PREDICTING MAPPING PENALTIES WITH A ‘FROM-SCRATCH’ ARTIFICIAL NEURAL NETWORK (ANN)

Report

Saf Flatters

Bachelor of Science
Major: Data Science
Minors: Geospatial Tech, Artificial Intelligence

Contents

Introduction 2
Methodology 3
Data Description 3
Preprocessing & Encoding 3
Model Architectures 4
Hyperparameter Tuning 4
Training Procedure 5
Results 7
Comparison Graphs 8
Epoch Vs Loss 8
Learning Rate Vs Loss 9
Activation Function vs Loss 9
Batch Size vs Epoch Time 10
Discussion & Conclusion 10
References 12
Appendix 13
Sample Mappings 13
Predicted vs True Summary Table 14

Introduction

Artificial Neural Networks (ANN) are powerful tools for predictive modelling as they are able to learn complex non-linear relationships from structured data (Mirjalili, 2019). In this assignment, a Feed-Forward ANN was implemented from scratch (no libraries) to predict the penalty score of a mapping between 10 tasks assigned to five employees. Each mapping's penalty score reflected how well the task-to-employee assignments met five given constraints, with lower penalty indicating a better solution. The Feed-Forward ANN was able to predict the penalty score without the constraints calculation explicitly given to it.

The Python Code notebook **'ANN_from_scratch_Code.ipynb'** pre-processed and encoded 100 imported mappings (from **'population.csv'**) and converted each mapping into a 110-dimensional input vector. These input vectors along with their associated penalty score were split into training, validation and test sets. Two ANN shape models with different amounts of hidden layers and neurons within those layers were constructed for comparison. Each model was trained using mini-batch gradient descent. For each batch, the model performed Forward Propagation to compute predicted penalties, followed by Back Propagation to compute loss gradients. The parameters (weights and biases) were then updated using gradient descent. The chosen non-linear Activation Functions were applied in the hidden layers.

Prior to training the training set, Hyperparameter tuning was employed via grid search to find the optimal learning rate used in the Back Propagation, the type of Activation function used in the hidden layers (Sigmoid or ReLU) and the batch size.

Model performance was evaluated using Mean Squared Error (MSE) on the validation set each iteration and then finally the best performing model was evaluated (with MSE) on the test set. Validation loss was used to guide the selection of optimal hyperparameters and parameters (weights and biases). Using the validation loss metric helped ensure better generalisation to unseen data and reduced overfitting.

Visualisations were generated to compare model performance and computational efficiency across training epochs, activation function types, learning rates and batch sizes.

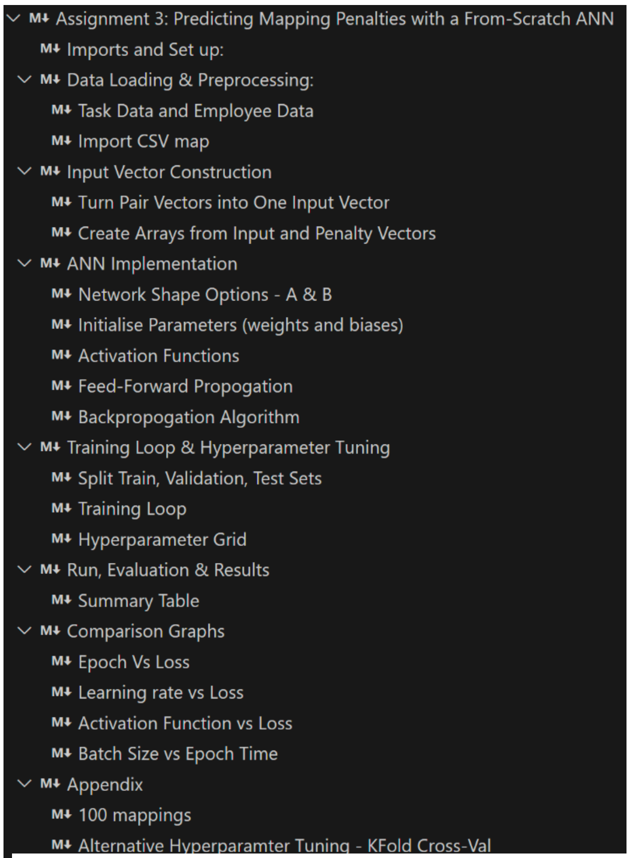

- Assignment 3: Predicting Mapping Penalties with a From-Scratch ANN
  - Imports and Set up:
  - Data Loading & Preprocessing:
    - Task Data and Employee Data
    - Import CSV map
  - Input Vector Construction
    - Turn Pair Vectors into One Input Vector
    - Create Arrays from Input and Penalty Vectors
  - ANN Implementation
    - Network Shape Options - A & B
    - Initialise Parameters (weights and biases)
    - Activation Functions
    - Feed-Forward Propogation
    - Backpropogation Algorithm
  - Training Loop & Hyperparameter Tuning
    - Split Train, Validation, Test Sets
    - Training Loop
    - Hyperparameter Grid
  - Run, Evaluation & Results
    - Summary Table
  - Comparison Graphs
    - Epoch Vs Loss
    - Learning rate vs Loss
    - Activation Function vs Loss
    - Batch Size vs Epoch Time
  - Appendix
    - 100 mappings
    - Alternative Hyperparamter Tuning - KFold Cross-Val

Figure 1: Code Notebook Outline

Methodology

Data Description

Following on from a previous project, a modified version of the code was created to generate 100 chromosomes. Each chromosome represented a random potential solution to assigning 10 tasks among 5 employees. For example, Chromosome [5, 3, 2, 3, 5, 3, 1, 3, 3, 4] meant that the Employee 5 was assigned to Task 1, Employee 3 was assigned to Task 2, Employee 2 was assigned to Task 3 and so on. The modified code also evaluated the fitness of each chromosome by calculating penalties for overload, skill mismatch, difficulty and deadline violations and unique task assignments. This penalty score was coupled with the chromosome in a tuple. 100 tuples were outputted as a Pandas dataframe in 'population.csv'. This csv file was imported into this project using the loadPopulationCSV() function in Code Notebook section: ***Data Loading and Preprocessing*** along with the supplied Task and Employee data in the chunk above.

Preprocessing & Encoding

To prepare each chromosome for training the ANN, each task-assignment pair (or genome) in the chromosome was transformed into an 11-dimensional numeric vector in the Code Notebook section: ***Input Vector Construction*** with the createVectors() function. The first 6 elements of the input vector represented Task Features - time, difficulty, deadline and skill. The next 5 elements of the input vector represented Employee features - hours, skill_level and skills. Both skill and skills with "A", "B" and "C" represented as binary values. For example, [1, 0, 0] for Employee skills indicated an employee possessing skill "A" but not "B" or "C".

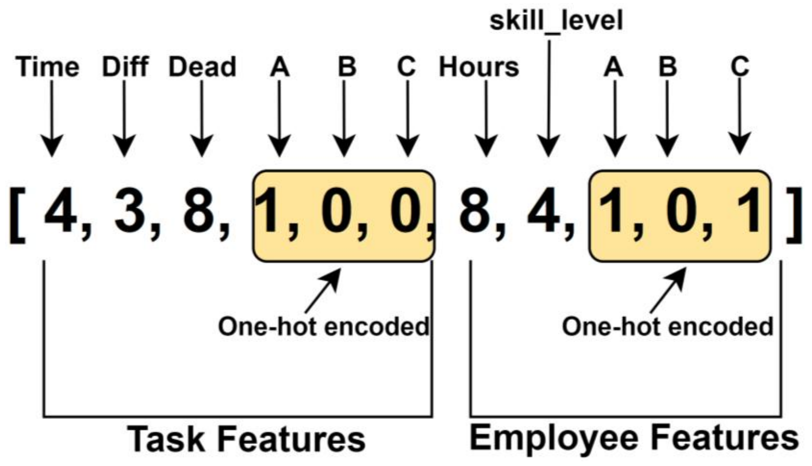


Figure 2: Example of 11-dimensional task-assignment pair

Each pairing vector was concatenated into a 110-dimensional input vector to represent the entire chromosome. These 100 chromosomes were then represented by their corresponding input vectors, forming a 100 x 110 matrix. This matrix was converted into an Numpy array named "X". The penalty associated with each input vector (inherited from the chromosome) was stored in a 100 x 1 array named "y" following common practices for supervised machine learning, where "X" represents the feature matrix and "y" represents the target vector.

The "X" and "y" arrays were split into three with the splitTrainValTestSets() function. The first 70 rows became the training set, the next 15 became the validation set and the final 15 became the test set.

Model Architectures

Weights and biases are critical components of ANNs, as optimising them allows the network to learn and fit to data. The architecture of the network determines the number of weights (which control the strength of connections between neurons) and biases, both necessary for modelling complex relationships (Starmer, n.d.). The number of weights between two layers is determined by the product of the number of neurons in those layers, and each neuron has one bias term.

Two ANN architectures were modelled in this project (specified in function networkShape()):

- Shape Model A had the shape [110, 256, 1], where 110 input features were passed into a hidden layer with 256 neurons. This resulted in 28,160 weights (W1) and 256 biases (b1) for the hidden layer, along with 256 weights (W2) and 1 bias (b2) for the output layer.
- Shape Model B had the shape [110, 128, 128, 1], where 110 input features were passed into two hidden layers, each with 128 neurons, resulting in 14,080 weights (W1) and 128 biases (b1) for the first hidden layer, 16,384 weights (W2) and 128 biases (b2) for the second hidden layer, and 128 weights (W3) and 1 bias (b3) for the output layer.

Hyperparameter Tuning

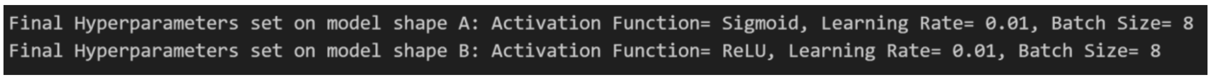
Before training the ANN, key hyperparameters were chosen to ensure optimal results were achieved. Each model (Shape A and B) underwent Hyperparameter tuning in the tuneHyper() function in the ***Training Loop & Hyperparameter Tuning*** section of the Notebook. A manual grid search was performed, iterating over combinations of key hyperparameters.

These included:

- Learning rates (0.01, 0.001, 0.0001) – which controls how quickly a model adapts to the problem, used inside the backwardPropogation() function.
- Batch sizes (8, 16, 32) – which determines how many input vectors are processed together before the model updates its weights, used inside the trainEpoch() function
- Activation types ("ReLU", "Sigmoid") – which impacts how the ANN captures non-linear relationships, used inside the feedforward() function.

Each combination was trained for 200 epochs. The minimum validation loss achieved during training for each hyperparameter combination was recorded. The minimum overall validation loss for each model shape was then used to select the best hyperparameters by returning the configuration that resulted in that lowest validation loss overall.

The hyperparameters selected by this process was:



```
Final Hyperparameters set on model shape A: Activation Function= Sigmoid, Learning Rate= 0.01, Batch Size= 8
Final Hyperparameters set on model shape B: Activation Function= ReLU, Learning Rate= 0.01, Batch Size= 8
```

The hyperparameters selected for learning rate and batch size were the same across both models – however, the activation function selected was different. This may be due to Model Shape B being a more complex model and therefore requiring the advantages ReLU has over Sigmoid. This includes the ability to alleviate the vanishing gradient problem (finding local optimum instead of global optimum) (Brownlee, 2020).

Note: In the appendix of the code Notebook – I have written an alternative to the hyperparameter tuning function called tuneHyper_KFold(). This function required sklearn.model_selection library which I was not sure we were allowed so I left it out and wrote a manual grid search function instead.

Training Procedure

1. **Initialise parameters**

Once the optimal hyperparameters were chosen, the starterParameters() function was deployed to initialise the weights and biases for each layer of the ANN – this is called from trainingLoop(). The initial parameters are randomly chosen small numbers and inputted into arrays of the sizes mentioned in the above Data Description section of this report.

2. **Shuffle Data (each Epoch)**

For each epoch, the training data (both X and y) are shuffled, to ensure each batch is a random sample of the dataset and prevents the ANN memorising a sequence pattern in the data.

3. **Split into Batches (each Epoch)**

The training data is then partitioned into mini batches of size selected by the hyperparameter tuning prior to this loop.

4. **Forward Propagation (each Epoch)**

The training data is passed through the layers of the ANN (using the feedforward() function in the ***ANN Implementation*** section of the Notebook) and using the weights and biases found at the previous epoch via Backwards Propagation. Each layer in the network involves a linear transformation followed by an activation function (ReLU or Sigmoid) in all hidden layer.

$$z_i = W_i \cdot a_{i-1} + b_i$$

$$a_i = f(z_i)$$

Where $W_i$and b_i are the weights and biases of layer i , f is the activation function and $a_{i-1}$is the output from the previous layer.

All layers except the output layer includes the activation function. The activation function found by hyperparameter tuning to learn complex non-linear patterns can be either ReLU or Sigmoid in this ANN. Sigmoid is

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

and ReLU is

$$ReLU(x) = max(0, x)$$

so if the x is negative – ReLU returns 0, otherwise it returns x .

The feedforward() function will return the parameter cache (final layer predictions, weights and biases).

5. **Backwards Propagation (each Epoch)**

The training data and parameter cache are then passed to the backwardsPropogation() function which computes the gradients required to update the networks weights and biases. First this function determines the error (dZ) at the output layer by subtracting the actual penalties from the final layer predictions,

$$dZ = \hat{y} - y.$$

Then, passing in reverse through the network layers, the gradients are computed:

- dW (the gradient with respect to weights) is the dot product of the error (dZ) and the activations (outputs) from the previous layer divided by the batch size (for normalisation).
$$dW = \frac{dZ \cdot A_{previous}}{n_{batch}}$$
- The second gradient db is the sum of the errors (dZ) divided by batch size (for normalisation).

Using the chain rule, in the hidden layers the error is propagated backwards by multiplying the current layers error by weights and the derivative of the activation function used in the previous layer:

$$dZ_{previouslayer} = dA_{previouslayer} \cdot f'(z).$$

Then to update the parameters by gradient descent, it is the current weight or bias minus the gradient with respect to weight or bias times the chosen learning rate. The backwardsPropagation() function outputs these weights and biases.

6. **Feed Forward again (each Epoch)**

The feedForward() function now uses the new found parameters to run through the layers.

7. **Calculate Training and Validation loss (each Epoch)**

Using the meanSquaredError() function, the loss is evaluated between the predicted penalties returned by the final Feed Forward of the Epoch and the actual penalties. The loss calculation is

$$loss(MSE) = \sum (y - \hat{y})^2$$

The validation set is then passed through the ANN model using feedForward() once to determine how well the updated weights and biases would work on unseen data. The mean squared error loss is calculated and this is what ultimately selects the final model parameters.

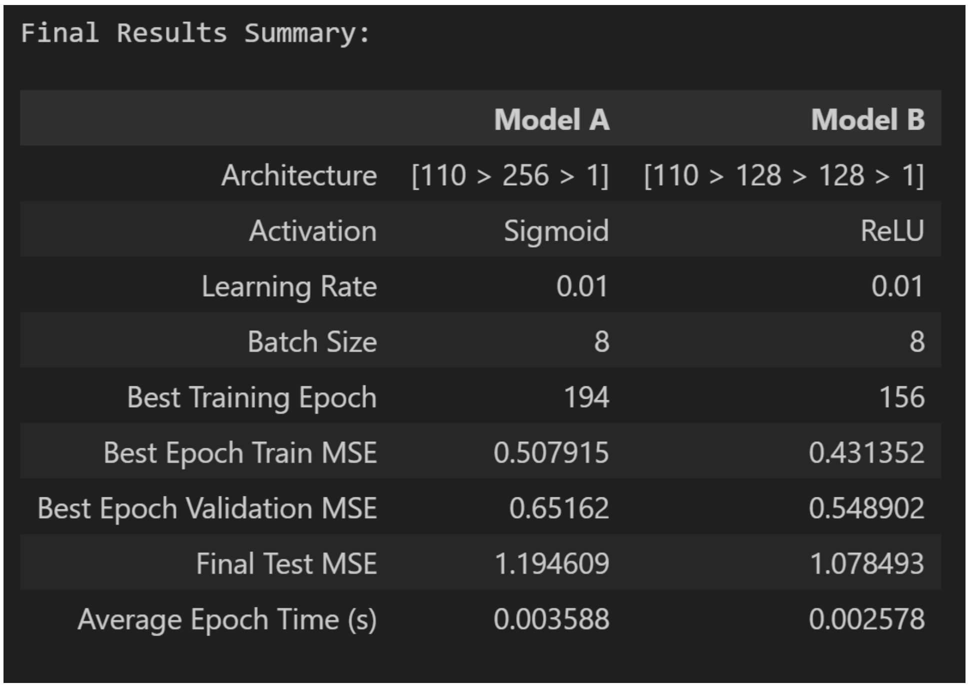
8. **Optional Early Stopping**

Originally, I was testing 10,000 epochs. Due to the excessive computational time – I implemented early stopping capabilities. This stops the search for the lowest validation loss if the results have not improved for 150 epochs. For the sake of this assignment and 100-200 epochs being specified in the Assignment brief, I have kept this early stopping algorithm in the Training Loop in case I am to present the project running for greater iterations.

Results

The final models were evaluated using the test set in the ***Run, Evaluation & Results*** section of the Notebook. For each model, the best parameters found during training were used in a forward pass with the test set data to generate penalty predictions. These predictions were then compared to the actual penalties, and the mean squared error (MSE) was calculated.

Figure 3 is a summary output of the results (for maximum epochs set to 200). Model B, which featured two hidden layers and used the ReLU activation function, outperformed Model A in all metrics. Model B achieved a lower best validation MSE (0.5489 vs. 0.6516) and a lower final test MSE (1.0785 vs. 1.1946), showing improved generalisation to unseen data. Model B also trained slightly faster per epoch on average. These results suggest that a deeper network with ReLU activation is better suited to this task than a shallower network with Sigmoid activation.



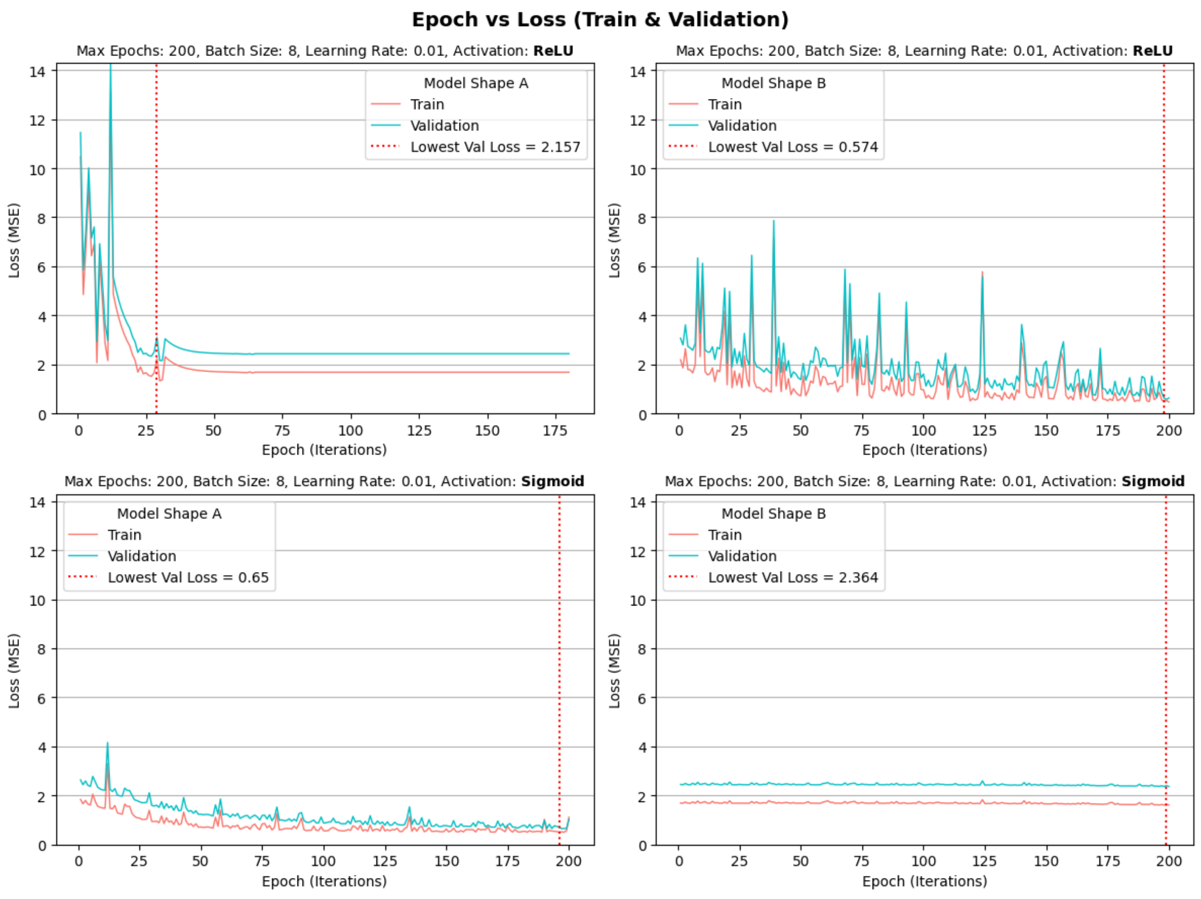
```
Final Results Summary:
```

	Model A	Model B
Architecture	[110 > 256 > 1]	[110 > 128 > 128 > 1]
Activation	Sigmoid	ReLU
Learning Rate	0.01	0.01
Batch Size	8	8
Best Training Epoch	194	156
Best Epoch Train MSE	0.507915	0.431352
Best Epoch Validation MSE	0.65162	0.548902
Final Test MSE	1.194609	1.078493
Average Epoch Time (s)	0.003588	0.002578

Figure 3: Summary Table from Notebook

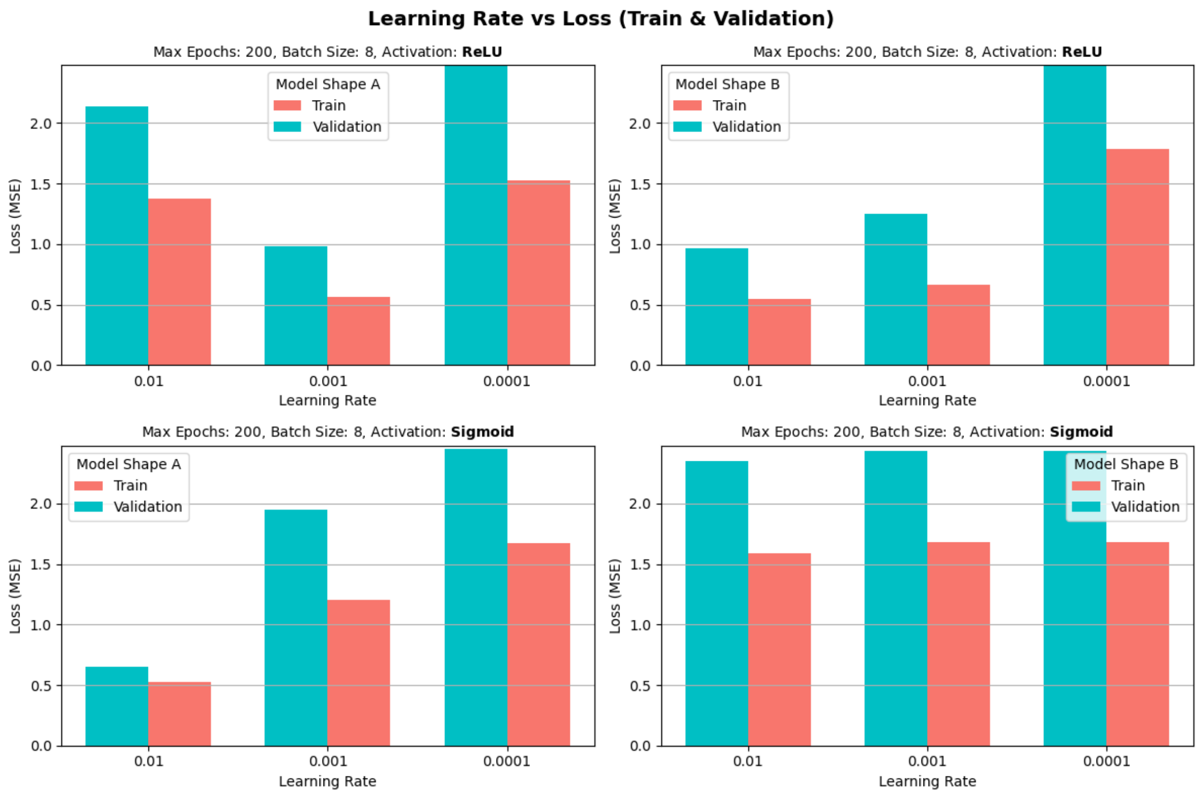
Comparison Graphs

Epoch Vs Loss



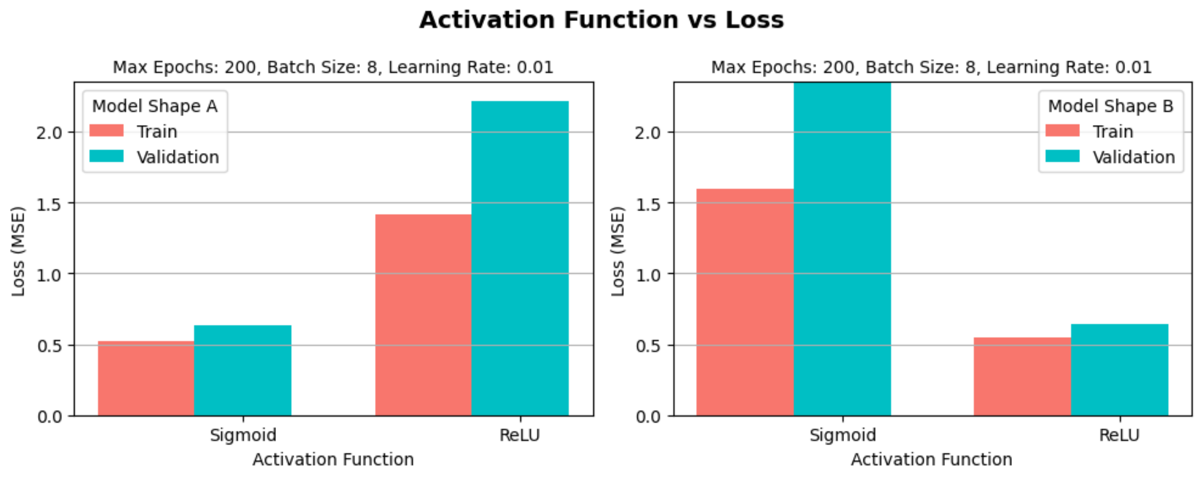
In ***Comparison Graphs*** – Epoch vs Loss plot shows Model A on the left two plots and Model B on the right two plots. The top two plots use the ReLU function and the bottom two plots use the Sigmoid function. Each subplot displays both training and validation loss across epochs for the corresponding model and activation function. There is significantly more variance in the loss history of models employing the ReLU function than the Sigmoid function. This is due to ReLU not being bounded between 0 and 1, allowing weights to be larger and more unpredictable. It is clear why Hyperparameter tuning chose Sigmoid for Model A and ReLU for Model B as the activation function greatly affected the lowest validation result per epoch recorded. None of the four models showed evidence of overfitting within 200 epochs. Overfitting would show training loss dropping while validation loss rising. Not seeing this indicates a good match of network shape and activation type for both models.

Learning Rate Vs Loss



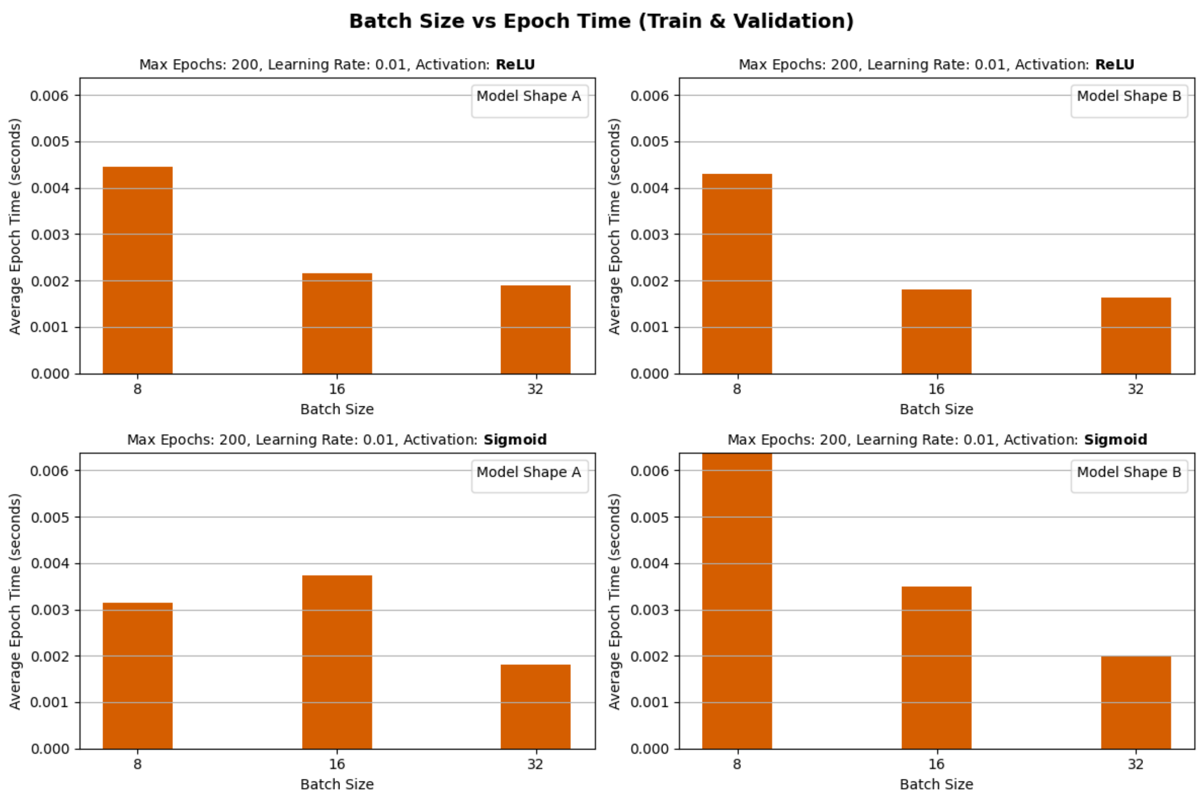
This plot compared the same four models as the last graph but showed how the learning rate used in the Backpropagation function affected MSE. Once again, red represents the training loss and teal represents the validation loss. In both of the models that were selected during hyperparameter tuning - Model A: Sigmoid (bottom left) and Model B: ReLU (top right) – 0.01 Learning Rate had a clear advantage. As for ReLU with Model A showed 0.001 Learning Rate to be better and no discernible difference between Learning Rates in Model B: Sigmoid. These results highlight the importance of hyperparameter tuning for each network shape.

Activation Function vs Loss



The Activation Function vs Loss plot reinforces the hyperparameter tuning selection by showing a significant difference between Sigmoid and ReLU for each model. As for the other activation functions (not chosen for each model) – this plot shows evidence of overfitting, seen by the gap between Train and Validation loss. This reinforces the selection of Sigmoid for Model A and ReLU for Model B.

Batch Size vs Epoch Time



This plot demonstrates the relationship between batch size and training speed. As the batch size gets larger, the average Epoch time (in seconds) decreases for both models. This is because processing large batches is faster. Time was not a factor in hyperparameter tuning so this was not taken into account when tuning. Both models ultimately had a batch size of 8 which took slightly longer to process due to it having more frequent weight updates.

Discussion & Conclusion

This project allowed for comparison between a shallow ANN architecture and a more complex deeper one. The more complex shape B, with two hidden layers outperformed model shape A in all metrics. It achieved lower training and validation loss and ultimately lower test MSE – the metric we chose to compare models.

There didn’t seem to be many trade-offs with choosing this more complex design, possibly because it was only 2 hidden layer vs 1 hidden layers. Deeper networks can be prone to overfitting but this did not happen here. Through my research, it seems common to use ReLU over Sigmoid as the activation functions due to vanishing gradient issue (Brownlee, 2020). However Sigmoid was clearly the appropriate activation function to use for Model A – highlighting the importance of hyperparameter tuning rather than relying on common conventions.

Due to the small size of the test set (15 observations), a table comparing both models' predicted penalties and the actual penalties was included (see Appendix 2). The standard deviation of the actual penalties was much higher than that of the model predictions. This suggests that the models tend to predict values closer to the average penalty and are less likely to produce extreme (high or low) predictions compared to the true distribution. This behaviour may result from the models generalising towards the mean, especially when trained on a limited dataset.

Out of curiosity, I tested both ANN models at 10,000 epochs instead of 200. The hyperparameter chosen remained the same but the validation and test results showed improvement with lower loss values. If this ANN was to be deployed, training and validating for more epochs would improve accuracy.

Further work could be to make the dataset bigger for more training capability, add more network shapes with more hidden layers to experiment with optimality vs complexity. Also employing cross validation (Kfolds) within the hyperparameter tuning could improve the results.

References

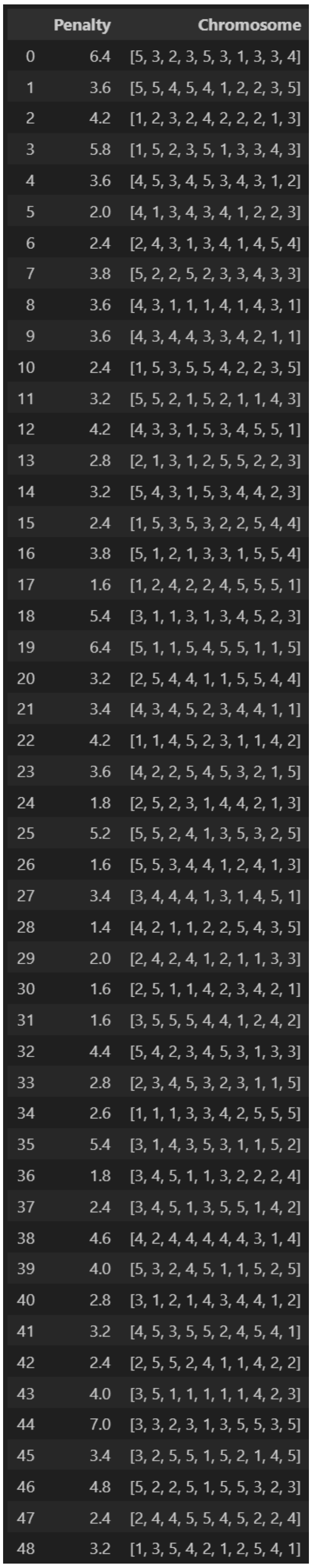
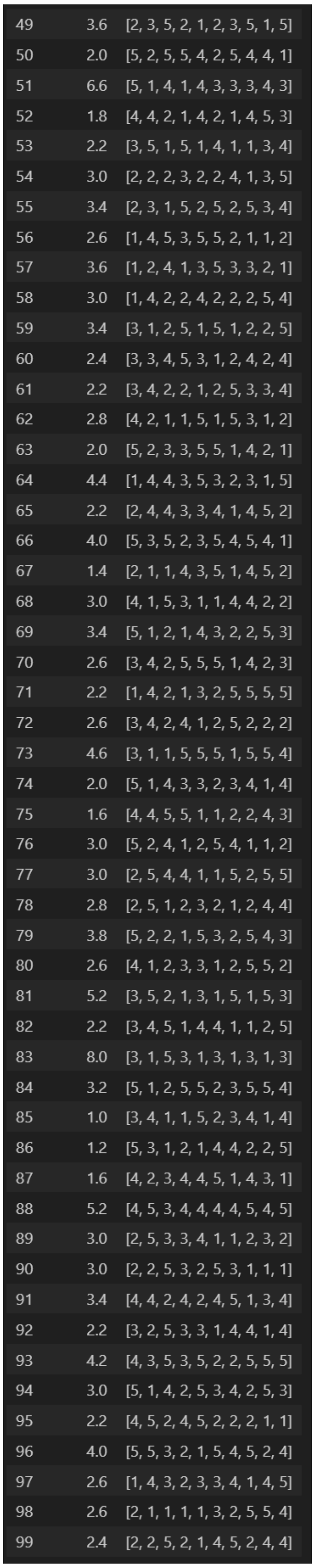
Brownlee, J. (2020, August 25). How to fix the vanishing gradients problem using the ReLU. *Machine Learning Mastery*. https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-relu/

Mirjalili, S. (2019). Evolutionary algorithms and neural networks. In *Studies in computational intelligence* (Vol. 780). Springer International Publishing. https://doi.org/10.1007/978-3-319-93025-1_6

Starmer, J. (n.d.). *StatQuest*. https://statquest.org/

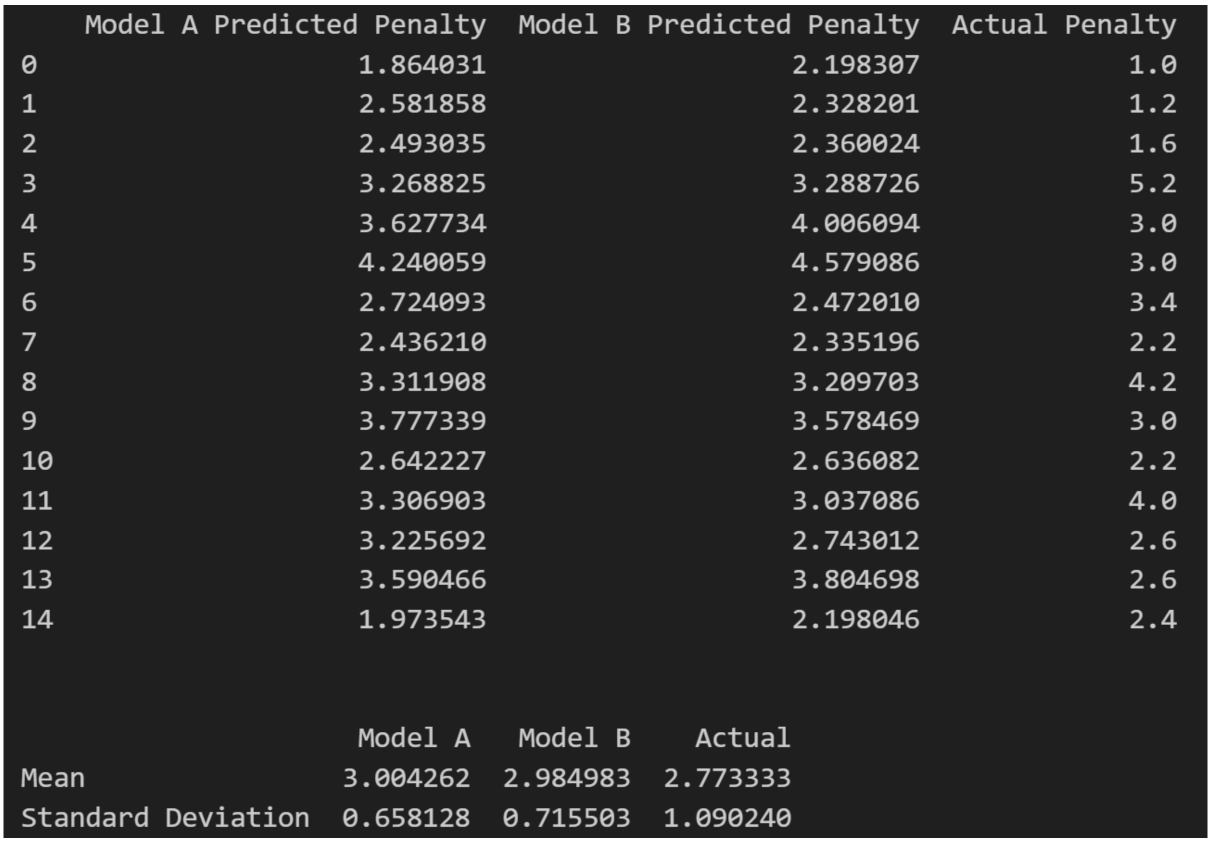
Appendix

Sample Mappings

	Penalty	Chromosome
0	6.4	[5, 3, 2, 3, 5, 3, 1, 3, 3, 4]
1	3.6	[5, 5, 4, 5, 4, 1, 2, 2, 3, 5]
2	4.2	[1, 2, 3, 2, 4, 2, 2, 2, 1, 3]
3	5.8	[1, 5, 2, 3, 5, 1, 3, 3, 4, 3]
4	3.6	[4, 5, 3, 4, 5, 3, 4, 3, 1, 2]
5	2.0	[4, 1, 3, 4, 3, 4, 1, 2, 2, 3]
6	2.4	[2, 4, 3, 1, 3, 4, 1, 4, 5, 4]
7	3.8	[5, 2, 2, 5, 2, 3, 3, 4, 3, 3]
8	3.6	[4, 3, 1, 1, 1, 4, 1, 4, 3, 1]
9	3.6	[4, 3, 4, 4, 3, 3, 4, 2, 1, 1]
10	2.4	[1, 5, 3, 5, 5, 4, 2, 2, 3, 5]
11	3.2	[5, 5, 2, 1, 5, 2, 1, 1, 4, 3]
12	4.2	[4, 3, 3, 1, 5, 3, 4, 5, 5, 1]
13	2.8	[2, 1, 3, 1, 2, 5, 5, 2, 2, 3]
14	3.2	[5, 4, 3, 1, 5, 3, 4, 4, 2, 3]
15	2.4	[1, 5, 3, 5, 3, 2, 2, 5, 4, 4]
16	3.8	[5, 1, 2, 1, 3, 3, 1, 5, 5, 4]
17	1.6	[1, 2, 4, 2, 2, 4, 5, 5, 5, 1]
18	5.4	[3, 1, 1, 3, 1, 3, 4, 5, 2, 3]
19	6.4	[5, 1, 1, 5, 4, 5, 5, 1, 1, 5]
20	3.2	[2, 5, 4, 4, 1, 1, 5, 5, 4, 4]
21	3.4	[4, 3, 4, 5, 2, 3, 4, 4, 1, 1]
22	4.2	[1, 1, 4, 5, 2, 3, 1, 1, 4, 2]
23	3.6	[4, 2, 2, 5, 4, 5, 3, 2, 1, 5]
24	1.8	[2, 5, 2, 3, 1, 4, 4, 2, 1, 3]
25	5.2	[5, 5, 2, 4, 1, 3, 5, 3, 2, 5]
26	1.6	[5, 5, 3, 4, 4, 1, 2, 4, 1, 3]
27	3.4	[3, 4, 4, 4, 1, 3, 1, 4, 5, 1]
28	1.4	[4, 2, 1, 1, 2, 2, 5, 4, 3, 5]
29	2.0	[2, 4, 2, 4, 1, 2, 1, 1, 3, 3]
30	1.6	[2, 5, 1, 1, 4, 2, 3, 4, 2, 1]
31	1.6	[3, 5, 5, 5, 4, 4, 1, 2, 4, 2]
32	4.4	[5, 4, 2, 3, 4, 5, 3, 1, 3, 3]
33	2.8	[2, 3, 4, 5, 3, 2, 3, 1, 1, 5]
34	2.6	[1, 1, 1, 3, 3, 4, 2, 5, 5, 5]
35	5.4	[3, 1, 4, 3, 5, 3, 1, 1, 5, 2]
36	1.8	[3, 4, 5, 1, 1, 3, 2, 2, 2, 4]
37	2.4	[3, 4, 5, 1, 3, 5, 5, 1, 4, 2]
38	4.6	[4, 2, 4, 4, 4, 4, 4, 3, 1, 4]
39	4.0	[5, 3, 2, 4, 5, 1, 1, 5, 2, 5]
40	2.8	[3, 1, 2, 1, 4, 3, 4, 4, 1, 2]
41	3.2	[4, 5, 3, 5, 5, 2, 4, 5, 4, 1]
42	2.4	[2, 5, 5, 2, 4, 1, 1, 4, 2, 2]
43	4.0	[3, 5, 1, 1, 1, 1, 1, 4, 2, 3]
44	7.0	[3, 3, 2, 3, 1, 3, 5, 5, 3, 5]
45	3.4	[3, 2, 5, 5, 1, 5, 2, 1, 4, 5]
46	4.8	[5, 2, 2, 5, 1, 5, 5, 3, 2, 3]
47	2.4	[2, 4, 4, 5, 5, 4, 5, 2, 2, 4]
48	3.2	[1, 3, 5, 4, 2, 1, 2, 5, 4, 1]
49	3.6	[2, 3, 5, 2, 1, 2, 3, 5, 1, 5]
50	2.0	[5, 2, 5, 5, 4, 2, 5, 4, 4, 1]
51	6.6	[5, 1, 4, 1, 4, 3, 3, 3, 4, 3]
52	1.8	[4, 4, 2, 1, 4, 2, 1, 4, 5, 3]
53	2.2	[3, 5, 1, 5, 1, 4, 1, 1, 3, 4]
54	3.0	[2, 2, 2, 3, 2, 2, 4, 1, 3, 5]
55	3.4	[2, 3, 1, 5, 2, 5, 2, 5, 3, 4]
56	2.6	[1, 4, 5, 3, 5, 5, 2, 1, 1, 2]
57	3.6	[1, 2, 4, 1, 3, 5, 3, 3, 2, 1]
58	3.0	[1, 4, 2, 2, 4, 2, 2, 2, 5, 4]
59	3.4	[3, 1, 2, 5, 1, 5, 1, 2, 2, 5]
60	2.4	[3, 3, 4, 5, 3, 1, 2, 4, 2, 4]
61	2.2	[3, 4, 2, 2, 1, 2, 5, 3, 3, 4]
62	2.8	[4, 2, 1, 1, 5, 1, 5, 3, 1, 2]
63	2.0	[5, 2, 3, 3, 5, 5, 1, 4, 2, 1]
64	4.4	[1, 4, 4, 3, 5, 3, 2, 3, 1, 5]
65	2.2	[2, 4, 4, 3, 3, 4, 1, 4, 5, 2]
66	4.0	[5, 3, 5, 2, 3, 5, 4, 5, 4, 1]
67	1.4	[2, 1, 1, 4, 3, 5, 1, 4, 5, 2]
68	3.0	[4, 1, 5, 3, 1, 1, 4, 4, 2, 2]
69	3.4	[5, 1, 2, 1, 4, 3, 2, 2, 5, 3]
70	2.6	[3, 4, 2, 5, 5, 5, 1, 4, 2, 3]
71	2.2	[1, 4, 2, 1, 3, 2, 5, 5, 5, 5]
72	2.6	[3, 4, 2, 4, 1, 2, 5, 2, 2, 2]
73	4.6	[3, 1, 1, 5, 5, 5, 1, 5, 5, 4]
74	2.0	[5, 1, 4, 3, 3, 2, 3, 4, 1, 4]
75	1.6	[4, 4, 5, 5, 1, 1, 2, 2, 4, 3]
76	3.0	[5, 2, 4, 1, 2, 5, 4, 1, 1, 2]
77	3.0	[2, 5, 4, 4, 1, 1, 5, 2, 5, 5]
78	2.8	[2, 5, 1, 2, 3, 2, 1, 2, 4, 4]
79	3.8	[5, 2, 2, 1, 5, 3, 2, 5, 4, 3]
80	2.6	[4, 1, 2, 3, 3, 1, 2, 5, 5, 2]
81	5.2	[3, 5, 2, 1, 3, 1, 5, 1, 5, 3]
82	2.2	[3, 4, 5, 1, 4, 4, 1, 1, 2, 5]
83	8.0	[3, 1, 5, 3, 1, 3, 1, 3, 1, 3]
84	3.2	[5, 1, 2, 5, 5, 2, 3, 5, 5, 4]
85	1.0	[3, 4, 1, 1, 5, 2, 3, 4, 1, 4]
86	1.2	[5, 3, 1, 2, 1, 4, 4, 2, 2, 5]
87	1.6	[4, 2, 3, 4, 4, 5, 1, 4, 3, 1]
88	5.2	[4, 5, 3, 4, 4, 4, 4, 5, 4, 5]
89	3.0	[2, 5, 3, 3, 4, 1, 1, 2, 3, 2]
90	3.0	[2, 2, 5, 3, 2, 5, 3, 1, 1, 1]
91	3.4	[4, 4, 2, 4, 2, 4, 5, 1, 3, 4]
92	2.2	[3, 2, 5, 3, 3, 1, 4, 4, 1, 4]
93	4.2	[4, 3, 5, 3, 5, 2, 2, 5, 5, 5]
94	3.0	[5, 1, 4, 2, 5, 3, 4, 2, 5, 3]
95	2.2	[4, 5, 2, 4, 5, 2, 2, 2, 1, 1]
96	4.0	[5, 5, 3, 2, 1, 5, 4, 5, 2, 4]
97	2.6	[1, 4, 3, 2, 3, 3, 4, 1, 4, 5]
98	2.6	[2, 1, 1, 1, 1, 3, 2, 5, 5, 4]
99	2.4	[2, 2, 5, 2, 1, 4, 5, 2, 4, 4]

Predicted vs True Summary Table



```
    Model A Predicted Penalty  Model B Predicted Penalty  Actual Penalty
0                    1.864031                   2.198307             1.0
1                    2.581858                   2.328201             1.2
2                    2.493035                   2.360024             1.6
3                    3.268825                   3.288726             5.2
4                    3.627734                   4.006094             3.0
5                    4.240059                   4.579086             3.0
6                    2.724093                   2.472010             3.4
7                    2.436210                   2.335196             2.2
8                    3.311908                   3.209703             4.2
9                    3.777339                   3.578469             3.0
10                   2.642227                   2.636082             2.2
11                   3.306903                   3.037086             4.0
12                   3.225692                   2.743012             2.6
13                   3.590466                   3.804698             2.6
14                   1.973543                   2.198046             2.4


                     Model A   Model B    Actual
Mean                3.004262  2.984983  2.773333
Standard Deviation  0.658128  0.715503  1.090240
```