[AI METHODS]

ANN IMPLEMENTATION

(s) Rahmath Ilham

F211372

Table of Contents

Data Pre- Processing

Splitting our data sets

In the data pre-processing phase for an Artificial Neural Network (ANN), we meticulously prepared our dataset to ensure its suitability for training, testing, and validation phases in Excel.
The strategic split of the dataset into proportions of 60%, 20%, and 20% for training, testing, and validation, respectively, was conducted with an eye toward achieving a balanced representation of the underlying data distributions. The 60% allocation for training, comprising 348 instances, was carefully curated to ensure a diverse and representative sample of the data complexities, aiming to teach the model the broad spectrum of patterns without favouring any trend.
The equal division of the remaining 40% of the dataset into testing and validation sets, each consisting of 116 points, was designed to critically assess the model's performance and its ability to generalize from learned patterns to unseen data. This approach helps in identifying any biases the model may have acquired during the training phase, allowing for adjustments before the model is finalized.

Deletion of Data such as Anomalies and Outliers

Table 1: Reason for Deletion and corresponding data rows

Reason for Deletion	Data Set Deleted
Existence of -999	Rows 80, 182, 337, 549
Existence of invalid characters	Rows 69, 115, 293
Empty data	Rows 539, 588
Outliers (AREA)	Rows 22, 141, 577
Outliers (PROPWET)	Rows 340
Outliers (LDP)	Rows 22, 301
Outliers (RMED-1D)	Rows 149, 177, 526

Additionally, we addressed potential data anomalies that could compromise the model's accuracy and reliability. This included the removal of entries marked with -999, indicative of missing/unrecorded data, which could introduce bias or false patterns into the model. We also rectified empty values and non-numeric characters in numeric fields, which could lead to incomplete information and parsing errors, respectively. By cleansing the dataset of these irregularities, we ensured that the ANN is trained, tested, and validated on clean, consistent data. This meticulous approach to data pre-processing enhances the model's ability to learn effectively from the training data and generalize well to new, unseen data, thereby improving its predictive performance in real-world applications.

As part of our comprehensive data pre-processing strategy for the ANN model, special attention was given to the identification and removal of outliers. Outliers, by their very nature, can significantly skew data analysis, leading to potential inaccuracies in model training and predictions. To mitigate this risk, we employed scatter graphs for an in-depth visual examination of each critical predictor field within our dataset.

Initially, scatter plots were generated for key predictors such as “AREA”, “BFIHOST”, “FARL”, “FPEXT”, “LDP”, “PROPWET”, “RMED-ID” and “SAAR”. These visual aids enabled us to meticulously identify data points that starkly deviated from the overall distribution, marking them as potential outliers. For instance, in the “AREA” field, data points in rows 22, 141, and 577 were flagged; in “PROPWET”, a deviation was noted in row 340; “LDP” outliers were identified in rows 22 and 301; and “RMED-1D” presented irregularities in rows 149, 177, and 526.

Following the identification process, the marked outliers were systematically removed from the dataset which can aid in reducing biases. This crucial step was undertaken to purify the dataset, ensuring that the remaining data accurately reflected the genuine patterns and relationships inherent within, devoid of extreme value-induced biases.

This outlier removal process is crucial for enhancing the model's robustness and reliability, as outliers can significantly distort the model's training, leading to overfitting or underfitting. By refining the dataset in this manner, we ensure that the ANN's training, testing, and validation phases are based on a dataset that is both representative and conducive to high model performance in real-world applications.

To transparently communicate the effect of our outlier removal process, I have included 'before' and 'after' scatter graphs below. This graphical representation not only underscores the rigor of our pre-processing approach but also reinforces the reliability of the subsequent ANN model training, testing, and validation phases.

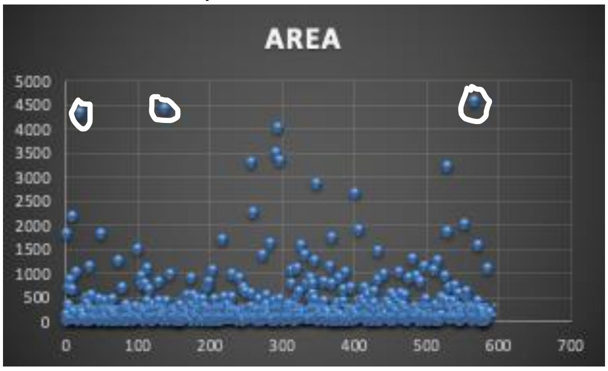


Figure 1: Before removing the marked outliers

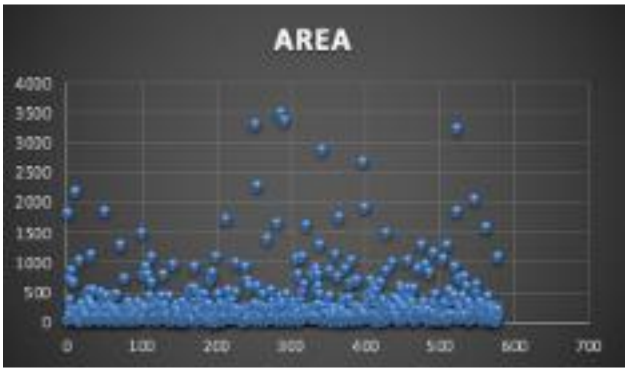


Figure 2: After removing the marked outliers.

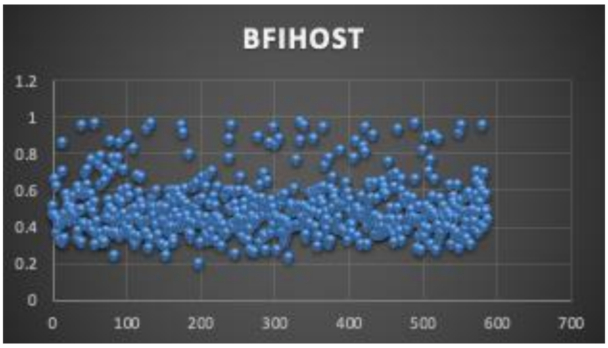


Figure 3: No outliers were found

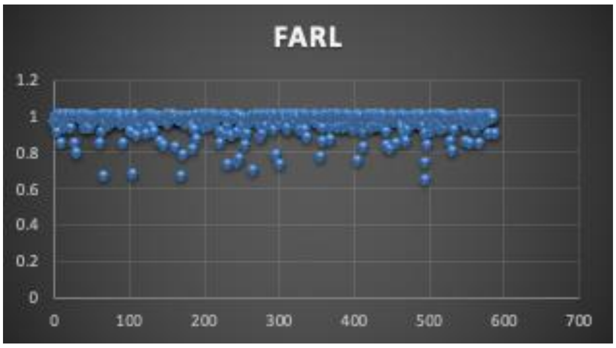


Figure 4: No outliers were found.

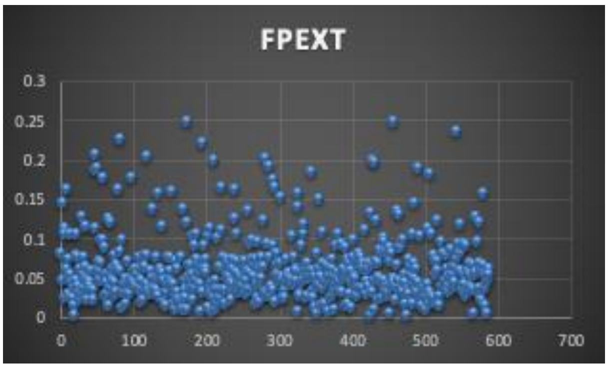


Figure 5: No outliers were found.

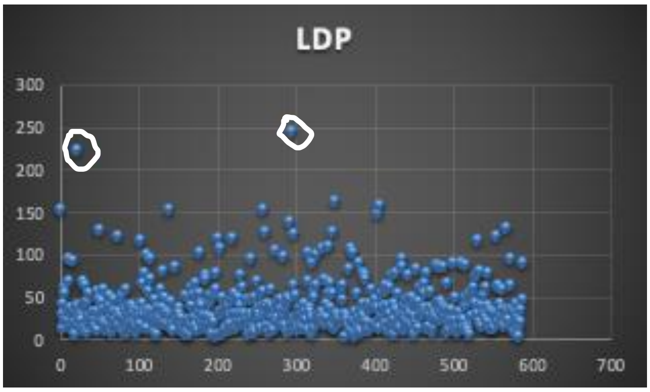


Figure 6: Before removing the marked outliers

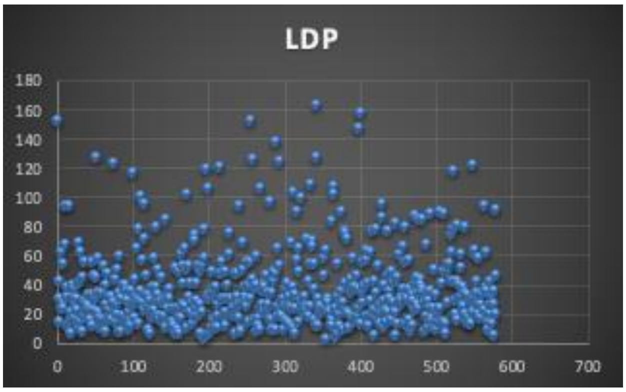


Figure 7: After removing the marked outliers.

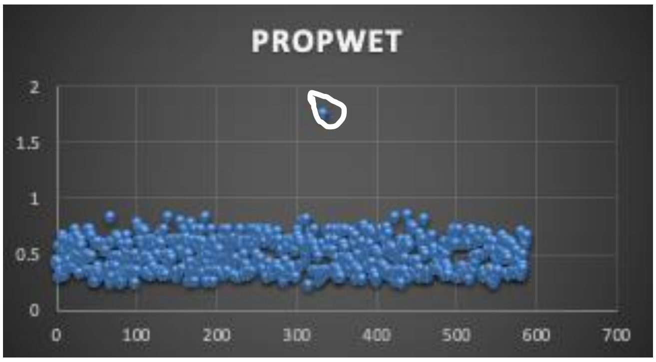


Figure 8: Before removing the marked outliers

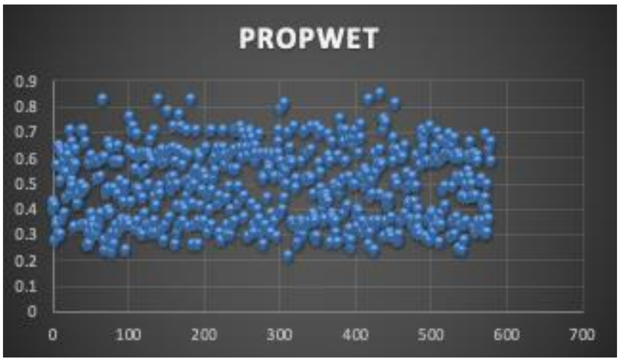


Figure 9: After removing the marked outliers.

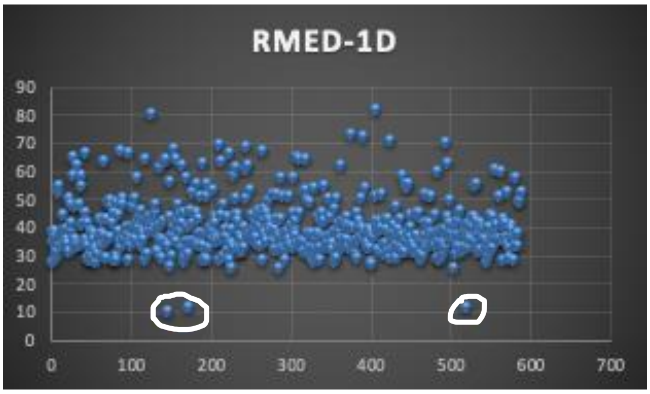


Figure 10: Before removing the marked outliers

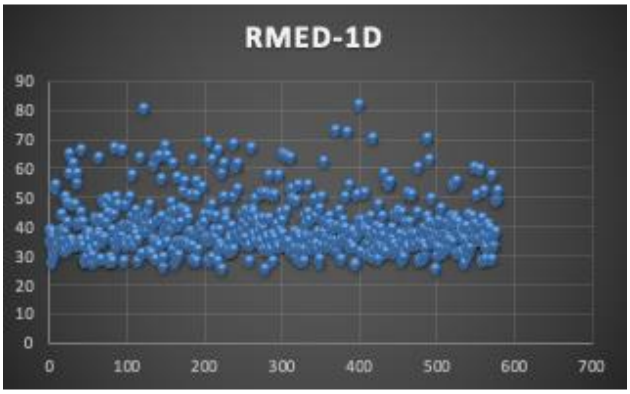


Figure 11: After removing the marked outliers.

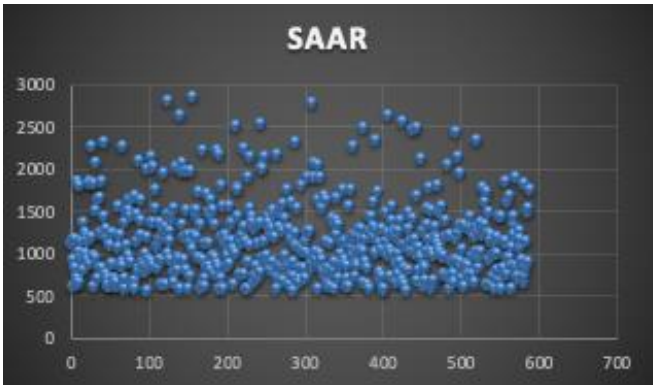


Figure 12: No outliers were found.

By adhering to this meticulous outlier management protocol, we have significantly bolstered the robustness and predictive accuracy of our ANN model, ensuring it is well-poised to deliver high-performance outcomes in real-world applications.

Rationale for the Inclusion of All Predictors in ANN Model

In the process of constructing a robust ANN model for flood prediction, each predictor variable is carefully considered for its potential contribution, regardless of the strength of its linear correlation with the predictand. The correlation matrix serves as an initial guide, delineating the direct linear relationships between predictors and the Index Flood. However, the true complexity of hydrological phenomena requires a more nuanced approach.

Predictors such as 'LDP' (Longest drainage path) and 'AREA' (Catchment area) exhibit strong positive correlations with the Index Flood, suggesting a more direct influence on flood magnitudes. This is expected as larger areas can accumulate more runoff, and longer paths may imply a greater accumulation of flow, both of which are critical factors in the mediation of flood events.

Conversely, variables like 'BFIHOST' (Base flow index) present negative correlations, indicating an inverse relationship. This could be interpreted as areas with higher base flow indexes, which are less likely to contribute to rapid runoff, thus potentially reducing flood magnitudes.

'FARL' (Flood attenuation due to reservoirs and lakes) also demonstrates a negative correlation. Reservoirs and lakes can attenuate flood waves, thus reducing the peak flow reaching downstream areas. The negative correlation reflects the mitigation impact these features have on flood severity.

It is important to note that weaker correlations do not negate the significance of a predictor. Variables with weaker correlations might capture more complex, non-linear interactions that are not reflected in the correlation coefficient but are nonetheless vital to the understanding and prediction of flood events. These predictors may have conditional influences that are only apparent under specific circumstances or in conjunction with other variables.

Furthermore, given the dynamic nature of catchment responses to hydrological processes, the inclusion of all predictors allows the ANN to learn from a comprehensive dataset, capturing subtle but potentially critical patterns that may emerge under varying conditions.

The ANN's capability to identify and leverage complex, non-linear relationships between variables justifies the inclusion of all predictors in the model. This holistic approach ensures the model is not only informed by the most apparent relationships but is also equipped to uncover and utilize subtle patterns within the dataset, leading to a more accurate, reliable, and generalizable flood forecasting model.

Table 2: Cross- Correlation values between predictors and predictand

Predictand	AREA	BFIHOST	FARL	FPEXT	LDP	PROPWET	RMED-1D	SAAR	INDEX FLOOD
AREA	1.0000	-0.0195	-0.0703	0.1260	0.8892	0.0801	-0.0851	-0.0520	0.7606
BFIHOST	-0.0195	1.0000	0.1076	0.1720	-0.0149	-0.5030	-0.3301	-0.4117	-0.2696
FARL	-0.0703	0.1076	1.0000	0.0423	-0.0557	-0.2867	-0.3472	-0.3926	-0.0691
FPEXT	0.1260	0.1720	0.0423	1.0000	0.1617	-0.3633	-0.4166	-0.4016	-0.0981
LDP	0.8892	-0.0149	-0.0557	0.1617	1.0000	0.0825	-0.1219	-0.0929	0.6969
PROPWET	0.0801	-0.5030	-0.2867	-0.3633	0.0825	1.0000	0.5948	0.7490	0.4054
RMED-1D	-0.0851	-0.3301	-0.3472	-0.4166	-0.1219	0.5948	1.0000	0.9005	0.1788
SAAR	-0.0520	-0.4117	-0.3926	-0.4016	-0.0929	0.7490	0.9005	1.0000	0.2384
INDEX FLOOD	0.7606	-0.2696	-0.0691	-0.0981	0.6969	0.4054	0.1788	0.2384	1.0000

Standardising Our Training, Validation and Test Datasets

Incorporating data standardisation into your ANN model's pre-processing phase ensures that all input variables (predictors and predictands) contribute equally to the analysis, preventing any single feature with a broader range from dominating the model's behavior. This process involves scaling the data to fit within a specified range, in this case, between 0.1 and 0.9, using the formula:

$$S_i = 0.8\left(\frac{R_i - Min}{Max - Min}\right) + 0.1$$

$$S_i: Standardised\ value$$

$$R_i: Raw\ value$$

And you de-standardise it with the following equation:

$$R_i = \left(\frac{S_i - 0.1}{0.8}\right)(Max - Min) + Min$$

The wisdom behind choosing this specific range lies in optimizing the ANN's learning process. The scaled values, confined between 0.1 and 0.9, maintain a margin from the extremes of 0 and 1. By avoiding the extreme ends, the standardisation ensures that the gradients remain significant enough during backpropagation, facilitating a more efficient and stable convergence during training.

For the standardisation process, it's crucial to compute the minimum (Min) and maximum (Max) values from the training and validation sets only, excluding the test set. This approach adheres to the principle of simulating a real-world scenario where future data (analogous to the test set) is unseen and should not influence the model's training phase. By deriving the scaling parameters (Min and Max) solely from the training and validation data, we ensure that the standardisation of the test set is consistent with the conditions under which the model was trained and validated.

This method of standardisation, applied uniformly across all data sets (training, validation, and test), guarantees that the ANN model evaluates and learns from the data in a scale-invariant manner, leading to improved model generalization and predictive performance across diverse data distributions.

To illustrate the transformative impact of standardisation on our dataset, particularly for key variables example: 'AREA' and 'INDEX', I have provided a comparative visualisation through scatter graphs. These graphs, presented for both 'before' and 'after' standardisation, offer a glimpse into how the values now adhere to a consistent scale ranging between 0.1 and 0.9.

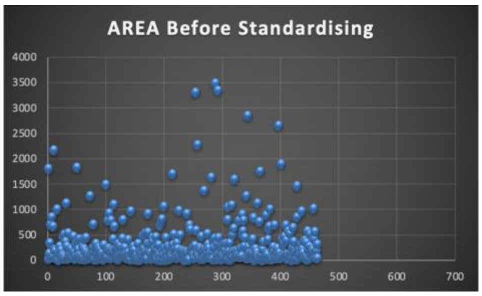


Figure 13: Area before standardising

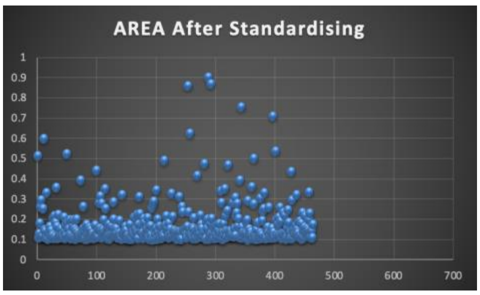


Figure 14: Area after standardising.

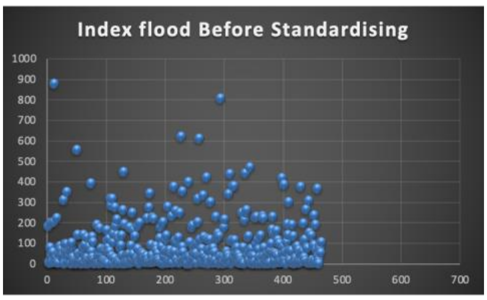


Figure 15: Index flood before standardising

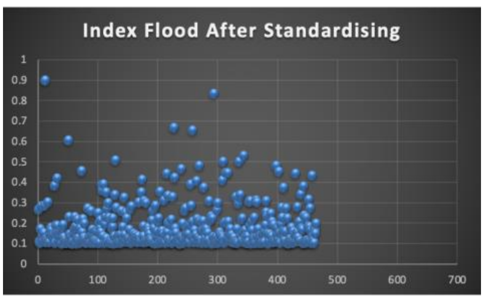


Figure 16: Index flood after standardising!

Implementation of MLP

Choice of Language

For the implementation of our MLP model, I opted for Java due to its robustness, portability, and extensive library support. Java's strong object-oriented programming capabilities facilitate a clear structuring of complex machine learning models, allowing for encapsulation of model components and modular code that is easy to maintain and extend. Moreover, Java's platform independence ensures that our model can be deployed across various environments without modification, a critical advantage for wide-ranging applications in catchment analysis.

Libraries Used

My implementation leverages several core Java libraries to manage data processing, file operations, and mathematical computations efficiently:

java.io: This library is fundamental for input/output operations, enabling my model to read from and write to files. Specifically, BufferedReader and IOException classes allow for efficient reading of dataset files, while handling potential IO exceptions gracefully.

java.nio.charset.StandardCharsets and java.nio.file.Paths: These libraries provide modern approaches to handling file paths and character encoding, ensuring my data reads and writes are reliable and compatible with various system environments.

java.util: The utility classes from this library, such as ArrayList, Arrays, List, Scanner, and Random, are instrumental in managing collections of data, parsing user input, and generating random numbers (for initializing weights and biases), respectively. Their inclusion significantly simplifies the handling of datasets, dynamic array structures, and initialization procedures in the MLP model.
The Random class provides flexible, high-performance data structures for managing inputs, weights, and biases, along with the capability to initialize these parameters in a statistically sound manner.

Math Library: Though not imported in a traditional sense like other libraries, Java's built-in Math library is extensively used for mathematical operations critical to the MLP model's functionality. Operations such as exponentiation (Math.exp) are crucial for implementing the activation function and its derivative.

Limitations

A notable limitation of our approach is the singular reliance on the sigmoid activation function and the decision to utilize all available inputs from the dataset. While the sigmoid function has historically been instrumental in the development of neural networks, its use as the sole activation mechanism might not fully leverage the complex relationships within the data as effectively as newer functions like tanh could. Furthermore, the strategy of employing all dataset inputs, although selected to ensure comprehensive model learning, might not always lead to optimal performance due to potential noise or irrelevant features. However, this approach was adopted with the belief that a model trained on a broader spectrum of inputs offers a more holistic understanding of the underlying patterns, thus holding promise for improved future predictions.

Use of OOP Principles

In developing the Multilayer Perceptron (MLP) model for catchment analysis, we employed a robust Object-Oriented Programming (OOP) approach, leveraging several core principles such as Encapsulation, Inheritance, Polymorphism, and Abstraction. This methodology not only facilitated a modular and scalable design but also streamlined the implementation of various enhancements including backpropagation, momentum, annealing, bold driver, and weight decay. Here's a detailed overview suitable for inclusion in your report:

Object-Oriented Programming Principles Applied:

- Encapsulation: Our MLP model encapsulates both data (weights, biases) and behaviour (methods for forward pass, backward pass, and training) within classes. This encapsulation ensures data integrity and hides the internal state and functionality from the outside world, offering a clear interface for interaction.
- Inheritance: Through inheritance, we extend a base CatchmentMLP class to introduce additional functionalities (momentum, annealing, bold driver, weight decay) without duplicating code. This not only promotes code reuse but also eases future extensions and modifications.
- Polymorphism: By overriding methods such as train in subclasses, our design allows objects of different classes to be treated as objects of a common superclass. This enables the model to exhibit different behaviours (e.g., training processes) under the same interface, depending on the specific enhancements being applied.
- Abstraction: The MLP model abstracts the complexity behind a simple interface, allowing users to leverage its capabilities without delving into the underlying computational details. This abstraction simplifies interactions with the model and enhances usability.

Utilized Data Structures:

- Arrays and Multi-Dimensional Arrays: Serve as the backbone for storing neural network weights and biases, facilitating efficient matrix operations essential for neural computations. Some of them are listed below just for understanding.

```
private double[][] inputToHiddenWeights; // Matrix to store weights from the input layer to the hidden layer.
private double[] hiddenBiases; // Array to store biases for the hidden layer neurons.
private double[] hiddenToOutputWeights; // Array to store weights from the hidden layer to the output layer.
// Array to store the hidden layer outputs for use in weight updates.
double[] hiddenOutputs = new double[numHiddenNeurons];
```

- Lists: Employed to dynamically manage datasets (training, validation, test sets) and track training metrics (e.g., MSEs) over epochs, providing flexibility in handling varying dataset sizes and monitoring model performance. Some of them are listed below just for understanding.

```
// Load the datasets from their respective CSV files.
List<double[]> trainingData = loadData(TRAINING_SET_PATH);
List<double[]> validationData = loadData(VALIDATION_SET_PATH);
List<double[]> testData = loadData(TEST_SET_PATH);
// Lists to hold training and validation mean squared error (MSE) values.
List<String> trainingMSEList = new ArrayList<>();
List<String> validationMSEList = new ArrayList<>();
```

- Scalars and Variables: Used to store and adjust model parameters (learning rate, momentum) and control training dynamics, pivotal for tuning the model's learning process. Some of them are listed below just for understanding.

```
private static final double learningRate = 0.1; // Learning rate for the network's training phase.
```

```
double range = 4.0 / numInputs;// Declare range between -2/n and 2/n
double error = actualOutput - predictedOutput; // Store the difference in values
// Calculate the output layer's delta (error gradient).
double deltaOutput = error * sigmoidDerivative(predictedOutput);
private double momentum = 0.9;// Momentum factor to accelerate convergence in the
desired direction
```

Integration of Data Structures with OOP Principles:

- Encapsulated Data Management: The MLP's design encapsulates critical data structures (arrays for weights/biases, lists for datasets/metrics), ensuring secure and straightforward access mechanisms via class methods.
- Inheritance for Data Handling: Inherited classes utilize and extend the base class's data structures (e.g., weights and biases arrays), allowing for specialized behaviour while maintaining a consistent data handling approach.
- Abstracted Computational Processes: Complex data operations (e.g., forward, and backward propagation) are abstracted behind simple method calls, hiding the intricate matrix operations and algorithmic steps from the end-user.

In conclusion, the adoption of OOP principles in conjunction with carefully selected data structures has significantly contributed to the development of a flexible, maintainable, and efficient MLP model. This approach has not only streamlined the model's implementation but also provided a solid foundation for integrating advanced features and future enhancements. Through this design, we aim to ensure that the MLP model remains at the forefront of innovation in catchment analysis, offering both robust performance and adaptability to evolving computational needs.

Overall Implementation

In the overall implementation of the Multilayer Perceptron (MLP) model for catchment analysis, a user-centric and adaptable approach has been adopted to navigate the complexities of machine learning. The design allows for **user input** to define key parameters, such as the number of epochs for the training process and the number of neurons in the hidden layer, thereby eliminating any hardcoding and enhancing the flexibility and adaptability of the model. This approach ensures that the model can be fine-tuned for various datasets and objectives, facilitating a wide range of experimental scenarios and optimizations.

Model Variations

The project encompasses six distinct models, each building upon the foundational backpropagation algorithm with successive layers of complexity and optimization techniques:

1. Backpropagation Alone: Serves as the baseline, leveraging gradient descent for weight adjustments based solely on the error gradient.
2. Backpropagation + Momentum: Enhances the baseline by incorporating momentum, which accelerates convergence and helps navigate the error landscape more effectively.
3. Backpropagation + Momentum + Annealing: Introduces a dynamic learning rate adjustment (annealing), further refining the training process by adapting the learning rate over time for improved convergence.
4. Backpropagation + Momentum + Bold Driver: Applies the Bold Driver adjustment strategy to dynamically modify the learning rate based on performance, unable to be combined with annealing in this framework.

5. Backpropagation + Momentum + Annealing + Weight Decay: Combines momentum and annealing with weight decay to regularize the model, preventing overfitting by penalizing large weights.
6. Backpropagation + Momentum + Bold Driver + Weight Decay: Merges momentum, Bold Driver adjustment, and weight decay, offering an alternative strategy to annealing for learning rate adaptation and regularization.

Training and Validation

Training is conducted on 60% of the dataset, employing both forward and backward passes to adjust model weights based on the calculated error. Validation occurs every 100 epochs using 20% of the dataset, with a forward pass only to assess the model's generalization capability without influencing its weights. This periodic validation helps monitor the model's performance on unseen data, aiding in the selection of the best epoch and configuration (e.g., the number of nodes) based on the mean squared error (MSE) metric, calculated as the average of the squares of the differences between each actual and predicted value.

Testing and Learning Rate Observation

After completing the training and validation phases, the model undergoes a final evaluation on the test set, comprising 20% of the dataset, to simulate real-world predictions and assess its generalization performance. Throughout this process, the learning rate is meticulously observed and adjusted based on the model's training progression and validation feedback, ensuring optimal learning dynamics and convergence.

This comprehensive and methodical approach underscores a commitment to developing a robust, adaptable, and high-performing MLP model. By meticulously tuning and evaluating the model across different configurations and optimization strategies, this project demonstrates a nuanced understanding of machine learning principles and their practical application in environmental data analysis.

Back Propagation Implementation + Extensions/ Improvements

In this section, we delve into the implementation of a Multilayer Perceptron (MLP), a core component of our machine learning framework designed to predict and analyse catchment data. The MLP's architecture and learning process are tailored to capture the intricate relationships within environmental data, ensuring accurate predictions and insightful analyses.
In a nutshell, this is the back propagation algorithm- our core algorithm I followed:

1. Initialize weights and biases to a random starting point.
2. Select a data point (Cycle through)
3. Make a forward pass and calculate error.
4. Make a backward pass adjusting weights according to the error.
5. Repeat

In addition, I will delve into the extensions and improvements that have been integrated into the core program, enhancing its performance and adaptability.

Basic Backpropagation

Some of the Key Methods:

- initializeWeightsAndBiases (): Initializes weights and biases within a specific range (-2/n to 2/n) to start the training from a diverse state. This approach helps avoid premature convergence to local minima. "n" denotes the number of input neurons, which corresponds to the number of predictors in the data set.
- forwardPass (double [] inputs): Implements the forward propagation through the network using the sigmoid activation function. This method calculates the network's output for given inputs based on current weights and biases. The sigmoid activation function, $\sigma(x) = \frac{1}{1+e^{-x}}$, introduces non-linearity, enabling the MLP to learn and model complex patterns. Its smooth, differentiable nature allows for efficient gradient calculation during backpropagation.
- sigmoid (double x), sigmoidDerivative (double x): Implement the sigmoid activation function and its derivative, essential for forward propagation and backpropagation, respectively.
- train (List<double []> trainingData, int epochs, List<double []> validationData): Manages the training process over multiple epochs, adjusting weights using the backpropagation method based on the calculated error.
- backwardPass (double [] inputs, double predictedOutput, double error): Implements backpropagation for error correction. Updates weights and biases in the direction that minimizes the error between predicted and actual outputs.

Backpropagation Equations involve calculating the gradient of the error with respect to each weight, adjusting weights in the opposite direction of the gradient to minimize error.

Forward Pass:

"forwardPass ()": The forward pass involves calculating the weighted sum of inputs for each neuron in the hidden layer, adding a bias, and applying a non-linear activation function. This process transforms the linear input into a format that the network can use to capture and model complex, non-linear relationships inherent in the data. The output neuron sums the weighted outputs from the hidden layer, adds the output bias, and applies the sigmoid function to produce the final output.
The sigmoid activation function, $\sigma(x) = \frac{1}{1+e^{-x}}$, introduces non-linearity, enabling the MLP to learn and model complex patterns. Its smooth, differentiable nature allows for efficient gradient calculation during backpropagation.
Hidden Layer Computation: Each neuron in the hidden layer calculates a weighted sum of its inputs, incorporating a bias term. This is mathematically represented as:

$$hi = \sigma(\sum_{j=1}^{numInputs} \left(x^j . w^{ji}\right) + bi$$

Here hi represents the output of the ith hidden neuron, x^j are the input values, w^{ji} denotes the weights connecting to input j to hidden neuron, are weights from input to hidden layer and bi are hidden layer biases for the ith hidden neuron and $\sigma(x) = \frac{1}{1+e^{-x}}$, is the sigmoid activation function.

Output Layer Computation: The network's final output is calculated by aggregating the outputs from the hidden layer, again applying weights and a bias followed by the sigmoid function.

$$Output = \sigma(\sum_{i=1}^{numHiddenNeurons} (hi . wi) + boutput$$

In this equation O is the final output, hi are the outputs from the hidden layer neurons, wi are weights from hidden neurons to output layer neuron, and $boutput$ is the output layer bias.
This corresponds to the following 2 equations:

$$S_j = \sum_i w_{i,j} u_i \qquad u_j = f(S_j) = \frac{1}{1+e^{-S_j}}$$

Backward Pass:

"backwardPass ()": Backpropagation is where the network learns from its mistakes. By computing the derivative of the error with respect to each weight (using the chain rule), the network identifies how changes to weights impact the overall error.

The sigmoid derivative is used in backpropagation to determine how much each neuron's output contributed to the error. This derivative reflects how changes to the neuron's input affect its output, critical for adjusting weights in the right direction.
Error calculation: The error is calculated as the difference between the actual output (y) and the predicted output (o) from the forward pass: $\delta_{output} = y - o$
The derivative of the sigmoid function $\sigma'(x) = x(1-x)$ is used to compute gradients.
Weights are updated using the delta rule, which involves the learning rate α=0.1 the computed gradients, and the respective inputs or neuron outputs;

$$w_{new} = w_{old} + \ \alpha * \ \delta * input$$

This corresponds to the following equation: $w^*_{i,j} = w_{i,j} + \rho\delta_j u_i$
Application of the sigmoid derivative in backpropagation:
For output weights, the error gradient δ_{output} is calculated first, factoring in the derivative of the hidden neuron's activation:
$\delta_{output} = y - o.\sigma'(o)$.

For weights from the input to the hidden layer (w_{ji}), the error gradient $\delta^{hiddeni}$is calculated by propagating the output layer's error back through the network, accounting for the derivative pf the hidden neuron's activation:
$\delta^{hiddeni} = \delta^{output}.w_i.\sigma'(h_i)$ where h_i is the output of the i^{th} hidden neuron.

These equations of backward pass correspond to the following:

$f'(S_j) = u_j(1-u_j)$ (if we are using sigmoid functions)

$\delta_O = (C-u_O)\,f'(S_O)$ O is the output node (do this first)

$\delta_j = w_{j,O}\delta_O f'(S_j)$ for hidden layer nodes (any order)

Bias updates: Similarly biases for both hidden and output neurons are updated using their respective error gradients and the learning rate:

$$b_{new} = b + \ \alpha * \ \delta * input$$

This also corresponds to the following equation: $w^*_{i,j} = w_{i,j} + \rho\delta_j u_i$ where w can be weights/ biases.
The backward pass thus serves as a mechanism for the MLP to introspect on its performance and iteratively refine its parameters, aiming for a model that accurately maps inputs to outputs with minimal error. The learning rate (α=0.1) plays a crucial role in modulating the magnitude of these updates, striking a balance between rapid learning and the stability of convergence.

"train ()": Training involves iteratively presenting the network with input-output pairs, allowing it to adjust its weights and biases to minimize prediction errors. This iterative refinement, conducted over multiple epochs selected by the user, helps the network to converge towards a set of weights that can generalize well across unseen data.

In the development of our Multilayer Perceptron (MLP) for catchment analysis, we employ a deliberate strategy by dedicating 60% of the entire dataset exclusively to the training phase. This decision is rooted in a balanced approach to machine learning, where the dataset is segmented into distinct sets for training, validation, and testing, each serving a unique purpose in the model's lifecycle.

Momentum

Incorporating momentum into the weight update process of your Multilayer Perceptron (MLP) model introduces a crucial enhancement that aids in the convergence of the training process. By adding a fraction of the previous weight update to the current update, the model can navigate the optimization landscape more effectively.

Momentum is incorporated by adding 0.9 times the previous weight update to the current adjustment, enhancing the model's ability to navigate the error landscape smoothly.

$weightChangeHO\ and\ weightChangeIH\ holds\ the\ following$: $\Delta w_{i,j} = w^*_{i,j} - w_{i,j}$

For hidden-to-output-weights ("hiddenToOutputWeights"), the update with momentum is computed as:

$$hiddenToOutputWeights[i] += currentUpdateHO + (0.9 * weightChangeHO)$$

here $currentUpdateHO$ is the product of the learning rate, the delta of the output layer and the output of the hidden layer neuron. $weightChangeHO$ is the difference in weight from the previous iteration, captures the 'velocity' of weight change from the previous iteration, introducing a memory-like effect that smooths the update path.

Similarly, for input-to-hidden weights ("inputToHiddenWeights"), the update with momentum is:

$$inputToHiddenWeights[j][i] += currentUpdateIH + (0.9 * weightChangeIH)$$

In this case, $currentUpdateIH$ is derived from the learning rate, the hidden layer's delta, and the input value, reflecting the direct update for the current step. $weightChangeIH$ denotes the previous update's influence, adding a directional persistence to the weight adjustments.

This corresponds to the following equation: $w^*_{i,j} = w_{i,j} + \rho\delta_j u_i + \alpha\Delta w_{i,j}$ where $\alpha = 0.9$

Incorporating momentum addresses training challenges like erratic update paths and slow convergence by smoothing the trajectory of weight updates and injecting a form of inertia. This modification not only accelerates the model's training but also enhances its ability to escape local minima and navigate plateaus in the error landscape. Consequently, the MLP model achieves more robust and reliable predictive performance, particularly crucial in the complex domain of catchment analysis, thereby significantly contributing to the model's efficacy and efficiency.

Annealing

To further refine the training process of our Multilayer Perceptron (MLP) model, we introduce the concept of annealing, a technique inspired by the metallurgical process to improve material properties. In the context of machine learning, annealing involves the dynamic adjustment of the learning rate over time, allowing for a more nuanced approach to convergence.

Our AnnealingMLP class, an extension of the base CatchmentMLP model, incorporates annealing through a sigmoid-based schedule that adjusts the learning rate for each epoch. This is achieved by modifying the train method to calculate an adjusted learning rate as follows:

Adjusted Learning Rate: For each epoch x, the learning rate is adjusted using the formula:
$adjustedLearningRate = p + (q - p) * (1 - \frac{1}{1+e^{10-\frac{20x}{r}}});$
Where p is the final learning rate, q is the initial learning rate, and r is the total number of epochs and x is the epochs so far. This sigmoidal function ensures a gradual decrease from q to p facilitating a smooth transition from exploratory to exploitative learning phases.

Start Learning Rate (q = 0.1): The initial learning rate is set to 0.1 to facilitate a phase of rapid exploration across the error landscape at the onset of training. A relatively higher learning rate enables the model to traverse the error surface broadly, making substantial strides towards the optimal regions. This setting is particularly effective in the early stages of training, where the goal is to quickly converge towards a promising area of the error landscape.

End Learning Rate (p = 0.01): As training advances, the learning rate is gradually reduced to 0.01, transitioning the model into a phase of fine-tuning and exploitation. This lower learning rate allows for more precise adjustments to the model's weights, minimizing the risk of overshooting the minima. By the time the learning rate approaches p, the model is expected to be in the vicinity of an optimal solution, and smaller steps are necessary to refine its parameters without disturbing the already achieved progress.
This corresponds to the following equation:

$$f(x) = p + (q-p)\left(1 - \frac{1}{1+e^{10-\frac{20x}{r}}}\right)$$

Example:
p = end parm: 0.01
q = start parm: 0.1
r = max epochs = 3000
x = epochs so far

In the implementation of the annealing technique within your Multilayer Perceptron (MLP) model, the parameters p and q play pivotal roles in defining the learning rate's dynamic range over the training epochs. The choice of p = 0.01 and q = 0.1 is strategic, designed to optimize the learning trajectory from an initially higher rate to a significantly lower rate as training progresses.

Bold Driver

Building upon our established foundation of backpropagation and momentum, we further refined our Multilayer Perceptron (MLP) model's training process by integrating the Bold Driver method by encapsulating it in my BoldDriverMLP class. This advanced strategy dynamically adjusts the learning rate based on the model's performance changes, leveraging the strengths of basic backpropagation and momentum to accelerate convergence and improve stability.

The BoldDriverMLP class, an extension of our CatchmentMLP model, implements the Bold Driver method by overriding the trainEpoch method to include a learning rate adjustment mechanism based on epoch-to-epoch performance feedback. This process involves several key functions and steps directly derived from your code:

1. Storing Pre-Update weights and biases: At the start of each epoch, the current weights and biases are saved, allowing for potential rollbacks if the model's performance degrades. ("inputToHiddenWeightsBeforeUpdate", "hiddenBiasesBeforeUpdate", "hiddenToOutputWeightsBeforeUpdate, and "outputBiasBeforeUpdate")

2. Calculating MSE: Post-training, the Mean Squared Error (MSE) is computed for the training dataset using "calculateMSE" and for the validation dataset using "calculateValidationMSE" facilitating an informed decision on the learning rate adjustment.
3. Adjust the learning rate: Based on the MSE comparison ("adjustLearningRate" function), the learning rate is modified,

- If the performance worsens significantly (currentMSE > previousMSE * (1 + 0.04)), the learning rate is reduced (learningRate *= 0.7) to temper the subsequent epoch's updates.
- If performance improves (currentMSE < previousMSE), the learning rate is increased (learningRate *= 1.05) to expedite convergence.

4. Rolling Back Updates: In cases of performance degradation, the previously saved weights and biases are restored to undo the last updates ("undoWeightUpdates" function), a direct countermeasure against potential divergence.

To optimize our Multilayer Perceptron (MLP) model's learning efficiency and generalization capability, validation and learning rate adjustments are performed every 1000 epochs. This approach is designed to mitigate the risk of premature adjustments that could destabilize the training process.

- Validation Interval: Conducting validation every 1000 epochs allows the model ample time to learn and adapt from a significant amount of data, offering a more stable and clear indication of its performance trend over time. This strategic pacing ensures that model evaluations are both meaningful and computationally efficient.
- Learning Rate Adjustment: Similarly, adjusting the learning rate at these intervals ensures that changes are based on substantial learning progress, avoiding the 'snowball effect' where the learning rate could either decrease too rapidly, hampering learning, or increase uncontrollably, leading to potential divergence.

These deliberate intervals for validation and learning rate adjustment are crucial in maintaining a balance between rapid learning and the stability of convergence, ensuring the model's training process is both effective and efficient.

Note: We cannot do bold driver and annealing together as they both adjust the learning paramter, so I try different models.

Weight Decay

Incorporating weight decay into the training process of our Multilayer Perceptron (MLP) model represents a significant advancement towards enhancing model generalization and mitigating overfitting. This section elaborates on the implementation details, underlying equations, and the structured approach utilized in integrating weight decay alongside annealing.

'train': Orchestrates the training process, distinguishing between phases with and without weight decay application. It leverages a Boolean flag to toggle the application of weight decay.

'calculateWeightDecayTerm': Dynamically computes the weight decay term based on the adjusted learning rate and epoch, ensuring that the regularization strength is appropriately scaled throughout the training.

'applyWeightDecay': Directly applies the calculated weight decay term to all model parameters, including weights and biases, effectively shrinking their magnitudes to prevent overfitting.

The core equations are as follows:

Weight Decay Term Calculation:
The weight decay term (Weight Decay Term) is calculated using:
Weight Decay Term= $\nu . \omega$

Where ν (regularization parameter) is determined by: $\upsilon = \frac{1}{\rho e}$ where it ranges in between 0.001 and 0.1.

And Ω is determined by:

$$\Omega = \frac{1}{2n}\sum_{i=1}^{n} w_i^2$$

where n: weights and biases

Penalty Application: In applyWeightDecay, the calculated decay term is subtracted from all model weights and biases. This direct adjustment effectively reduces their magnitudes, promoting a model that is complex enough to learn from data but simple enough to generalize well.
And this adds a "penalty term" to the error function as follows:

$$\widetilde{E} = E + \upsilon\Omega$$

$(upsilon, omega)$

$\delta_O = \underbrace{(C\text{-}u_O)}_{E} f'(S_O)$

Becomes:

$\delta_O = (C\text{-}u_O + \upsilon\Omega) f'(S_O)$

Training and Network Selection

The performance of Multi-Layer Perceptron's (MLPs), a type of Artificial Neural Network (ANN), is fundamentally influenced by the training process and network configuration. Training optimizes the network's weights to minimize prediction errors, which is crucial for the model's ability to generalize to new data. Moreover, the architecture of the network, including the number of hidden layers and nodes, along with the training parameters, plays a significant role in determining its efficiency and accuracy. Consequently, exploring a variety of training strategies and configurations is essential for developing high-performing ANNs.

Our objectives include assessing the impact of diverse training techniques on MLP models and understanding how changes in network setup, such as the number of hidden nodes and training epochs, affect performance. This encompasses the evaluation of basic backpropagation, enhancements like momentum, annealing, bold driver, and weight decay, and their collective effects on reducing mean squared error (MSE) on a validation set.

We have conducted experiments across five models to span a comprehensive range of training methodologies:

1. **Basic Backpropagation**: Serves as a foundational comparison, utilizing the standard error backpropagation algorithm.
2. **Backpropagation with Momentum, Bold Driver, and Weight Decay**: Incorporates momentum and weight decay while leveraging bold driver for dynamic adjustments of the learning rate based on performance.
3. **Backpropagation with Momentum, Annealing, and Weight Decay**: Enhances the basic approach by introducing mechanisms for accelerated convergence, adaptive learning rates, and overfitting prevention.
4. **Backpropagation with Momentum and Annealing**: A focused examination of the synergy between momentum and annealing techniques without weight decay, to isolate their effects on model training and performance. (No graphs included, but detailed table is listed)
5. **Backpropagation with Momentum and Bold Driver**: Investigates the combination of momentum with the bold driver approach, omitting weight decay to specifically understand the impact of these two strategies on learning dynamics. (No graphs included, but detailed table is listed)

These models were rigorously tested across configurations employing 6, 8, 10, 12, and 14 hidden nodes, trained over epochs of 10,000; 20,000; and 50,000. The aim of this diverse experimental setup is to identify the most effective configurations for learning efficiency and model generalization, by exploring the balance between network complexity and training duration.

While we have foregone the inclusion of detailed graphical representations in the interest of simplicity and clarity, comprehensive tables have been provided to effectively convey the outcomes of our experiments. These tables detail critical information for each model, including:

Model Name: Clearly identifies each model configuration for ease of reference.

Epoch at Min MSE: Specifies the training epoch at which the minimum mean squared error was observed, offering insights into the model's convergence behavior.
Hidden Neurons: Indicates the number of neurons in the hidden layer, shedding light on the model's complexity.
Min MSE: The lowest mean squared error achieved, highlighting the model's best performance.
Max MSE: The highest mean squared error observed, which, when contrasted with the Min MSE, provides a sense of the model's performance range over the training period.

These tables serve as a concise yet informative summary of our findings, supporting a clear comparison across different model configurations and training strategies. Through this analysis, we aim to contribute valuable insights into the optimisation of MLPs for diverse applications and datasets.

Basic Back Propagation Training and Network Selection

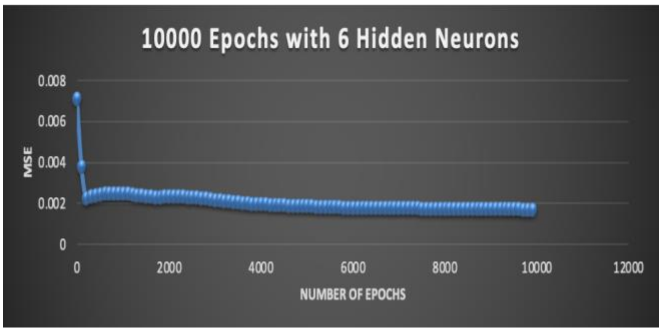


Figure 17: Model 1.1.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.1.1	9900	6	0.001713745	0.007107145

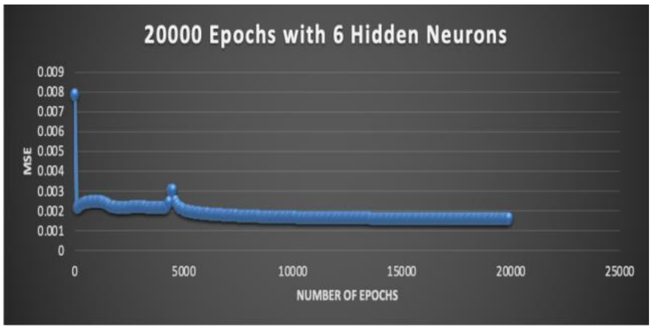


Figure 18: Model 1.1.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.1.2	19900	6	0.001602579	0.007855668

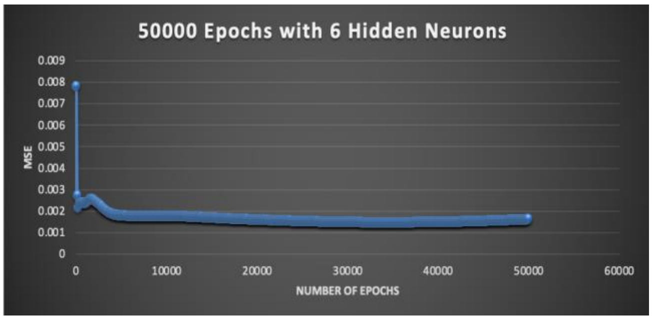


Figure 19: Model 1.1.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.1.3	34300	6	0.00150634	0.007833026

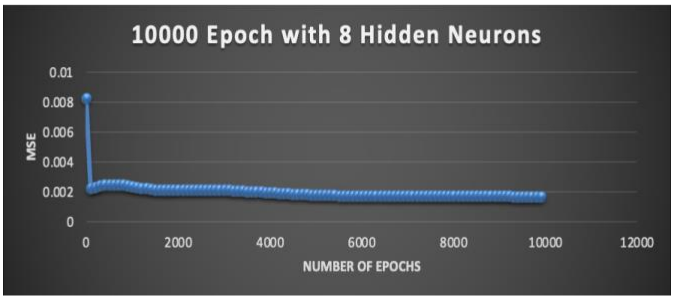


Figure 20: Model 1.2.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.2.1	9900	8	0.001693965	0.008243871

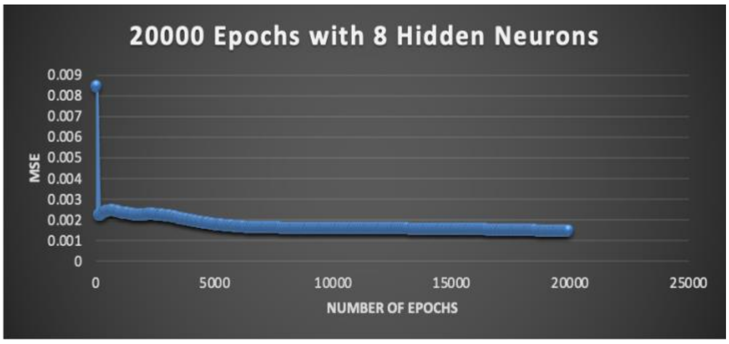


Figure 21: Model 1.2.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.2.2	19900	8	0.001476558	0.008495455

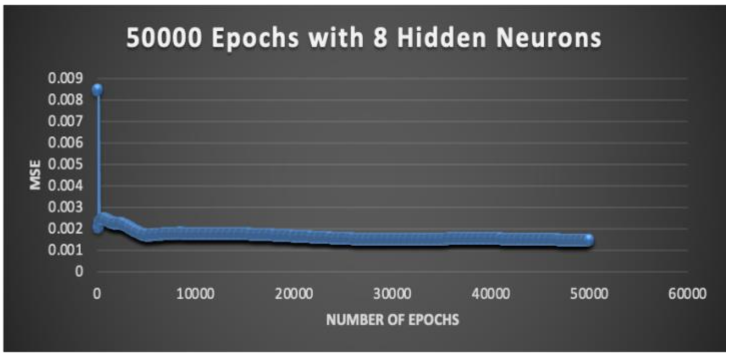


Figure 22: Model 1.2.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.2.3	49900	8	0.001466679	0.008479477

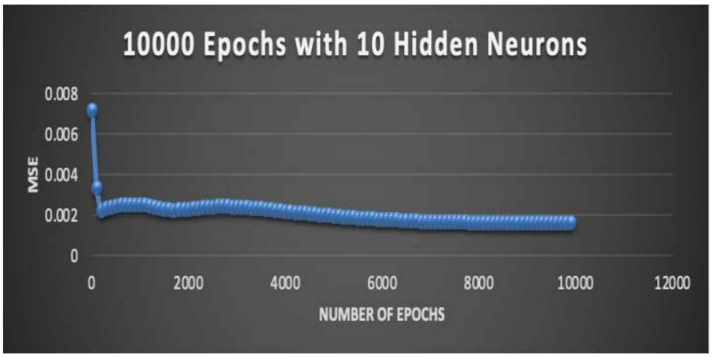


Figure 23: Model 1.3.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.3.1	9900	10	0.001642757	0.007161811

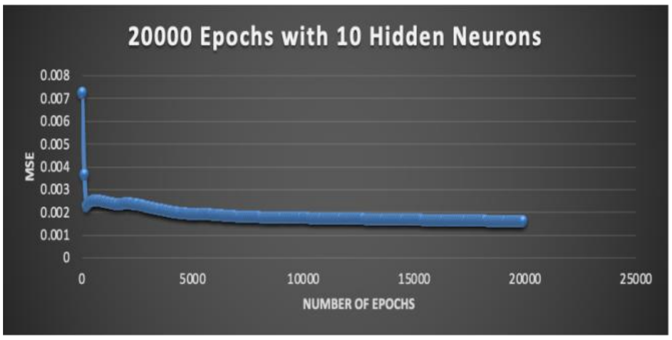


Figure 24: Model 1.3.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.3.2	19900	10	0.001616787	0.007256961

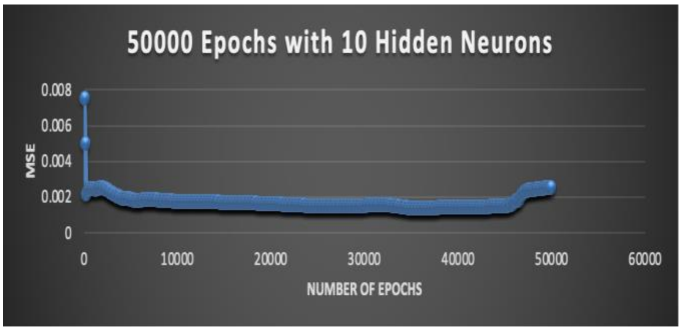


Figure 25: Model 1.3.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.3.3	35600	10	0.001397205	0.007534174

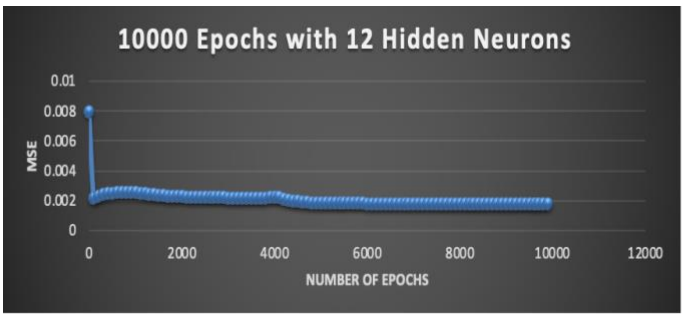


Figure 26: Model 1.4.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.4.1	9900	12	0.001777619	0.007963056

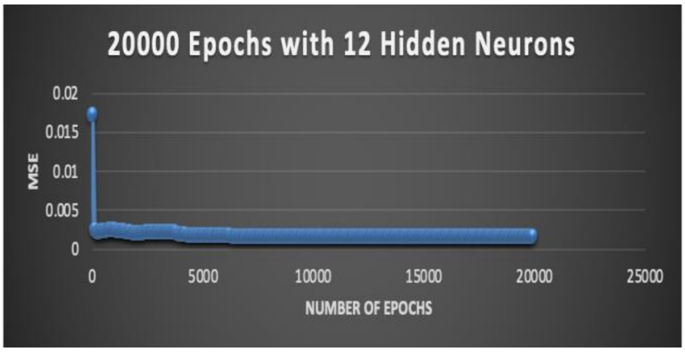


Figure 27: Model 1.4.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.4.2	19900	12	0.001651405	0.017319189

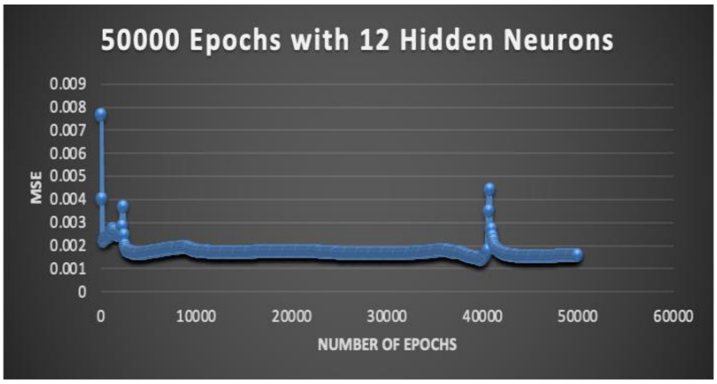


Figure 28: Model 1.4.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.4.3	39800	12	0.001429522	0.007674836

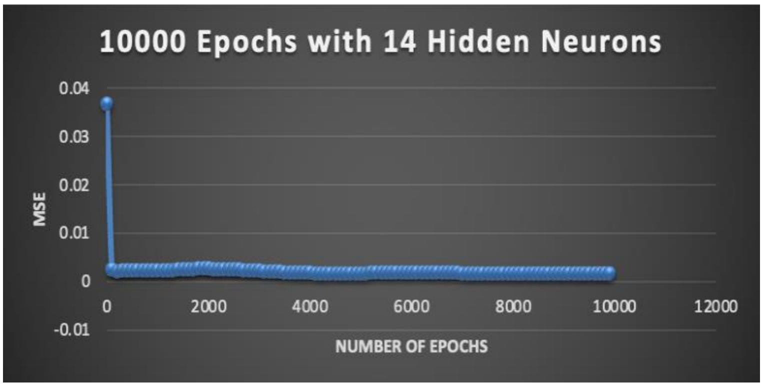


Figure 29: Model 1.5.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.5.1	8800	14	0.001770308	0.036871148

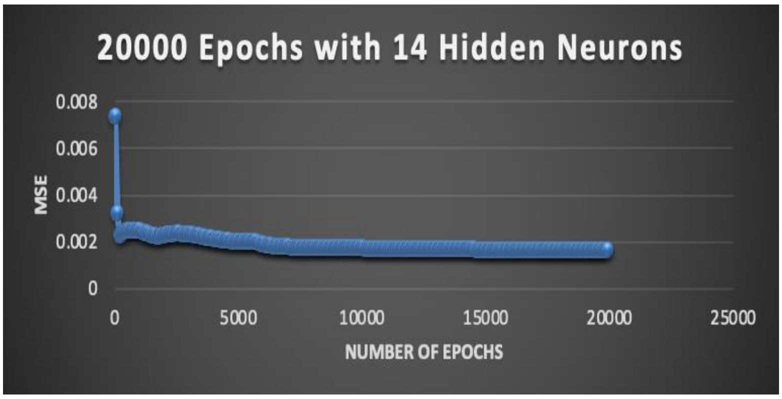


Figure 30: Model 1.5.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.5.2	19900	14	0.001638372	0.007403091

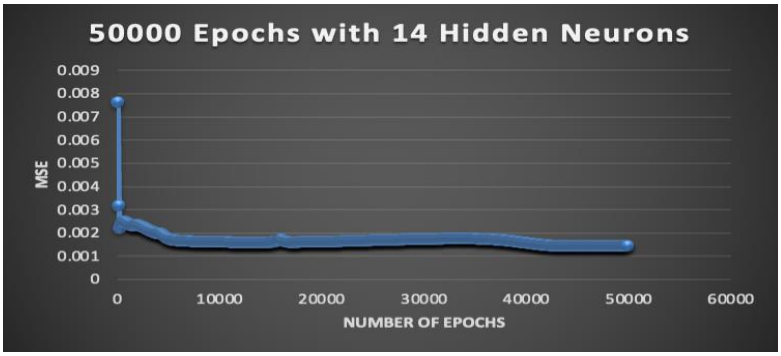


Figure 31: Model 1.5.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
1.5.3	45100	14	0.001432961	0.007649626

Backpropagation with Momentum, Bold Driver, and Weight Decay Training and Network Selection

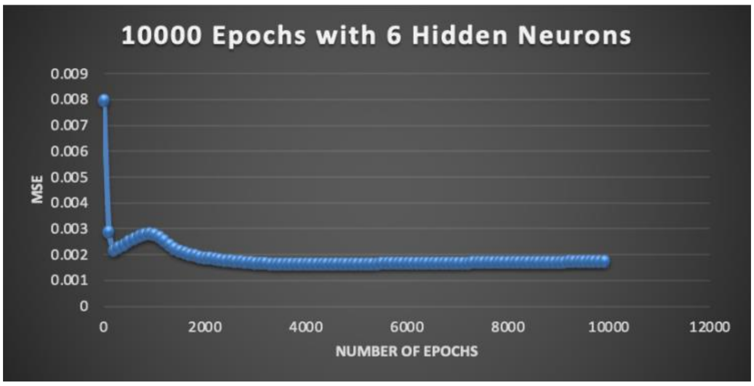


Figure 32: Model 2.1.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.1.1	4100	6	0.001630545	0.0080

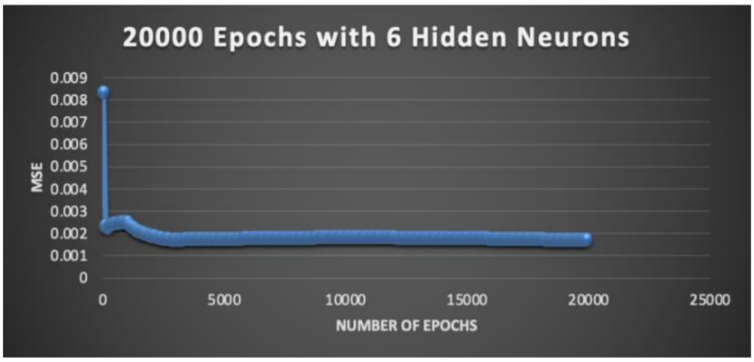


Figure 33: Model 2.1.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.1.2	19900	6	0.001710186	0.0083

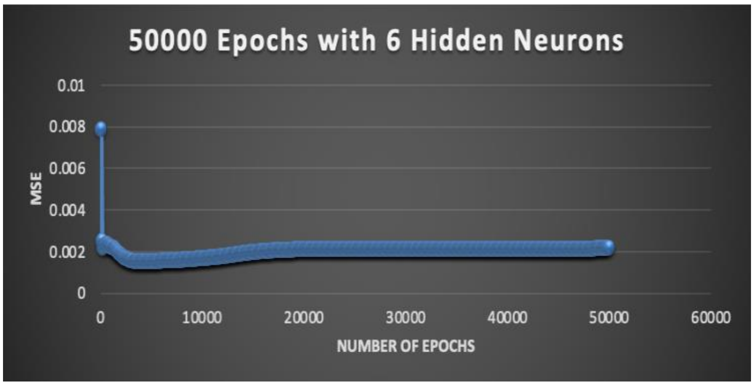


Figure 34: Model 2.1.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.1.3	4000	6	0.00155906	0.0079

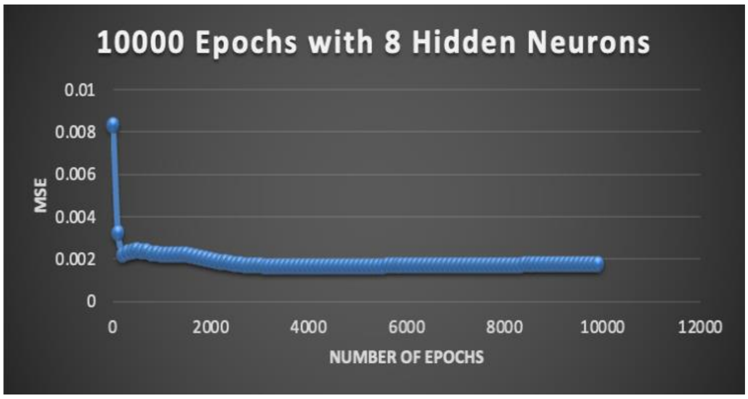


Figure 35: Model 2.2.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.2.1	4000	8	0.001644684	0.0083

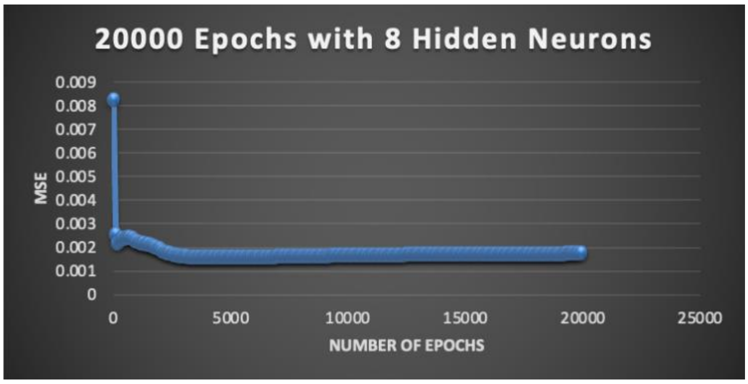


Figure 36: Model 2.2.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.2.2	4100	8	0.001599487	0.0082

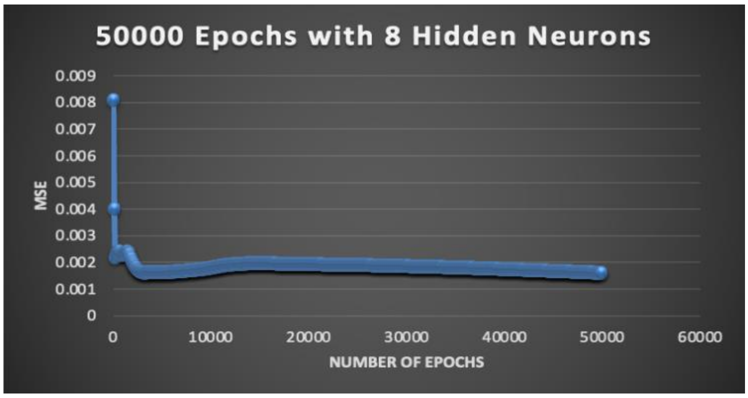


Figure 37: Model 2.2.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.2.3	49900	8	0.001614598	0.0081

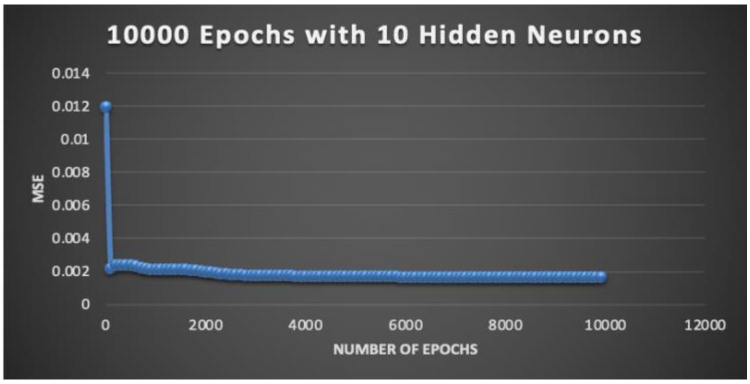


Figure 38: Model 2.3.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.3.1	9900	10	0.001651451	0.0012

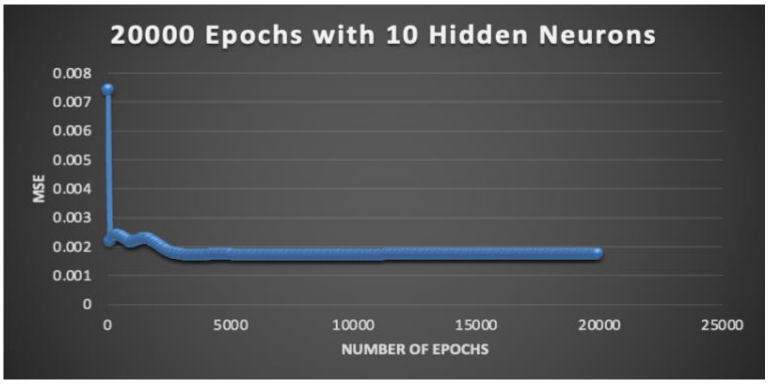


Figure 39: Model 2.3.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.3.2	7100	10	0.001724006	0.0074

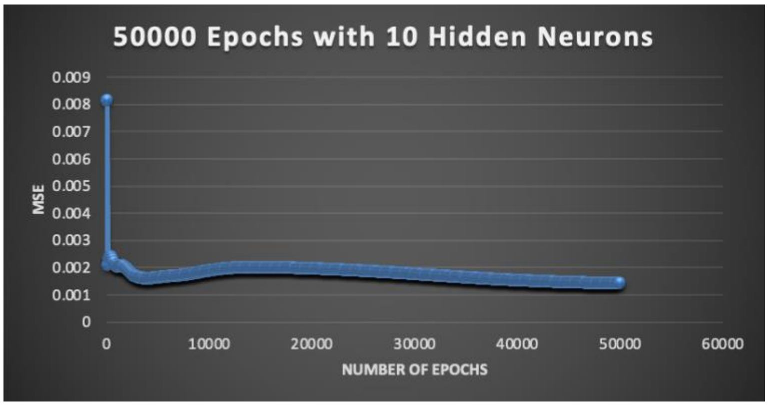


Figure 40: Model 2.3.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.3.3	49900	10	0.001396205	0.0081

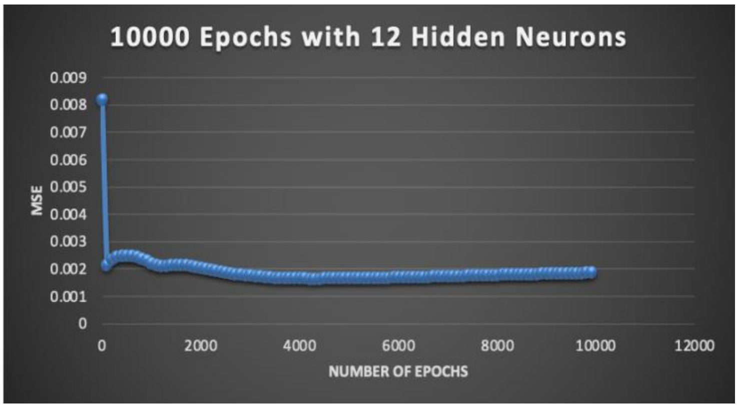


Figure 41: Model 2.4.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.4.1	4300	12	0.001658848	0.0082

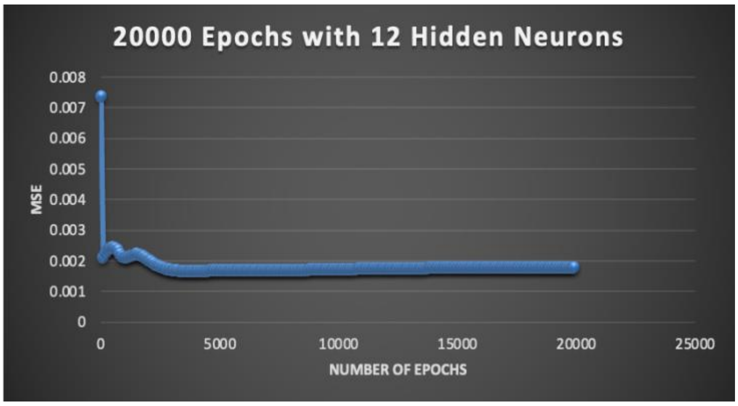


Figure 42: Model 2.4.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.4.2	3800	12	0.001673207	0.0074

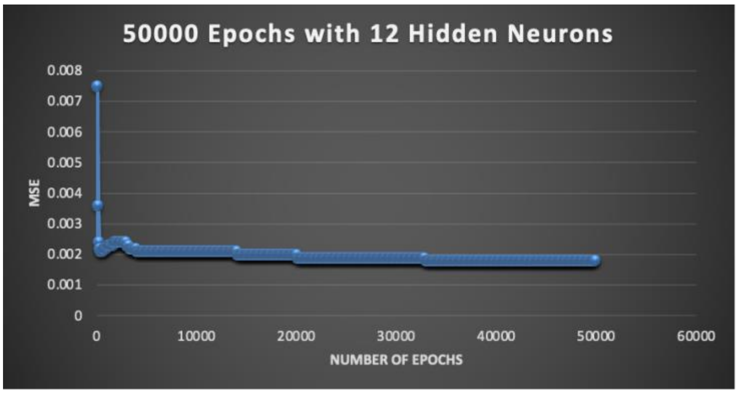


Figure 43: Model 2.4.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.4.3	49900	12	0.001533295	0.69

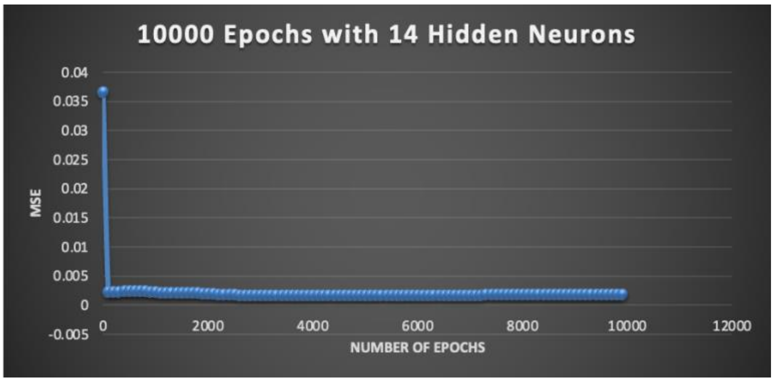


Figure 44: Model 2.5.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.5.1	3300	14	0.001673282	0.037

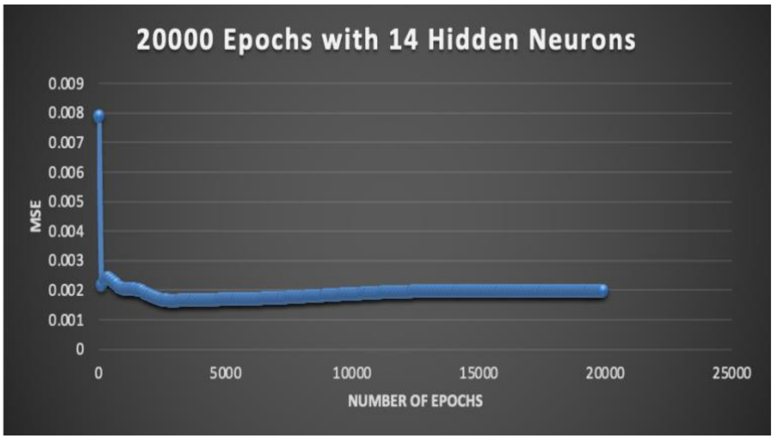


Figure 45: Model 2.5.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.5.2	2900	14	0.00165688	0.0079

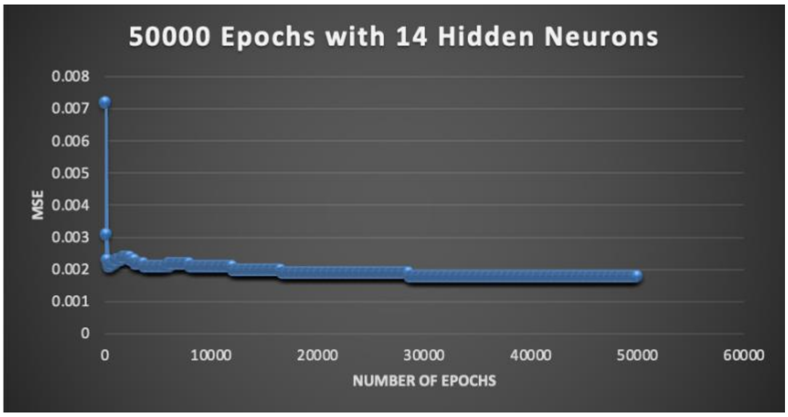


Figure 46: Model 2.5.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 2.5.3	43400	14	0.001180324	0.69

Backpropagation with Momentum, Annealing, and Weight Decay Training and Network Selection

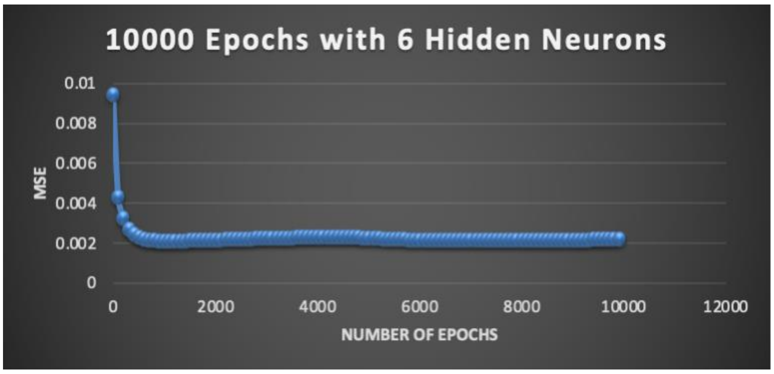


Figure 47: Model 3.1.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.1.1	1100	6	0.002108268	0.009426704

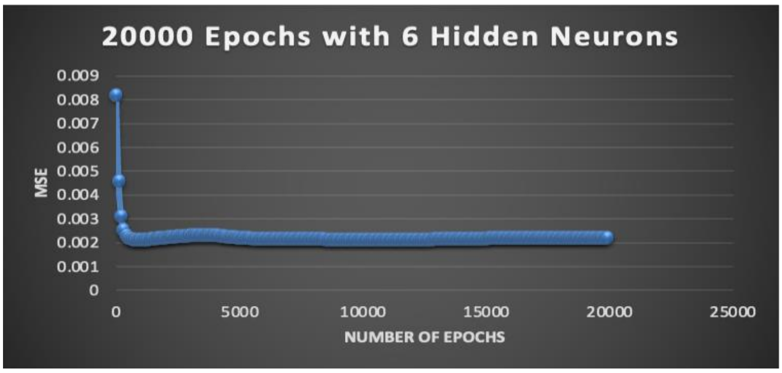


Figure 48: Mode. 3.1.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.1.2	900	6	0.002118888	0.008177954

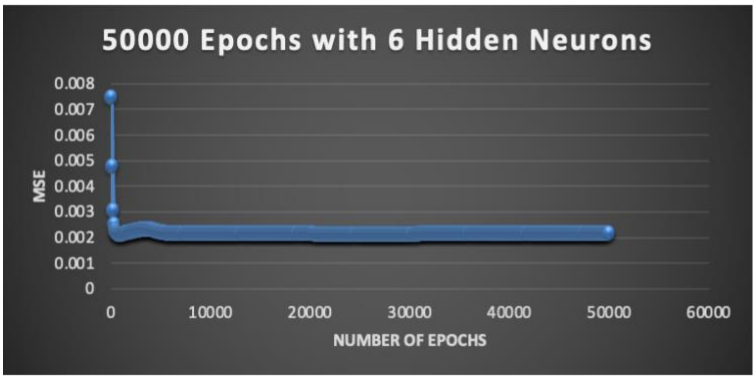


Figure 49: Model 3.1.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.1.3	800	6	0.00213185	0.007503333

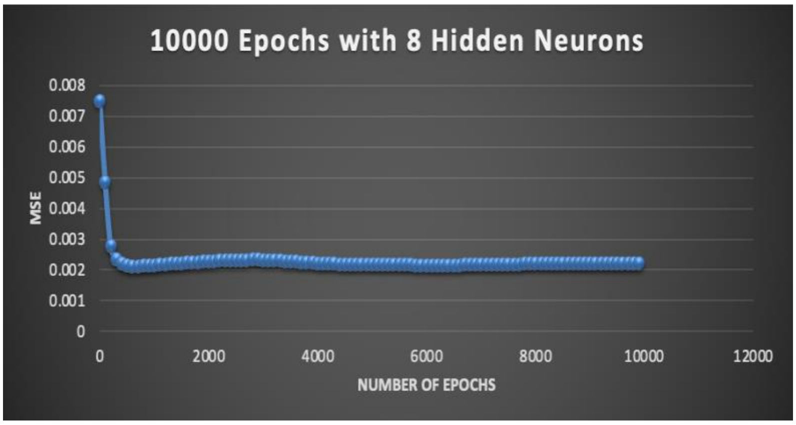


Figure 50: Mode. 3.2.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.2.1	700	8	0.002126413	0.007507919

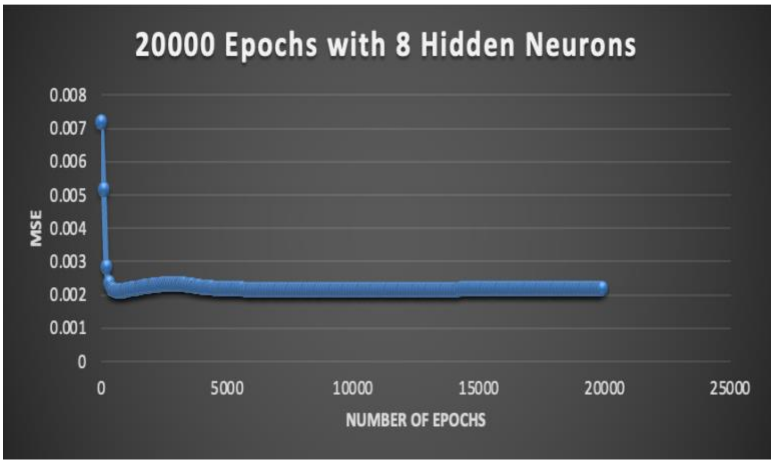


Figure 51: Model 3.2.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.2.2	700	8	0.002119492	0.007176506

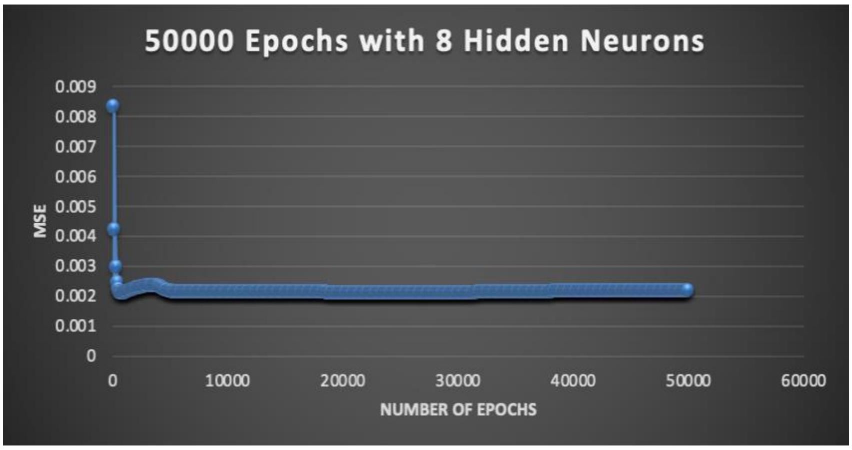


Figure 52: Model 3.2.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.2.3	700	8	0.002113376	0.008365742

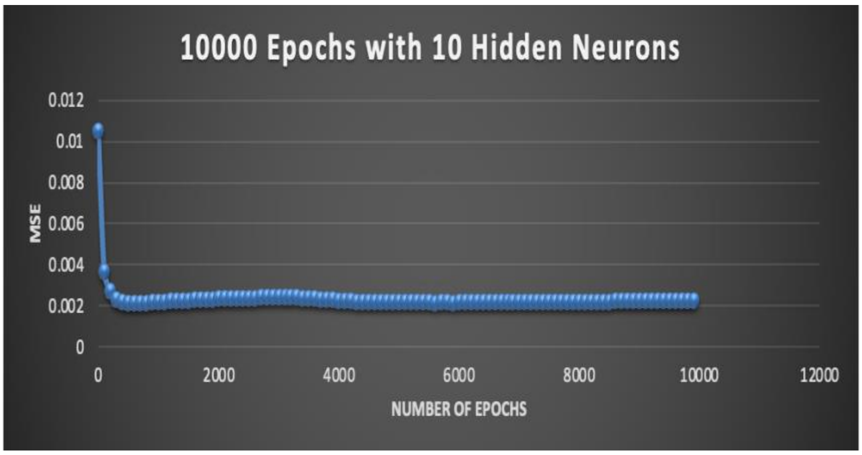


Figure 53: Model 3.3.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.3.1	600	10	0.002122705	0.010512065

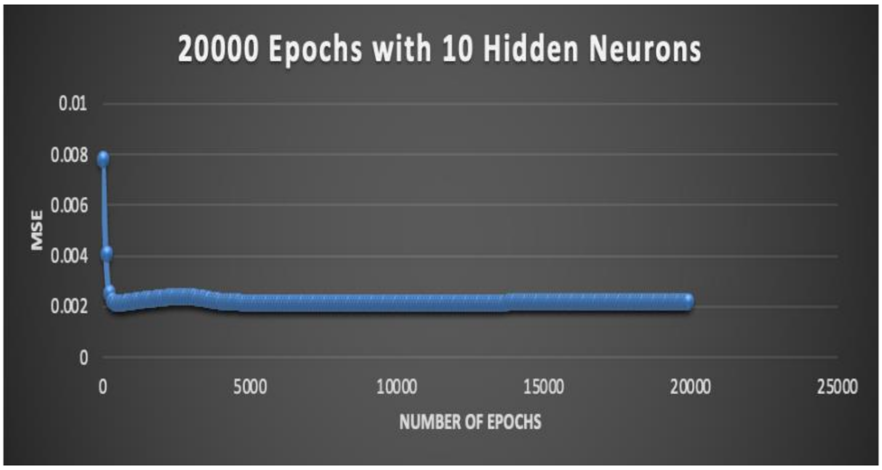


Figure 54: Model 3.3.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.3.2	500	10	0.002129476	0.007807334

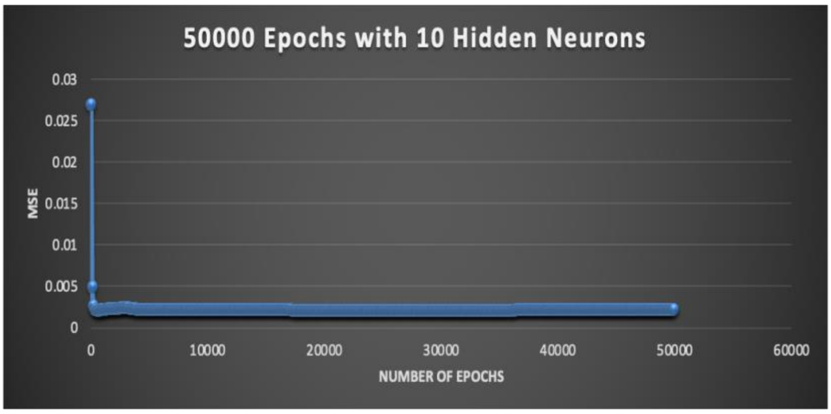


Figure 55: Model 3.3.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.3.3	28400	10	0.002095472	0.02694126

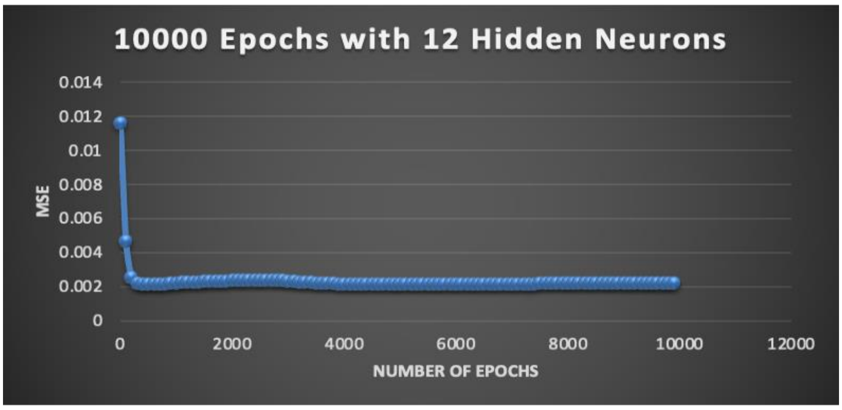


Figure 56: Model 3.4.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.4.1	500	12	0.002125352	0.011610569

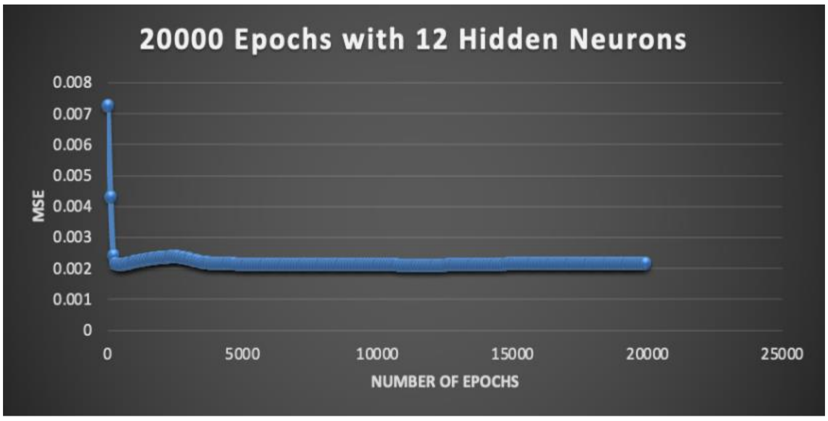


Figure 57: Model 3.4.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.4.2	11400	12	0.00210817	0.00726238

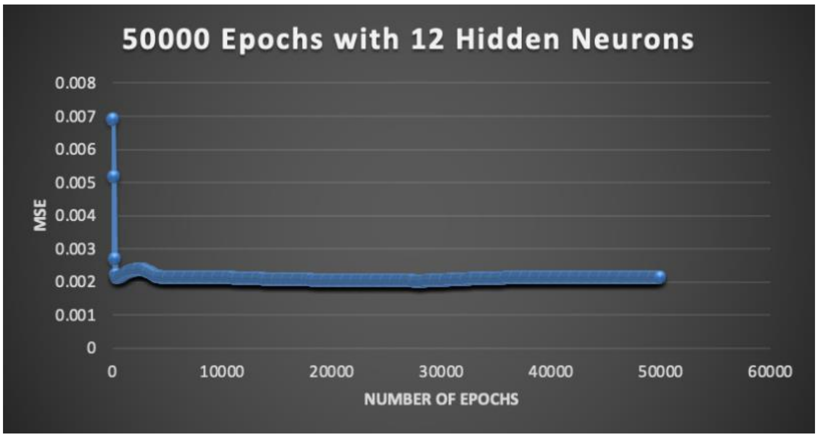


Figure 58: Model 3.4.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.4.3	28400	12	0.002031565	0.006922088

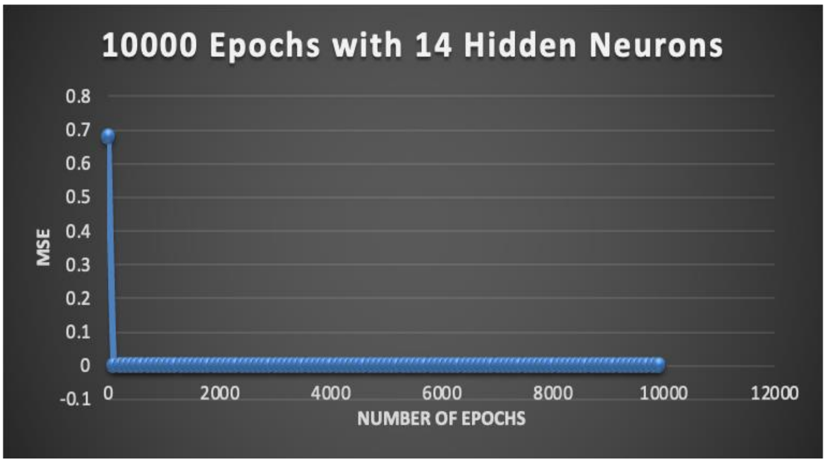


Figure 59: Model 3.5.1

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.5.1	500	14	0.00211476	0.681935639

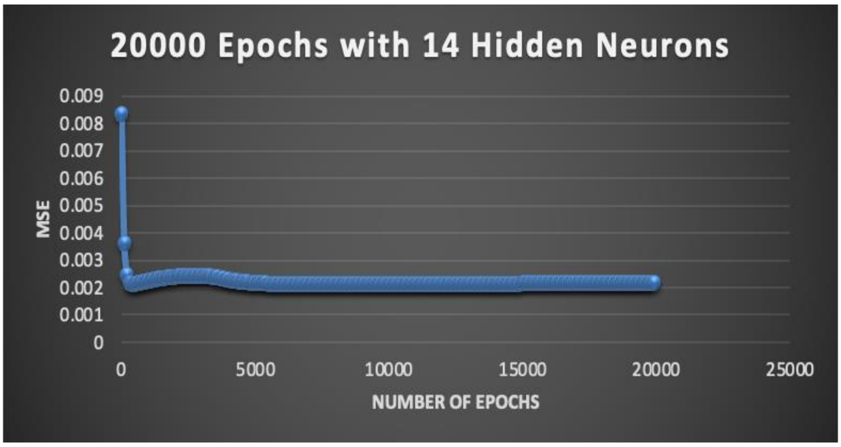


Figure 60: Model 3.5.2

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.5.2	11400	14	0.002117946	0.0083505

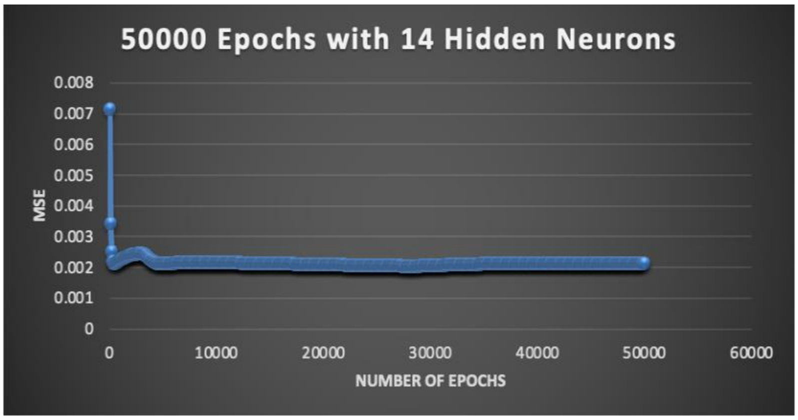


Figure 61: Model 3.5.3

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
Model 3.5.3	28400	14	0.002047309	0.007165889

Backpropagation with Momentum and Bold Driver- No graphs included to maintain simplicity for the reader.

Table 3: Tabulating Epoch and Min MSE for the model of Back Propagation with Momentum + Bold Driver

Model Name	Total Epochs	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
4.1.1	10000	4100	6	0.001630545	0.007971138
4.1.2	20000	19900	6	0.001710186	0.008334156
4.1.3	50000	4000	6	0.00155906	0.007874261
4.2.1	10000	4000	8	0.001644684	0.008342775
4.2.2	20000	4100	8	0.001599487	0.008237243
4.2.3	50000	49900	8	0.001614598	0.008085914
4.3.1	10000	9900	10	0.001651451	0.011931769
4.3.2	20000	7100	10	0.001724006	0.007423611
4.3.3	50000	49900	10	0.001432053	0.008143554
4.4.1	10000	4300	12	0.001658848	0.008190071
4.4.2	20000	3800	12	0.001673207	0.007369986

4.4.3	50000	49900	12	0.001533295	0.687714841
4.5.1	10000	3300	14	0.001673282	0.036601637
4.5.2	20000	2900	14	0.00165688	0.007903117
4.5.3	50000	43400	14	0.001180324	0.685579708

Backpropagation with Momentum and Annealing- No graphs included to maintain simplicity for the reader

Table 4: Tabulating Epoch and Min MSE for the model of Back Propagation with Momentum + Annealing

Model Name	Total Epochs	Epoch at MIN MSE	Hidden Neurons	MIN MSE	MAX MSE
4.1.1	10000	9900	6	0.001856328	0.007752139
4.1.2	20000	8300	6	0.001780368	0.008603112
4.1.3	50000	10100	6	0.001646897	0.008027619
4.2.1	10000	9900	8	0.001891264	0.025178084
4.2.2	20000	8100	8	0.001853464	0.009399924
4.2.3	50000	8500	8	0.00177572	0.008040738
4.3.1	10000	9900	10	0.001977161	0.035483179
4.3.2	20000	8700	10	0.001696442	0.008309246
4.3.3	50000	12200	10	0.001641034	0.008261501
4.4.1	10000	9900	12	0.002060221	0.028539949
4.4.2	20000	8200	12	0.001805364	0.008359321
4.4.3	50000	3500	12	0.001718258	0.008161959
4.5.1	10000	9900	14	0.001950844	0.031167391
4.5.2	20000	8700	14	0.001669405	0.007923081
4.5.3	50000	16300	14	0.001776393	0.036676646

Evaluation of Final Model

In the culmination of our comprehensive analysis of Multi-Layer Perceptron's (MLPs) trained with various configurations, we have distilled our findings to concentrate on three pivotal models. This focus allows us to delve deeper into the effects of specific training enhancements on model performance, specifically examining their impact on reducing mean squared error (MSE) on a validation set. The models selected for this final evaluation include:

1. Basic Backpropagation
2. Backpropagation with Momentum, Weight Decay, and Annealing
3. Backpropagation with Momentum, Weight Decay, and Bold Driver

Basic Backpropagation

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE
1.1.1	9900	6	0.001713745
1.1.2	19900	6	0.001602579
1.1.3	34300	6	0.00150634
1.2.1	9900	8	0.001693965

1.2.2	19900	8	0.001476558
1.2.3	49900	8	0.001466679
1.3.1	9900	10	0.001642757
1.3.2	19900	10	0.001616787
1.3.3	35500	10	0.001397205
1.4.1	9900	12	0.001777619
1.4.2	19900	12	0.001651405
1.4.3	39800	12	0.001429522
1.5.1	8800	14	0.001770308
1.5.2	19900	14	0.001638372
1.5.3	45100	14	0.001432961

Backpropagation with Momentum, Weight Decay, and Bold Driver

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE
Model 2.1.1	4100	6	0.001630545
Model 2.1.2	19900	6	0.001710186
Model 2.1.3	4000	6	0.00155906
Model 2.2.1	4000	8	0.001644684
Model 2.2.2	4100	8	0.001599487
Model 2.2.3	49900	8	0.001614598
Model 2.3.1	9900	10	0.001651451
Model 2.3.2	7100	10	0.001724006
Model 2.3.3	49900	10	0.001397203
Model 2.4.1	4300	12	0.001658848
Model 2.4.2	3800	12	0.001673207
Model 2.4.3	49900	12	0.001533295
Model 2.5.1	3300	14	0.001673282
Model 2.5.2	2900	14	0.00165688
Model 2.5.3	43400	14	0.001180324

Backpropagation with Momentum, Weight Decay, and Annealing

Model Name	Epoch at MIN MSE	Hidden Neurons	MIN MSE
Model 3.1.1	1100	6	0.002108268
Model 3.1.2	900	6	0.002118888
Model 3.1.3	800	6	0.00213185
Model 3.2.1	700	8	0.002126413
Model 3.2.2	700	8	0.002119492
Model 3.2.3	700	8	0.002113376
Model 3.3.1	600	10	0.002122705
Model 3.3.2	500	10	0.002129476
Model 3.3.3	28400	10	0.002095472
Model 3.4.1	500	12	0.002125352
Model 3.4.2	11400	12	0.00210817
Model 3.4.3	28400	12	0.002031565
Model 3.5.1	500	14	0.00211476
Model 3.5.2	11400	14	0.002117946
Model 3.5.3	28400	14	0.002047309

I have highlighted the Minimum MSE from my three models above.

After extensive experimentation and evaluation, the best-performing model was identified as version 2.3.3, which utilizes a combination of bold driver optimization, weight decay regularization, and momentum in conjunction with basic backpropagation. This model achieved a minimum mean squared error (MSE) of 0.001397203 at epoch 49900, with an architecture consisting of 10 hidden nodes. The low MSE indicates a high level of prediction accuracy, making this model particularly effective for our specific application. The convergence at a relatively high number of epochs suggests that the model benefits from extended training, potentially due to the effective synergy between the applied optimization and regularization techniques.

Having identified the model version 2.3.3 as our best-performing configuration based on its minimum mean squared error (MSE) during the validation phase, we are now poised to further validate its predictive capabilities on an entirely unseen dataset—the test set. This crucial step is aimed at assessing the model's generalization ability, a vital aspect of machine learning models that determines their applicability in real-world scenarios.

The test set, which was meticulously segregated from the training and validation datasets at the outset of our experiment, will provide the grounds for this evaluation. By comparing the model's predictions against the actual values within this dataset, we will gain insights into how well the model has learned to generalize from the patterns observed during training to make accurate predictions on new, unseen data.

To visually depict the accuracy of our model's predictions against the actual values in the test set, we have prepared a graph that plots both sets of values. This graphical representation will allow us to easily observe the degree of alignment between the predicted and actual values, offering a clear and intuitive measure of the model's performance on the test set.

A pivotal aspect of this evaluation is illustrated through a scatter graph, contrasting the model's predictions against actual values within the test set. Remarkably, this visual analysis yielded an R-squared value of 0.9829, attesting to an exceptionally high degree of correlation between predicted and actual values. Such a high R-squared value not only confirms the model's acute predictive accuracy but also visually demonstrates its capability to closely mirror real-world outcomes, further evidenced by the graph's tight clustering around the line of unity.

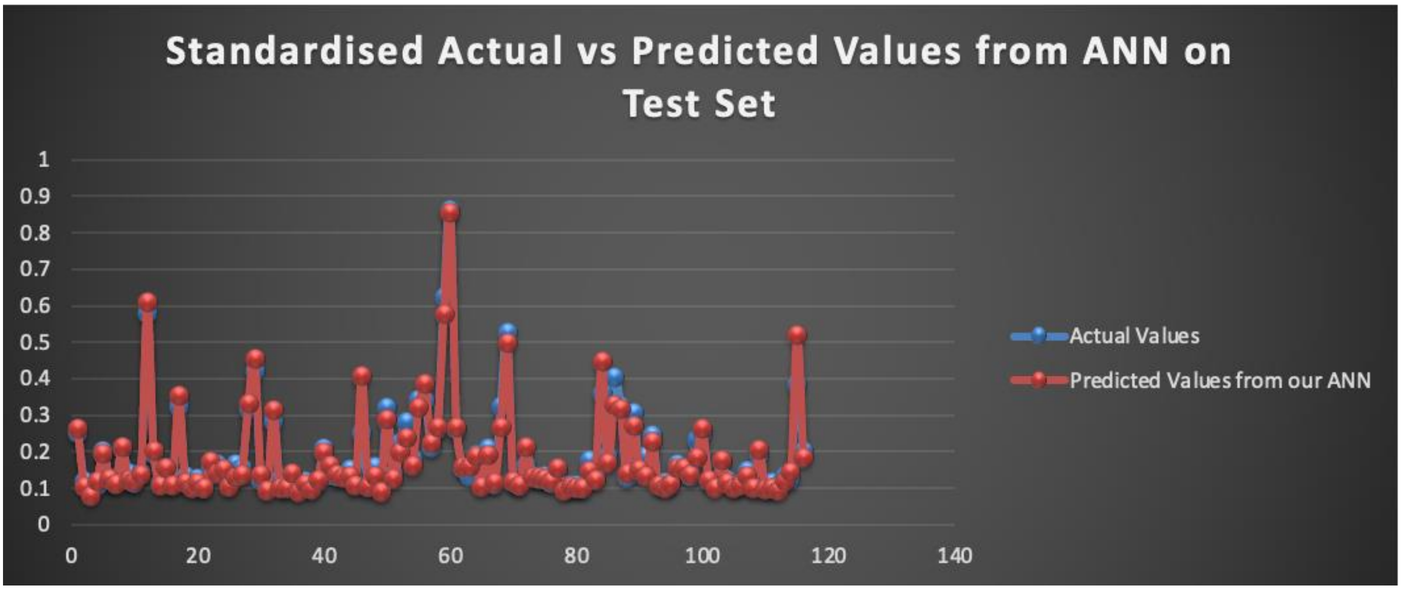


Figure 62: Standardised Actual vs Predicted values

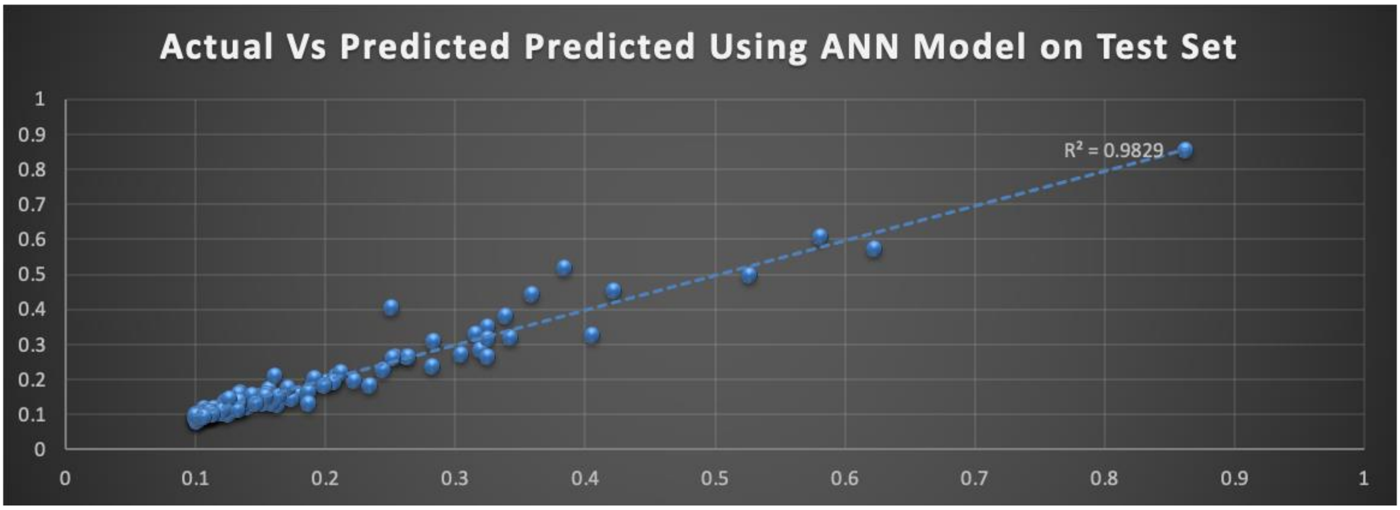


Figure 63: Standardised Actual vs Predicted values using a scatter graph

The closer the predicted values are to the actual values, as illustrated in the graph, the better the model is at making accurate predictions on unseen data. This comparison not only serves as a testament to the model's predictive accuracy but also highlights the effectiveness of the chosen features, model architecture, and training regimen in capturing the underlying patterns of the dataset.

In our evaluation, a critical component of validating our model's effectiveness involves not only analysing the performance metrics such as the mean squared error (MSE) on standardized data but also ensuring that our model's predictions maintain high accuracy when transformed back to the original scale of the data, known as un standardisation.

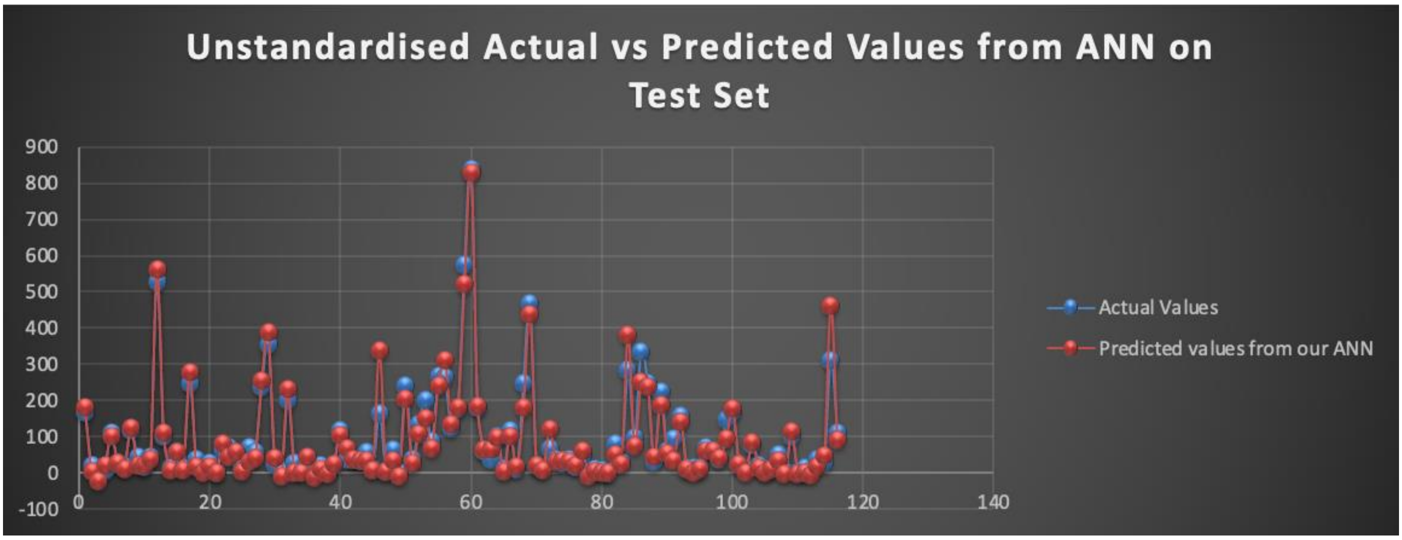


Figure 64: Unstandardised Actual vs Predicted values

The accompanying graph presents a comparison of our model's predictions against the actual values within the test set, both expressed in their original scale to help observe the true reality. Observing a close alignment between these unstandardised predicted and actual/ modelled values not only underscores the model's ability to make accurate predictions but also reinforces its utility in practical scenarios where decisions rely on the original data scale.

Through this comprehensive evaluation—spanning standardised comparisons for model optimization and unstandardized comparisons for practical applicability—we demonstrate version 1.3.3's robustness and readiness to address real-world challenges. The detailed approach, from data pre-processing to final performance validation, ensures that our model is not only statistically effective but also pragmatically valuable.

Comparison with Other Models

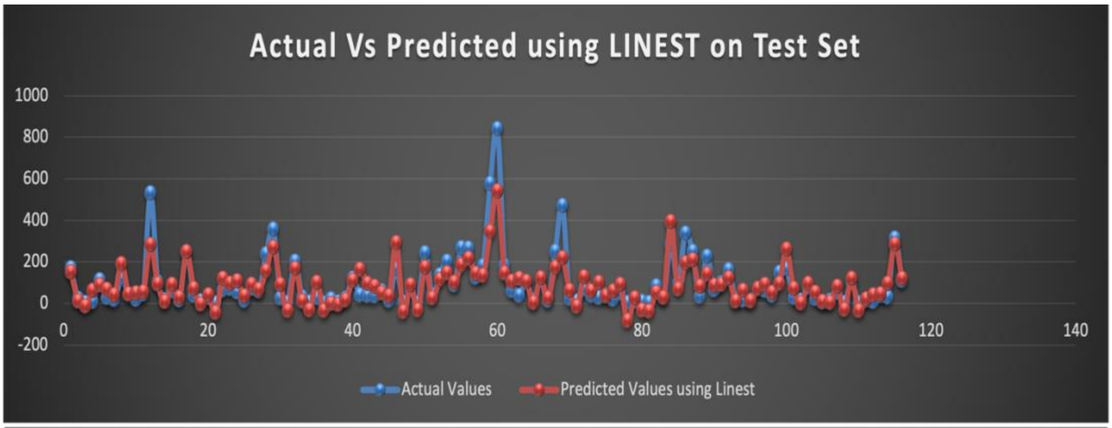
Method:
We conducted a regression analysis using Excel's LINEST function, drawing from our training and validation cleaned datasets to establish a predictive model. This model was then applied to our test dataset to evaluate its predictive capability. To visually assess the model's performance, a graph plotting predicted versus actual values was generated on the Test Set, highlighting the model's accuracy in predicting unseen data.

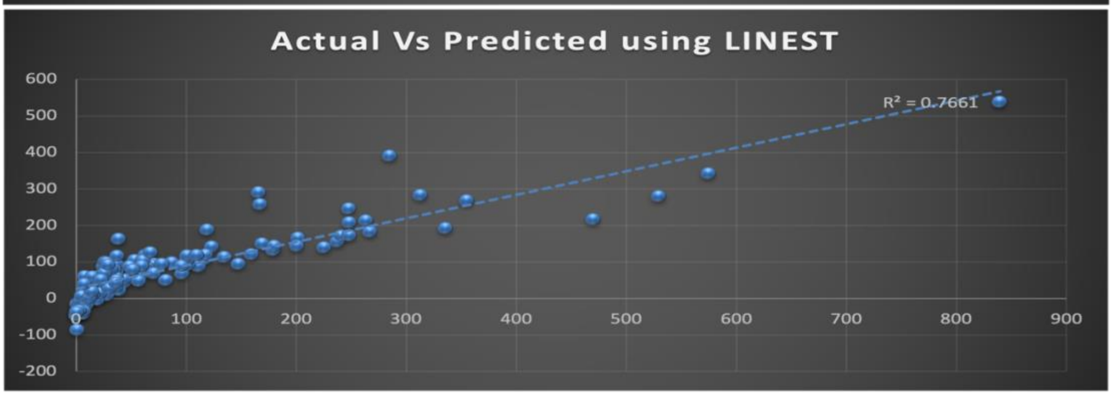
Regression Coefficients:

SARR	RMED-1D	PROPWET	LDP	FPEXT	FARL	BFI HOST	AREA	Y-INTERCEPT
-0.0282579	2.01857881	244.968601	1.11922531	-199.30618	278.728031	-82.542071	0.12209849	-375.87029
0.02653939	0.99435196	43.205279	0.25265763	86.0895399	62.6066324	24.0355113	0.01513001	73.1584527
0.685976	66.2107355							
124.241729	455							
4357268.27	1994656.98							

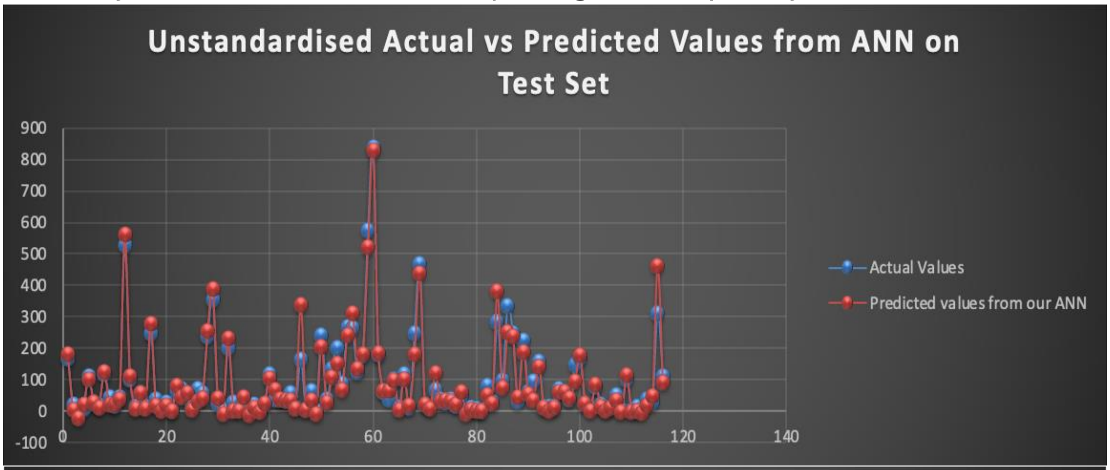
Model Evaluation:
The comparison of predicted versus actual values indicated a moderate alignment, suggesting a moderate level of predictive accuracy. The R^2 value, which quantifies the model's explanatory power, was found to be 0.7661, indicating that 76.6% of the variance in the dependent variable is accounted for by the model. This level of performance underscores the model's capability but also points towards potential areas for refinement.

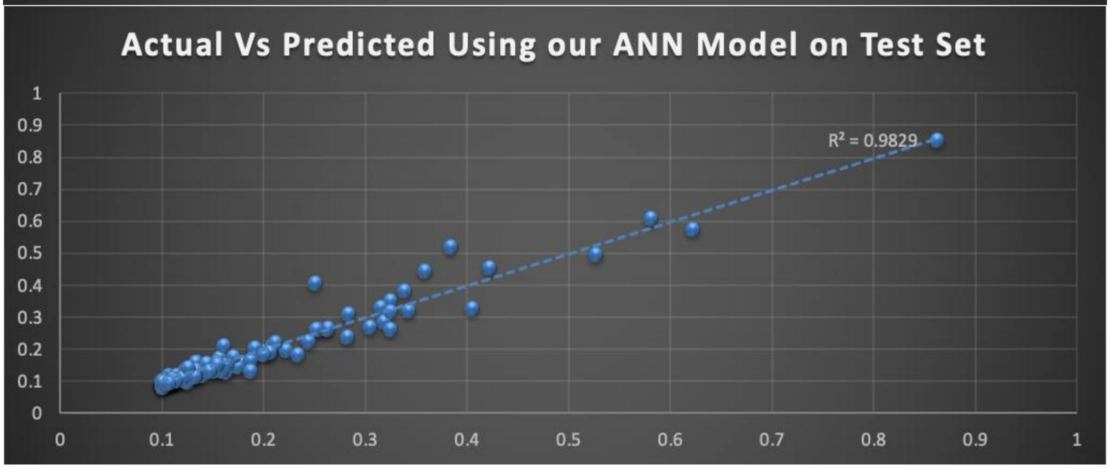
To visually assess the model's performance, a graph plotting predicted versus actual values was generated (see Figures below). This visual representation highlights the model's accuracy in predicting unseen data, illustrating a moderate level of predictive accuracy through the alignment of predicted versus actual values.





Similarly, the ANN model's superior predictive capability is demonstrated through a graph comparing predicted to actual values (see Figures below). This graph shows a tighter clustering of data points around the line of best fit, visually indicating the model's higher R^2 value of 0.9829 giving us 98% of accuracy and its effectiveness in capturing the complex dynamics of INDEX FLOOD prediction.





Predictor Influence:

$$Ratio = \frac{Coefficient\ of\ Each\ Predictor}{Standard\ Error\ of\ Each\ Coefficient}$$

Calculated Ratios:

SARR	-1.064752178
RMED-1D	2.030044573
PROPWET	5.669876627
LDP	4.429810016
FPEXT	-2.315103315
FARL	4.452052774
BFI HOST	-3.434171608
AREA	8.069952555

The regression analysis revealed that AREA, with a ratio of 8.07, exhibits the strongest positive influence on the dependent variable among the predictors analysed, suggesting that it is a critical factor in predicting INDEX FLOOD. Similarly, PROPWET and FARL also demonstrate strong positive influences, with ratios of 5.67 and 4.45, respectively. On the other hand, BFI HOST presents a notable negative impact, with a ratio of -3.43, indicating its importance in decreasing the value of the dependent variable. These findings underscore the diverse roles of different environmental and operational factors in influencing the catchment analysis.

Conclusion:
The ANN model, with its impressive R^2 value of 0.9457, markedly outperforms the linear regression model in predicting INDEX FLOOD, reflecting its advanced capability to handle the intricate dynamics of the dataset. This analysis not only showcases the potential of ANN models in hydrological forecasting but also prompts a re-evaluation of modelling strategies to embrace more sophisticated, data-driven approaches. Moving forward, the focus will be on optimizing the ANN architecture, exploring the integration of additional predictive factors, and possibly combining various modelling techniques to create a robust, versatile forecasting tool.

The demonstrated predictive superiority of the ANN model guides our future direction in model selection, especially for projects demanding high accuracy in complex scenarios. However, the linear regression model's advantages—interpretability and lower computational demand—will render it the model of choice for preliminary analyses or when simplicity and transparency are paramount.

Our findings underscore the necessity of balancing model complexity, interpretability, and predictive power. While ANN models offer unmatched accuracy, their complexity and opaque nature pose challenges. Conversely, linear regression models, with their straightforwardness and ease of interpretation, provide valuable insights despite lower predictive power. Future methodologies will thus aim to strike a balance, leveraging each model's strengths according to the specific requirements of the task at hand.

Appendix A: Basic Back Propagation

- I have highlighted my back propagation implementation.
- I have also commented my other enhancements for easier capture.
- I have only commented code which is not repeated.

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {
    // Constants for paths to the CSV files containing training, validation, and test datasets.
    private static final String TRAINING_SET_PATH = "trainingSet1.csv";
    private static final String VALIDATION_SET_PATH = "validationSet.csv";
    private static final String TEST_SET_PATH = "testSet.csv";

    // The main method is the entry point of the application.
    public static void main(String[] args) {
        // Initialize a Scanner object for reading input from the console.
        Scanner scanner= new Scanner(System.in);
        // Prompt the user to enter the number of epochs for training and hidden neurons
        System.out.println("Enter the number of epochs:");
        int epochs= scanner.nextInt();
        System.out.println("Enter the number of hidden neurons:");
        int numHiddenNeurons= scanner.nextInt();
        // Load the datasets from their respective CSV files.
        List<double[]> trainingData = loadData(TRAINING_SET_PATH);
        List<double[]> validationData = loadData(VALIDATION_SET_PATH);
        List<double[]> testData = loadData(TEST_SET_PATH);

        // Create an instance of the CatchmentMLP class with the specified number of hidden
neurons.
```

```java
        CatchmentMLP mlp = new CatchmentMLP(numHiddenNeurons);
        // Train the MLP model using the training data and validate it using the validation data 
for the given number of epochs.
        mlp.train(trainingData, epochs, validationData);
        // After training, output the actual versus predicted outputs for the test dataset.
        System.out.println("\nActual vs Predicted Outputs (Test Set):");
        for (double[] dataRow : testData) {
            double[] inputs = new double[CatchmentMLP.getNumInputs()];
            System.arraycopy(dataRow, 0, inputs, 0, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];
            double predictedOutput = mlp.forwardPass(inputs);
            System.out.println("Actual:         " + actualOutput + "              Predicted: 
" + predictedOutput);
        }
    }
    // Method to remove UTF-8 Byte Order Mark (BOM) from a string.
    private static String removeUtf8Bom(String s) {
        if (s.startsWith("\uFEFF")) {
            s = s.substring(1);
        }
        return s;
    }
    // Method to load data from a CSV file and convert it into a list of double arrays.
    private static List<double[]> loadData(String filePath) {
        List<double[]> data = new ArrayList<>();
        try (BufferedReader br = Files.newBufferedReader(Paths.get(filePath), 
StandardCharsets.UTF_8)) {
            String line;
            while ((line = br.readLine()) != null) {
                line = removeUtf8Bom(line);
                // Split the line into values using a comma as the delimiter.
                String[] values = line.split(",");
                double[] doubleValues = new double[values.length];
                for (int i = 0; i < values.length; i++) {
                    // Convert each value to double and store in the array.
                    doubleValues[i] = Double.parseDouble(values[i]);
                }
                // Add the array of doubles to the list representing the dataset.
                data.add(doubleValues);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return data;
    }
}

import java.util.ArrayList;
import java.util.Random;
import java.util.List;

// Define the CatchmentMLP class representing a simple multi-layer perceptron neural 
network.
public class CatchmentMLP {

    private double[][] inputToHiddenWeights; // Matrix to store weights from the input 
layer to the hidden layer.

    private double[] hiddenBiases; // Array to store biases for the hidden layer 
neurons.
    private double[] hiddenToOutputWeights; // Array to store weights from the hidden 
layer to the output layer.
    private double outputBias; // Bias for the output neuron.
    private static final int numInputs = 8; // Constant representing the number of 
input neurons/features.
    private Random rand = new Random(); // Random number generator for initializing 
weights and biases.
    private static final double learningRate = 0.1; // Learning rate for the network's 
training phase.

    private final int numHiddenNeurons; // The number of neurons in the hidden layer,
```

```
defined at runtime entered by the user

    // Constructor for the CatchmentMLP class.
    public CatchmentMLP(int numHiddenNeurons) {
        this.numHiddenNeurons = numHiddenNeurons; // Initialize the number of hidden
neurons based on user input.
        initializeWeightsAndBiases();// Call method to initialize weights and biases.
    }
    // Getter for number of inputs
    public static int getNumInputs() {
        return numInputs;
    }
    // Range for weights initialization: -2/n to 2/n, where n is the number of inputs
    private void initializeWeightsAndBiases() {
        double range = 4.0 / numInputs;
        // Initialize weights and biases for each input neuron
        inputToHiddenWeights = new double[numInputs][numHiddenNeurons];
        hiddenBiases = new double[numHiddenNeurons];         // Initialize the array
for hidden layer biases.
        hiddenToOutputWeights = new double[numHiddenNeurons];         // Initialize the
array for hidden to output layer weights.
        // Randomly initialize weights and biases within the calculated range.
        for (int i = 0; i < numInputs; i++) {
            for (int j = 0; j < numHiddenNeurons; j++) {
                inputToHiddenWeights[i][j] = range * (rand.nextDouble() - 0.5);
            }
        }
        for (int i = 0; i < numHiddenNeurons; i++) {
            hiddenBiases[i] = range * (rand.nextDouble() - 0.5);
            hiddenToOutputWeights[i] = range * (rand.nextDouble() - 0.5);
        }
        // Initialize the output bias.
        outputBias = (numHiddenNeurons / 2.0) * (2.0 * rand.nextDouble() - 1.0);
    }
    // Perform the forward pass through the network and return the output.
    public double forwardPass(double[] inputs) {
        double[] hiddenOutputs = new double[numHiddenNeurons]; // Array to store the
outputs from the hidden layer neurons.
        // Calculate the output for each hidden neuron using the inputs, weights, and
biases.
        for (int i = 0; i < numHiddenNeurons; i++) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; j++) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]); // Apply the sigmoid
activation function to the hidden layer output.
        }
        // Calculate the final output using the hidden layer outputs, weights, and
output bias.
        double output = outputBias;
        for (int i = 0; i < numHiddenNeurons; i++) {
            output += hiddenOutputs[i] * hiddenToOutputWeights[i];
        }
        // Apply the sigmoid activation function to the final output.
        return sigmoid(output);
    }
    // Sigmoid activation function.
    private double sigmoid(double x) {
        return 1.0 / (1.0 + Math.exp(-x));
    }
    // Derivative of the sigmoid function, used during backpropagation.
    private double sigmoidDerivative(double x) {
        return x * (1.0 - x);
    }
    // Train the neural network using the provided training and validation data.
```

```
    public void train(List<double[]> trainingData, int epochs, List<double[]>
validationData) {
        // Lists to hold training and validation mean squared error (MSE) values.
        List<String> trainingMSEList = new ArrayList<>();
        List<String> validationMSEList = new ArrayList<>();

        // Iterating over epochs, computing training error, validating error and
updating model
        for (int epoch = 0; epoch < epochs; epoch++) {
            // Variable to accumulate total training error.
            double totalTrainingError = 0;
            // This loop processes each row of training data to predict output,
calculate error, and update the model.
            // For each row, it extracts inputs, computes the predicted output using a
forward pass, determines the error, accumulates the squared error for the epoch,
            // and adjusts weights and biases through a backward pass.
            for (double[] row : trainingData) {
                double[] inputs = new double[numInputs];
                System.arraycopy(row, 0, inputs, 0, numInputs);
                double actualOutput = row[numInputs];
                double predictedOutput = forwardPass(inputs);
                double error = actualOutput - predictedOutput; // Store the difference
in values
                totalTrainingError += Math.pow(error, 2);
                backwardPass(inputs, predictedOutput, error);
            }
            // Every 100 epochs, calculate and log the MSE for training and validation.
            if (epoch % 100 == 0) {
                // Compute Validation MSE
                double totalValidationMSE = 0;
                for (double[] valRow : validationData) {
                    double[] valInputs = new double[numInputs];
                    System.arraycopy(valRow, 0, valInputs, 0, numInputs);
                    double actualValOutput = valRow[numInputs];
                    double predictedValOutput = forwardPass(valInputs);
                    double error = actualValOutput - predictedValOutput;
                    totalValidationMSE += error * error;
                }
                double validationMSE = totalValidationMSE / validationData.size();
                validationMSEList.add("Epoch:           " + epoch + "
Validation MSE:            " + validationMSE);

                // Compute Training MSE
                double totalTrainingMSE = 0;
                for (double[] trainRow : trainingData) {
                    double[] trainInputs = new double[numInputs];
                    System.arraycopy(trainRow, 0, trainInputs, 0, numInputs);
                    double actualTrainOutput = trainRow[numInputs];
                    double predictedTrainOutput = forwardPass(trainInputs);
                    double error = actualTrainOutput - predictedTrainOutput;
                    totalTrainingMSE += error * error;
                }
                // Training MSE calculation.
                double trainingMSE = totalTrainingMSE / trainingData.size();
                trainingMSEList.add("Epoch:           " + epoch + "           Training
MSE:           " + trainingMSE);
            }
        }
        // Print training MSE values.
        System.out.println("Training MSEs:");
        for (String mse : trainingMSEList) {
            System.out.println(mse);
        }
        // Print validation MSE list
        System.out.println("\nValidation MSEs:");
        for (String mse : validationMSEList) {
```

```
            System.out.println(mse);
        }
    }
    // Perform the backward pass for backpropagation and update weights and biases.
    private void backwardPass(double[] inputs, double predictedOutput, double error) {
        // Calculate the output layer's delta (error gradient).
        double deltaOutput = error * sigmoidDerivative(predictedOutput);
        // Array to store the hidden layer outputs for use in weight updates.
        double[] hiddenOutputs = new double[numHiddenNeurons];
        // Calculate each hidden neuron's output again (as done in forward pass).
        for (int i = 0; i < numHiddenNeurons; i++) {
            for (int j = 0; j < numInputs; j++) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }
        // Update weights and biases for hidden to output layer and input to hidden
layer.
        for (int i = 0; i < numHiddenNeurons; i++) {
            double deltaHidden = deltaOutput * hiddenToOutputWeights[i] *
sigmoidDerivative(hiddenOutputs[i]);
            for (int j = 0; j < numInputs; j++) {
                inputToHiddenWeights[j][i] += learningRate * deltaHidden * inputs[j];
            }
            hiddenBiases[i] += learningRate * deltaHidden;
        }
        for (int i = 0; i < numHiddenNeurons; i++) {
            hiddenToOutputWeights[i] += learningRate * deltaOutput * hiddenOutputs[i];
        }
        // Update the output bias.
        outputBias += learningRate * deltaOutput;
    }
}
```

Appendix B: Basic Back Propagation + Momentum

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;


public class Main {
    private static final String TRAINING_SET_PATH = "trainingSet1.csv";
    private static final String VALIDATION_SET_PATH = "validationSet.csv";
    private static final String TEST_SET_PATH = "testSet.csv";


    public static void main(String[] args) {
        Scanner scanner= new Scanner(System.in);
        System.out.println("Enter the number of epochs:");
        int epochs= scanner.nextInt(); // Read user input for number of epochs
        System.out.println("Enter the number of hidden neurons:");
        int numHiddenNeurons= scanner.nextInt(); // Read user input for number of
neurons

        // List to hold training data and validation data
        List<double[]> trainingData = loadData(TRAINING_SET_PATH);
        List<double[]> validationData = loadData(VALIDATION_SET_PATH);
        List<double[]> testData = loadData(TEST_SET_PATH);
```

```
        // Output the training data
        for (double[] rowData : trainingData) {
            for (double value : rowData) {
                System.out.print(value + " ");
            }
            System.out.println();
        }

        CatchmentMLP mlp = new CatchmentMLP(numHiddenNeurons);
        // Train the model and validate for a specified number of epochs
        mlp.train(trainingData, epochs, validationData);

        for (double[] dataRow : trainingData) {
            double[] inputs = new double[CatchmentMLP.getNumInputs()];
            System.arraycopy(dataRow, 0, inputs, 0, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];
            double predictedOutput = mlp.forwardPass(inputs);
            System.out.println(actualOutput + "                         "
+predictedOutput);
        }
        System.out.println("\nActual vs Predicted Outputs (Test Set):");
        for (double[] dataRow : testData) {
            double[] inputs = new double[CatchmentMLP.getNumInputs()];
            System.arraycopy(dataRow, 0, inputs, 0, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];
            double predictedOutput = mlp.forwardPass(inputs);
            System.out.println("Actual:         " + actualOutput + "
Predicted:          " + predictedOutput);
        }
    }
    // Method to remove UTF-8 Byte Order Mark (BOM) from a string.
    private static String removeUtf8Bom(String s) {
        if (s.startsWith("\uFEFF")) {
            s = s.substring(1);
        }
        return s;
    }
    // Method to load data from a CSV file and convert it into a list of double
arrays.- Just as before
    private static List<double[]> loadData(String filePath) {
        List<double[]> data = new ArrayList<>();
        try (BufferedReader br = Files.newBufferedReader(Paths.get(filePath),
StandardCharsets.UTF_8)) {
            String line;
            while ((line = br.readLine()) != null) {
                line = removeUtf8Bom(line);
                String[] values = line.split(",");
                double[] doubleValues = new double[values.length];
                for (int i = 0; i < values.length; i++) {
                    doubleValues[i] = Double.parseDouble(values[i]);
                }
                data.add(doubleValues);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return data;
    }

}
```

```
import java.util.ArrayList;
import java.util.Random;
import java.util.List;
```

```java
public class CatchmentMLP {
    public double[][] inputToHiddenWeights;
    private double[][] previousInputToHiddenUpdates; // Stores previous updates for input-to-hidden weights to implement momentum in training.
    public double[] hiddenBiases; // Array to store biases for the hidden layer neurons.
    public double[] hiddenToOutputWeights; // Array to store weights from the hidden layer to the output layer.
    private double[] previousHiddenToOutputUpdates; // Stores previous weight updates from hidden to output layer for momentum calculation.
    private double momentum = 0.9;// Momentum factor to accelerate convergence in the desired direction
    public double outputBias;
    private static final int numInputs = 8; // Use all 8 inputs
    private Random rand = new Random();
    public static final double learningRate = 0.1; // Learning rate is fixed
    private final int numHiddenNeurons; // This is entered by the user

    public CatchmentMLP(int numHiddenNeurons) {
        this.numHiddenNeurons = numHiddenNeurons; // Set the number of hidden neurons using the User's input
        initializeWeightsAndBiases();
    }
    public static int getNumInputs() {
        return numInputs;
    }
    // Range for weights  and bias initialization: -2/n to 2/n, where n is the number of inputs

    private void initializeWeightsAndBiases() {
        double range = 4.0 / numInputs;
        previousInputToHiddenUpdates = new double[numInputs][numHiddenNeurons];
        previousHiddenToOutputUpdates = new double[numHiddenNeurons];
        // Initialize weights and biases for each input neuron
        inputToHiddenWeights = new double[numInputs][numHiddenNeurons];
        hiddenBiases = new double[numHiddenNeurons];
        hiddenToOutputWeights = new double[numHiddenNeurons];
        for (int i = 0; i < numInputs; i++) {
            for (int j = 0; j < numHiddenNeurons; j++) {
                previousInputToHiddenUpdates[i][j] = 0.0; // Set to zero
                inputToHiddenWeights[i][j] = range * (rand.nextDouble() - 0.5);
            }
        }
        for (int i = 0; i < numHiddenNeurons; i++) {
            previousHiddenToOutputUpdates[i] = 0.0; // Set to zero
            hiddenBiases[i] = range * (rand.nextDouble() - 0.5);
            hiddenToOutputWeights[i] = range * (rand.nextDouble() - 0.5);
        }
        outputBias = (numHiddenNeurons / 2.0) * (2.0 * rand.nextDouble() - 1.0);
    }
    // Forward pass algorithm as explained before, no changes done
    public double forwardPass(double[] inputs) {
        double[] hiddenOutputs = new double[numHiddenNeurons];
        for (int i = 0; i < numHiddenNeurons; i++) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; j++) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }

        double output = outputBias;
        for (int i = 0; i < numHiddenNeurons; i++) {
            output += hiddenOutputs[i] * hiddenToOutputWeights[i];
        }
        return sigmoid(output);
```

```
    }
    // Activation functions as explained before
    private double sigmoid(double x) {
        return 1.0 / (1.0 + Math.exp(-x));
    }
    private double sigmoidDerivative(double x) {
        return x * (1.0 - x);
    }
    // Training function as explained before performing forward and backward pass only
on training set
    // Performs only forward pass on validation set
    public void train(List<double[]> trainingData, int epochs, List<double[]>
validationData) {
        List<String> trainingMSEList = new ArrayList<>();
        List<String> validationMSEList = new ArrayList<>();
        for (int epoch = 0; epoch < epochs; epoch++) {
            double totalTrainingError = 0;
            for (double[] row : trainingData) {
                double[] inputs = new double[numInputs];
                System.arraycopy(row, 0, inputs, 0, numInputs);
                double actualOutput = row[numInputs];
                double predictedOutput = forwardPass(inputs);
                double error = actualOutput - predictedOutput;
                totalTrainingError += Math.pow(error, 2);

                // Backward pass and weight update
                backwardPass(inputs, predictedOutput, error);
            }
            if (epoch % 100 == 0) {
                // Compute Validation MSE
                double totalValidationMSE = 0;
                for (double[] valRow : validationData) {
                    double[] valInputs = new double[numInputs];
                    System.arraycopy(valRow, 0, valInputs, 0, numInputs);
                    double actualValOutput = valRow[numInputs];
                    double predictedValOutput = forwardPass(valInputs);
                    double error = actualValOutput - predictedValOutput;
                    totalValidationMSE += error * error;
                }
                double validationMSE = totalValidationMSE / validationData.size();
                validationMSEList.add("Epoch:           " + epoch + "
Validation MSE:           " + validationMSE);

                // Compute Training MSE
                double totalTrainingMSE = 0;
                for (double[] trainRow : trainingData) {
                    double[] trainInputs = new double[numInputs];
                    System.arraycopy(trainRow, 0, trainInputs, 0, numInputs);
                    double actualTrainOutput = trainRow[numInputs];
                    double predictedTrainOutput = forwardPass(trainInputs);
                    double error = actualTrainOutput - predictedTrainOutput;
                    totalTrainingMSE += error * error;
                }
                double trainingMSE = totalTrainingMSE / trainingData.size();
                trainingMSEList.add("Epoch:           " + epoch + "           Training
MSE:           " + trainingMSE);
            }
        }
        // Print training MSE and validation MSE list
        System.out.println("Training MSEs:");
        for (String mse : trainingMSEList) {
            System.out.println(mse);
        }
        System.out.println("\nValidation MSEs:");
        for (String mse : validationMSEList) {
            System.out.println(mse);
```

```
        }
    }
    // Perform the backward pass for backpropagation and update weights and
    // biases as explained before but intergrating momentum
    private void backwardPass(double[] inputs, double predictedOutput, double error) {
        double[] hiddenOutputs = new double[numHiddenNeurons];

        // Calculate hidden layer outputs
        for (int i = 0; i < numHiddenNeurons; ++i) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; ++j) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }

        // Calculate output layer delta
        double deltaOutput = error * sigmoidDerivative(predictedOutput);

        // Update weights and biases for hidden-to-output weights
        for (int i = 0; i < numHiddenNeurons; ++i) {
            // Calculate the current update without momentum
            double currentUpdateHO = learningRate * deltaOutput * hiddenOutputs[i];
            // Calculate the weight change (new weight - old weight)
            double weightChangeHO = hiddenToOutputWeights[i] -
previousHiddenToOutputUpdates[i];
            // Apply the momentum term
            hiddenToOutputWeights[i] += currentUpdateHO + (momentum * weightChangeHO);
            // Update the previous weight for the next iteration
            previousHiddenToOutputUpdates[i] = hiddenToOutputWeights[i];
        }
        outputBias += learningRate * deltaOutput;

        // Update weights and biases for input-to-hidden weights
        for (int i = 0; i < numHiddenNeurons; ++i) {
            double deltaHidden = deltaOutput * hiddenToOutputWeights[i] *
sigmoidDerivative(hiddenOutputs[i]);
            for (int j = 0; j < numInputs; ++j) {
                // Calculate the current update without momentum
                double currentUpdateIH = learningRate * deltaHidden * inputs[j];
                // Calculate the weight change (new weight - old weight)
                double weightChangeIH = inputToHiddenWeights[j][i] -
previousInputToHiddenUpdates[j][i];
                // Apply the momentum term
                inputToHiddenWeights[j][i] += currentUpdateIH + (momentum *
weightChangeIH);
                // Update the previous weight for the next iteration
                previousInputToHiddenUpdates[j][i] = inputToHiddenWeights[j][i];
            }
            hiddenBiases[i] += learningRate * deltaHidden;
        }
    }
}
```

Appendix C: Basic Back Propagation + Momentum + Annealing + Bold Driver

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Arrays;
```

```
import java.util.List;
import java.util.Scanner;

// as explained before
public class Main {
    public static final String TEST_SET_PATH ="testSet.csv";
    private static final String TRAINING_SET_PATH = "trainingSet1.csv"; // Update this
with the path to your CSV file
    private static final String VALIDATION_SET_PATH = "validationSet.csv";


    public static void main(String[] args) {
        Scanner scanner= new Scanner(System.in);
        System.out.println("Enter the number of epochs:");
        int epochs= scanner.nextInt(); // Read user input for number of epochs
        System.out.println("Enter the number of hidden neurons:");
        int numHiddenNeurons= scanner.nextInt(); // Read user input for number of
neurons

        // List to hold training data and validation data as explained before
        List<double[]> trainingData = loadData(TRAINING_SET_PATH);
        List<double[]> validationData = loadData(VALIDATION_SET_PATH);
        List<double[]> testData = loadData(TEST_SET_PATH);

        CatchmentMLP mlp = new CatchmentMLP(numHiddenNeurons);
        // Train the model and validate for a specified number of epochs
        mlp.train(trainingData, epochs, validationData);
        // Instantiate the Annealing class for a given number of hidden neurons
        AnnealingMLP annealingMLP = new AnnealingMLP(numHiddenNeurons);
        annealingMLP.train(trainingData, epochs, validationData);
        // Print out the comparisons
        System.out.println("\nComparisons:");
        for (double[] dataRow : trainingData) {
            double[] inputs = Arrays.copyOf(dataRow, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];

            double predictedOutputMomentum = mlp.forwardPass(inputs);
            double predictedOutputAnnealing = annealingMLP.forwardPass(inputs);

            System.out.println("Actual Output:      " + actualOutput +
                    "           With Momentum:        " + predictedOutputMomentum +
                    "           With Annealing:       " + predictedOutputAnnealing);
        }
        // Loading the test data
        System.out.println("\nComparisons on Test Data:");
        for (double[] dataRow : testData) {
            double[] inputs = Arrays.copyOf(dataRow, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];

            double predictedOutputMomentum = mlp.forwardPass(inputs);
            double predictedOutputAnnealing = annealingMLP.forwardPass(inputs);

            System.out.println("Actual Output:      " + actualOutput +
                    "           With Momentum:        " + predictedOutputMomentum +
                    "           With Annealing:       " + predictedOutputAnnealing);
        }
    }
    // Method to remove UTF-8 Byte Order Mark (BOM) from a string.
    private static String removeUtf8Bom(String s) {
        if (s.startsWith("\uFEFF")) {
            s = s.substring(1);
        }
        return s;
    }
    // Loading data method
    static List<double[]> loadData(String filePath) {
```

```
        List<double[]> data = new ArrayList<>();
        try (BufferedReader br = Files.newBufferedReader(Paths.get(filePath),
StandardCharsets.UTF_8)) {
            String line;
            while ((line = br.readLine()) != null) {
                line = removeUtf8Bom(line);
                String[] values = line.split(","); // Make sure to use the correct
delimiter
                double[] doubleValues = new double[values.length];
                for (int i = 0; i < values.length; i++) {
                    doubleValues[i] = Double.parseDouble(values[i]);
                }
                data.add(doubleValues);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return data;
    }

}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;
import java.util.List;

public class CatchmentMLP {
    public double[][] inputToHiddenWeights;
    private double[][] previousInputToHiddenUpdates;
    public double[] hiddenBiases;
    public double[] hiddenToOutputWeights;
    private double[] previousHiddenToOutputUpdates;
    public double outputBias;
    public static final int numInputs = 8;
    private Random rand = new Random();
    public static double learningRate = 0.1;
    public final int numHiddenNeurons;

    public CatchmentMLP(int numHiddenNeurons) {
        this.numHiddenNeurons = numHiddenNeurons; // Set the number of hidden neurons
using the User's input
        initializeWeightsAndBiases();
    }
    public static int getNumInputs() {
        return numInputs;
    }
    // Range for weights initialization: -2/n to 2/n as explained before
    private void initializeWeightsAndBiases() {
        double range = 4.0 / numInputs;
        previousInputToHiddenUpdates = new double[numInputs][numHiddenNeurons];
        previousHiddenToOutputUpdates = new double[numHiddenNeurons];
        inputToHiddenWeights = new double[numInputs][numHiddenNeurons];
        hiddenBiases = new double[numHiddenNeurons];
        hiddenToOutputWeights = new double[numHiddenNeurons];
        for (int i = 0; i < numInputs; i++) {
            for (int j = 0; j < numHiddenNeurons; j++) {
                previousInputToHiddenUpdates[i][j] = 0.0;
                inputToHiddenWeights[i][j] = range * (rand.nextDouble() - 0.5);
            }
        }
        for (int i = 0; i < numHiddenNeurons; i++) {
            previousHiddenToOutputUpdates[i] = 0.0;
            hiddenBiases[i] = range * (rand.nextDouble() - 0.5);
            hiddenToOutputWeights[i] = range * (rand.nextDouble() - 0.5);
```

```
        }
        outputBias = (numHiddenNeurons / 2.0) * (2.0 * rand.nextDouble() - 1.0);
    }
    // Forward pass algorithm as explained before
    public double forwardPass(double[] inputs) {
        double[] hiddenOutputs = new double[numHiddenNeurons];
        for (int i = 0; i < numHiddenNeurons; i++) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; j++) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }

        double output = outputBias;
        for (int i = 0; i < numHiddenNeurons; i++) {
            output += hiddenOutputs[i] * hiddenToOutputWeights[i];
        }
        return sigmoid(output);
    }
    // Activation functions as explained before
    private double sigmoid(double x) {
        return 1.0 / (1.0 + Math.exp(-x));
    }
    private double sigmoidDerivative(double x) {
        return x * (1.0 - x);
    }
    // Training algorithm as explained before
    public void train(List<double[]> trainingData, int epochs, List<double[]>
validationData) {
        List<String> trainingMSEList = new ArrayList<>();
        List<String> validationMSEList = new ArrayList<>();
        // Create an instance of BoldDriverMLP with a given number of hidden neurons
        BoldDriverMLP boldDriverMLP = new BoldDriverMLP(numHiddenNeurons,
validationData);


        for (int epoch = 0; epoch < epochs; epoch++) {
            double totalTrainingError = 0;
            boldDriverMLP.trainEpoch(trainingData, learningRate, epoch);

            for (double[] row : trainingData) {
                double[] inputs = new double[numInputs];
                System.arraycopy(row, 0, inputs, 0, numInputs);
                double actualOutput = row[numInputs];

                double predictedOutput = forwardPass(inputs);
                double error = actualOutput - predictedOutput;
                totalTrainingError += Math.pow(error, 2);

                // Backward pass and weight update
                backwardPass(inputs, predictedOutput, error, learningRate);
            }
            if (epoch % 100 == 0) {
                // Compute Validation MSE
                double totalValidationMSE = 0;
                for (double[] valRow : validationData) {
                    double[] valInputs = new double[numInputs];
                    System.arraycopy(valRow, 0, valInputs, 0, numInputs);
                    double actualValOutput = valRow[numInputs];
                    double predictedValOutput = forwardPass(valInputs);
                    double error = actualValOutput - predictedValOutput;
                    totalValidationMSE += error * error;
                }
                double validationMSE = totalValidationMSE / validationData.size();
                validationMSEList.add("Epoch:           " + epoch + "
```

```
Validation MSE:           " + validationMSE);

                // Compute Training MSE
                double totalTrainingMSE = 0;
                for (double[] trainRow : trainingData) {
                    double[] trainInputs = new double[numInputs];
                    System.arraycopy(trainRow, 0, trainInputs, 0, numInputs);
                    double actualTrainOutput = trainRow[numInputs];
                    double predictedTrainOutput = forwardPass(trainInputs);
                    double error = actualTrainOutput - predictedTrainOutput;
                    totalTrainingMSE += error * error;
                }
                double trainingMSE = totalTrainingMSE / trainingData.size();
                trainingMSEList.add("Epoch:           " + epoch + "            Training
MSE:           " + trainingMSE);
            }

        }
        List<double[]> testData = Main.loadData(Main.TEST_SET_PATH);

        System.out.println("\nComparisons on Test Data with Bold Driver:");
        for (double[] dataRow : testData) {
            double[] inputs = Arrays.copyOf(dataRow, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];

            // Use the trained BoldDriverMLP to make predictions
            double predictedOutputBoldDriver = boldDriverMLP.forwardPass(inputs);

            System.out.println("Actual Output: " + actualOutput +
                    "   With Bold Driver: " + predictedOutputBoldDriver);
        }
        System.out.println("Training MSEs For Momentum:");
        for (String mse : trainingMSEList) {
            System.out.println(mse);
        }

        // Print validation MSE list
        System.out.println("\nValidation MSEs for Momentum:");
        for (String mse : validationMSEList) {
            System.out.println(mse);
        }
    }
    // basic train method that is overidden in different classes
    protected void trainEpoch(List<double[]> trainingData, double learningRate, int
epoch) {
        double totalError = 0;

        // Iterate over each data row in the training set
        for (double[] row : trainingData) {
            double[] inputs = Arrays.copyOf(row, numInputs);
            double actualOutput = row[numInputs];

            double predictedOutput = forwardPass(inputs);
            double error = actualOutput - predictedOutput;
            totalError += Math.pow(error, 2);

            backwardPass(inputs, predictedOutput, error, learningRate); // Perform
backpropagation
        }
//        double mse = totalError / trainingData.size(); // Calculate mean squared
error for the epoch
    }

    // Basic backward pass with only momentum+ back prop as explained before
    private void backwardPass(double[] inputs, double predictedOutput, double error,
double learningRate) {
```

```
        double[] hiddenOutputs = new double[numHiddenNeurons];

        // Calculate hidden layer outputs
        for (int i = 0; i < numHiddenNeurons; ++i) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; ++j) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }
        // Calculate output layer delta
        double deltaOutput = error * sigmoidDerivative(predictedOutput);
        double momentum = 0.9;
        for (int i = 0; i < numHiddenNeurons; ++i) {
            double currentUpdateHO = learningRate * deltaOutput * hiddenOutputs[i];
            double weightChangeHO = hiddenToOutputWeights[i] -
previousHiddenToOutputUpdates[i];
            hiddenToOutputWeights[i] += currentUpdateHO + (momentum * weightChangeHO);
            previousHiddenToOutputUpdates[i] = hiddenToOutputWeights[i];
        }
        outputBias += learningRate * deltaOutput;
        for (int i = 0; i < numHiddenNeurons; ++i) {
            double deltaHidden = deltaOutput * hiddenToOutputWeights[i] *
sigmoidDerivative(hiddenOutputs[i]);

            for (int j = 0; j < numInputs; ++j) {
                double currentUpdateIH = learningRate * deltaHidden * inputs[j];
                double weightChangeIH = inputToHiddenWeights[j][i] -
previousInputToHiddenUpdates[j][i];
                inputToHiddenWeights[j][i] += currentUpdateIH + (momentum *
weightChangeIH);
                previousInputToHiddenUpdates[j][i] = inputToHiddenWeights[j][i];
            }
            hiddenBiases[i] += learningRate * deltaHidden;
        }
    }
}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class AnnealingMLP extends CatchmentMLP {

    public AnnealingMLP(int numHiddenNeurons) {
        super(numHiddenNeurons);
    }
    @Override
    public void train(List<double[]> trainingData, int epochs, List<double[]>
validationData) {
        // List to store validation MSE for each epoch
        List<Double> validationMSEList = new ArrayList<>();
        List<Double> trainingMSEList = new ArrayList<>();
        List<Integer> annealingEpochs = new ArrayList<>();
        List<Double> adjustedLearningRates = new ArrayList<>();

        for (int epoch = 0; epoch < epochs; epoch++) {
            double x = epoch; // Calculate the annealed learning rate for this epoch
            double r = epochs; // total epochs
            double p = 0.01; // End learning rate parameter
            double q = 0.1; // Start learning rate parameter
            double adjustedLearningRate = p + (q - p) * (1 - 1 / (1 + Math.exp(10 - (20
* x / r)))); // Equation for adjusting the learning rate

            // Train for one epoch using the adjusted learning rate
            super.trainEpoch(trainingData, adjustedLearningRate, epoch);
```

```
            if (epoch % 100 == 0 || epoch == 0) {
                annealingEpochs.add(epoch);
                adjustedLearningRates.add(adjustedLearningRate);
            }
            // Print the adjusted learning rate every 100 epochs
            if (epoch % 100 == 0) {
                double totalTrainingMSE = computeTrainingMSE(trainingData);
                trainingMSEList.add(totalTrainingMSE); // Store training MSE for this
epoch
                double validationMSE = computeValidationMSE(validationData); // Compute
validation MSE
                validationMSEList.add(validationMSE); // Store validation MSE for this
epoch
            }
        }
        System.out.println("Annealing Training MSE and Validation MSE:");

        System.out.println("Epoch           Training MSE               Validation
MSE");
        for (int i = 0; i < validationMSEList.size(); i++) {
            System.out.println(i + "            " + trainingMSEList.get(i) + "
" + validationMSEList.get(i));
        }
        System.out.println("Annealing Epochs                Adjusted Learning Rates:");
// Print annealing epochs and adjusted learning rates
        for (int i = 0; i < annealingEpochs.size(); i++) {
            System.out.println(annealingEpochs.get(i) + "
" + adjustedLearningRates.get(i));
        }
    }
    // Compute the validation MSE
    private double computeValidationMSE(List<double[]> validationData) {
        double totalValidationMSE = 0;
        int dataSize = validationData.size();

        for (double[] valRow : validationData) {
            double[] inputs = Arrays.copyOf(valRow, numInputs);
            double actualOutput = valRow[numInputs];
            double predictedOutput = forwardPass(inputs);
            double error = actualOutput - predictedOutput;
            totalValidationMSE += Math.pow(error, 2);
        }
        return totalValidationMSE / dataSize;
    }
    // Compute the training MSE
    private double computeTrainingMSE(List<double[]> trainingData) {
        double totalTrainingMSE = 0;
        int dataSize = trainingData.size();

        for (double[] trainRow : trainingData) {
            double[] inputs = Arrays.copyOf(trainRow, numInputs);
            double actualOutput = trainRow[numInputs];
            double predictedOutput = forwardPass(inputs);
            double error = actualOutput - predictedOutput;
            totalTrainingMSE += Math.pow(error, 2);
        }
        return totalTrainingMSE / dataSize;
    }
}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
// Extends CatchmentMLP to include Bold Driver in addition to back propogation
public class BoldDriverMLP extends CatchmentMLP {
    private final double[][] inputToHiddenWeightsBeforeUpdate; // Holds model's state
```

```
before updates for rollback in case of performance decrease after an update.
    private final double[] hiddenBiasesBeforeUpdate; // Stores hidden layer biases
before updates to allow reverting if needed.
    private final double[] hiddenToOutputWeightsBeforeUpdate; // Keeps track of weights
from hidden to output layer before updates.
    private double outputBiasBeforeUpdate; // Records the output bias before its update
for possible rollback.
    private double previousMSE = Double.MAX_VALUE; // Store the MSE of the previous
epoch
    private final List<Double> trainingMSEList = new ArrayList<>();
    private final List<Double> validationMSEList = new ArrayList<>();
    private final List<double[]> validationData;

    // Constructor initializes model and sets up for Bold Driver adaptation.
    public BoldDriverMLP(int numHiddenNeurons,List<double[]> validationData) {
        super(numHiddenNeurons);
        inputToHiddenWeightsBeforeUpdate = new double[numInputs][numHiddenNeurons];
        hiddenBiasesBeforeUpdate = new double[numHiddenNeurons];
        hiddenToOutputWeightsBeforeUpdate = new double[numHiddenNeurons];
        this.validationData = validationData;
    }

    // Overrides trainEpoch to include Bold Driver's dynamic learning rate adjustment.
    @Override
    protected void trainEpoch(List<double[]> trainingData, double learningRate, int
epoch) {
        // Store the weights and biases before update
        for (int i = 0; i < numInputs; i++) {
            System.arraycopy(inputToHiddenWeights[i], 0,
inputToHiddenWeightsBeforeUpdate[i], 0, numHiddenNeurons);
        }
        System.arraycopy(hiddenBiases, 0, hiddenBiasesBeforeUpdate, 0,
numHiddenNeurons);
        System.arraycopy(hiddenToOutputWeights, 0, hiddenToOutputWeightsBeforeUpdate,
0, numHiddenNeurons);
        outputBiasBeforeUpdate = outputBias;
        super.trainEpoch(trainingData, learningRate, epoch); // Call the parent method
to train for one epoch
        // After training, calculate the MSE and adjust learning rate
        double currentMSE = calculateMSE(trainingData);
        double currentValidationMSE = calculateValidationMSE(); // Calculate validation
MSE
        trainingMSEList.add(currentMSE);
        validationMSEList.add(currentValidationMSE);
        adjustLearningRate(currentMSE, epoch);

        if (epoch % 1000 == 0) {
            adjustLearningRate(currentMSE, epoch);
            System.out.println("Epoch for Bold Driver:                  " + epoch + "
Learning Rate:                    " + learningRate + "                Training MSE:
" + currentMSE + "                 Validation MSE:                 " +
calculateValidationMSE());

        } else {
            // Adjust learning rate every 2000 epochs
            adjustLearningRate(currentMSE, epoch);
        }

    }
    private double calculateMSE(List<double[]> data) {
        double totalError = 0;
        for (double[] row : data) {
            double[] inputs = Arrays.copyOf(row, numInputs); // Extract input features
from the row
            double actualOutput = row[numInputs]; // Actual output is the last element
in the row
```

```
            double predictedOutput = forwardPass(inputs); // Get the predicted output
from the neural network
            double error = actualOutput - predictedOutput; // Calculate the error
            totalError += Math.pow(error, 2); // Sum the squared error
        }
        return totalError / data.size(); // Return the mean squared error
    }
    private double calculateValidationMSE() {
        return calculateMSE(validationData); // Calculate MSE using validation data
    }
    // Adjusts the learning rate based on comparison of current MSE to previous MSE,
implementing the Bold Driver strategy.
    private void adjustLearningRate(double currentMSE, int epoch) {
        // Update learning rate and weights only every few thousand epochs
        if (epoch % 1000 == 0) {
            if (currentMSE > previousMSE * (1 + 0.04)) {
                // Factor to decrease the learning rate if performance worsens
                double decreaseFactor = 0.7;
                learningRate *= decreaseFactor;
                // Minimum boundary for the learning rate
                double minLearningRate = 0.01;
                if (learningRate < minLearningRate) learningRate = minLearningRate;
                undoWeightUpdates(); // Undo the last weight updates because the error
increased
            } else if (currentMSE < previousMSE) {
                // Factor to increase the learning rate if performance improves
                double increaseFactor = 1.05;
                learningRate *= increaseFactor;
                // Maximum boundary for the learning rate
                double maxLearningRate = 0.5;
                if (learningRate > maxLearningRate) learningRate = maxLearningRate;
            }
            previousMSE = currentMSE;
        }
    }
    // Reverts weights and biases to their previous state before the last update if the
MSE increases.
    private void undoWeightUpdates() {
        for (int i = 0; i < numInputs; i++) {
            System.arraycopy(inputToHiddenWeightsBeforeUpdate[i], 0,
inputToHiddenWeights[i], 0, numHiddenNeurons);
        }
        System.arraycopy(hiddenBiasesBeforeUpdate, 0, hiddenBiases, 0,
numHiddenNeurons);
        System.arraycopy(hiddenToOutputWeightsBeforeUpdate, 0, hiddenToOutputWeights,
0, numHiddenNeurons);
        outputBias = outputBiasBeforeUpdate;
    }
}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class BoldDriverMLP extends CatchmentMLP {
    private final double[][] inputToHiddenWeightsBeforeUpdate; // Holds model's state
before updates for rollback in case of performance decrease after an update.
    private final double[] hiddenBiasesBeforeUpdate; // Stores hidden layer biases
before updates to allow reverting if needed.
    private final double[] hiddenToOutputWeightsBeforeUpdate; // Keeps track of weights
from hidden to output layer before updates.
    private double outputBiasBeforeUpdate; // Records the output bias before its update
for possible rollback.
    private double previousMSE = Double.MAX_VALUE; // Store the MSE of the previous
epoch
```

```
    private final List<Double> trainingMSEList = new ArrayList<>();
    private final List<Double> validationMSEList = new ArrayList<>();
    private final List<double[]> validationData;

    // Constructor initializes model and sets up for Bold Driver adaptation.
    public BoldDriverMLP(int numHiddenNeurons,List<double[]> validationData) {
        super(numHiddenNeurons);
        inputToHiddenWeightsBeforeUpdate = new double[numInputs][numHiddenNeurons];
        hiddenBiasesBeforeUpdate = new double[numHiddenNeurons];
        hiddenToOutputWeightsBeforeUpdate = new double[numHiddenNeurons];
        this.validationData = validationData;
    }

    // Overrides trainEpoch to include Bold Driver's dynamic learning rate adjustment.
    @Override
    protected void trainEpoch(List<double[]> trainingData, double learningRate, int
epoch) {
        // Store the weights and biases before update
        for (int i = 0; i < numInputs; i++) {
            System.arraycopy(inputToHiddenWeights[i], 0,
inputToHiddenWeightsBeforeUpdate[i], 0, numHiddenNeurons);
        }
        System.arraycopy(hiddenBiases, 0, hiddenBiasesBeforeUpdate, 0,
numHiddenNeurons);
        System.arraycopy(hiddenToOutputWeights, 0, hiddenToOutputWeightsBeforeUpdate,
0, numHiddenNeurons);
        outputBiasBeforeUpdate = outputBias;
        super.trainEpoch(trainingData, learningRate, epoch); // Call the parent method
to train for one epoch
        // After training, calculate the MSE and adjust learning rate
        double currentMSE = calculateMSE(trainingData);
        double currentValidationMSE = calculateValidationMSE(); // Calculate validation
MSE
        trainingMSEList.add(currentMSE);
        validationMSEList.add(currentValidationMSE);
        adjustLearningRate(currentMSE, epoch);

        if (epoch % 1000 == 0) {
            adjustLearningRate(currentMSE, epoch);
            System.out.println("Epoch for Bold Driver:                  " + epoch + "
Learning Rate:                  " + learningRate + "                  Training MSE:
" + currentMSE + "                  Validation MSE:                  " +
calculateValidationMSE());

        } else {
            // Adjust learning rate every 1000 epochs
            adjustLearningRate(currentMSE, epoch);
        }

    }
    private double calculateMSE(List<double[]> data) {
        double totalError = 0;
        for (double[] row : data) {
            double[] inputs = Arrays.copyOf(row, numInputs); // Extract input features
from the row
            double actualOutput = row[numInputs]; // Actual output is the last element
in the row
            double predictedOutput = forwardPass(inputs); // Get the predicted output
from the neural network
            double error = actualOutput - predictedOutput; // Calculate the error
            totalError += Math.pow(error, 2); // Sum the squared error
        }
        return totalError / data.size(); // Return the mean squared error
    }
    private double calculateValidationMSE() {
        return calculateMSE(validationData); // Calculate MSE using validation data
```

```
    }
    // Adjusts the learning rate based on comparison of current MSE to previous MSE,
implementing the Bold Driver strategy.
    private void adjustLearningRate(double currentMSE, int epoch) {
        // Update learning rate and weights only every few thousand epochs
        if (epoch % 1000 == 0) {
            if (currentMSE > previousMSE * (1 + 0.04)) {
                // Factor to decrease the learning rate if performance worsens
                double decreaseFactor = 0.7;
                learningRate *= decreaseFactor;
                // Minimum boundary for the learning rate
                double minLearningRate = 0.01;
                if (learningRate < minLearningRate) learningRate = minLearningRate;
                undoWeightUpdates(); // Undo the last weight updates because the error
increased
            } else if (currentMSE < previousMSE) {
                // Factor to increase the learning rate if performance improves
                double increaseFactor = 1.05;
                learningRate *= increaseFactor;
                // Maximum boundary for the learning rate
                double maxLearningRate = 0.5;
                if (learningRate > maxLearningRate) learningRate = maxLearningRate;
            }
            previousMSE = currentMSE;
        }
    }
    // Reverts weights and biases to their previous state before the last update if the
MSE increases.
    private void undoWeightUpdates() {
        for (int i = 0; i < numInputs; i++) {
            System.arraycopy(inputToHiddenWeightsBeforeUpdate[i], 0,
inputToHiddenWeights[i], 0, numHiddenNeurons);
        }
        System.arraycopy(hiddenBiasesBeforeUpdate, 0, hiddenBiases, 0,
numHiddenNeurons);
        System.arraycopy(hiddenToOutputWeightsBeforeUpdate, 0, hiddenToOutputWeights,
0, numHiddenNeurons);
        outputBias = outputBiasBeforeUpdate;
    }
}
```

Appendix D: Basic Back Propagation + Momentum + Annealing + Weight Decay

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

public class Main {
    private static final String TRAINING_SET_PATH = "trainingSet1.csv"; // Update this
with the path to your CSV file
    private static final String VALIDATION_SET_PATH = "validationSet.csv";
    public static final String TEST_SET_PATH ="testSet.csv";

    public static void main(String[] args) {
        Scanner scanner= new Scanner(System.in);
        System.out.println("Enter the number of epochs:");
        int epochs= scanner.nextInt(); // Read user input for number of epochs
        System.out.println("Enter the number of hidden neurons:");
        int numHiddenNeurons= scanner.nextInt(); // Read user input for number of
neurons
        // List to hold training data and validation data
        List<double[]> trainingData = loadData(TRAINING_SET_PATH);
        List<double[]> validationData = loadData(VALIDATION_SET_PATH);
        List<double[]> testData = loadData(TEST_SET_PATH);

        CatchmentMLP mlp = new CatchmentMLP(numHiddenNeurons);
        // Train the model and validate for a specified number of epochs
        mlp.train(trainingData, epochs, validationData);
        AnnealingMLP annealingMLP = new AnnealingMLP(numHiddenNeurons);
        annealingMLP.train(trainingData, epochs, validationData);

        // Print comparisons
        System.out.println("\nComparisons:");
        for (double[] dataRow : trainingData) {
            double[] inputs = Arrays.copyOf(dataRow, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];

            double predictedOutputMomentum = mlp.forwardPass(inputs);
            double predictedOutputAnnealing = annealingMLP.forwardPass(inputs);

            System.out.println("Actual Output:      " + actualOutput +
                    "             With Momentum:        " + predictedOutputMomentum +
                    "             With Annealing:       " + predictedOutputAnnealing);
        }
        // Assuming AnnealingMLP has been trained as 'annealingMLP'
        System.out.println("\nEvaluating Test Data with Annealing and Weight Decay:");

        for (double[] testRow : testData) {
            double[] inputs = Arrays.copyOf(testRow, CatchmentMLP.getNumInputs());
            double actualOutput = testRow[CatchmentMLP.getNumInputs()];
            double predictedOutputMomentum = mlp.forwardPass(inputs);
            double predictedOutput = annealingMLP.forwardPass(inputs);

            System.out.println("Actual Output: " + actualOutput + "          Predicted
Output: " + predictedOutput + "           With momentum alone:"
+predictedOutputMomentum);
        }
```

```
        // Inside your Main class, right before test evaluations
        double checksumBefore = annealingMLP.calculateChecksum();
        System.out.println("Checksum before test evaluations: " + checksumBefore);

        // Calculate checksum before and after test evaluations to detect changes in
weights.
        double checksumAfter = annealingMLP.calculateChecksum();
        System.out.println("Checksum after test evaluations: " + checksumAfter);
        // Verify if weight changes occurred during test evaluations.
        if (checksumBefore == checksumAfter) {
            System.out.println("No weight changes detected.");
        } else {
            System.out.println("Weight changes detected.");
        }
    }
    // Helper method to remove UTF-8 BOM from strings if present.
    private static String removeUtf8Bom(String s) {
        if (s.startsWith("\uFEFF")) {
            s = s.substring(1);
        }
        return s;
    }
    // Loads dataset from a CSV file, converts string values to doubles, and returns a
list of data points.
    private static List<double[]> loadData(String filePath) {
        List<double[]> data = new ArrayList<>();
        try (BufferedReader br = Files.newBufferedReader(Paths.get(filePath),
StandardCharsets.UTF_8)) {
            String line;
            while ((line = br.readLine()) != null) {
                line = removeUtf8Bom(line);
                String[] values = line.split(","); // Make sure to use the correct
delimiter
                double[] doubleValues = new double[values.length];
                for (int i = 0; i < values.length; i++) {
                    doubleValues[i] = Double.parseDouble(values[i]);
                }
                data.add(doubleValues);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return data;
    }

}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;
import java.util.List;
// Parent class to be extended
public class CatchmentMLP {
    // as commented before everything stays constant
    public double[][] inputToHiddenWeights;
    private double[][] previousInputToHiddenUpdates;
    public double[] hiddenBiases;
    public double[] hiddenToOutputWeights;
    private double[] previousHiddenToOutputUpdates;
    public double outputBias;
    public static final int numInputs = 8;
    private Random rand = new Random();
    public static double learningRate = 0.1;
    public final int numHiddenNeurons;
```

```java
    public CatchmentMLP(int numHiddenNeurons) {
        this.numHiddenNeurons = numHiddenNeurons;
        initializeWeightsAndBiases();
    }

    public static int getNumInputs() {
        return numInputs;
    }
    // Initializes weights and biases with random values to start training from a non-
uniform initial state.
    private void initializeWeightsAndBiases() {
        double range = 4.0 / numInputs;
        previousInputToHiddenUpdates = new double[numInputs][numHiddenNeurons];
        previousHiddenToOutputUpdates = new double[numHiddenNeurons];
        // Initialize weights and biases for each input neuron
        inputToHiddenWeights = new double[numInputs][numHiddenNeurons];
        hiddenBiases = new double[numHiddenNeurons];
        hiddenToOutputWeights = new double[numHiddenNeurons];
        for (int i = 0; i < numInputs; i++) {
            for (int j = 0; j < numHiddenNeurons; j++) {
                previousInputToHiddenUpdates[i][j] = 0.0; // Set to zero
                inputToHiddenWeights[i][j] = range * (rand.nextDouble() - 0.5);
            }
        }
        for (int i = 0; i < numHiddenNeurons; i++) {
            previousHiddenToOutputUpdates[i] = 0.0; // Set to zero
            hiddenBiases[i] = range * (rand.nextDouble() - 0.5);
            hiddenToOutputWeights[i] = range * (rand.nextDouble() - 0.5);
        }
        outputBias = (numHiddenNeurons / 2.0) * (2.0 * rand.nextDouble() - 1.0);
    }
    // Forward pass algorithm just as explained before
    public double forwardPass(double[] inputs) {
        double[] hiddenOutputs = new double[numHiddenNeurons];
        for (int i = 0; i < numHiddenNeurons; i++) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; j++) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }

        double output = outputBias;
        for (int i = 0; i < numHiddenNeurons; i++) {
            output += hiddenOutputs[i] * hiddenToOutputWeights[i];
        }
        return sigmoid(output);
    }
    // Activation function
    private double sigmoid(double x) {
        return 1.0 / (1.0 + Math.exp(-x));
    }
    private double sigmoidDerivative(double x) {
        return x * (1.0 - x);
    }
    // Training method as explained before
    public void train(List<double[]> trainingData, int epochs, List<double[]>
validationData) {
        List<String> trainingMSEList = new ArrayList<>();
        List<String> validationMSEList = new ArrayList<>();
        BoldDriverMLP boldDriverMLP = new BoldDriverMLP(numHiddenNeurons,
validationData); // Create an instance of BoldDriverMLP

        for (int epoch = 0; epoch < epochs; epoch++) {
            double totalTrainingError = 0;
            boldDriverMLP.trainEpoch(trainingData, learningRate, epoch);
```

```
            for (double[] row : trainingData) {
                double[] inputs = new double[numInputs];
                System.arraycopy(row, 0, inputs, 0, numInputs);
                double actualOutput = row[numInputs];

                double predictedOutput = forwardPass(inputs);
                double error = actualOutput - predictedOutput;
                totalTrainingError += Math.pow(error, 2);

                // Backward pass and weight update
                backwardPass(inputs, predictedOutput, error, learningRate);
            }
            if (epoch % 100 == 0) {
                // Compute Validation MSE
                double totalValidationMSE = 0;
                for (double[] valRow : validationData) {
                    double[] valInputs = new double[numInputs];
                    System.arraycopy(valRow, 0, valInputs, 0, numInputs);
                    double actualValOutput = valRow[numInputs];
                    double predictedValOutput = forwardPass(valInputs);
                    double error = actualValOutput - predictedValOutput;
                    totalValidationMSE += error * error;
                }
                double validationMSE = totalValidationMSE / validationData.size();
                validationMSEList.add("Epoch:           " + epoch + "
Validation MSE:           " + validationMSE);

                // Compute Training MSE
                double totalTrainingMSE = 0;
                for (double[] trainRow : trainingData) {
                    double[] trainInputs = new double[numInputs];
                    System.arraycopy(trainRow, 0, trainInputs, 0, numInputs);
                    double actualTrainOutput = trainRow[numInputs];
                    double predictedTrainOutput = forwardPass(trainInputs);
                    double error = actualTrainOutput - predictedTrainOutput;
                    totalTrainingMSE += error * error;
                }
                double trainingMSE = totalTrainingMSE / trainingData.size();
                trainingMSEList.add("Epoch:           " + epoch + "            Training
MSE:           " + trainingMSE);
            }
        }
        System.out.println("Training MSEs For Momentum:");
        for (String mse : trainingMSEList) {
            System.out.println(mse);
        }

        // Print validation MSE list
        System.out.println("\nValidation MSEs for Momentum:");
        for (String mse : validationMSEList) {
            System.out.println(mse);
        }
    }
    // train Epoch method to be overidden
    protected void trainEpoch(List<double[]> trainingData, double learningRate, int
epoch) {
        double totalError = 0;
        // Iterate over each data row in the training set
        for (double[] row : trainingData) {
            double[] inputs = Arrays.copyOf(row, numInputs);
            double actualOutput = row[numInputs];
            double predictedOutput = forwardPass(inputs);
            double error = actualOutput - predictedOutput;
            totalError += Math.pow(error, 2);
            backwardPass(inputs, predictedOutput, error, learningRate);
```

```
        }
        double mse = totalError / trainingData.size();
    }
    // backward pass as explained before
    private void backwardPass(double[] inputs, double predictedOutput, double error,
double learningRate) {
        double[] hiddenOutputs = new double[numHiddenNeurons];
        // Calculate hidden layer outputs
        for (int i = 0; i < numHiddenNeurons; ++i) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; ++j) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }
        double deltaOutput = error * sigmoidDerivative(predictedOutput);
        double momentum = 0.9;
        for (int i = 0; i < numHiddenNeurons; ++i) {
            double currentUpdateHO = learningRate * deltaOutput * hiddenOutputs[i];
            double weightChangeHO = hiddenToOutputWeights[i] -
previousHiddenToOutputUpdates[i];
            hiddenToOutputWeights[i] += currentUpdateHO + (momentum * weightChangeHO);
            previousHiddenToOutputUpdates[i] = hiddenToOutputWeights[i];
        }
        outputBias += learningRate * deltaOutput;
        for (int i = 0; i < numHiddenNeurons; ++i) {
            double deltaHidden = deltaOutput * hiddenToOutputWeights[i] *
sigmoidDerivative(hiddenOutputs[i]);

            for (int j = 0; j < numInputs; ++j) {
                double currentUpdateIH = learningRate * deltaHidden * inputs[j];
                double weightChangeIH = inputToHiddenWeights[j][i] -
previousInputToHiddenUpdates[j][i];
                inputToHiddenWeights[j][i] += currentUpdateIH + (momentum *
weightChangeIH);
                previousInputToHiddenUpdates[j][i] = inputToHiddenWeights[j][i];
            }
            hiddenBiases[i] += learningRate * deltaHidden;
        }
    }
}
```

```
import java.util.Arrays;
import java.util.List;
// Extends CatchmentMLP to include advanced training features like annealing and weight
decay.
public class AnnealingMLP extends CatchmentMLP {
    // Constructor initializes AnnealingMLP with a specified number of hidden neurons,
leveraging the base class constructor.
    public AnnealingMLP(int numHiddenNeurons) {
        super(numHiddenNeurons);
    }
    // Overrides train method to implement a two-phase training process: with and
without weight decay.
    @Override
    public void train(List<double[]> trainingData, int epochs, List<double[]>
validationData) {
//          Phase 1 is commented out but can be enabled for training without weight
decay
//         System.out.println("Phase 1: Training with Momentum and Annealing");
//         performTraining(trainingData, epochs, validationData, false);
        // Phase 2: Training with Momentum, Annealing, and Weight Decay
        System.out.println("\nPhase 2: Training with Momentum, Annealing, and Weight
Decay");
        performTraining(trainingData, epochs, validationData, true);
```

```
    }
    // Core training loop adjusting learning rate per epoch and applying weight decay
based on the boolean flag.
    private void performTraining(List<double[]> trainingData, int epochs,
List<double[]> validationData, boolean applyWeightDecay) {
        for (int epoch = 0; epoch < epochs; epoch++) {
            // Adjust learning rate based on the current epoch, simulating annealing
effect.
            double adjustedLearningRate = adjustLearningRate(epoch, epochs);
            // Proceed with a training epoch, leveraging adjusted learning rates.
            super.trainEpoch(trainingData, adjustedLearningRate, epoch);
            // Apply weight decay after the epoch if flagged to do so.
            if (applyWeightDecay) {
                double weightDecayTerm = calculateWeightDecayTerm(adjustedLearningRate,
epoch + 1);
                applyWeightDecay(weightDecayTerm);
            }

            // Periodically print MSEs every 100 epochs or at specific intervals
            if (epoch % 100 == 0 || epoch == 0) {
                printMSEs(epoch, trainingData, validationData);
            }
        }
    }
    // Outputs Mean Squared Error (MSE) for both training and validation datasets at
specified epochs.
    private void printMSEs(int epoch, List<double[]> trainingData, List<double[]>
validationData) {
        double trainingMSE = computeTrainingMSE(trainingData);
        double validationMSE = computeValidationMSE(validationData);
        System.out.println("Epoch:           " + epoch + "             Training MSE:
" + trainingMSE + "          Validation MSE:          " + validationMSE);
    }
    // Computing the validation MSE
    private double computeValidationMSE(List<double[]> validationData) {
        double totalValidationMSE = 0;
        int dataSize = validationData.size();

        for (double[] valRow : validationData) {
            double[] inputs = Arrays.copyOf(valRow, numInputs);
            double actualOutput = valRow[numInputs];
            double predictedOutput = forwardPass(inputs);
            double error = actualOutput - predictedOutput;
            totalValidationMSE += Math.pow(error, 2);
        }
        return totalValidationMSE / dataSize;
    }
    // Sums up all weights and biases to create a checksum, useful for ensuring
stability across training/testing phases.
    public double calculateChecksum() {
        double checksum = 0;
        for (double[] layer : inputToHiddenWeights) {
            for (double weight : layer) {
                checksum += weight;
            }
        }
        for (double weight : hiddenToOutputWeights) {
            checksum += weight;
        }
        for (double bias : hiddenBiases) {
            checksum += bias;
        }
        checksum += outputBias;
        return checksum;
    }
    // Computes Training mse
```

```
    private double computeTrainingMSE(List<double[]> trainingData) {
        double totalTrainingMSE = 0;
        int dataSize = trainingData.size();

        for (double[] trainRow : trainingData) {
            double[] inputs = Arrays.copyOf(trainRow, numInputs);
            double actualOutput = trainRow[numInputs];
            double predictedOutput = forwardPass(inputs);
            double error = actualOutput - predictedOutput;
            totalTrainingMSE += Math.pow(error, 2);
        }
        return totalTrainingMSE / dataSize;
    }
    // Calculates the term for weight decay based on adjusted learning rate and epoch
count, aiming to mitigate overfitting by penalizing large weights.
    private double calculateWeightDecayTerm(double learningRate, int epoch) {
        // 'upsilon' is adjusted to ensure the regularization strength is appropriate
for the current stage of training.
        double upsilon = Math.max(0.001, Math.min(0.1, 1 / (learningRate * epoch)));
        double omega = 0; // Initialize sum of squares of all weights and biases.
        // Sum the squares of weights from input to hidden layer to contribute to
'omega'.
        for (double[] weightsLayer : inputToHiddenWeights) {
            for (double weight : weightsLayer) {
                omega += weight * weight;
            }
        }
        // Add the sum of squares of weights from hidden to output layer to 'omega'.
        for (double weight : hiddenToOutputWeights) {
            omega += weight * weight;
        }
        // Include the sum of squares of all biases in 'omega'.
        for (double bias : hiddenBiases) {
            omega += bias * bias;
        }
        omega += outputBias * outputBias; // Add the square of output bias to 'omega'.

        // Calculate omega by dividing by the total number of weights and biases
        int totalNumberOfWeightsAndBiases = inputToHiddenWeights.length *
inputToHiddenWeights[0].length + hiddenToOutputWeights.length + hiddenBiases.length +
1;
        omega /= (2 * totalNumberOfWeightsAndBiases);
        // The final weight decay term combines 'upsilon' and 'omega'.
        double weightDecayTerm = upsilon * omega;
        return weightDecayTerm; // Return the computed weight decay term.
    }
    // Applies the calculated weight decay term to reduce the magnitude of weights and
biases across the model.
    private void applyWeightDecay(double weightDecayTerm) {
        // Subtract the weight decay term from each weight in the input to hidden layer
matrix.
        for (int i = 0; i < inputToHiddenWeights.length; i++) {
            for (int j = 0; j < inputToHiddenWeights[i].length; j++) {
                inputToHiddenWeights[i][j] -= weightDecayTerm *
inputToHiddenWeights[i][j];
            }
        }
        // Apply weight decay to weights from hidden to output layer.
        for (int i = 0; i < hiddenToOutputWeights.length; i++) {
            hiddenToOutputWeights[i] -= weightDecayTerm * hiddenToOutputWeights[i];
        }
        // Apply weight decay to all biases in the hidden layer.
        for (int i = 0; i < hiddenBiases.length; i++) {
            hiddenBiases[i] -= weightDecayTerm * hiddenBiases[i];
        }
        // Output bias is also adjusted by subtracting the weight decay term.
```

```
        outputBias -= weightDecayTerm * outputBias;
    }
    // Dynamically adjusts the learning rate based on the epoch, employing an annealing
schedule.
    private double adjustLearningRate(int epoch, int totalEpochs) {
        double startLearningRate = 0.3;  // Had to change to make sure my outputs
weren't converging
        double endLearningRate = 0.05;  // Had to change to make sure my outputs
weren't converging
        double x = epoch;
        double r = totalEpochs;
        double adjustedLearningRate = endLearningRate + (startLearningRate -
endLearningRate) * (1 - 1 / (1 + Math.exp(10 - (20 * x / r))));
        adjustedLearningRate = Math.max(0.001, Math.min(0.1, adjustedLearningRate));
        return adjustedLearningRate;
    }
}
```

Appendix E: Basic Back Propagation + Momentum + Bold Driver + Weight Decay

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;


public class Main {
    private static final String TRAINING_SET_PATH = "trainingSet1.csv"; // Update this
with the path to your CSV file
    private static final String VALIDATION_SET_PATH = "validationSet.csv";
    public static final String TEST_SET_PATH ="testSet.csv";

    public static void main(String[] args) {
        Scanner scanner= new Scanner(System.in);
        System.out.println("Enter the number of epochs:");
        int epochs= scanner.nextInt();
        System.out.println("Enter the number of hidden neurons:");
        int numHiddenNeurons= scanner.nextInt();

        // List to hold training data and validation data
        List<double[]> trainingData = loadData(TRAINING_SET_PATH);
        List<double[]> validationData = loadData(VALIDATION_SET_PATH);
        List<double[]> testData = loadData(TEST_SET_PATH);

        CatchmentMLP mlp = new CatchmentMLP(numHiddenNeurons);
        // Train the model and validate for a specified number of epochs
        mlp.train(trainingData, epochs, validationData);
        AnnealingMLP annealingMLP = new AnnealingMLP(numHiddenNeurons);
        annealingMLP.train(trainingData, epochs, validationData);

        System.out.println("\nComparisons:");
        for (double[] dataRow : trainingData) {
            double[] inputs = Arrays.copyOf(dataRow, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];

            double predictedOutputMomentum = mlp.forwardPass(inputs);
            double predictedOutputAnnealing = annealingMLP.forwardPass(inputs);

            System.out.println("Actual Output:      " + actualOutput +
                    "            With Momentum:        " + predictedOutputMomentum +
                    "            With Annealing:       " + predictedOutputAnnealing);
        }
    }
    // Method to remove UTF-8 Byte Order Mark (BOM) from a string.
    private static String removeUtf8Bom(String s) {
        if (s.startsWith("\uFEFF")) {
            s = s.substring(1);
        }
        return s;
    }
    // Loading data method
    static List<double[]> loadData(String filePath) {
        List<double[]> data = new ArrayList<>();
        try (BufferedReader br = Files.newBufferedReader(Paths.get(filePath),
StandardCharsets.UTF_8)) {
            String line;
            while ((line = br.readLine()) != null) {
```

```
                line = removeUtf8Bom(line);
                String[] values = line.split(","); // Make sure to use the correct
delimiter
                double[] doubleValues = new double[values.length];
                for (int i = 0; i < values.length; i++) {
                    doubleValues[i] = Double.parseDouble(values[i]);
                }
                data.add(doubleValues);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return data;
    }

}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;
import java.util.List;
// Parent class to be extended
public class CatchmentMLP {
    // as commented before everything stays constant
    public double learningRate = 0.1;
    public double[][] inputToHiddenWeights;
    private double[][] previousInputToHiddenUpdates; // For momentum
    public double[] hiddenBiases;
    public double[] hiddenToOutputWeights;
    private double[] previousHiddenToOutputUpdates; // For momentum
    public double outputBias;
    public static final int numInputs = 8; // Use all 8 inputs
    private Random rand = new Random();
    public final int numHiddenNeurons; // This is entered by the user

    public static final String TEST_SET_PATH ="testSet.csv";


    public CatchmentMLP(int numHiddenNeurons) {
        this.numHiddenNeurons = numHiddenNeurons; // Set the number of hidden neurons
using the User's input
        initializeWeightsAndBiases();
    }

    // Getter for number of inputs
    public static int getNumInputs() {
        return numInputs;
    }
    // Initializes weights and biases with random values to start training from a non-
uniform initial state.
    private void initializeWeightsAndBiases() {
        double range = 4.0 / numInputs;
        previousInputToHiddenUpdates = new double[numInputs][numHiddenNeurons];
        previousHiddenToOutputUpdates = new double[numHiddenNeurons];
        // Initialize weights and biases for each input neuron
        inputToHiddenWeights = new double[numInputs][numHiddenNeurons];
        hiddenBiases = new double[numHiddenNeurons];
        hiddenToOutputWeights = new double[numHiddenNeurons];
        for (int i = 0; i < numInputs; i++) {
            for (int j = 0; j < numHiddenNeurons; j++) {
                previousInputToHiddenUpdates[i][j] = 0.0; // Set to zero
                inputToHiddenWeights[i][j] = range * (rand.nextDouble() - 0.5);
            }
        }
        for (int i = 0; i < numHiddenNeurons; i++) {
```

```
            previousHiddenToOutputUpdates[i] = 0.0; // Set to zero
            hiddenBiases[i] = range * (rand.nextDouble() - 0.5);
            hiddenToOutputWeights[i] = range * (rand.nextDouble() - 0.5);
        }
        outputBias = (numHiddenNeurons / 2.0) * (2.0 * rand.nextDouble() - 1.0);

    }
    // Forward pass algorithm just as explained before
    public double forwardPass(double[] inputs) {
        double[] hiddenOutputs = new double[numHiddenNeurons];
        for (int i = 0; i < numHiddenNeurons; i++) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; j++) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }

        double output = outputBias;
        for (int i = 0; i < numHiddenNeurons; i++) {
            output += hiddenOutputs[i] * hiddenToOutputWeights[i];
        }
        return sigmoid(output);
    }
    // Activation function
    private double sigmoid(double x) {
        return 1.0 / (1.0 + Math.exp(-x));
    }
    private double sigmoidDerivative(double x) {
        return x * (1.0 - x);
    }
    // Training method as explained before
    public void train(List<double[]> trainingData, int epochs, List<double[]>
validationData) {
        // Instantiate the Bold Driver class  with weight decay
        BoldDriverMLP modelWithWeightDecay = new BoldDriverMLP(numHiddenNeurons,
validationData, true);
        System.out.println("Training with Weight Decay:");
        trainModel(modelWithWeightDecay, trainingData, epochs, validationData);

        System.out.println("\nActual vs Predicted Outputs (With Weight Decay):");
        for (int i = 0; i < modelWithWeightDecay.actualOutputs.size(); i++) {
            System.out.println("Actual: " + modelWithWeightDecay.actualOutputs.get(i) +
", Predicted: " + modelWithWeightDecay.predictedOutputs.get(i));
        }

        // Instance without weight decay
        BoldDriverMLP modelWithoutWeightDecay = new BoldDriverMLP(numHiddenNeurons,
validationData, false);
        System.out.println("\nTraining without Weight Decay:");
        trainModel(modelWithoutWeightDecay, trainingData, epochs, validationData);

        System.out.println("\nActual vs Predicted Outputs (Without Weight Decay):");
        for (int i = 0; i < modelWithoutWeightDecay.actualOutputs.size(); i++) {
            System.out.println("Actual: " +
modelWithoutWeightDecay.actualOutputs.get(i) + ", Predicted: " +
modelWithoutWeightDecay.predictedOutputs.get(i));
        }
        List<double[]> testData = Main.loadData(TEST_SET_PATH);

        System.out.println("\nComparisons on Test Data final final:");
        for (double[] dataRow : testData) {
            double[] inputs = Arrays.copyOf(dataRow, CatchmentMLP.getNumInputs());
            double actualOutput = dataRow[CatchmentMLP.getNumInputs()];

            double predictedOutputWeightDecay =
```

```
modelWithWeightDecay.forwardPass(inputs);

            System.out.println("Actual Output:       " + actualOutput+
                    "              With weight decay:       " +
predictedOutputWeightDecay);
        }


    }
    private void trainModel(BoldDriverMLP model, List<double[]> trainingData, int
epochs, List<double[]> validationData) {
        for (int epoch = 0; epoch < epochs; epoch++) {
            model.trainEpoch(trainingData, model.learningRate, epoch); // Access
learningRate through the model instance

            if (epoch % 1000 == 0) {
                double trainingMSE = model.calculateMSE(trainingData);
                double validationMSE = model.calculateValidationMSE();
            }
        }
    }
    // train Epoch method to be overidden
    protected void trainEpoch(List<double[]> trainingData, double learningRate, int
epoch) {
        double totalError = 0;


        // Iterate over each data row in the training set
        for (double[] row : trainingData) {
            double[] inputs = Arrays.copyOf(row, numInputs); // Extract input features
            double actualOutput = row[numInputs]; // Extract actual output

            double predictedOutput = forwardPass(inputs); // Get model's prediction
            double error = actualOutput - predictedOutput; // Calculate error

            totalError += Math.pow(error, 2); // Accumulate squared error

            backwardPass(inputs, predictedOutput, error, learningRate); // Perform
backpropagation
        }

        double mse = totalError / trainingData.size(); // Calculate mean squared error
for the epoch
    }

    // backward pass as explained before
    private void backwardPass(double[] inputs, double predictedOutput, double error,
double learningRate) {
        double[] hiddenOutputs = new double[numHiddenNeurons];
        for (int i = 0; i < numHiddenNeurons; ++i) {
            hiddenOutputs[i] = hiddenBiases[i];
            for (int j = 0; j < numInputs; ++j) {
                hiddenOutputs[i] += inputs[j] * inputToHiddenWeights[j][i];
            }
            hiddenOutputs[i] = sigmoid(hiddenOutputs[i]);
        }
        double deltaOutput = error * sigmoidDerivative(predictedOutput);
        double momentum = 0.9;
        for (int i = 0; i < numHiddenNeurons; ++i) {
            double currentUpdateHO = learningRate * deltaOutput * hiddenOutputs[i];
            double weightChangeHO = hiddenToOutputWeights[i] -
previousHiddenToOutputUpdates[i];
            hiddenToOutputWeights[i] += currentUpdateHO + (momentum * weightChangeHO);
            previousHiddenToOutputUpdates[i] = hiddenToOutputWeights[i];
        }
        outputBias += learningRate * deltaOutput;
```

```
        for (int i = 0; i < numHiddenNeurons; ++i) {
            double deltaHidden = deltaOutput * hiddenToOutputWeights[i] *
sigmoidDerivative(hiddenOutputs[i]);

            for (int j = 0; j < numInputs; ++j) {
                double currentUpdateIH = learningRate * deltaHidden * inputs[j];
                double weightChangeIH = inputToHiddenWeights[j][i] -
previousInputToHiddenUpdates[j][i];
                inputToHiddenWeights[j][i] += currentUpdateIH + (momentum *
weightChangeIH);
                previousInputToHiddenUpdates[j][i] = inputToHiddenWeights[j][i];
            }

            hiddenBiases[i] += learningRate * deltaHidden;
        }
    }
}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
// Extends the CatchmentMLP class to implement bold driver strategy for learning rate
adjustment and optionally applies weight decay.
public class BoldDriverMLP extends CatchmentMLP {
    private boolean applyWeightDecay; // Indicates whether weight decay regularization
is applied.
    // Stores weights and biases before update for possible rollback in bold driver
adjustment.
    private final double[][] inputToHiddenWeightsBeforeUpdate;
    private final double[] hiddenBiasesBeforeUpdate;
    private final double[] hiddenToOutputWeightsBeforeUpdate;
    private double outputBiasBeforeUpdate;
    private double previousMSE = Double.MAX_VALUE; // Stores the MSE from the previous
training epoch for comparison.
    private final List<Double> trainingMSEList = new ArrayList<>();
    private final List<Double> validationMSEList = new ArrayList<>();
    List<Double> actualOutputs = new ArrayList<>();
    List<Double> predictedOutputs = new ArrayList<>();
    private final List<double[]> validationData; // Store validation data
    // Constructor initializes the MLP with given settings, including the option for
weight decay.
    public BoldDriverMLP(int numHiddenNeurons,List<double[]> validationData,boolean
applyWeightDecay) {
        super(numHiddenNeurons);
        // Pre-allocate memory for storing weights and biases before updates.
        inputToHiddenWeightsBeforeUpdate = new double[numInputs][numHiddenNeurons];
        hiddenBiasesBeforeUpdate = new double[numHiddenNeurons];
        hiddenToOutputWeightsBeforeUpdate = new double[numHiddenNeurons];
        this.validationData = validationData;
        this.applyWeightDecay=applyWeightDecay;
    }
    // Overridden trainEpoch method includes mechanisms for bold driver adjustments and
weight decay application.
    @Override
    protected void trainEpoch(List<double[]> trainingData, double learningRate, int
epoch) {
        // Store the weights and biases before update
        for (int i = 0; i < numInputs; i++) {
            // Backup current weights and biases before the epoch begins.
            System.arraycopy(inputToHiddenWeights[i], 0,
inputToHiddenWeightsBeforeUpdate[i], 0, numHiddenNeurons);
        }
        System.arraycopy(hiddenBiases, 0, hiddenBiasesBeforeUpdate, 0,
numHiddenNeurons);
        System.arraycopy(hiddenToOutputWeights, 0, hiddenToOutputWeightsBeforeUpdate,
```

```
0, numHiddenNeurons);
        outputBiasBeforeUpdate = outputBias;
        super.trainEpoch(trainingData, learningRate, epoch);
        // After training, calculate the MSE and adjust learning rate
        double currentMSE = calculateMSE(trainingData);
        double currentValidationMSE = calculateValidationMSE();
        trainingMSEList.add(currentMSE);
        validationMSEList.add(currentValidationMSE);
        adjustLearningRate(currentMSE, epoch);
        // If weight decay is enabled & it's not the first epoch, apply weight decay.
        if (applyWeightDecay && epoch > 0) {
            double weightDecayTerm = calculateWeightDecayTerm(learningRate, epoch);
            applyWeightDecay(weightDecayTerm); // Apply weight decay.
        }
        if (epoch % 100 == 0) {
            System.out.printf("Epoch: %d,                              Learning Rate:
%.4f                                  Training MSE:                          %.4f
Validation MSE:                                        %.4f\n",
                    epoch, learningRate, currentMSE, currentValidationMSE);
        }else {
            // Adjust learning rate every 2000 epochs
            adjustLearningRate(currentMSE, epoch);
        }
        // After the training logic, populate the lists
        actualOutputs.clear(); // Clear the lists at the beginning of each epoch to
avoid duplicate entries
        predictedOutputs.clear();
        for (double[] dataRow : trainingData) {
            double[] inputs = Arrays.copyOf(dataRow, numInputs);
            double actualOutput = dataRow[numInputs];
            double predictedOutput = forwardPass(inputs);
            actualOutputs.add(actualOutput);
            predictedOutputs.add(predictedOutput);
        }
    }
    // Calculate the weight decay term based on the learning rate and epoch number.
    private double calculateWeightDecayTerm(double learningRate, int epoch) {
        // Initialize sum of squares of all weights and biases
        double omega = 0;
        // Sum the squares of weights from input to hidden layer to contribute to
'omega'.
        for (double[] layer : inputToHiddenWeights) {
            for (double weight : layer) {
                omega += weight * weight;
            }
        }
        // Add the sum of squares of weights from hidden to output layer to 'omega'.
        for (double weight : hiddenToOutputWeights) {
            omega += weight * weight;
        }
        // Calculate omega by dividing by the total number of weights and biases
        omega /= (2 * (inputToHiddenWeights.length * inputToHiddenWeights[0].length +
hiddenToOutputWeights.length + hiddenBiases.length + 1));
        // 'upsilon' is adjusted to ensure the regularization strength is appropriate
for the current stage of training.
        double upsilon = 1.0 / (learningRate * epoch); // Calculate upsilon
        upsilon = Math.max(0.001, Math.min(0.1, upsilon)); // Clamp upsilon to be
within the typical range
        return upsilon * omega; // Return the weight decay term
    }
    // Applies the calculated weight decay term to reduce the magnitude of weights and
biases across the model.
    private void applyWeightDecay(double weightDecayTerm) {
        // Apply weight decay to all weights
        // Subtract the weight decay term from each weight in the input to hidden layer
matrix.
```

```
        for (int i = 0; i < inputToHiddenWeights.length; i++) {
            for (int j = 0; j < inputToHiddenWeights[i].length; j++) {
                inputToHiddenWeights[i][j] -= weightDecayTerm *
inputToHiddenWeights[i][j];
            }
        }
        // Apply weight decay to weights from hidden to output layer.
        for (int i = 0; i < hiddenToOutputWeights.length; i++) {
            hiddenToOutputWeights[i] -= weightDecayTerm * hiddenToOutputWeights[i];
        }
        // Apply weight decay to all biases in the hidden layer.
        for (int i = 0; i < hiddenBiases.length; i++) {
            hiddenBiases[i] -= weightDecayTerm * hiddenBiases[i];
        }
        // Output bias is also adjusted by subtracting the weight decay term.
        outputBias -= weightDecayTerm * outputBias;
    }
    double calculateMSE(List<double[]> data) {
        double totalError = 0;
        for (double[] row : data) {
            double[] inputs = Arrays.copyOf(row, numInputs);
            double actualOutput = row[numInputs];
            double predictedOutput = forwardPass(inputs);
            double error = actualOutput - predictedOutput;
            totalError += Math.pow(error, 2);
        }
        return totalError / data.size();
    }
    double calculateValidationMSE() {
        return calculateMSE(validationData); // Calculate MSE using validation data
    }
    // Adjusts the learning rate based on comparison of current MSE to previous MSE,
implementing the Bold Driver strategy.
    private void adjustLearningRate(double currentMSE, int epoch) {
        // Adjust learning rate only every 1000 epochs
        if (epoch % 1000 == 0 && epoch > 0) {
            if (currentMSE > previousMSE * 1.04) {
                // Factor to decrease the learning rate if performance worsens
                this.learningRate *= 0.7;
                undoWeightUpdates(); // Undo the last weight updates because the error
increased
            } else if (currentMSE < previousMSE) {
                // Factor to increase the learning rate if performance improves
                this.learningRate *= 1.05;
            }
            // Minimum boundary for the learning rate and Maximum Boundary
            this.learningRate = Math.max(0.01, Math.min(this.learningRate, 0.5));
            previousMSE = currentMSE;
        }
    }
    // Reverts weights and biases to their previous state before the last update if the
MSE increases.
    private void undoWeightUpdates() {
        for (int i = 0; i < numInputs; i++) {
            System.arraycopy(inputToHiddenWeightsBeforeUpdate[i], 0,
inputToHiddenWeights[i], 0, numHiddenNeurons);
        }
        System.arraycopy(hiddenBiasesBeforeUpdate, 0, hiddenBiases, 0,
numHiddenNeurons);
        System.arraycopy(hiddenToOutputWeightsBeforeUpdate, 0, hiddenToOutputWeights,
0, numHiddenNeurons);
        outputBias = outputBiasBeforeUpdate;
    }
}
```