, Gradient
0.602109/1e-06
   TRAINCGF-srchcha, Epoch 10/300, MSE 0.0095015/1e-05, Gradient
0.197436/1e-06
   TRAINCGF-srchcha, Epoch 15/300, MSE 0.000508668/1e-05,
Gradient 0.0439273/1e-06
   TRAINCGF-srchcha, Epoch 17/300, MSE 1.33611e-06/1e-05,
Gradient 0.00562836/1e-06
   TRAINCGF, Performance goal met.
a = sim(net,p)
a =
   -1.0001   -1.0023    0.9999    1.0002
```

The conjugate gradient algorithms are usually much faster than variable learning rate backpropagation, and are sometimes faster than `trainrp`, although the results will vary from one problem to another. The conjugate gradient algorithms require only a little more storage than the simpler algorithms, so they are often a good choice for networks with a large number of weights.

Try the Neural Network Design Demonstration `nnd12cg` [HDB96] for an illustration of the performance of a conjugate gradient algorithm.

Polak-Ribiére Update (traincgp)

Another version of the conjugate gradient algorithm was proposed by Polak and Ribiére. As with the Fletcher-Reeves algorithm, the search direction at each iteration is determined by

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}$$

For the Polak-Ribiére update, the constant β_k is computed by

$$\beta_k = \frac{\Delta \mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}$$

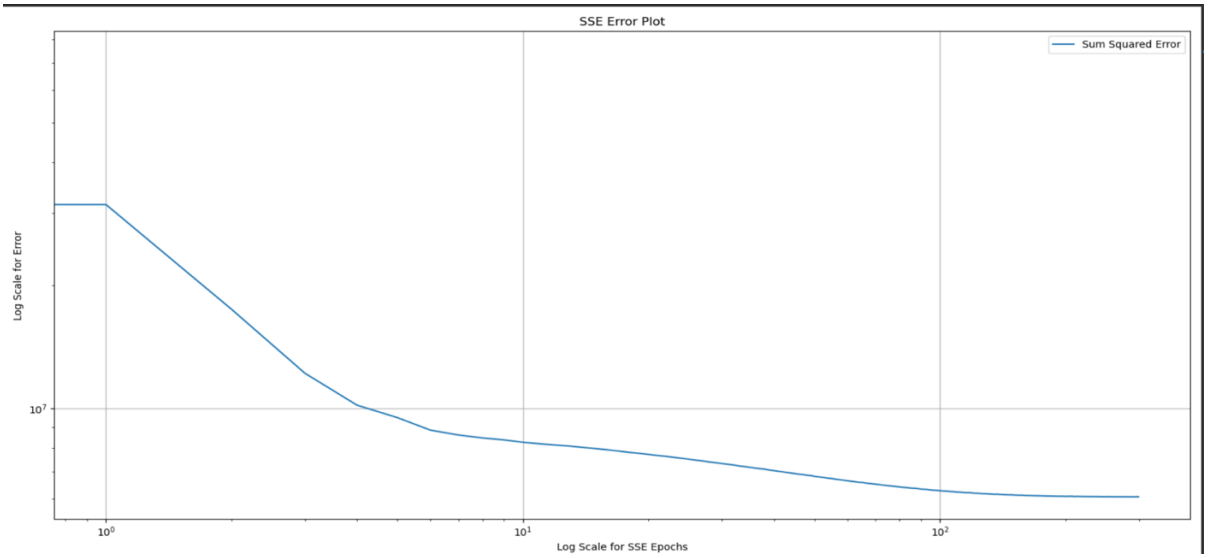
This is the inner product of the previous change in the gradient with the current gradient divided by the norm squared of the previous gradient. See [FlRe64] or [HDB96] for a discussion of the Polak-Ribiére conjugate gradient algorithm.

In the following code, we recreate our previous network and train it using the Polak-Ribiére version of the conjugate gradient algorithm. The training parameters for `traincgp` are the same as those for `traincgf`. The default line search routine `srchcha` is used in this example. The parameters `show` and `epoch` are set to the same values as they were for `traincgf`.

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traincgp');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
   TRAINCGP-srchcha, Epoch 0/300, MSE 1.21966/1e-05, Gradient
1.77008/1e-06
   TRAINCGP-srchcha, Epoch 5/300, MSE 0.227447/1e-05, Gradient
0.86507/1e-06
   TRAINCGP-srchcha, Epoch 10/300, MSE 0.000237395/1e-05,
Gradient 0.0174276/1e-06
   TRAINCGP-srchcha, Epoch 15/300, MSE 9.28243e-05/1e-05,
Gradient 0.00485746/1e-06
   TRAINCGP-srchcha, Epoch 20/300, MSE 1.46146e-05/1e-05,
Gradient 0.000912838/1e-06
   TRAINCGP-srchcha, Epoch 25/300, MSE 1.05893e-05/1e-05,
Gradient 0.00238173/1e-06
   TRAINCGP-srchcha, Epoch 26/300, MSE 9.10561e-06/1e-05,
Gradient 0.00197441/1e-06
   TRAINCGP, Performance goal met.
a = sim(net,p)
a =
   -0.9967   -1.0018    0.9958    1.0022
```

The `traincgp` routine has performance similar to `traincgf`. It is difficult to predict which algorithm will perform best on a given problem. The storage requirements for Polak-Ribiére (four vectors) are slightly larger than for Fletcher-Reeves (three vectors).

The subsequent results of the Conjugate gradient on the artificially generate sinusoidal data is :



The MSE of implementing conjugate gradient of on the artificially generate sinusoidal data is:

```
In [6]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: import pandas as pd
   ...: from sklearn.model_selection import train_test_split
   ...:
   ...: import sys
   ...: sys.path.append('../Final_integration.py')
   ...: from Final_integration import *
```

```
   ...: network2 = Generalized_NeuralNetwork_Backpropagation([1,10,1],['sigmoid','linear'])
   ...: p = np.linspace(-2,2,100).reshape(100,1)
   ...: g = np.exp(-np.abs(p))*np.sin(np.pi*p).reshape(100,1)
   ...: network2.train(p,g,learning_rate=0.01,epochs=300,optimizer='conjugate_gradient',batch_size=20)
   ...: p3 = network2.prediction(p)
   ...: e3 = g-p3
   ...: print(np.mean(e3**2))
```

```
0.11017475120392933
```

The MSE of implementing conjugate gradient of the large dataset using just PC1 is :

```
In [19]: Xtrain3, Xtest3, ytrain3, ytest3 = train_test_split( Xdf3,ydf3,test_size=0.2, random_state=42 )
    ...: network34 = Generalized_NeuralNetwork_Backpropagation([1,100,1],['sigmoid','linear'])
    ...: network34.train(Xtrain3,ytrain3,learning_rate=0.01,epochs=300,optimizer='conjugate gradient',batch_size=20)
    ...: p3 = network13.prediction(Xtest3)
    ...: error3 = ytest3-p3
    ...: print(np.mean(error3**2))
    ...:
32605.683436889965
```

The challenges faced while implementing the pseudo code for conjugate gradient are :

a. Memory requirements: Conjugate gradient requires storing the gradients and directions for each weight, which can become a memory issue for large neural networks. This can lead to slower computations and may require a lower batch size to avoid out-of-memory errors.

b. Non-convex optimization: While conjugate gradient is a powerful optimization algorithm, it is not guaranteed to converge to the global minimum for non-convex loss functions. This can lead to suboptimal solutions or slow convergence if the initial weight values are far from the global minimum.

c. Numerical instability: Conjugate gradient involves performing a number of matrix operations, which can be numerically unstable in some cases. This can lead to issues such as vanishing or exploding gradients, or numerical errors that accumulate over time and affect the performance of the optimization process.

4. Momentum and variable learning rate (attempted):

Momentum and variable learning rate are two common techniques used in optimization algorithms, including in backpropagation for neural network training.

Momentum: Momentum is a technique that helps the optimization algorithm to accelerate the convergence by keeping track of the past gradients and adding a fraction of the previous update to the current update. In other words, momentum helps the optimizer to maintain its direction of movement in cases where the gradient direction changes rapidly. The momentum term is typically a hyperparameter that ranges between 0 and 1, where a value of 0 corresponds to no momentum and a value of 1 corresponds to full momentum.

The pseudo-code for incorporating momentum into the optimization algorithm is as follows:

1. Initialize the velocity V of the weights as zero
2. Compute the gradient of the loss function with respect to the weights
3. Update the velocity as V = alpha * V - learning_rate * gradient
4. Update the weights as W = W + V
5. Repeat steps 2-4 until convergence

Here, alpha is the momentum coefficient, which determines the extent to which the previous update affects the current update. When alpha is set to 0, the momentum term is not used, and the algorithm reduces to the standard gradient descent algorithm.

Variable learning rate: Variable learning rate is a technique that adjusts the learning rate during the optimization process to improve convergence. The idea behind this technique is to start with a relatively large learning rate that allows the optimizer to take large steps towards the minimum of the loss function, and then gradually decrease the learning rate as the optimizer gets closer to the minimum. There are several ways to implement variable learning rate, such as step decay, exponential decay, and adaptive learning rate methods.

The pseudo-code for incorporating variable learning rate using exponential decay is as follows:

1. Initialize the learning rate alpha
2. Initialize the decay rate gamma
3. Initialize the weights W
4. Compute the gradient of the loss function with respect to the weights
5. Update the weights as W = W - alpha * gradient
6. Update the learning rate as alpha = alpha * gamma
7. Repeat steps 4-6 until convergence

Here, gamma is the decay rate, which is a hyperparameter that determines how fast the learning rate should decay. Typically, gamma is set to a value between 0.95 and 0.99.

Momentum and variable learning rate can be incorporated into the backpropagation algorithm to improve the convergence and stability of the optimization process in training neural networks.

To incorporate momentum into the backpropagation algorithm, the update rule for the weights can be modified to include a momentum term as follows:

1. Initialize the velocity V of the weights as zero
2. Compute the gradient of the loss function with respect to the weights
3. Update the velocity as V = alpha * V - learning_rate * gradient
4. Update the weights as W = W + V
5. Repeat steps 2-4 until convergence

Here, alpha is the momentum coefficient that controls the contribution of the previous velocity to the current update, and learning_rate is the learning rate that determines the step size taken in the direction of the gradient.

To incorporate variable learning rate into the backpropagation algorithm, the learning rate can be modified during the optimization process based on some pre-defined schedule. For example, the learning rate can be reduced over time using a decay schedule.

Here, gamma is the decay rate that controls how fast the learning rate should decay over time. By decreasing the learning rate over time, the optimization algorithm can better navigate the loss surface and converge to a better solution.

Both momentum and variable learning rate can be used together in the backpropagation algorithm to achieve even better performance. One common approach is to use a schedule that starts with a high learning rate and high momentum at the beginning of the optimization process and gradually reduces both as the optimization progresses. This approach can help the optimization algorithm to overcome local minima and converge to a better global minimum.

To implement Momentum and variable learning rate together in backpropagation, we can modify the update rule for the weights as follows:

1. Initialize the velocity V of the weights as zero
2. Initialize the learning rate alpha
3. Initialize the momentum coefficient beta
4. Initialize the decay rate gamma
5. Initialize the weights W
6. Compute the gradient of the loss function with respect to the weights
7. Update the velocity as V = beta * V - (1 - beta) * gradient
8. Update the weights as W = W + alpha * V
9. Update the learning rate as alpha = alpha * gamma
10. Repeat steps 6-9 until convergence

Here, beta is the momentum coefficient that controls the contribution of the previous velocity to the current update, and gamma is the decay rate that controls how fast the learning rate should decay over time. The update rule combines the momentum and variable learning rate techniques by applying momentum to the gradient and then adjusting the step size using the learning rate that decreases over time.

The momentum term in step 7 helps the optimizer to maintain its direction of movement in cases where the gradient direction changes rapidly. The variable learning rate term in step 8 helps the optimizer to take smaller steps as it gets closer to the minimum of the loss function.

The values of the hyperparameters beta, gamma, and alpha need to be tuned based on the problem at hand. Typically, beta is set to a value between 0.9 and 0.99, gamma is set to a value between 0.95 and 0.99, and alpha is set to a value that allows the optimizer to converge without oscillating or diverging.

Using Momentum and variable learning rate together in backpropagation can help improve the convergence speed and stability of the optimization process, and can help the optimizer to find better solutions by overcoming local minima.

Even though I tried to implement this algorithm, due to the storage of time I failed to incorporate it in the final training file.

5. L-M Algorithm(Pseudo code implementation attempted):

The Levenberg-Marquardt (LM) algorithm is an optimization algorithm that is commonly used to solve nonlinear least-squares problems. The LM algorithm is a modification of the Gauss-Newton algorithm, which is used to solve linear least-squares problems.

The LM algorithm is often used when the Jacobian matrix (i.e., the matrix of partial derivatives) of the objective function is not full rank, which can occur when there are more parameters than data points. In such cases, the Gauss-Newton algorithm may fail to converge or produce poor results, whereas the LM algorithm is often more robust.

The basic idea of the LM algorithm is to add a damping term to the normal equations used in the Gauss-Newton algorithm. This damping term helps to prevent the algorithm from taking large steps that might cause it to diverge or oscillate. The amount of damping is controlled by a parameter called the LM parameter, which is adjusted during the optimization process.

At each iteration of the LM algorithm, the objective function is approximated by a quadratic function that depends on the current estimate of the parameters. This quadratic function is then minimized to obtain a new estimate of the parameters. The LM parameter is used to control the amount of damping applied to the optimization process.

The LM algorithm is commonly used in a variety of applications, including data fitting, computer vision, and robotics. It is also used as a subroutine in many optimization algorithms, such as the nonlinear conjugate gradient method and the Levenberg-Marquardt-BFGS method.

The Levenberg-Marquardt (LM) algorithm can be used to optimize the parameters of a neural network trained using backpropagation. In this context, the LM algorithm can be used to adjust the weights of the network to minimize the difference between the actual output of the network and the desired output.

To apply the LM algorithm to backpropagation, we can use the following steps:

Initialize the weights of the network to small random values.

Compute the gradient of the cost function with respect to the weights using backpropagation.

Compute the LM parameter for each layer of the network.

Update the weights using the LM parameter and the computed gradients.

Check for convergence, and if the convergence criterion is not met, go back to step 2.

The LM parameter for each layer is computed using the following formula:

lm_param = nu * max(diag(Hessian)) * I

where nu is a damping factor, diag(Hessian) is the diagonal of the Hessian matrix (i.e., the matrix of second partial derivatives), and I is the identity matrix. The value of nu can be adjusted during the optimization process to control the amount of damping applied to the optimization.

The Hessian matrix can be approximated using the outer product of the gradient vector:

Hessian = J.T @ J + mu * I

where J is the Jacobian matrix (i.e., the matrix of partial derivatives) of the output of the network with respect to the weights, mu is a small constant added to the diagonal of the Hessian to ensure that it is positive definite, and I is the identity matrix.

The weights of the network can be updated using the following formula:

weights_new = weights_old - inv(J.T @ J + lm_param) @ J.T @ error

where weights_old are the current weights of the network, J is the Jacobian matrix, error is the difference between the actual output of the network and the desired output, and lm_param is the LM parameter for the current layer.

The optimization process can be stopped when the norm of the difference between the current weights and the previous weights is less than a convergence tolerance, or when the maximum number of iterations is reached.

The pseudo code that I followed was provided in another course that I am enrolled in (DATS 6313: TIME SERIES ANALYSIS AND MODELLING)

Levenberg Marquardf Algorithm - Step 0 & 1

Step 0: $\theta_0 \leftarrow 0$ (only during the first iteration)
Step 1:

$$\mathbf{e}(\theta) \leftarrow \begin{pmatrix} e(1) \\ e(2) \\ \vdots \\ e(N) \end{pmatrix} \text{ where } \theta \leftarrow \begin{pmatrix} a_1 \\ \vdots \\ a_{n_a} \\ b_1 \\ \vdots \\ b_{n_b} \end{pmatrix}_{(n_a+n_b)\times 1}$$

$SSE(\theta) \leftarrow \mathbf{e}^T\mathbf{e}$
$\mathbf{x}_i \leftarrow \frac{\mathbf{e}(\theta_1,\ldots,\theta_n)-\mathbf{e}(\theta_1,\ldots,\theta_i+\delta,\ldots,\theta_n)}{\delta}, 1 \leq i \leq n, \delta = 10^{-6}$
$\mathbf{X} \leftarrow [\mathbf{x}_1^T, \mathbf{x}_2^T, \ldots, \mathbf{x}_n^T]_{N\times n}$
$\mathbf{A} \leftarrow \mathbf{X}^T\mathbf{X}$
$\mathbf{g} \leftarrow \mathbf{X}^T\mathbf{e}$

Step 2:

$\Delta\theta \leftarrow [A + \mu I]^{-1}g$, I is a $n \times n$ identity matrix

$\mu \leftarrow 0.01$, μ will be updated during the program

$\theta_{new} \leftarrow \theta + \Delta\theta$

$SSE(\theta_{new}) \leftarrow e(\theta_{new})^T e(\theta_{new})$

Levenberg Marquardt Algorithm-Step 3

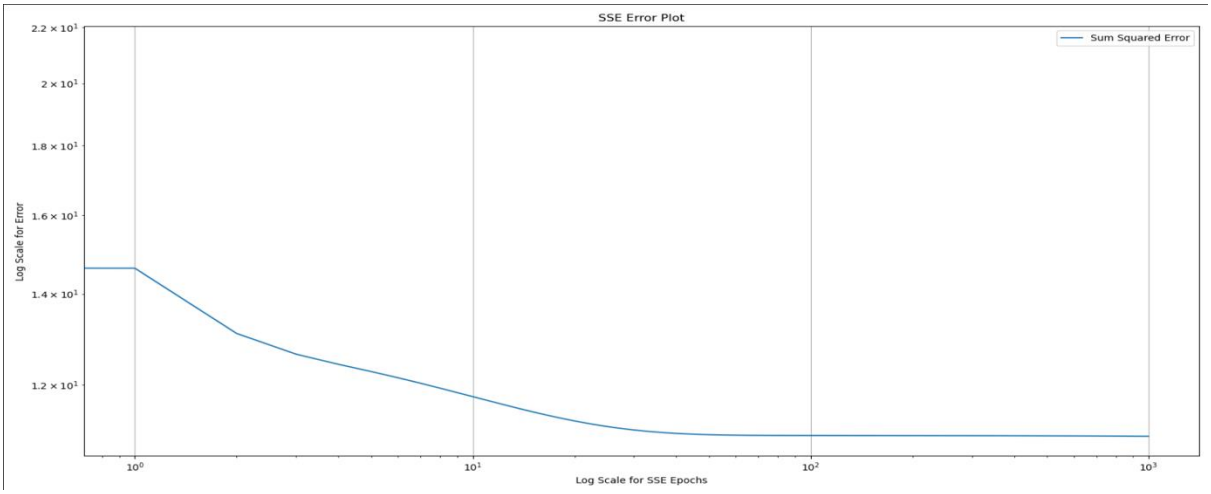
```
if # of iterations < MAX then
    if SSE(θ_new) < SSE(θ) then
        if ||Δθ|| < ε(10^-3) then
            θ̂ = θ_new;
            σ̂_e^2 = SSE(θ_new)/(N-n);
            côv(θ̂) = σ̂_e^2 . A^-1;
            stop;
        else
            θ = θ_new;
            μ = μ/10;
    while SSE(θ_new) >= SSE(θ) do
        μ = μ * 10;
        if μ > μ_max then
            Print out results and print error message;
            stop;
        Return to step 2
    # of iterations +=1;
    if # of iterations > MAX then
        Print out the results;
        Error Message;
        stop;
    θ = θ_new;
    Return to step 1;
    Return to step 2;
```

Please note that this is only for forward propagation and a single input and single output. So, the next steps were to generalise the process while was done by another team member.

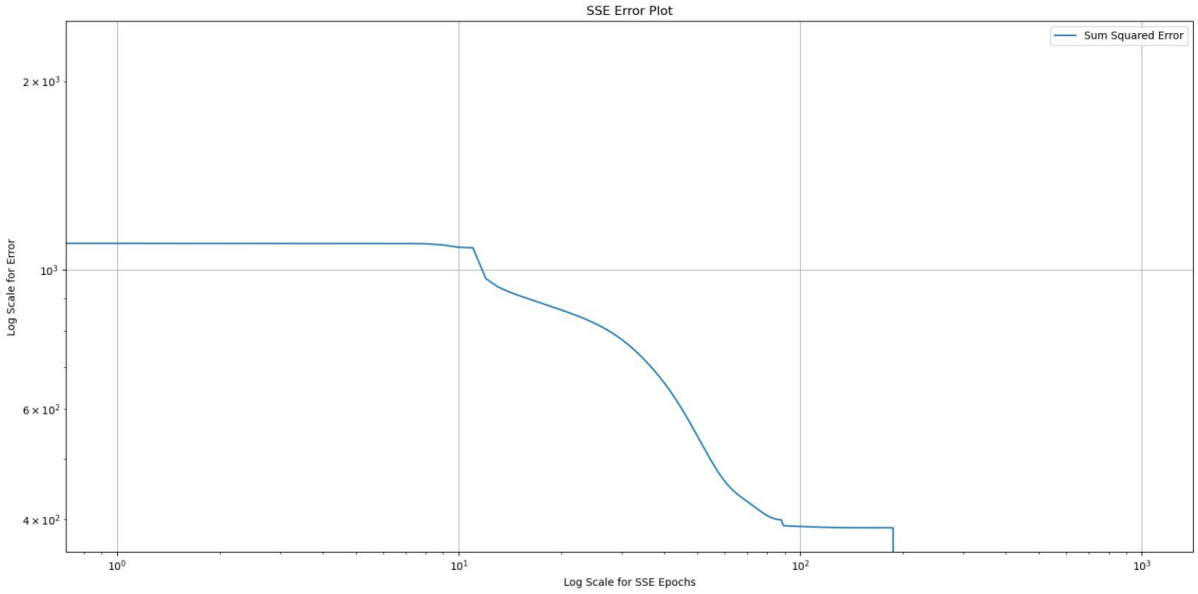
6. Implementation of the algorithms on the datasets :
   After compiling the entire code with backpropagation with gradient descent , conjugate gradient and L-M algorithm into one train function with some additional feature. I ran the function for 2 different use cases. The first one is for the synthetic sinusoidal data that was generated for our Homework 6. While running that I generated 3 Mean Square Errors for each of the methods. The results are as follows:

```
In [6]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: import pandas as pd
   ...: from sklearn.model_selection import train_test_split
   ...:
   ...: import sys
   ...: sys.path.append('../Final_integration.py')
   ...: from Final_integration import *
   ...:
   ...: network = Generalized_NeuralNetwork_Backpropagation([1,10,1],['sigmoid','linear'])
   ...: p = np.linspace(-2,2,100).reshape(100,1)
   ...: g = np.exp(-np.abs(p))*np.sin(np.pi*p).reshape(100,1)
   ...: network.train(p,g,learning_rate=0.01,epochs=300,batch_size=20)
   ...: p1 = network.prediction(p)
   ...: e1 = g-p1
   ...: print(np.mean(e1**2))
   ...:
   ...: network1 = Generalized_NeuralNetwork_Backpropagation([1,10,1],['sigmoid','linear'],seed=1234)
   ...: p = np.linspace(-2,2,100).reshape(100,1)
   ...: g = np.exp(-np.abs(p))*np.sin(np.pi*p).reshape(100,1)
   ...: network1.train(p,g,learning_rate=0.01,epochs=300,optimizer='lm')
   ...: p2 = network1.prediction(p)
   ...: e2 = g-p2
   ...: print(np.mean(e2**2))
   ...:
   ...: network2 = Generalized_NeuralNetwork_Backpropagation([1,10,1],['sigmoid','linear'])
   ...: p = np.linspace(-2,2,100).reshape(100,1)
   ...: g = np.exp(-np.abs(p))*np.sin(np.pi*p).reshape(100,1)
   ...: network2.train(p,g,learning_rate=0.01,epochs=300,optimizer='conjugate_gradient',batch_size=20)
   ...: p3 = network2.prediction(p)
   ...: e3 = g-p3
   ...: print(np.mean(e3**2))
   ...:
0.11017475120392933
Algorithm has converged in 86 epoch.
0.03997260786601465
0.11017475120392933
```

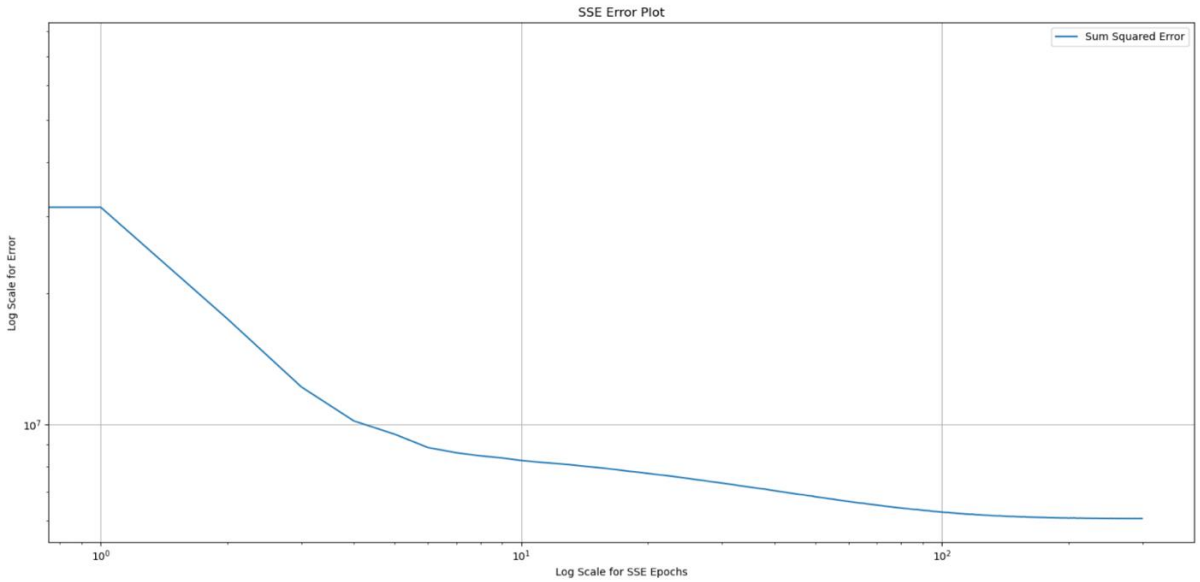
Here, clearly LM has the lowest MSE. I have also generated 3 SSE curves:
SSE for steepest descent:



SSE for LM:



SSE for Conjugate Gradient:



I also ran the datasets for conjugate gradient for the first PC and calculated the SSE:

```
...:
...: Xtrain1, Xtest1, ytrain1, ytest1 = train_test_split( Xdf1,ydf1,test_size=0.2, random_state=42 )
...: network13 = Generalized_NeuralNetwork_Backpropagation([1,10,1],['sigmoid','linear'])
...: network13.train(Xtrain1,ytrain1,learning_rate=0.01,epochs=300,optimizer='conjugate gradient',batch_size=20)
...: p1 = network13.prediction(Xtest1)
...: error1 = ytest1-p1
...: # x = error1[~np.isnan(error1)]
...: print(np.sum(error1**2))
...:
...: Xtrain2, Xtest2, ytrain2, ytest2 = train_test_split( Xdf2,ydf2,test_size=0.2, random_state=42 )
...: network23 = Generalized_NeuralNetwork_Backpropagation([1,10,1],['sigmoid','linear'])
...: network23.train(Xtrain2,ytrain2,learning_rate=0.01,epochs=300,optimizer='conjugate gradient',batch_size=20)
...: p2 = network13.prediction(Xtest2)
...: error2 = ytest2-p2
...: # x = error2[~np.isnan(error2)]
...: print(np.sum(error2**2))
...:
...: Xtrain3, Xtest3, ytrain3, ytest3 = train_test_split( Xdf3,ydf3,test_size=0.2, random_state=42 )
...: network34 = Generalized_NeuralNetwork_Backpropagation([1,100,1],['sigmoid','linear'])
...: network34.train(Xtrain3,ytrain3,learning_rate=0.01,epochs=300,optimizer='conjugate gradient',batch_size=20)
...: p3 = network13.prediction(Xtest3)
...: error3 = ytest3-p3
...: print(np.mean(error3**2))
...:
9.493081330737644e+17
1.0895335760787229e+24
31879.04742498154
```

Given the fact that only the first PC was considered. It feels like a reasonable SSE.
Note: The other datasets and the algorithms have not been considered as there were some dimensionality issues.
Further Research :

1. Fixing LM and the Steepest descent algorithms to run for all real world datasets
2. Generalizing the Conjugate Gradient algorithm to accept a custom number of features.
3. Running conventional models and the pipeline packages as well as hyperparameter tuning with Keras in neural networks to compare to these algorithms.

Code Percent:

BCKPROP WITH MOMENYUM AND VBLR.py = 218

Conjugate_TRY_1.py = 74

CONJUGATE GRADIENT_FINAL_TRY.py = 246

PCA_SAMPLE_TO_IMPLEMENT_LATER.py = 32

TrywithdatasetMS_LASTFILE_IMPORTS TRAIN FUNCTION FROM ANOTHER BRANCH.py = 69

DATASET_LARGE.py = 149

DATASET_MID.py = 177

DATASET_SMALL.py =130

LM_ALG_FWD_PROP_PSEUDO CODE.py = 158

LM_ALG_TRY_1.py = 127

TOTAL = 1380

Note : none of the codes are copied from the internet however a significant amount of the back propagation code for conjugate gradient is from the original backpropagation file . Hence the calculation of the total code written by me is = 1300.

As for the percentage calculation for CONJUGATE GRADIENT_FINAL_TRY.py = (200/250)x100 = 80 percent.

References:

1. “Neural Network Design” (2nd Ed), by Martin T Hagan, ISBN 0971732116
2. Henry P. Gavin (2022), “The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems”, Duke University, Department of Civil & Environmental Engineering.
3. Rauf Bhat (2020), “Gradient Descent with Momentum”, Towards Data Science.
4. Khan et al (2020), "Design of Neural Network with Levenberg-Marquardt and Bayesian Regularization Backpropagation for Solving Pantograph Delay Differential Equations" in IEEE Access, vol. 8, pp. 137918-137933, doi: 10.1109/ACCESS.2020.3011820.