Imprint
Prof. Dr.-Ing. Oliver Nelles
Measurement and Control Engineering - Mechatronics
Department of mechanical engineering
University of Siegen
57068 Siegen

Institute of Mechanics and Control Engineering - Mechatronics
Measurement and Control Engineering - Mechatronics
Prof. Dr.-Ing. Oliver Nelles

Studienarbeit

Latent Space Optimization Of Autoencoders Using L1 Regularization

Author:	Neel Patel
Mat.-No.:	1653726
Supervising tutor:	Prof. Dr.-Ing. Oliver Nelles
	M.Sc. Fabian Schneider

University of Siegen
Faculty of Natural Sciences and Engineering
Department mechanical engineering
Paul-Bonatz-Straße 9-11
D-57068 Siegen

Abstract

This project work delves into the optimization of the latent space in autoencoders through the application of L1 regularization. Autoencoders are neural networks with 3 basic sections: encoder, latent space, and decoder. Latent space carries the information about the input dataset which is extracted by the encoder. High-dimensional datasets are explored via the construction of a latent low-dimensional space which enables convenient visualization, feature extraction, and efficient predictive modeling or clustering. We use L1 regularization which enables some neurons in the latent space to be deactivated, thus eliminating redundant neurons in lower dimensional representation. It might also lead to a decrease in the inference time as we have lower dimensions than the original layout, still maintaining the useful information and the structure of the input.

In this work, we regularize the weights of some neurons exactly to zero, thus eliminating them completely from recreating the output. After deactivating some neurons, we got a very high test error on a new dataset. We also had no control over the number of neurons going to zero. An increase in the regularization strength marginally created a much sparse representation of the latent space. We came to know the fact that it is very hard to distinguish the usefulness of the neurons of the latent space. Each one of them carries some useful information and eliminating some of them may lead to higher reconstruction error.

Preface

The purpose of this project is to find the relation between the reconstruction error and the size of the latent space of an autoencoder on a time series dataset. If we manage to somehow reduce the size of the latent space while maintaining the most useful information of the input dataset, then it may lead to reducing the reconstruction error and the inference time. The inference time creates a big impact in the case where the structure of the autoencoder is very large and the size of the input dataset is also large. Among many regularization techniques, which mainly focus to reduce overfitting, thus maintaining the lower variance error, we use L1 regularization. It gives us the privilege to deactivate some neurons of the latent space. The emphasis of this project is to try different techniques and algorithms to deactivate some neurons and eliminate redundant neurons. After achieving this objective, the second is to have a small reconstruction error and lower inference time.

Contents

Glossary

Symbol	Explanation
λ	Regularization strength
X	Matrix of an input dataset
$\underline{w}$	Parameter vector
$\underline{y}$	Output vector
$\mathcal{L}()$	Loss function
y	True output of a process
$\hat{y}$	Model output
I	Identity Matrix
σ	variance
α	Learning rate
$S_\lambda()$	Soft threshold function
$\underline{z}$	Latent space

List of Figures

List of Tables

Abbreviations

Abbreviation	Explanation
mse	Mean squared error
mae	Mean absolute error
NRMSE	Normalized root mean squared error
GD	Gradient Descent
CD	Coordinate descent
COBYLA	Constraint optimization by linear approximation
L-BFGS-B	Limited memory Broyden Fletcher Goldfarb Shanno algorithm
PCA	Principal Component Analysis

1 Introduction

Dimensionality reduction is a key component in data compression, data visualization, and feature extraction. Autoencoders are extensively used for these functions. It contains an encoder part, which extracts the information and complexity of the input dataset. The latent space represents the information extracted by the encoder. The information can be used for further processing like new data formation, and dimensionality reduction. the decoder uses the information available in the latent space to recreate the input, thus enabling the latent space to learn useful information about the input. While designing an autoencoder, the size of the latent space plays a crucial role while reconstructing the input in the decoder. The difference between an input and an output is called the reconstruction error.

A very large autoencoder often suffers from the problem of overfitting if the input data is noisy, which is often the case, and some redundant information in the latent space if the size is big enough. If the size of the latent space is too small, we might lose some useful information about the input. Thus, it is very important to have the right balance in the size of the latent space. To reduce overfitting, we have many techniques called regularisation techniques. L1 regularization is one of the techniques used to reduce overfitting by penalizing the weights in the loss function with regularization strength. The idea is to eliminate some input features completely which are not much relevant to the output. L1 regularization makes some of the redundant features exactly zero,

thus eliminating them completely. The number of features going to zero depends on the regularization strength.

Many popular algorithms are developed to achieve this exact function of applying L1 regularization. Least angle regression (LARS), Coordinate descent, subgradient descent, and Proximal descent are some algorithms for the task. We have tried LARS and Coordinate descent along with the most popular algorithm gradient descent. Some Newtonian algorithms work best in nonlinear cases. As the Autoencoders are highly nonlinear in nature, we will explore some of them like Constrained optimization by linear approximation (COBYLA), Nelder Mead, and L-BFGS-B.

To evaluate which method and algorithm work best, we will first implement all of them on simple linear and nonlinear systems. After evaluating them on both systems, we will apply them to a much more complex nonlinear autoencoder.

2 Methods

In this chapter, in the first section information about the autoencoders, their structure, type, and usage is given. In the second session, the bias-variance trade-off and regularization techniques are discussed. In the third section, different algorithms used to implement regularization techniques are discussed.

2.1 Autoencoder

Autoencoders are a type of artificial neural network that is used for unsupervised learning. They are designed to copy their input to their output. Internally it has a hidden layer h that describes a code used to represent the input. The network may be viewed as consisting of two parts: an encoder function $h = f(x)$. and a decoder that produces a reconstruction $r = g(h)$. This architecture is given in Fig. 2.1. If an autoencoder succeeds in simply learning to set $g(f(x)) = x$ everywhere, then it is not especially useful. Instead, they are designed to be unable o learn to copy perfectly. They are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. As the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data. [5]

The complete architecture of an autoencoder can be realized as shown in Fig. 2.2

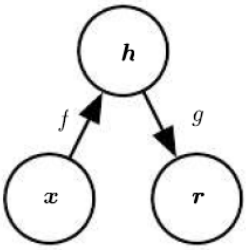

Figure 2.1: The general structure of an autoencoder, mapping an input x to an output r through an internal representation or code h [5]

one way to obtain useful features from the autoencoder is to constrain h to have smaller dimensions than input x. An autoencoder whose code dimension or latent space is less than the input dimension is called undercomplete. It captures the most salient features of the training data. The aim is to minimize the loss function:

$$\mathcal{L}(x, g(f(x)))$$

where L is the loss function penalizing $g(f(x))$ for being dissimilar from x, such as the mean squared error.
Even the undercomplete autoencoder can learn to copy the input to output if we allow the encoder and decoder are allowed too much capacity, thus making autoencoders perform operations without learning anything new. A similar problem occurs if the hidden code is allowed to have dimensions equal to the input, and in an overcomplete case in which latent space has dimensions greater than the input. In these cases, even a linear encoder and a linear decoder can learn to copy the input to output without learning anything useful about the data distribution.

Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides

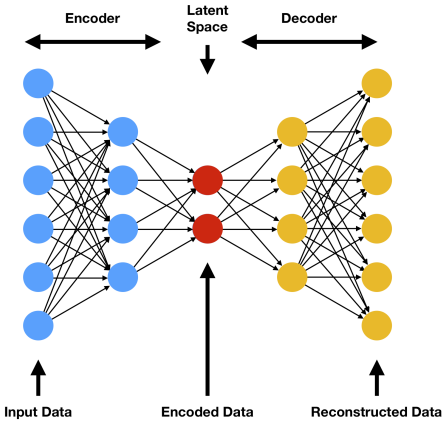

Figure 2.2: Autoencoder architecture with the encoder, latent space, and the decoder

Source: `https://www.compthree.com/blog/autoencoder/`

the ability to copy its input to its output. Sparse autoencoders have simply sparsity penalty $\Omega(h)$ on the code layer h, in addition to the reconstruction error:[2]

$$\mathcal{L}(x, g(f(x))) + \Omega(h),$$

where, $g(h)$ is the decoder output, and typically we have $h = f(x)$, the encoder output.We can think of the penalty $\Omega(h)$ as a regularized term added to the feedforward network.

Autoencoders can be used for various tasks, such as dimensionality reduction, data compression, and anomaly detection. They can also be used as a building block for more complex models, such as variational autoencoders and generative adversarial networks.

There are several types of autoencoders, such as:

- **Standard autoencoder:** This is the basic architecture described above, with a single encoding and decoding layer.
- **Deep autoencoder:** This architecture consists of multiple encoding and decoding layers, which allows for a more complex mapping of the input data to the latent space.
- **Convolutional autoencoder:** This type of autoencoder uses convolutional layers instead of fully connected layers, which is useful for processing image data.
- **Recurrent autoencoder:** This type of autoencoder uses recurrent layers, which is useful for processing sequential data.

Autoencoders can also be regularized to prevent overfitting, such as by adding L1 or L2 regularization to the weights, or by using dropout.
Autoencoders possess specific properties as shown below:

- **Data-oriented:** Autoencoders can produce the desired outcome on data on which are trained. It tries to understand the structure and distribution of data in the latent space and from the latent space, it reconstructs the input. If the autoencoder is trained to denoise the images, then we cannot expect it to use to compress the image.
- **Imperfect output**: The output of the autoencoder is not exactly the same as the input, as it tries to regenerate from the lower or higher dimensioned latent space. Autoencoders are not allowed to generate perfect output, if we do so, it will not learn any important features or distribution of the input data. It will simply perform the copying operation.
- **Unsupervised:** Autoencoders do not require any specific labels to train, so it is considered as an Unsupervised learning technique. We can treat it as self-supervised learning, as the ideal output of

the encoders is the input itself. The training data itself can be considered as the true output of the model.

Overall, autoencoders are a powerful tool for learning a compressed representation of input data. They have many applications in various fields, such as computer vision, natural language processing, and signal processing.

2.1.1 Types

- **Standard Autoencoder:** This is the basic form of an autoencoder that consists of an encoder that compresses the input data and a decoder that reconstructs the input data from the compressed representation.
- **Denoising Autoencoder:** This type of autoencoder is used to remove noise from input data. The input is corrupted with noise, and the autoencoder is trained to reconstruct the original input from the corrupted input.
- **Convolutional Autoencoder**: This type of autoencoder is used for image data, where the encoder and decoder consist of convolutional layers. It can learn a compressed representation of image data while preserving spatial information.
- **Variational Autoencoder:** This type of autoencoder is a generative model that can generate new data samples similar to the input data. It learns a probability distribution over the compressed representation, allowing it to generate new samples.
- **Recurrent Autoencoder:** This type of autoencoder is used for sequential data, such as time series or natural language. The en-

coder and decoder consist of recurrent neural network layers that can learn the temporal structure of the input data.

- **Adversarial Autoencoder:** This type of autoencoder combines the generative power of the variational autoencoder with the adversarial training of the generative adversarial network. It can learn a compressed representation of input data that can be used to generate new samples similar to the input data.

2.1.2 Usage and Application

Autoencoders have a wide range of applications in various fields. Here are some of the most common use cases:

- **Dimensionality Reduction:** Autoencoders can be used to reduce the dimensionality of high-dimensional data, making it easier to visualize and process. This is particularly useful in fields like computer vision and natural language processing.
- **Anomaly Detection:** Autoencoders can be used to identify anomalies in data. The autoencoder is first trained on normal data, and then when it is evaluated with an anomalous input, it will produce a reconstruction error that is significantly higher than for normal inputs.
- **Image Denoising:** Autoencoders can be used to remove noise from images. The autoencoder is trained to reconstruct clean images from noisy ones and then can be used to denoise new images.
- **Feature Extraction:** Autoencoders can be used to extract useful features from data. The encoder part of the autoencoder learns to

compress the input data into a lower-dimensional representation, which can then be used as a feature vector for further tasks like classification.

- **Data Generation:** Autoencoders can be used to generate new data that is similar to the input data. By sampling points in the latent space, the decoder can be used to generate new data that resembles the original input data.

Overall, autoencoders are a powerful tool for data analysis, and their applications are only growing as more researchers discover their capabilities.

2.2 Regularization

In this section, the importance of regularization is discussed along with many regularization techniques commonly used.

2.2.1 Bias-Variance tradeoff [6]

To describe any physical process in mathematical form, we make the model, which tries to mimic the actual process and establishes the parameters of the process as near as possible. One such process and model is described in Fig. 2.3.

Fig. 2.3 depicts a process with its true output y_u disturbed by the noise n, resulting in the measurable process output y. The model with output $\hat{y}$ will describe the process. This is achieved by minimizing some loss function depending on the error e with respect to the model parameters.

The model $(y_u - \hat{y})$ error can be written as follow:

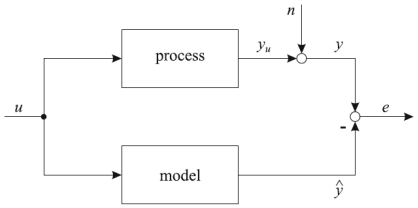

Figure 2.3: Process and Model. $\hat{y}$ is the model output, y_u is the process output [6]

$$E[(y_u - \hat{y})^2] = [y_u - E\{\hat{y}\}^2] + E\{[\hat{y} - E\{\hat{y}\}]^2\}$$

since the first term on the right-hand side is deterministic and all cross terms vanish. This decomposition can also be expressed as

$$(\text{model error})^2 = (\text{bias error})^2 + \text{variance error}$$

The variance-bias tradeoff is a fundamental concept in machine learning and statistical modeling. It refers to the balance between the complexity of a model and its ability to generalize well to new, unseen data.

2.2.1.1 Bias Error

Bias error occurs as a result of restricted model complexity. Most of the real processes are too complex to be modeled exactly. Even if the model parameters are at optimum values, we have a bias error due to the lack of complexity of the model. For example, if we try to model the process of 5^{th} order with 3rd order polynomial, we surely have the error in the model. It is called the bias error. We also need to take the properties of a process, whether it is linear or nonlinear. If the process is nonlinear and of the 5^{th} order, there will be a bias error if we estimate it with the linear model of the 5^{th} order.

For a nonlinear system, the bias error can never be zero unless we know the true structure of the system. We can reduce the bias error with the help of neural networks, fuzzy systems, and polynomials. Bias error decreases as we increase the complexity of the model.

2.2.1.2 Variance Error

The variance error is caused by the sensitivity of the model with respect to the training data. The variance error can also be seen as that part of the model error that is due to a deviation of the estimated parameters from their (unknown) perfect values. Since, in practice, the model parameters are estimated from a finite and noisy data set, these parameters usually deviate from their perfect values. This introduces an error, which is called the variance error. In other words, the variance error describes that part of the model error that is due to uncertainties in the estimated parameters. High variance means that the model is too complex and is overfitting the data. A model with high variance may fit the training data very well but will perform poorly on new, unseen data.

Ideally, for infinite training data, the variance error will be zero, as the parameters can be estimated perfectly. For data same as the parameters, the variance error is maximum, as the parameters reflect the value of noise. It is highly recommended to use the large training dataset to make the variance error as small as possible. On the other hand, as the number of parameters increases, the variance error increases. The expression for the variance error can be shown as:

$$\text{variance error} \sim \sigma^2 \, \frac{n}{N}$$

in above expression, σ is the noise variance, n is the number of parameteres and N is the number of training data samples.

2.2.1.3 Tradeoff

Figure 2.4 summarizes the effect of the bias and variance error on the model error. Obviously, a very simple model has a high bias but a low variance error, while a very complex model has a low bias but a high variance error. Somewhere in between lies the optimal model complexity. Figure 7.4a clearly shows that models, that are too simple, can be improved by the incorporation of additional parameters because the increase in the variance error is overcompensated by the decrease in the bias error. Contrary, a model, that is too complex, can be improved by discarding parameters because the increase in the bias error is overcompensated by the decrease in the variance error. The fact that the bias and variance error are in conflict (it is not possible to minimize both simultaneously) is often called the bias/variance dilemma.

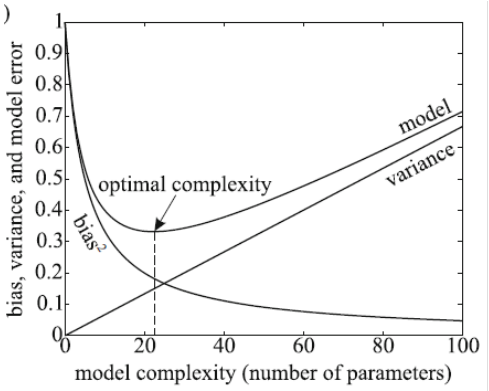

Figure 2.4: Bias/variance tradeoff. As one error decreases, the other starts to increase. the optimal complexity of the model shows the best value for the bias and variance error [6]

High variance and low bias mean that the model is too complex and is overfitting the data (right figure of Fig.2.5), Low variance and high bias means that the model is too simple (left figure of Fig.2.5). The good fit shows the optimum value of bias and variance (center figure of Fig.2.5).

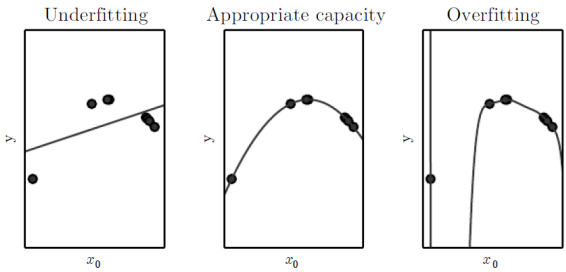

Figure 2.5: Types of fit of the model.(Left) figure shows the underfit, which fails to capture the curve of the data.(Centre) figure shows that model uses a quadratic function to generalize well to unseen points. (Right) figure shows that a polynomial of degree 9, that fits the training data perfectly [6]

Regularization techniques are used to help find this balance between bias and variance. They involve adding additional constraints or penalties to the model to prevent it from becoming too complex and overfitting the training data. Common regularization techniques are L1 and L2 regularization, dropout, and early stopping.

In summary, the variance-bias tradeoff is a critical concept in machine learning and statistical modeling. It refers to the tradeoff between the complexity of a model and its ability to generalize well to new, unseen data. Regularization techniques can help find the optimal balance between bias and variance and prevent overfitting, leading to better model performance.

2.2.2 Regularization Techniques

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce test error, possibly at the expense of increased training error. These

strategies are known collectively as regularization. There are several reasons why we need regularization techniques [1]:

- **The complexity of Models:** Modern machine learning models can be extremely complex, with many layers, neurons, and hyperparameters. These models have a large capacity to fit the training data, but this can also make them more prone to overfitting. Regularization techniques help to reduce the effectiveness of the complexity of these models, without sacrificing their predictive power. One thing to keep in mind is that these techniques do not reduce the complexity.

- **Limited Data:** In many real-world problems, we only have a limited amount of data available for training our models. This can make it difficult to generalize the model to new, unseen data. Regularization techniques help to ensure that the model doesn't overfit to the limited data, and can generalize better to new data.

- **Noisy Data:** Real-world data is often noisy, with errors and outliers that can influence the model's predictions. Regularization techniques help to reduce the impact of noisy data on the model, by focusing on the underlying patterns in the data instead of the noise.

- **Preventing Model Collapse:** In some cases, deep neural networks can suffer from "model collapse", where the network fails to learn anything useful and simply outputs the same value for all inputs. Regularization techniques can help to prevent model collapse, by encouraging the network to learn more diverse features and avoid over-reliance on a few input features.

There are several types of regularization techniques, including L1 and L2 regularization, dropout, early stopping, and data augmentation. These

techniques work by adding additional constraints or penalties to the model during training, in order to reduce overfitting and improve generalization performance.

1. **L1 and L2 regularization:** L1 and L2 regularization are techniques that add a penalty term to the loss function during model training. This encourages the model to learn simpler patterns and reduces the effect of noisy features in the data. L1 regularization adds the sum of the absolute values of the model weights to the loss function, while L2 regularization adds the sum of the squares of the model weights.

2. **Dropout:** Dropout is a regularization technique used to prevent overfitting by randomly dropping out (setting to zero) some of the neurons in a layer during training. We make the weights and bias coming out of the dropped-out neurons to zero, thus outcasting them in the model predictions. This forces the network to learn more robust features that are useful for making predictions even when some of the neurons are missing.

3. **Early stopping:** Early stopping is a technique used to prevent overfitting by monitoring the validation loss during training and stopping the training process when the validation loss stops improving. This prevents the model from continuing to learn the training data too well and overfitting.

4. **Data augmentation:** Data augmentation is a technique used to artificially increase the size of the training data by applying transformations to the existing data, such as rotating, flipping, or shifting the images. This helps to prevent overfitting by exposing the model to a wider range of variations in the data.

Overall, these techniques are powerful tools for improving the generalization performance of machine learning models and are commonly used

in practice.

2.2.3 L1 (Lasso) Regularization

L1 regularization is a technique used to prevent overfitting in machine learning models. It is a form of regularization that adds a penalty term to the objective function of the model during the training process. The penalty term is proportional to the sum of the absolute values of the weights of the model.

The L1 regularization penalty is also known as the Lasso (Least absolute shrinkage and selection operator) penalty, and it is a form of sparsity-inducing regularization. It encourages the model to have sparse solutions, i.e., solutions where many of the weights are set to zero. This is because the L1 penalty is non-zero only for non-zero weights. Therefore, by minimizing the L1 penalty, the model is encouraged to have fewer non-zero weights, leading to a more parsimonious model.

Mathematically, the L1 regularization penalty can be defined as:

$$L_1(\underline{w}) = \lambda \sum_{i=1}^{n} |\underline{w}_i|$$

where λ is a hyperparameter that controls the strength of the regularization, $\underline{w}$ is the vector of weights of the model, and n is the number of weights.

The addition of the L1 regularization penalty to the objective function of the model has the effect of shrinking the weights toward zero. This is because the objective function is minimized by finding the values of the weights that minimize both the data loss and the L1 penalty. The result is a trade-off between the data loss and the L1 penalty, which can be controlled by adjusting the hyperparameter λ.

L1 regularization is particularly useful in situations where there are many features or predictors in the dataset, and many of them are irrelevant or redundant. In such cases, the L1 penalty can help to identify and remove the irrelevant features by shrinking their corresponding weights to zero. This can lead to a more interpretable and accurate model.

If we take the example of the objective function of linear regression including the lasso penalty, it can be defined as below:

$$L(\underline{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(h_{\underline{\theta}}(\underline{x}^{(i)}), y^{(i)}) + \lambda\sum_{j=1}^{n}|\underline{\theta}_j|$$

Here, $\mathcal{L}(h_{\underline{\theta}}(\underline{x}^{(i)}), y^{(i)})$ is the loss function, which is the difference between actual output (y) and predicted output($h_{\underline{\theta}}(\underline{x})$. The lasso penalty is defined as $\lambda\sum_{j=1}^{n}|\underline{\theta}_j|$. Which is the deciding factor towards making weights to zero to increase sparsity.

2.2.4 L2 (Ridge) Regularization

Ridge regression is a regularization technique used in linear regression to address the issue of multicollinearity and overfitting. It extends the ordinary least squares (OLS) regression by adding a penalty term to the objective function, which helps to control the complexity of the model and prevent extreme parameter values.

The formula for ridge regression is as follows:

$$L(\underline{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(h_{\underline{\theta}}(\underline{x}^{(i)}), y^{(i)}) + \lambda\sum_{j=1}^{n}|\underline{\theta}_j|^2$$

Here, $\mathcal{L}(h_{\underline{\theta}}(\underline{x}^{(i)}), y^{(i)})$ is the loss function, which is the difference between actual output (y) and predicted output($h_{\underline{\theta}}(\underline{x})$. The Ridge penalty is defined as $\lambda\sum_{j=1}^{n}|\underline{\theta}_j|^2$.

The regularization term in ridge regression is usually defined as the L2 norm (Euclidean norm) of the parameter vector:

$$R(\underline{w}) = \|\underline{w}\|_2^2 = w_1^2 + w_2^2 + \ldots + w_p^2$$

Where $|\underline{w}\|_2$ represents the L2 norm of the parameter vector $\underline{w}$, and p is the number of parameters in the model.

The objective of ridge regression is to find the parameter values that minimize the overall error, which is a trade-off between the sum of squared errors and the regularization term. By adjusting the λ parameter, you can control the degree of regularization applied. A higher λ value results in stronger regularization, shrinking the parameter values more toward zero.

To find the optimal parameter values in ridge regression, various optimization algorithms can be used, such as gradient descent or closed-form solutions like the ridge regression equation:

$$\underline{w} = (X^T X + \alpha I)^{-1} X^T y$$

Where $\underline{w}$ is the parameter vector containing the regression coefficients.,X is the design matrix with the input features. y is the target variable or output.I is the identity matrix.

Ridge regression is a valuable tool when dealing with multicollinearity or when there are more predictors than observations. By introducing regularization, it helps to stabilize and improve the performance of the regression model.

2.2.5 Regularization technique selection

The end goal of this project is to deactivate some neurons in the latent space of the autoencoder, making it less prone to overfitting and better feature selection. To completely deactivate the neuron, we have make sure that the weights and biases going out of deactivated neurons are zero. This can only be achieved with the help of L1 regularization. L2 regularization can make weights very small, but cannot make them zero. The sparsity can only be achieved with the L1 norm.

Figure 3.13 compares the non-regularized least squares solution to the ridge and lasso estimate. One very appealing property of the lasso can already be observed in this figure: Lasso drives many parameters toward zero. Thus, it produces sparse parameter vectors, i.e., many parameters values are exactly zero. This is not the case for ridge regression because the square lightens the penalty; the closer one parameter value approaches zero. Therefore in the ridge regression case, there exists no mechanism to drive it exactly toward zeros. If the penalty is linear, as in the lasso case, the pressure on parameters toward zero remains intact [6].

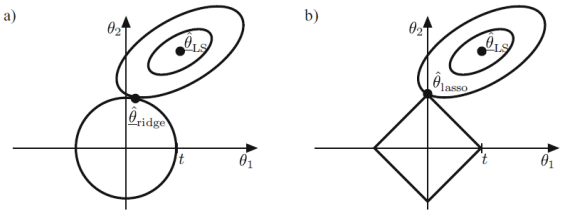

Figure 2.6: (a) L2 and (b) L1 regularization for a quadratic objective function- [6]

This can also be proved with the help of a formula. The minimization

problem for ridge regression is:

$$\underset{\underline{\beta}}{argmin}||\underline{y} - X\underline{\beta}||^2 + \sum_{j=1}^{n} \lambda||\underline{\beta}||^2$$

the solution for the above equation becomes:

$$\underline{\beta} = (X^T X + \lambda I)^{-1} X^T \underline{y}$$

Here, we are adding this λI (called the ridge) on the diagonal of the matrix that we invert. The effect this has on the matrix $X^T X$ is that it "pulls" the determinant of the matrix away from zero. Thus when you invert it, you do not get huge eigenvalues. The eigenvalues get shifted by λ. This happens to all eigenvalues, and all the values being real and positive, they all move away from zero.

The minimization problem for lasso regression is:

$$\underset{\underline{\beta}}{argmin}||\underline{y} - X\underline{\beta}||^2 + \sum_{j=1}^{n} \lambda||\underline{\beta}||$$

the solution for the above equation becomes:

$$\underline{\beta} = (2X\underline{y} - \lambda I)/2X^T X$$

Here, we are removing the λ from the denominator, shrinking the parameters towards zero.

2.3 Optimizer selection

The most popular and effective algorithm being used in almost all machine learning models is the Gradient Descent, which is the first-order

method and its variants like Stochastic gradient descent(SGD) and Adam optimizer.

However, models like lasso regression or support vector machine(SVM) contain a loss function that is not differentiable at the value zero. Because of this reason, gradient descent gives poor results on these models resulting in suboptimal performance and poor feature selection.

The general form of any loss function can be given below:

$$f(x) = g(x) + \sum_{i=1}^{n} h_i(x_i)$$

Where g is convex and differentiable and h_i is convex but non-differentiable. The gradient descent algorithm updates the weights in the opposite direction of the gradient of the cost function with respect to the weights. However, the L1 norm is not differentiable at 0, and the gradient of the L1 regularization term is not well-defined at this point. This can lead to the gradient descent algorithm getting stuck at 0, which can result in some of the weights never being updated and remaining at 0. This can result in suboptimal performance and poor feature selection.

To overcome this issue, we can use several different algorithms for Lasso regression, which is specifically designed to handle the non-differentiability of the L1 norm. The most popular methods to solve lasso regression are Least angle regression(LARS), Coordinate descent, Proximal Descent, and subgradient descent.

Coordinate descent works well for linear systems. On nonlinear systems, it sometimes fails to provide optimization. To apply optimization on the latent space of autoencoders which is highly nonlinear to the input and to the output, there are some other algorithms that give better results. Some of them are Nelder-Mead, Powell, and COBYLA. Some are discussed in the upcoming sections.

Newton-based algorithms can also be used for training neural networks. Which uses a Hessian matrix of the objective function to improve convergence. It can be especially useful for large-scale problems where first-order methods like stochastic gradient descent (SGD) can be slow to converge. The problem with Newton-based algorithms is the same as gradient descent, i.e. partial derivatives at zero are not defined, thus they do not give prominent results on L1 regularization. Owing to these reasons, they are not discussed in this report.

2.3.1 Least angle regression (LARS)

Least Angle Regression (LARS) is a regression algorithm designed for high-dimensional data analysis. It is particularly useful when the number of predictors (features) is large compared to the number of observations, making traditional regression methods computationally expensive or prone to overfitting. Least Angle Regression (LARS) relates to the classic model-selection method known as Forward Selection, or forward stepwise regression.

The goal of LARS is to efficiently select a subset of relevant predictors and estimate their coefficients in the linear regression model. It achieves this by iteratively adding predictors to the model, gradually moving towards the least squares solution. Unlike forward selection or stepwise regression, LARS simultaneously estimates the coefficients of all predictors, providing a more comprehensive understanding of their impact on the response variable.

By combining LARS with Lasso, we obtain an enhanced regression method called LARS-Lasso or LARS with L1 regularization.

The main idea behind LARS-Lasso is to modify the LARS algorithm to introduce Lasso penalties during the coefficient estimation process. LARS-Lasso starts with an empty model and iteratively adds variables

to the model while adjusting the coefficients. It achieves this in a way that smoothly transitions between ridge regression and Lasso regression as the penalty parameter changes.

The algorithm begins with all coefficients set to zero. At each step, LARS identifies the predictor most correlated with the response variable and moves in its direction until another predictor becomes equally correlated. This process is performed gradually, controlling the speed at which each predictor enters the model. LARS employs a set of equiangular directions to update the coefficient estimates, ensuring that all predictors with the same absolute correlation enter the model simultaneously. As the algorithm progresses, the coefficient estimates move towards their final values, and at each step, LARS computes the entire solution path, capturing the evolution of the coefficients as predictors enter the model. The LARS algorithm terminates when it reaches a user-defined number of predictors or when all predictors have been included in the model.[3]

To incorporate Lasso regression, we include the penalty term at the time of updating the coefficient in the direction of the prediction. LARS-Lasso encourages sparsity in the coefficient estimates, leading to automatic variable selection. As the penalty parameter increases, more coefficients are driven to exactly zero, effectively performing the variable selection.

The advantage of using LARS-Lasso over standard LARS is that it combines the benefits of LARS in terms of computational efficiency and variable selection with the benefits of Lasso in terms of regularization and sparse solutions. It provides a flexible and efficient approach for simultaneous variable selection and regularization in regression problems.

2.3.2 Coordinate Descent

Coordinate descent is an optimization algorithm used for finding the minimum of a function by optimizing over one coordinate at a time. It is commonly used for solving Lasso and Elastic Net regression problems.

The basic idea behind coordinate descent is to optimize a function by repeatedly selecting a coordinate (dimension) to update, and updating it while holding all other coordinates fixed. The process is then repeated until convergence is achieved.[13]

More formally, given a function f(x) where x is a vector of p parameters, the algorithm updates one parameter at a time while keeping all other parameters fixed. The parameter to update is selected based on its partial derivative with respect to the loss function.

Let's assume that we want to minimize a function f(x) with respect to x = (x1, x2, ..., xp), where each xi is a scalar. The algorithm starts by initializing the values of xi to some starting point x0.

1. Select an index j from 1 to p

2. Update xj to minimize the function f(x) over xj, while holding all other coordinates fixed. This means that we need to solve the one-dimensional optimization problem: $minf(x_1, x_2, ..., x_{j-1}, x_j, x_{j+1}, ..., x_p)$.

3. Repeat steps 1 and 2 for all indices j until convergence is achieved.

The coordinate descent algorithm can be visualized as moving along the coordinate axes toward the minimum value of the function, as shown in Fig. 2.7

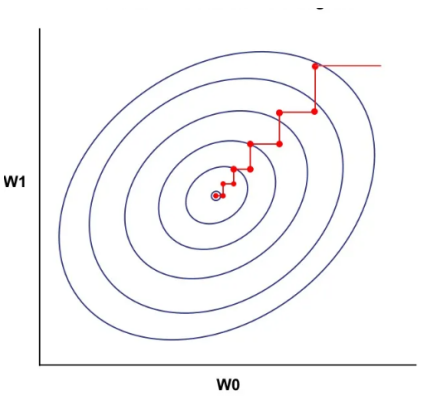

Figure 2.7: Visual representation of how coordinate descent reaches minima for two parameters W0 and W1

Source: `http://www.adeveloperdiary.com`

The steps shown above can be mathematically written as:[12]

$$
\begin{aligned}
x_1^{(k)} &\in \underset{x_1}{\operatorname{argmin}} f(x_1, x_2^{(k-1)}, x_3^{(k-1)}, \ldots, x_p^{(k-1)}) \\
x_2^{(k)} &\in \underset{x_2}{\operatorname{argmin}} f(x_1^{(k)}, x_2, x_3^{(k-1)}, \ldots, x_p^{(k-1)}) \\
x_3^{(k)} &\in \underset{x_3}{\operatorname{argmin}} f(x_1^{(k)}, x_2^{(k)}, x_3, \ldots, x_p^{(k-1)}) \\
&\vdots \\
x_p^{(k)} &\in \underset{x_p}{\operatorname{argmin}} f(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \ldots, x_p)
\end{aligned}
$$

The update for coordinate j can be expressed as follows:

$$xj = argmin(f(x1, x2, ..., xj - 1, xj, xj + 1, ..., xp))$$

The above update can be done using closed-form solutions for certain loss functions, such as L1 (Lasso) regularization

Soft Threshold function:

The soft threshold function is a special case of a more general class of functions known as the shrinkage function. These functions are used to shrink or reduce the magnitude of a parameter or coefficient toward zero. Other examples of shrinkage functions include the hard threshold function and the truncated threshold function.

The soft threshold function is a nonlinear function that is commonly used in signal processing and statistical analysis. The function is used for the regularization or denoising of data. It is applied to coefficients of a signal, which are then reduced or shrunk towards zero. The soft threshold function is defined as follows:[11]

$$S_\lambda(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ 0, & \text{if } -\lambda \leq x \leq \lambda \\ x + \lambda, & \text{if } x < -\lambda \end{cases}$$

where λ is a parameter that controls the amount of shrinkage. The soft threshold function has a "knee" at $x = \lambda$ and $x = -\lambda$, where it transitions from linear decrease (or increase) to zero. it can be visualized as follow:

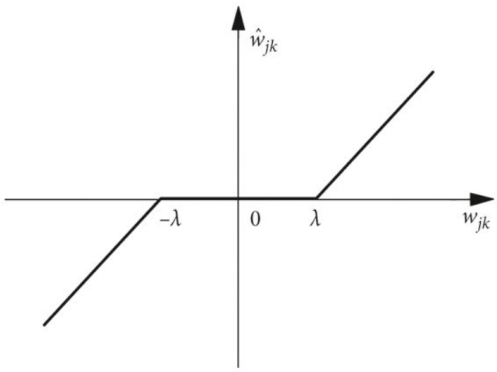

Figure 2.8: Graph of the Soft threshold function with λ as the shrinkage factor

Source: `https://www.researchgate.net/figure`

In signal processing, the soft threshold function is used as a regularization method to reduce noise in signals. For example, it can be applied to Fourier coefficients of a signal to reduce the effect of high-frequency noise, resulting in a smoother signal. In statistical analysis, the soft threshold function is used as a penalty term in regression methods to prevent overfitting of the data. For example, in the L1 update for linear regression, the soft threshold function can be expressed as:

$$xj = sgn(sj)(|sj| - \lambda)^{+}$$

where $s_j = \sum_{i=1}^{n} x_i y_i - x_i^2, \lambda$ is the regularization parameter, and sgn() is the sign function, which takes the sign of sj. xj is called the soft threshold function, which is the reason we are getting some weights to zero.

In the case of coordinate descent, we apply the soft threshold function after finding the argument that minimizes our weights to make sure that weights having a small effect on the output are zero.

The advantages of coordinate descent are that it can handle large-scale problems and can be faster than other optimization algorithms. Additionally, it has a closed-form solution for L1 regularization, which is not possible with gradient descent.

2.3.3 Constrained Optimization by linear approximation (COBYLA)

COBYLA which was first described by [10] approximates the objective function linearly over a small region of the search space around the current point, using a combination of function evaluations and the Hooke-Jeeves pattern search. Specifically, at each iteration, it constructs a linear approximation of the objective function in the neighborhood of

the current iterate and then uses a trust-region approach to determine a step size that improves the objective function.

The Hooke-Jeeves pattern search algorithm was developed in the 1960s by R. Hooke and T. A. Jeeves and has been widely used in various optimization problems, especially in situations where the objective function is noisy or does not have a smooth gradient. [9]

The trust region approach is a method for solving optimization problems, particularly nonlinear optimization problems. The idea is to focus on a small region around the current solution and perform optimization only within that region. This is done to ensure that the optimization remains within a "trusted" region of the solution space.

Hooke-Jeeves pattern search is an optimization algorithm that does not require the computation of gradients. It is a derivative-free method that iteratively searches for the optimal solution by moving from one point to another.

The algorithm starts with an initial point and a step size. It then evaluates the objective function at the current point. If a better point is found within the current neighborhood, the algorithm moves to that point and reduces the step size. If no better point is found, the algorithm expands the search neighborhood by increasing the step size and tries again.[9]

The trust region approach is based on the idea of building a quadratic model of the objective function in the neighborhood of the current solution. The size of the region is controlled by a trust region radius, which is a parameter that determines the size of the region around the current solution that is considered "trustworthy". The quadratic model is then minimized subject to a trust region constraint, which limits the movement of the optimizer to within the trust region.

The trust region approach is particularly useful for optimization problems with complex, non-convex objective functions that may have multiple local minima. By focusing on a small region around the current

solution, the trust region approach can help the optimizer avoid getting stuck in a local minimum and instead converge to a global minimum.

The linear approximation of the objective function is based on the first-order Taylor series expansion of the function, which assumes that the function is locally linear and can be approximated by its slope (i.e., derivative) at the current point. This approximation is used to construct a linear model of the objective function in the vicinity of the current point, which is then used to guide the search for the next point.

COBYLA uses this linear approximation to minimize the objective function subject to the constraints. It searches for the next point by iteratively solving a series of subproblems in which the objective function is minimized subject to linear constraints, where the constraints are constructed based on the linear approximation. The algorithm uses a trust-region approach to determine the step size, which ensures that the algorithm does not stray too far from the current iterate.

Overall, the linear approximation used by COBYLA is a simple but effective way to guide the search for the optimal solution of a nonlinear optimization problem, particularly when the objective function is not differentiable or the derivatives are not available or expensive to compute.

3 Implementation

In this chapter, Lasso regularization is applied to three different kinds of systems. Them being linear systems, lower dimensional non-linear systems, and high dimensional nonlinear systems of autoencoders. On each type of system, various methods mentioned in the last chapters are applied and compared with each other.

The primary goal of applying different optimizers on linear and nonlinear systems is to find out the best-performing algorithm in terms of efficiency and effectiveness. After getting the suitable optimizer, we can extend it to autoencoders. Where we apply it to the latent space to introduce sparsity.

3.1 Linear Systems

In this section, we have taken 2 variations of linear systems. The first is a linear system with 2 features and the second is with 10 features. After applying different optimizers and algorithms, the results are compared with each other.

The preprocessing steps and evaluation metrics are the same for both. Which are mentioned below.

3.1.1 Dataset Generation

The input dataset ($\underline{X}$) is generated using a random number generator (rng) with a shape of (1000, number of features or samples). Each element in the dataset is a random value between 0 and 1. It represents a linear dataset consisting of 1000 samples. The value of the input features is between 0 and 1. We have set the seed of random number generation as constant. So that, we get the same numbers every time.

The output dataset ($\underline{y}$) is calculated by taking the dot product of the input dataset ($\underline{X}$) and a weight vector $\underline{w}$. We have not added any bias to the output, to keep the model structure simple and ease of visualization of different algorithms in a plot.

3.1.2 Normalization

The dataset is then normalized with min-max normalization. The formula for any random variable $\underline{v}$ is shown below:

$$\underline{v}_n = \frac{\underline{v} - v_{min}}{v_{max} - v_{min}}$$

Here, v_{min} is the minimum value of the variable $\underline{v}$ and v_{max} is the maximum value of variable $\underline{v}$.
The purpose of the normalization is to make sure that the scale of all features is similar and to avoid the dominance of any one feature in the model's performance. It also makes the model run faster as the features are close to a standard distribution, and the algorithm can converge faster to find the optimum solution.

3.1.3 Dataset split into training and testing

After normalizing, the dataset is divided into training and testing. The ratio of training to testing dataset is 0.8 i.e. 800 samples are used for the training and 200 samples are used for the testing.

3.1.4 Evaluation metric

As the evaluation metric, we have chosen Normalized root mean square error (NRMSE). A few reasons behind the selection of the NRMSE are:

- **Scale-invariant:** NRMSE takes into account the scale of the data, making it suitable for comparing models that operate on different scales. By normalizing the error, it becomes independent of the magnitude of the target variable.
- **Interpretability:** NRMSE provides a normalized measure of the average error, allowing for better interpretability and comparison across different datasets or models. It helps in understanding the relative performance of models in terms of prediction accuracy.
- **Consistency:** NRMSE allows for consistent error comparisons across different datasets, even when the datasets have varying ranges or units. This makes it useful when dealing with diverse data sources.
- **Sensitivity to outliers:** NRMSE is less sensitive to outliers compared to other error metrics like Mean Squared Error (MSE). By normalizing the error, extreme values have a smaller impact on the overall error calculation, providing a more robust evaluation.

The formula to find NRMSE for the actual output $\underline{y}$ and the model output $\underline{\hat{y}}$ is given as:

$$e = \sum_{j=1}^{N} (\underline{y}(j) - \underline{\hat{y}}(j))^2$$

$$\text{NRMSE} = \frac{1}{N}\sqrt{e/\sigma(\underline{y})}$$

Here, $\underline{e}$ is the mean squared error, $\sigma(\underline{y})$ is the standard deviation of process output $\underline{y}$. N denotes the number of output points.

3.1.5 Loss Function

The loss function that needs to be optimized with respect to the weights is:

$$\mathcal{L} = \min_{\underline{w}} \frac{1}{2N_{samples}} ||\underline{X}\underline{w}^T - \underline{y}||_2^2 + \lambda ||\underline{w}||_1$$

Where $\underline{w}$ is the weight vector, y is the process output, $\underline{x}$ is the input features, and λ is the hyperparameter which represents the strength of regularization. $N_{samples}$ is the number of samples.

3.1.6 Linear system with 2 features

In this case, we have taken 2 features as the input. The 3D plot of both features with the output is shown in Fig. 3.1.

3.1.6.1 Co-ordinate descent (CD)

We have applied the coordinate descent optimizer to introduce sparsity, in which we initialized the optimizer with different values of the weight vector. Them being with ones, zeros, and random values between 0 and 1. We got the same results on all initializations. It concludes that the initialization does not have any effect on the end result. For termination

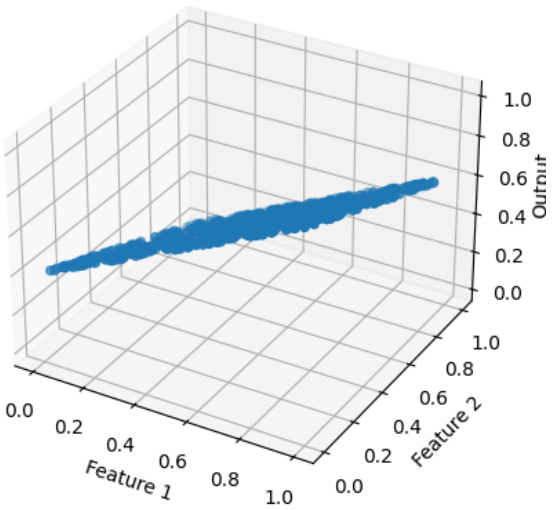

Figure 3.1: 3D Scatter Plot of Features and Output

criteria, the tolerance value between two consecutive update values of the weights is defined as 10^{-4} and the number of epochs is 10,000.

We applied the optimizers on different values of λ to get an idea of how the values of weights change. It can be visualized in Fig 3.2. It can be seen that as we increase the values of λ, weights approach to zero to introduce sparsity. At $\lambda = 1$, both weights become zero, and we have the highest NRMSE.

The effect of λ on NRMSE can be seen in Fig 3.3. A logarithmic scale is applied to the regularization strength λ. To visualize how the algorithm works and how parameters change after each epoch, we extracted both weights and plotted the contour plot of the loss function with both weights. The value of λ for the plot is taken as 0.001. The algorithm converged after 19 iterations. The progress of the optimizer can be seen in Fig. 3.4. Contour lines show the values of a loss function for different

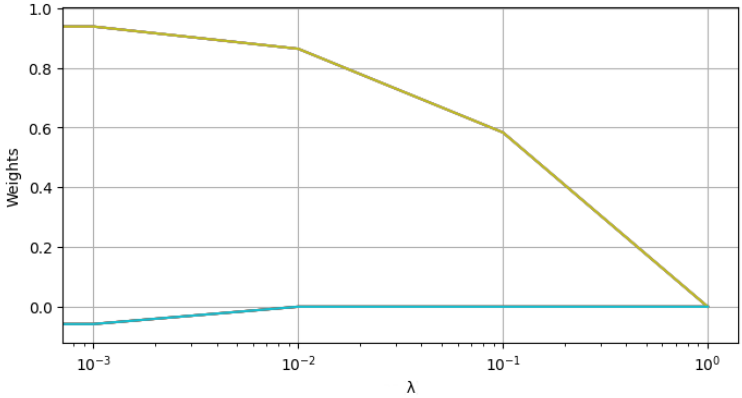

Figure 3.2: Change of the weights with different λ for the linear system with 2 features

weights and the direction of the arrow shows the change in the values. It can be clearly seen that Coordinate descent updates only one parameter at a time.

3.1.6.2 Least Angle Regression (LARS)

The termination criterion for this algorithm is the same as the coordinate descent mentioned previously. We applied the least angle regression(LARS) on the same dataset with $\lambda = 0.001$. The algorithm converged just after 4 iterations and reached global minima. Fig. 3.5. shows the working of LARS. Where the green dot represents the global minima. Fig. 3.5 also shows that the behavior of the weight updation is completely random. It converged faster than all other methods.

3.1.6.3 Gradient Descent (GD)

We also applied the most frequently and widely used algorithm Gradient Descent on lasso regularization. The number of epochs taken is 10,000.

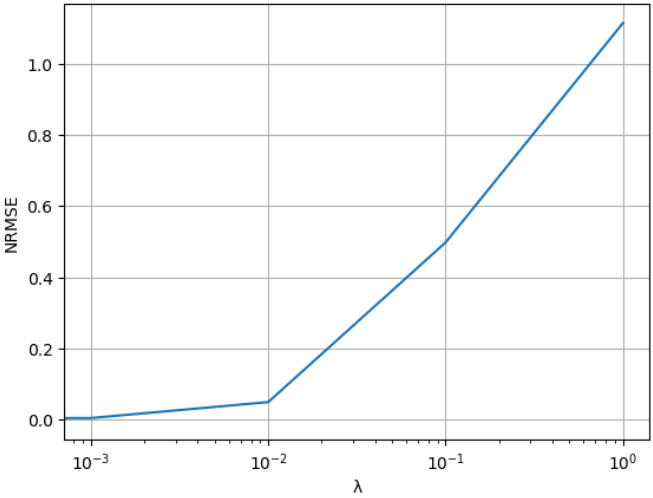

Figure 3.3: Change in NRMSE with λ for linear systems with 2 features

Which is sufficiently large. In some cases, this number was increased to check its effect on the convergence. Here, we have two hyperparameters, which are the learning rate α and regularization λ. The step size is kept constant for an entire training cycle. For the evaluation metric, we have considered weights coming out from the Coordinate descent as the Coordinate descent has converged to global minima.

The effect of epochs, learning rate, and hyperparameters on each other can be seen in table 3.1 From table 3.1, it can be clearly seen that algorithm highly depends on the amount of regularization λ and the learning rate to reach convergence.

For $\lambda = 0.001$ and $\lambda = 0.01$, the algorithm converged for all the values of the learning rate except $\lambda = 0.001$, and for $\alpha = 1.0$, we reached convergence quickly. For $\lambda = 0.1$ and $\lambda = 1.0$, the algorithm did not reach convergence for all the values of a learning rate. For $\alpha = 1$, the algorithm did not converge at all, as the update in the weight was too high, thus algorithm moved away from the global minima.

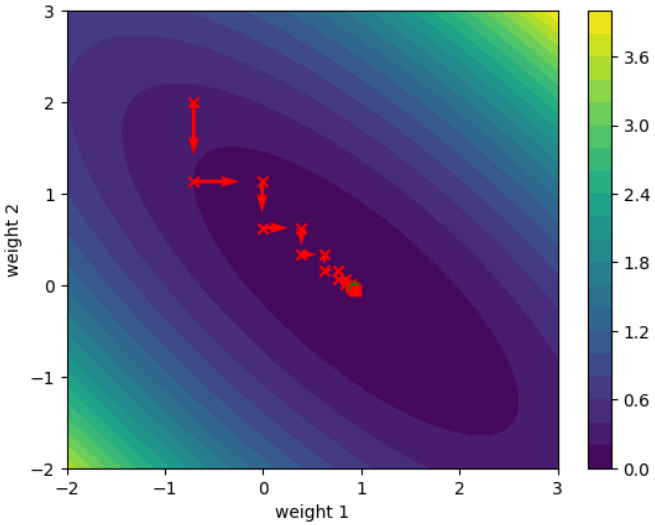

Figure 3.4: Visualization of the update rule for weights of Coordinate descent

3.1.6.4 Nelder-Mead

Nelder-Mead is a popular optimization algorithm used for the numerical optimization of nonlinear objective functions. It is an iterative method that does not require the calculation of gradients, making it suitable for optimizing functions that are not easily differentiable.

The Nelder-Mead algorithm belongs to the class of direct search or derivative-free optimization methods. It works by maintaining a set of simplex points in the parameter space and iteratively updating these points to search for the optimal solution. The simplex is a geometric shape in the parameter space that adapts and moves toward the optimal solution through a series of transformations.

To apply the method, we have used scipy library which is an in-built library of Python. We have used minimize function of it and selected

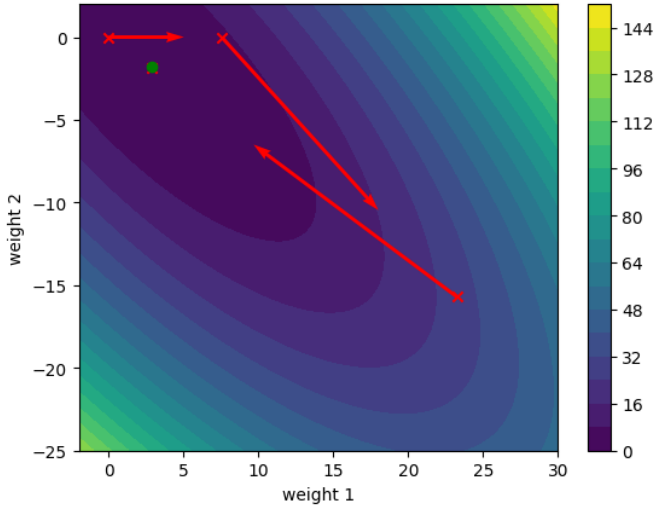

Figure 3.5: Visualization of the update rule for weights of LARS

'Nelder-Mead' as the method. The cost function is the same as mentioned above. The initialization of weights did not have any major effect on the result. The optimization converged after 90 iterations for the same tolerance value of 10^{-4}. The results are the same as Coordinate descent.

The update of the weights and the progress of the algorithm towards global minima(green point) can be seen in Fig. 3.6. We also applied other methods from the scipy library like 'COBYLA' and 'L-BFGS-B'. These are popular gradient-free methods, especially for nonlinear systems. We got similar results as Coordinate Descent and Nelder Mead.

λ	α	**No. of iterations to convergence**
0.001	0.001	10,000
	0.01	4250
	0.1	450
	1.0	47
0.01	0.001	10,000
	0.01	4500
	0.1	270
	1.0	32
0.1	0.001	10,000
	0.01	4300
	0.1	230
	1.0	did not converge
1.0	0.001	10,000
	0.01	4450
	0.1	235
	1.0	did not converge

Table 3.1: Relation between λ, α, and no. of iterations to converge for linear system with 2 features

3.1.7 Linear system 10 Features

3.1.7.1 Co-ordinate descent

In this algorithm. we initialized the optimizer with different values of the weight vector. As in the previous case, initialization did not have any effect on the end result. For termination criteria, the tolerance value between two consecutive update values of weights is taken as 10^{-4} and the number of epochs is 10,000.

We applied the optimizers on different values of λ to understand how the values of weights change. It can be visualized in Fig 3.2. It can be seen that as we increase the values of λ, weights approach to zero to introduce sparsity. At $\lambda = 1$, all weights become zero, and we have the highest NRMSE.

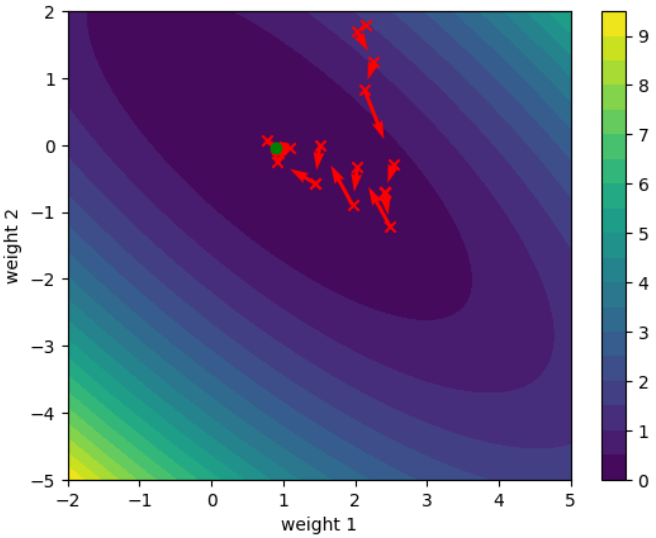

Figure 3.6: Visualization of the update rule for weights of Nelder Mead algorithm

The effect of λ on NRMSE can be seen in Fig 3.8. For all further methods, we will take the weights we get from convergence as a reference to compare the results coming from different methods.

3.1.7.2 Least angle regression (LARS)

Training and the termination criterion are the same as LARS in a linear system with 2 features. We trained the algorithm on different values of λ and predicted the output for the test dataset. After getting the output, we calculated NRMSE. The comparison of the NRMSE for LARS and CD can be seen in Fig. 3.9. It shows that the NRMSE for CD is small for the values of λ less than 0.1. After that, NRMSE continues to rise for CD, while it stays at 1 for LARS.

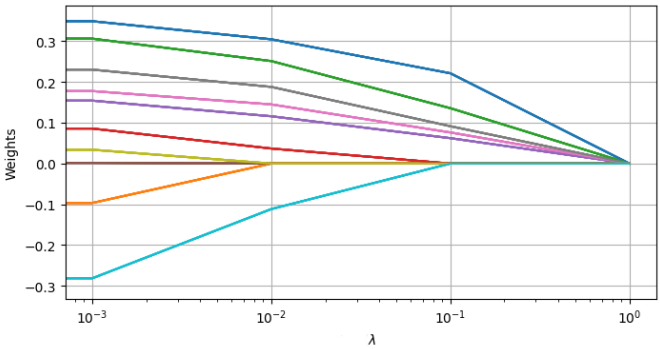

Figure 3.7: Change of the weights with different λ for the linear system with 10 features

3.1.7.3 Gradient Descent

Just like in the previous case, we applied the Gradient descent algorithm for different values of λ and the number of iterations and compared the weights after convergence with the Coordinate descent. The effect of iterations, learning rate α, and hyperparameters on each other can be seen in table 3.2 As we saw in the previous section for 2 parameters, convergence highly depends on hyperparameters α and λ.
For regularization $\lambda = 0.001$, and $\lambda = 0.01$, the algorithm converged for all the values of α except 0.001 and 1.0.
For $\alpha = 1$, and all the values of regularization λ the algorithm did not converge at all, as the update in the weight was too high, thus algorithm moved away from the global minima.
One thing to notice is that, if the algorithm reaches convergence, less learning rate gives a more sparse solution.

3.1.7.4 COBYLA, Nelder-Mead, and L-BFGS-B

In the first case, we evaluated all three methods with different hyperparameters and concluded the number of epochs needed to reach conver-

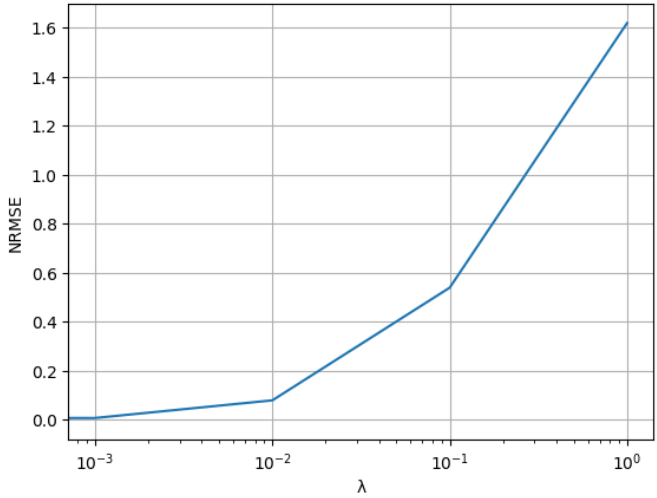

Figure 3.8: Change of the weights with different λ for the linear system with 10 features

gence. For all the methods, the change in the weights stops when the change in the value of the function evaluation is less than 10^{-8}.

A comparison between the three methods with hyperparameters and the number of epochs can be seen in the Table 3.3.

In the table 3.3, columns of Number of iteration shows the number of epochs needed to reach convergence. Here, convergence means whether the weights at the end of each method match the weights of coordinate descent or not.

For $\lambda = 0.001$ and 0.01, all methods reached convergence. For $\lambda = 0.1$, COBYLA performed much better than Nelder Mead and COBYLA, reaching almost the convergence. For $\lambda = 1$, L-BFGS-B introduced sparsity and reached the convergence, whereas COBYLA and Nelder Mead did not introduce any sparsity. In the second case, we used Coordinate descent along with the above-mentioned algorithms to find the solution.

λ	**Learning rate**	**No. of iterations to convergence**
0.001	0.001	10,000
	0.01	4300
	0.1	422
0.01	0.001	10,000
	0.01	4000
	0.1	425
0.1	0.001	4700
	0.01	2900
	0.1	2700
1.0	0.001	10,000
	0.01	4000
	0.1	430

Table 3.2: The relation between λ, learning rate, and no. of epochs for linear systems with 10 parameters

λ	**Number of iterations**		
	Nelder Mead	**COBYLA**	**L-BFGS-B**
0.001	3900	964	110
0.01	7279	428	561
0.1	9418	352	781
1.0	2592	467	1144

Table 3.3: The relation between λ and the Number of epochs to convergence for Nelder Mead, COBYLA, and L-BFGS-B for 10 features of a linear system

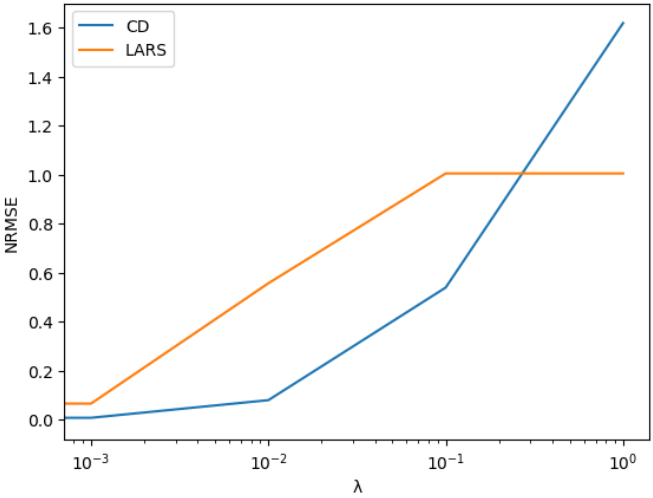

Figure 3.9: Comparison of NRMSE for LARS and CD for linear system with 10 features

For this case, for $\lambda = 0.001$ and 0.01, all three algorithms gave similar results as implementing alone without using Coordinate descent but took almost 3 times more time. For $\lambda = 0.1$, Nelder Mead performed best and reached convergence. For $\lambda = 1$, L-BFGS-B performed best and introduced sparsity in all weights, whereas Nelder Mead and COBYLA did not introduce sparsity.

3.1.8 Conclusion

From all the above methods and both types of linear systems, we can conclude that Coordinate descent, and Gradient Descent reached convergence and introduced sparsity.
Coordinate descent reached convergence quickly for both types of linear systems.
For Gradient Descent, we have to tune the learning rate and epochs to

reach convergence. It also works best for small values of regularization λ.
For Nelder Mead, COBYLA, and L-BFGS-B, it is hard to differentiate which method performs best, as different methods worked better in different cases and hyperparameters.
LARS converged the fastest among all the methods, yet it has comparatively high NRMSE values. Along with these, for nonlinear systems, we do not have any close-form solution to apply LARS. For these reasons, this method will not be evaluated for nonlinear systems and autoencoders.
All the methods do not depend on weight initialization.

3.2 Nonlinear systems

In this section, all different algorithms implemented earlier are implemented on nonlinear systems with different features to evaluate which algorithm works best.

3.2.1 Dataset Generation

The input dataset is generated like the data for the linear systems with random number generation (RNG) with different weights and features. The total number of samples is again 1000. Among 1000, 800 samples are used for the training and 200 are used for validation.

The output dataset has some nonlinear relation with the input dataset. It contains many nonlinear functions like sin, cos, power, exponential, logarithm, and also polynomials. The majority of weights are correlated nonlinearly with the output and some are also in linear correlation with the output.

The output is calculated with the help of a function called a model. Which takes the input features and weights as attributes. The function and the output for systems with 5 features in Python are as below:

def model($\underline{x}, \underline{w}$) :

return $(\sin(w_1 * x_1) + 1^{(w_2 * x_2)} + (x_3^2 - w_3) + w_4 * \log(x_4) + (\sin(x_5))^{w_5})$

Where $\underline{w}$ is the weight vector and $\underline{x}$ is the input feature vector.

3.2.2 Loss function

The loss function that needs to be optimized with respect to the weights is:

$$\mathcal{L} = \min_{\underline{w}} \frac{1}{2N_{samples}} ||\underline{\hat{y}} - \underline{y}||_2^2 + \lambda ||\underline{w}||_1$$

Where $\underline{w}$ is the weight vector, $\underline{y}$ is the process output, $\underline{x}$ is the input feature vector, and λ is the hyperparameter which represents the strength of regularization. $N_{samples}$ is the number of samples.

The evaluation metric is also the same as linear systems being normalized root mean square error (NRMSE).

For all different cases of nonlinear systems, we have considered 7 values of the hyperparameter λ. They are:

$\lambda = [0, 0.0001, 0.001, 0.01, 0.1, 1.0, 10]$

3.2.3 Nonlinear system with 5 features

For this type of nonlinear system, we have 5 features with 3 weights in the nonlinear relation with the output and 2 weights in the linear relation with the output. It is precisely similar as shown in the previous section.

λ	**No. of weights = 0**	**NRMSE (10^{-3})**
0	0	0.41
0.0001	0	1.28
0.001	0	10.18
0.01	0	67.95
0.1	1	295.02
1.0	4	779.11
10	5	1,612.43

Table 3.4: The relation between λ, the Number of weights going to zero, and NRMSE for a nonlinear system with 5 features

We have considered 3 methods for the evaluation of the system. Them being COBYLA, Nelder Mead, and L-BFGS-B with Coordinate descent, COBYLA, Nelder Mead, and L-BFGS-B without Coordinate descent, and Gradient descent.

3.2.3.1 COBYLA, Nelder Mead, and L-BFGS-B without Coordinate descent

Here, for all 3 methods, we update all the weights simultaneously hoping to get the convergence faster than Coordinate descent. The convergence criterion for all the methods is the change in the successive weights should be less than the tolerance of 10^{-8}.

We got almost similar results in terms of the weights and NRMSE as all the methods with Coordinate descent but with a much improved time. The reduction in time is almost five times less than with Coordinate descent.

As we increase the value of λ, the sparsity, and the NRMSE increases. The number of weights going to zero with the values of λ can be seen in the table 3.4 The change in NRMSE with λ can be seen in the Fig. 3.10 For this case, L-BFGS-B performed best in terms of timing and epochs to reach convergence.

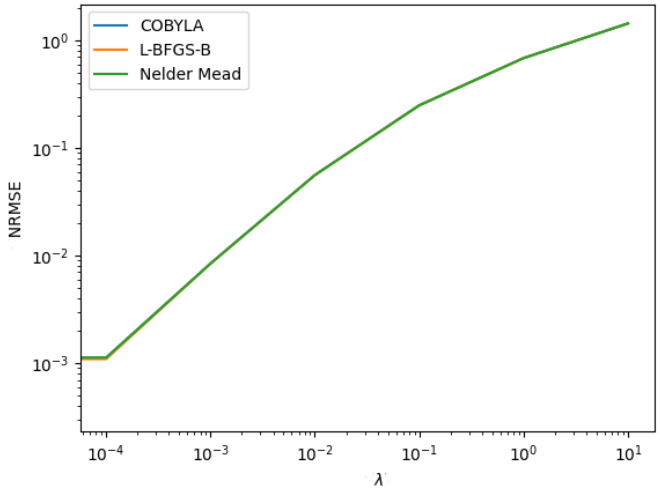

Figure 3.10: Comparison of change in NRMSE with λ for COBYLA, Nelder Mead, and L-BFGS-B without Coordinate descent for nonlinear systems with 5 features

3.2.3.2 COBYLA, Nelder Mead, and L-BFGS-B with Coordinate descent

In this method, we are combining two algorithms to get the result, as the solution by least square is not possible for nonlinear systems. First, we apply Coordinate descent in which we take only one weight at a time keeping all other weights constant, and find the optimum value of that weight with all 3 methods. The reason behind following this approach is the results we got from the experiments on linear systems. We saw that Coordinate descent gave the best results among all algorithms. We repeat this process until we reach the convergence criteria. We have repeated the process with different initialization of weights. We got the same results on all different initialization.

For this method, the convergence criterion for all methods is the tolerance of 10^{-8} and a maximum function evaluation of 10,000. For Coordinate

descent, the criterion is a tolerance of 10^{-6} between two consecutive weight updation values, and the number of iterations is 1,000.

For this case, L-BFGS-B and Nelder Mead performed almost similarly and took fewer epochs to converge than COBYLA.

For a comparison between the two cases, we have selected COBYLA as our primary algorithm. The change in NRMSE with hyperparameter λ for both cases can be seen in Fig. 3.11 . From the figure, we can see that there is not much difference in terms of NRMSE, but COBYLA without Coordinate descent took almost 5 times less time than with Coordinate descent.

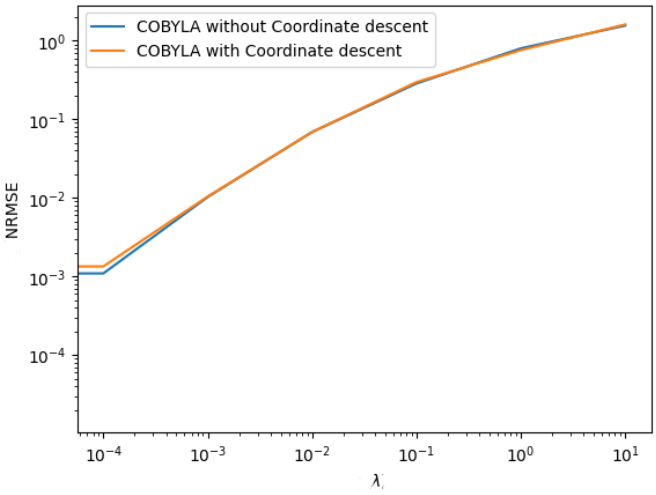

Figure 3.11: Comparison of Change in NRMSE with λ for nonlinear systems with 5 features for two methods: a) COBYLA with Coordinate descent, b) COBYLA without Coordinate descent

3.2.3.3 Gradient Descent (GD)

We applied gradient descent with different values of regularizations and learning rates. With a learning rate of 0.1 and a number of iterations of 3000, we got almost similar results as the other 3 algorithms. The only difference is that it took a much higher time to train and test the model.

Fig. 3.12 shows the comparison of NRMSE and regularization λ for all the methods.

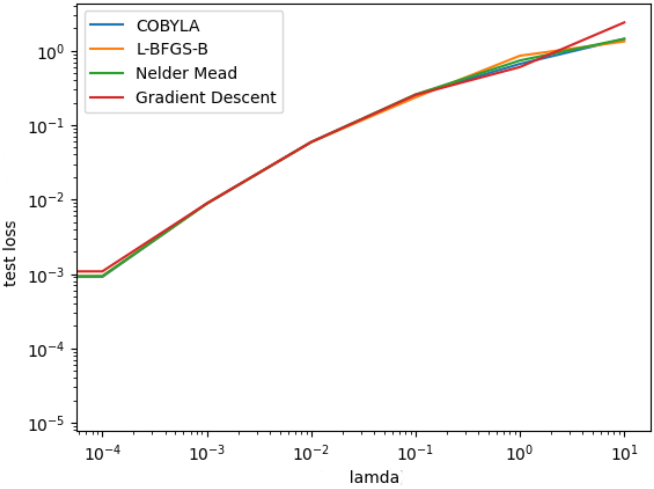

Figure 3.12: Comparison of change in NRMSE with λ for COBYLA, Nelder Mead, Gradient Descent, and L-BFGS-B with coordinate descent for nonlinear systems with 5 features

3.2.4 Nonlinear system with 10 Features

All previously mentioned methods have been applied to the nonlinear system with 10 features.

The application of the method, values of the hyperparameter, and the evaluation metric did not change.

3.2.4.1 COBYLA, Nelder Mead, and L-BFGS-B with Coordinate descent

For 10 features, Nelder Mead and L-BFGS-B took almost a similar number of iterations to converge.

Fig. 3.13 shows the change in NRMSE with λ. COBYLA and L-BFGS-B overlap each other. For $\lambda < 10^{-3}$, COBYLA and L-BFGS-B performed well. After that, all the methods performed equally.

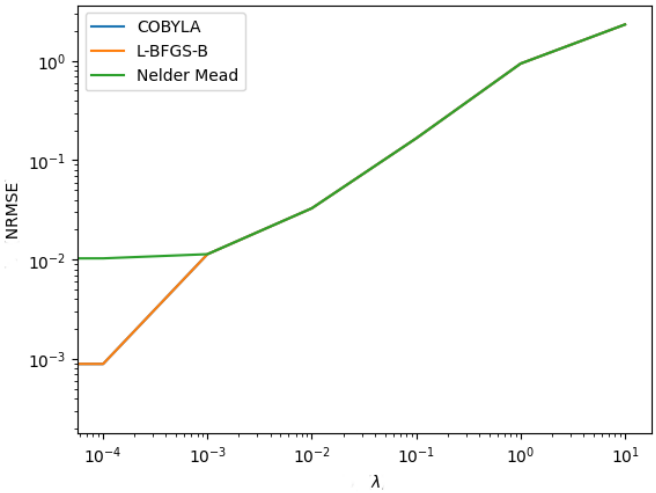

Figure 3.13: Comparison of change in NRMSE with λ for COBYLA, Nelder Mead, and L-BFGS-B with coordinate descent for nonlinear systems with 10 features

3.2.4.2 COBYLA, Nelder Mead, and L-BFGS-B without Coordinate descent

Just in the case of 5 parameters, methods without coordinate descent are much faster than without it. Here, all three methods performed faster than each other with different values of λ.

Fig. 3.14 shows the comparison of NRMSE between COBYLA with and without Coordinate descent with λ. It can be seen that COBYLA without Coordinate descent performed well in terms of NRMSE and also it is faster. The difference between both is not that prominent.

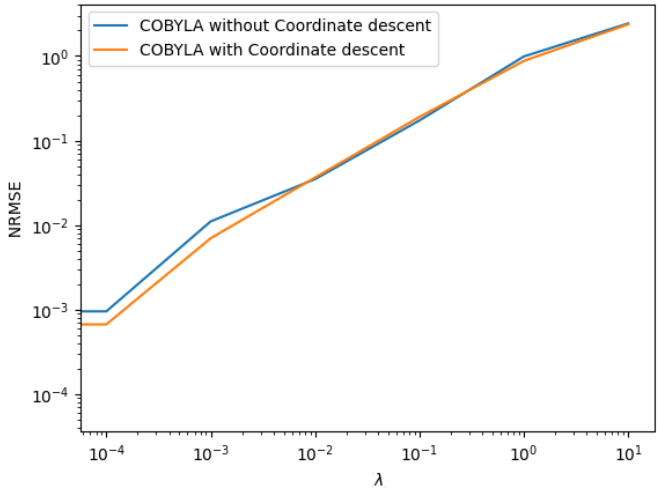

Figure 3.14: Comparison of Change in NRMSE with λ for nonlinear systems with 10 features for two methods: a) COBYLA with Coordinate descent, b) COBYLA without Coordinate descent

3.2.4.3 Gradient Descent (GD)

The training criterion is the same as a nonlinear system with 5 features and the results are almost similar to that.

Fig. 3.15 shows the change in NRMSE with λ. COBYLA, Gradient descent, and Nelder Mead overlap each other and performed well for all the values of λ. L-BFGS-B showed high errors.

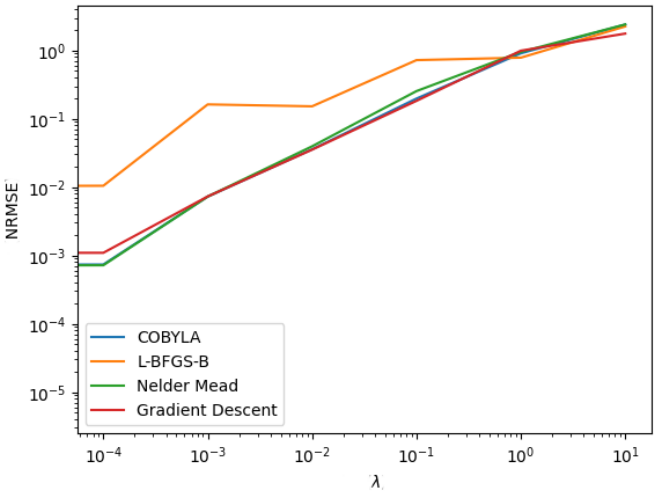

Figure 3.15: Comparison of change in NRMSE with λ for COBYLA, Nelder Mead, and L-BFGS-B without coordinate descent for nonlinear systems with 10 features

3.2.5 Nonlinear system with 20 Features

All the methods mentioned previously have been applied to the nonlinear system with 10 features.

The application of the method, values of the hyperparameter, and the evaluation metric did not change.

3.2.5.1 COBYLA, Nelder Mead, and L-BFGS-B with Coordinate descent

For 20 features, Nelder Mead and L-BFGS-B took almost a similar number of iterations to converge.

Fig. 3.16 shows the change in NRMSE with λ. All the methods performed equally.

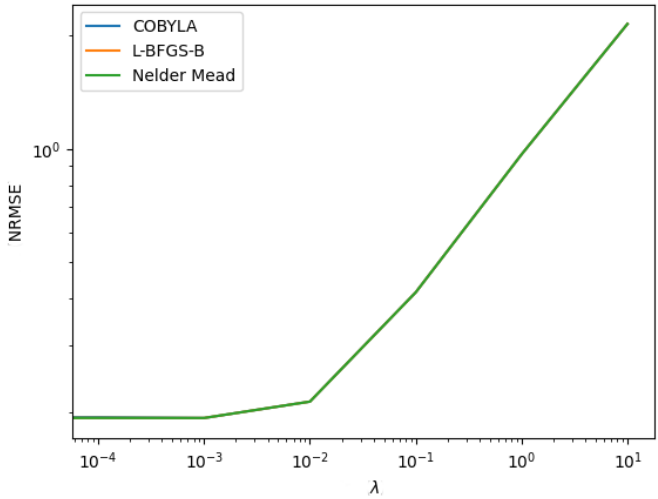

Figure 3.16: Comparison of change in NRMSE with λ for COBYLA, Nelder Mead, and L-BFGS-B with coordinate descent for nonlinear systems with 20 features

3.2.5.2 COBYLA, Nelder Mead, and L-BFGS-B without Coordinate descent

Just in the case of 5 parameters, methods without coordinate descent are much faster than without it. Here, all three methods performed faster than each other with different values of λ.

Fig. 3.17 shows the change in NRMSE with λ. Nelder Mead performed better than the other two methods. There is a very big change in NRMSE after $\lambda = 10^{-2}$ for Nelder Mead. COBYLA showed a slight improvement compared to L-BFGS-B.

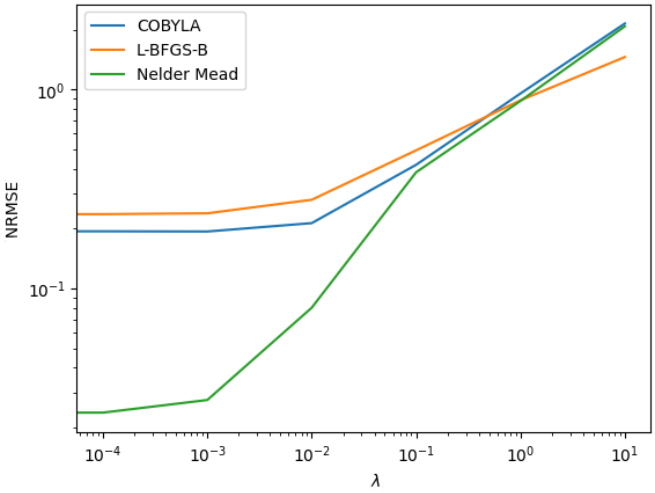

Figure 3.17: Comparison of change in NRMSE with λ for COBYLA, Nelder Mead, and L-BFGS-B without coordinate descent for nonlinear systems with 20 features

Fig. 3.18 compares NRMSE between COBYLA with and without Coordinate descent with different values of λ. It can be seen that COBYLA without coordinate descent performed significantly well for λ smaller than 0.1. This is true for Nelder Mead and L.BFGS-B as well.

3.2.5.3 Gradient Descent (GD)

Gradient Descent performed exactly similarly to COBYLA and Nelder Mead with the same training criterion as before.

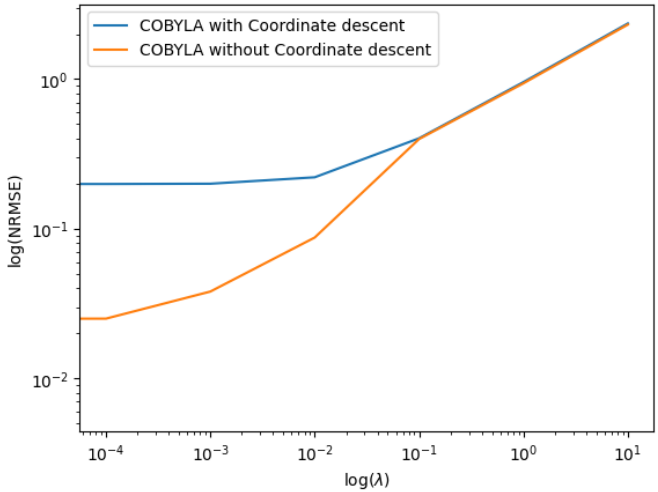

Figure 3.18: Comparison of Change in NRMSE with λ for nonlinear systems with 20 features for two methods: a) COBYLA with Coordinate descent, b) COBYLA without Coordinate descent

3.2.6 Conclusion

From all the different cases and methods, it is tough to conclude which method is better than the rest. COBYLA, Nelder Mead, and L-BFGS-B, all performed almost equally well.

Nelder Mead and COBYLA performed slightly well than L-BFGS-B. There were some random behaviors with L-BFGS-B in nonlinear systems with more features, which are hard to describe, at the same time it is the fastest among all.

Methods without and with Coordinate descent did not show any significant difference. The only difference is that methods without Coordinate descent are much faster. This is obvious as we optimize all parameters at the same time.

Gradient Descent is exactly similar to COBYLA and Nelder Mead, but it is much slower than both.

We will go with all the methods except Gradient Descent for the main task of autoencoders. The reason is it is very slow training time and proper hyperparameter tuning with almost the same results as the other methods.

3.3 Autoencoder

In this section, we have applied all the methods we discussed in the previous sessions on the latent space of autoencoder to introduce sparsity.

The input dataset is the ECG5000 dataset [4], which contains 5000 Electrocardiograms, each with 140 points. The shape of the dataset is (5000,140). It is divided into training and testing datasets with a ratio of 0.2. This means 4000 samples are used for the training and 1000 samples are used for the testing. It is normalized with min-max normalization.

3.3.1 Structure

After trying various architectures with different numbers of hidden layers and number of neurons in each layer, we finally selected the architecture of an autoencoder with the least mean squared error (mse) with 2 hidden layers in the encoder and decoder each, and one layer of latent space. The first hidden layer of an encoder has 64 neurons and the second has 32 neurons. Likewise, the first layer of the decoder has 32 neurons and the second layer has 64 neurons. The last layer, which is an output layer has 140 neurons same as the input layer. The number of neurons in the latent space has been decided where we have the smallest mean absolute error (mae).

Figure 3.19 shows the change in the mae error with the number of neurons in the latent space. It can be seen that after 11 neurons in the latent space, the error stays almost constant. We have selected the size of

the latent space as 20 to check whether we get the improvement in the prediction time and error after deactivating some neurons by applying lasso regularization. The structural graph can be seen in Fig. 3.20.

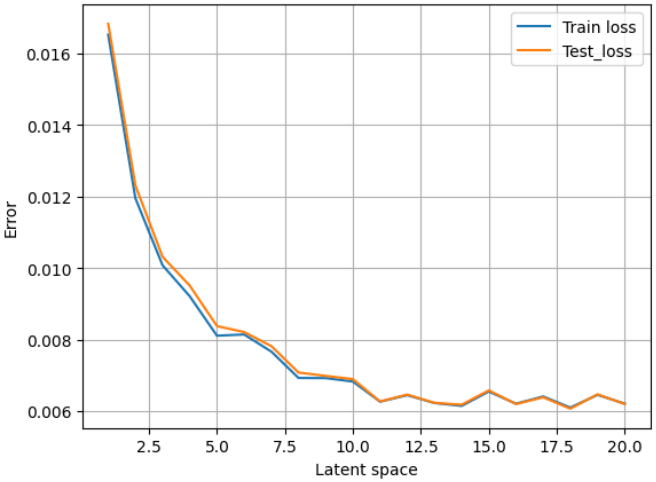

Figure 3.19: Change in the mean absolute error (mae) with the number of neurons in the latent space

Which shows the number of layers, the number of neurons in each layer, and the type of activation. encoder_dense _1 and encoder_dense _2 are two encoder layers with the rectified linear unit (Relu) activation. latent _space represents latent space with linear activation. decoder_dense _1 and decoder_dense _2 are two decoder layers with the rectified linear unit (Relu) activation. The output layer has sigmoid activation.

3.3.2 Training

The steps by which we have trained the autoencoder are shown below:

1. We first, train the autoencoder with Adam optimizer and mean absolute error(mae) as an evaluation metric.

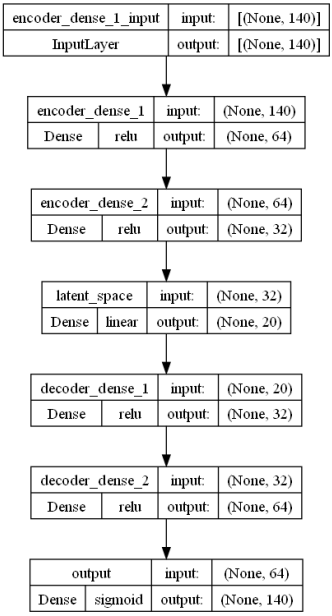

Figure 3.20: Structure of the autoencoder with 1 input layer, 2 encoder layers, one latent space, 2 decoder layers, and 1 output layer

2. Then we apply lasso regularization on the latent space by applying various optimizers discussed in the linear and nonlinear systems.

3. Some of the neurons are deactivated and the weights coming out from those neurons are exactly zero i.e. they do not play any role in the output.

4. We retrain the autoencoder with Adam optimizer but with the fixed deactivated neurons.

The criterion for training is early stopping. If the validation loss does not change for 20 consecutive epochs, we stop the training. We have kept 800 samples as a validation dataset from the training dataset itself. Fig. 3.21 shows how validation and training loss changes with epochs.

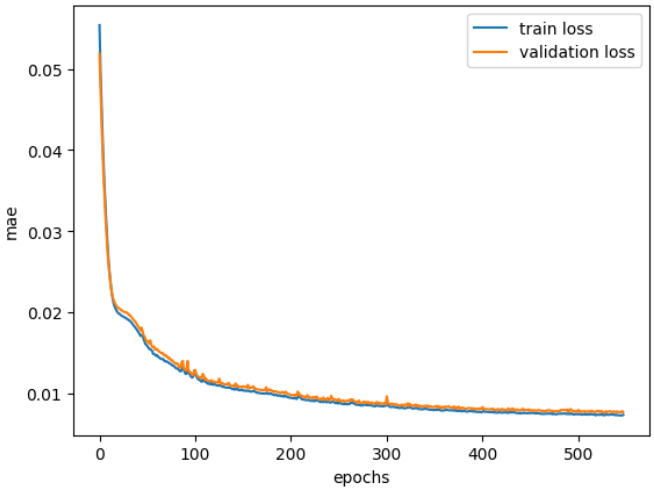

Figure 3.21: change in mean squared error(mae) for training and validation dataset with epochs

3.3.3 Application

For the optimization of the latent space, we have applied two different approaches. These are mentioned below:

1. In this approach, we multiply latent space with a vector of the same size and try to optimize the additional vector, hoping to completely eliminate some neurons.

2. In this approach, we extract the activation values of neurons of the latent space making them input features, and the values of all neurons before activation of an adjacent layer which is the first layer of the decoder making them labels. Thus we convert the nonlinear problem to a linear one and try to optimize the latent space.

3.3.3.1 First Approach

For all the algorithms mentioned below, we have multiplied a weight vector $\underline{\beta}$ with the latent space. The size of the $\underline{\beta}$ is the same as of latent space. The reason behind it is that we cannot make all the weights coming out of the latent space neurons exactly zero by applying lasso regularization. That is why we multiply a specific vector with latent space neurons and apply lasso regularization on this vector to make some of the weights zero. Neurons that are multiplied by the weights which are zero, will not have any effect on the output.

If encoder($\underline{x}$) is the output of the encoder, $\underline{x}$ is the input, and $\underline{\beta}$ is the weight vector, then latent space $\underline{z}$ can be described as:

$$\underline{z} = \underline{\beta} * \text{encoder}(\underline{x})$$

One thing to note is that these weights are not trainable when we compile the model. We initialized the weight vector with all the values as one. After the application of lasso regularization, whatever weight vector we got, we applied it back on the autoencoder and again trained the autoencoder keeping the weight vector the same and non-trainable.

Before we applied lasso regularization on $\underline{\beta}$, we tested the model on the test dataset. We extracted the mean of mae which is 0.00775 for the test dataset. We will use this error to compare different algorithms.

3.3.3.2 COBYLA with and without Coordinate Descent

We applied COBYLA with a maximum of 1,00,000 function evaluations and tolerance of 10^{-6} on the weight vector $\underline{\beta}$ of the latent space with and without Coordinate descent. After retraining the autoencoder, we tested it on the test dataset. Fig. 3.22 shows the comparison of mae with regularization λ. It can be seen that there is not much difference

between both. COBYLA without a Coordinate descent tool almost 5 times more time to train and retrain.

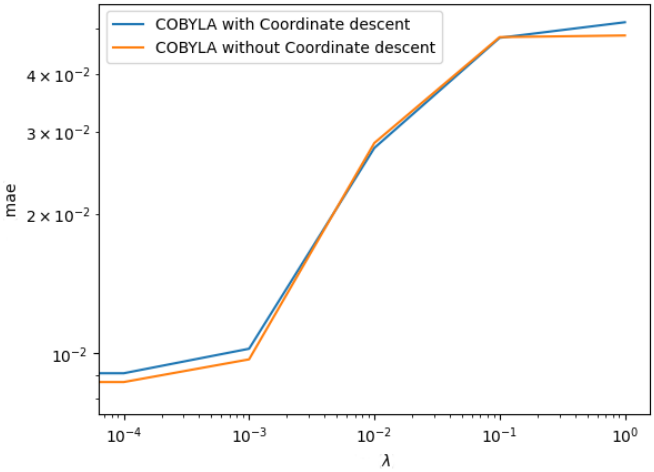

Figure 3.22: Change in mae with regularization λ for COBYLA with and without Coordinate descent for an autoencoder

Table 3.5 shows how many neurons are getting deactivated with different values of regularization λ and what are the values of mean absolute error (mae). We can see that as we increase the strength o regularization, we get more sparse latent space at the same time the error also increases exponentially.

3.3.3.3 Nelder Mead with and without Coordinate Descent

The implementation of the method is exactly similar to COBYLA. Fig. 3.23 shows the change in mar with λ for Nelder Mead with and without Coordinate descent. Same as COBYLA, the mean absolute error increases as we increase the strength of regularization. Table 3.5 can also be referred to for this method as the numbers of deactivated neurons and mae are exactly the same as COBYLA.

λ	**No. of deactivated neurons**	**mae**
0	0	0.0073
0.0001	0	0.0076
0.001	0	0.0088
0.01	4	0.025
0.1	15	0.045
1.0	20	0.055

Table 3.5: Relation between λ, Number of deactivated neurons of latent space, and mae for latent space of 20 with COBYLA

3.3.3.4 L-BFGS-B

We applied L-BFGS-B on the latent space in the same way as COBYLA and Nelder Mead. We got quite strange results unlike for linear and nonlinear systems, where it performed really well and took the least time to converge. L-BFGS-B did not introduce any sparsity for all values of regularization λ. The values of the weight vector remained exactly the same for different λ. We also got the lowest error than COBYLA and Nelder Mead, but it did not introduce even a slight amount of sparsity on the latent space.

Fig. 3.24 shows the change in mae with λ for with and without Coodinate descent. It can be seen that the values are constant for different λ. Fig. 3.25 and Fig. 3.26 show the comparison of all 3 methods in terms of change in mae with λ with and without Coordinate descent respectively. We can see that COBYLA and Nelder Mead performed almost equally, whereas L-BFGS-B performed poorly.

3.3.3.5 Second Approach

The steps involved in this method are:

1. Train the autoencoder model the same as in the previous section.

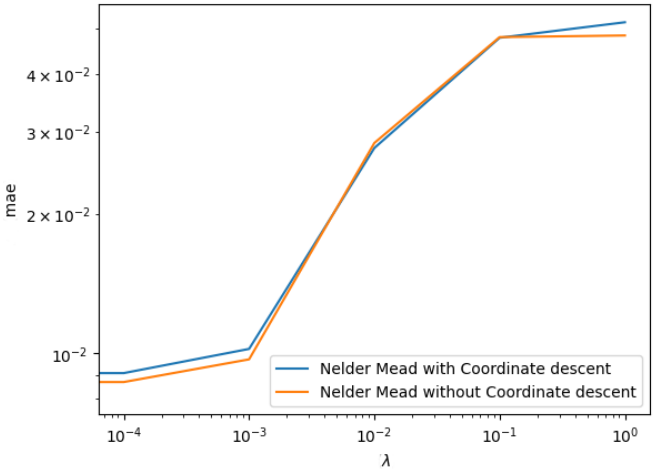

Figure 3.23: Change in mae with regularization λ for Nelder Mead with and without Coordinate descent for an autoencoder

2. Extract the values of neurons of the latent space. Which will act as the values of the features later.

3. Extract the values of the first layer of the decoder before ReLU activation, which will act as the labels later.

4. Apply coordinate descent on these values for different values of regularization.

Basically, we are trying to convert the nonlinear case to a linear case by extracting the neuron activation values before the activation function.

After following the above steps, we got different values of the NRMSE for different regularizations. Fig. 3.27 shows the relation between them.

To compare both approaches, we plotted the test loss with regularization. Fig. 3.28 shows that comparison.

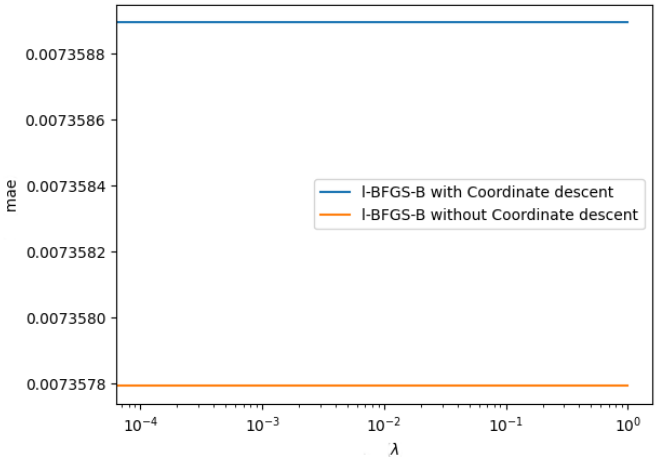

Figure 3.24: Change in mae with regularization λ for L-BFGS-B with and without Coordinate descent for an autoencoder

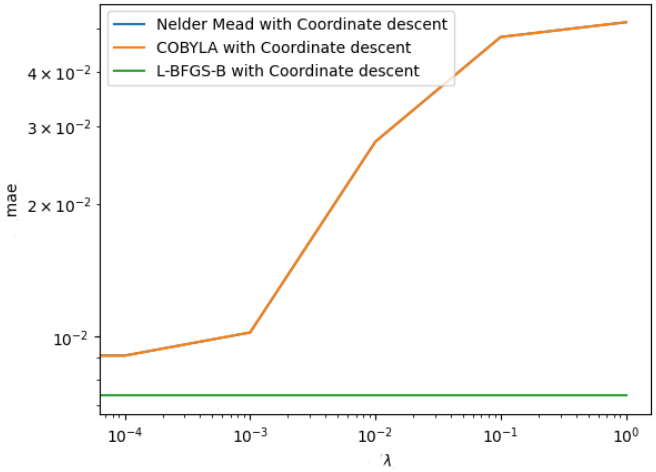

Figure 3.25: Comparison of change in mae with regularization λ for L-BFGS-B, COBYLA, and Nelder Mead with Coordinate descent for an autoencoder

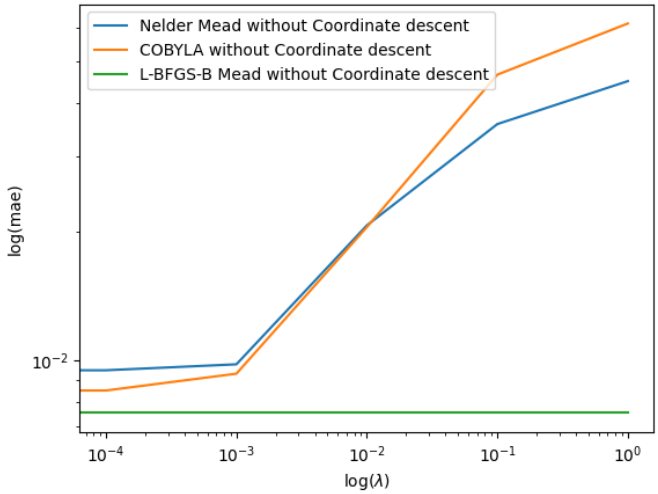

Figure 3.26: Comparison of change in mae with regularization λ for L-BFGS-B, COBYLA, and Nelder Mead without Coordinate descent for an autoencoder

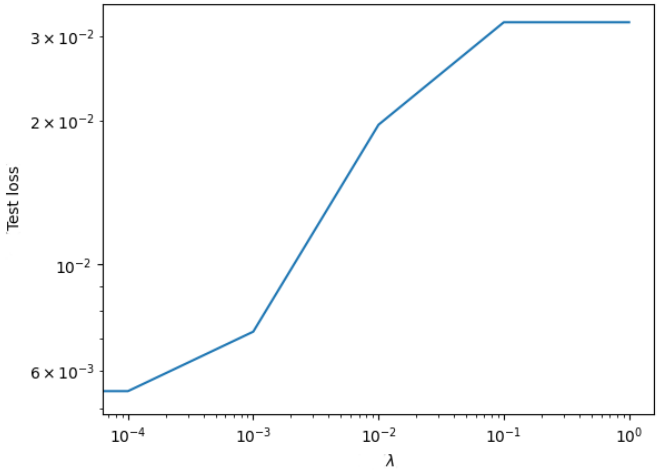

Figure 3.27: Change in the test loss with regularization λ for the second approach

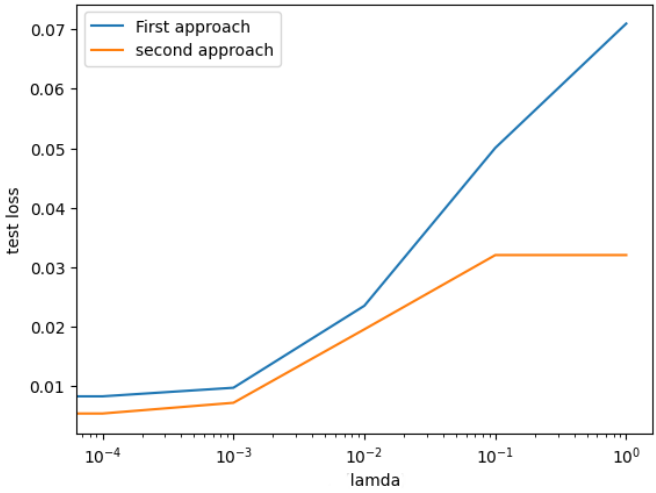

Figure 3.28: Comparison of the change in the test loss with regularization λ for both approaches

4 Conclusion

In this work, we tried to optimize the latent space of an autoencoder using L1 regularization using suitable algorithms which work better on L1 regularization in order to decrease the overfitting of the model along with inference time on the new dataset. The latent space of an autoencoder is a hyperparameter that plays a crucial role in deriving the output in the decoder layer. It can also be considered as a nonlinear Principal Component Analysis to reduce the dimension of the input dataset and maintain its prime information of it. The unregularized model may have some neurons in the latent space, which do not carry useful information. Applying L1 regularization in the latent space helps us to completely eliminate those neurons which have low relevance to the output, but are responsible for the low bias error and high variance error.

After applying L1 regularization to the latent space, we got a very high test error where the strength of regularization is high. For regularization strength of more than 0.1, the test error rises exponentially. Along with that, we do not have any control over the sparsity. If we increase the regularization strength λ from 0.01 to 0.1, we have 4 neurons deactivated and from 0.1 to 1, the number is 16 for 20 neurons latent space. It shows that the number of neurons that get deactivated is completely random and abrupt. We cannot decide the number of deactivated neurons. Even for a small change of 0.01 in regularization, we have a very high number of neurons that get deactivated. For a very small regularization of 0.001 and 0.01, we have smaller test errors, but they fail to introduce any sparsity in the latent space.

4.1 Outlook

Many research papers like [8] and [7] show that by applying group lasso and path lasso using proximal descent, we can achieve nonlinear sparsity in a particular layer of a neural network. The majority of their work is in the field of image. They used it in image denoising and image generation. The same concept can be used in the case of time series datasets in anomaly detection. In this work, we used a simple lasso penalty rather than a group lasso or path lasso, which did not give very good results on the time series dataset. The work of this project can also be extended to the images to check whether we get a lower reconstruction error or not.

Bibliography

[1] Adams, Ryan P.: Overfitting and Regularization. (2018)

[2] Bank, Dor ; Koenigstein, Noam ; Giryes, Raja: Autoencoders. (2020), 03

[3] Efron, Bradley ; Hastie, Trevor ; Johnstone, Iain ; Tibshirani, Robert: Least angle regression. In: *The Annals of Statistics* (2004). https://doi.org/10.1214/009053604000000067

[4] Goldberger AL, Glass L. Amaral LAN L. Amaral LAN: *BIDMC Congestive Heart Failure Database(chfdb)*. 2015

[5] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[6] Nelles, Oliver: *Nonlinear system identification: from classical approaches to neural networks and fuzzy models*. Springer, 2001

[7] Oskar Allerbo, Rebecka J.: Non-linear, Sparse Dimensionality Reduction via Path Lasso Penalized Autoencoders. (2021)

[8] OskarAllerbo: LassoRegularizedNeuralNetworks. (2020)

[9] Powell, M.: A View of Algorithms for Optimization Without Derivatives. In: *Mathematics TODAY* 43 (2007), 01

[10] POWELL, M.J.D.: A DIRECT SEARCH OPTIMIZATION METHOD THAT MODELS THE OBJECTIVE AND CONSTRAINT FUNCTIONS BY LINEAR INTERPOLATION. (1994)

[11] Rahul Mazumder, Jerome H. F. ; Hastie, Trevor: Coordinate Descent With Nonconvex Penalties. 106 (2011). `https://www.jstor.org/stable/23427579`

[12] Tibshirani, Robert: Regression Shrinkage and Selection via the Lasso. In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1996). `https://www.jstor.org/stable/2346178`

[13] Wright, Stephen: Coordinate Descent Algorithms. In: *Mathematical Programming* 151 (2015), 02. `http://dx.doi.org/10.1007/s10107-015-0892-3`. – DOI 10.1007/s10107–015–0892–3

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Siegen, July 31, 2023

Neel Patel