

MINERVA INVESTMENT MANAGEMENT SOCIETY
Quantitative Research Division



Forecasting 101

Neural Networks

Federico Lazzarini Head of Quantitative Research
Filippo Gombac Quantitative Research Analyst

Academic Year 2024/2025

Abstract

In this technical series on econometrics and statistics, we explore various econometric tools, providing the underlying theory behind these models while analyzing a dataset of house prices in California, which is featured in the book *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow* by Aurélien Géron.

DISCLAIMER

This is an academic paper related to an academic project that does not represent any investment recommendation nor solicitation to buy or sell securities. The opinions expressed are subject to change. References to specific securities, asset classes and financial markets are for illustrative purposes only and are not intended to be interpreted as recommendations. Reliance upon information in this material is at the sole risk and discretion of the reader. The material was prepared only in regard to the research activity of Minerva Investment Management Society.

1 Introduction

The objective of this analysis is to apply a neural network model to predict the median house value based on various housing features in a dataset. Unlike linear regression models, which assume a linear relationship between the predictors and the target variable, neural networks are highly flexible and capable of modelling complex, non-linear relationships. By learning from data through interconnected layers of nodes, neural networks can uncover intricate patterns and dependencies that may be difficult to capture with traditional methods.

In this implementation, we use a multi-layer neural network with carefully tuned hyperparameters to maximize predictive accuracy while mitigating the risk of overfitting. The architecture is designed to balance model complexity with generalization, incorporating techniques like dropout to improve robustness. Through empirical testing, we compare the performance of this neural network model with baseline models, analyzing its effectiveness in capturing non-linear relationships within the dataset and evaluating its ability to generalize to unseen data.

2 Theoretical Background

2.1 Singlelayer Neural Networks

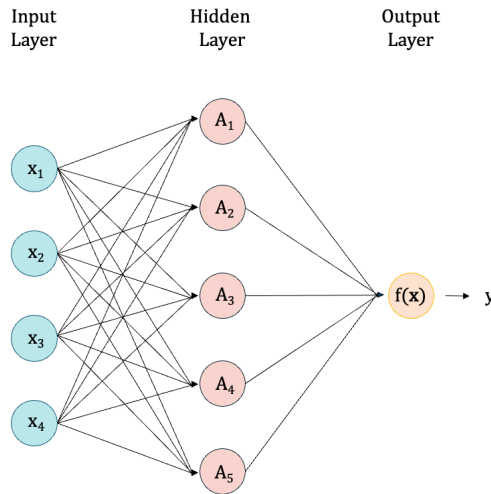


Figure 1: Neural network with a single hidden layer

A neural network takes an input vector $\mathbf{x} \in \mathbb{R}^p$ and builds a nonlinear function $f(\mathbf{x})$ to predict the response y . What distinguishes neural networks from traditional nonlinear prediction models (such as trees, boosting and generalized additive models) is their structure.

Figure 1 shows a simple feed-forward neural network for modeling a quantitative response using $p = 4$ predictors. In the terminology of neural networks, the four features $x_1, \dots, x_4$ make up the units in the input layer.

The arrows indicate that each of the inputs from the input layer feeds into each of the K hidden units (we get to pick K ; for this example we chose 5).

The neural network model has the form

$$f(\mathbf{x}) = \beta_0 + \sum_{k=1}^K \beta_k h_k(X) = \beta_0 + \sum_{k=1}^K \beta_k g \left(w_{k0} + \sum_{j=1}^p w_{kj} X_j \right). \quad (2.1)$$

It is built up here in two steps. First the K activations $A_k, k = 1, \dots, K$ in the hidden layer are computed as functions of the input features $x_1, \dots, x_p$,

$$A_k = h_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^p w_{kj} x_j) \quad (2.2)$$

Where $g(z)$ is a nonlinear activation function that is specified in advance. These K activations from the hidden layer then feed into the output layer, resulting in

$$f(\mathbf{x}) = \beta_0 + \sum_{k=1}^K \beta_k A_k \quad (2.3)$$

a linear regression model in the $K = 5$ activations. All the parameters $\beta_0, \dots, \beta_K$ and $w_0, \dots, w_K$ need to be estimated from the data. In the early instances of neural networks, the sigmoid activation function was favored.

$$g(z) = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}, \quad (2.4)$$

The preferred choice in modern neural networks is the *ReLU* (rectified linear unit) activation function, which takes the form

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise.} \end{cases} \quad (2.5)$$

A ReLU activation can be computed and stored more efficiently than a sigmoid activation. Although it thresholds at zero, because we apply it to a linear function, the constant term w_{k0} will shift this inflection point.

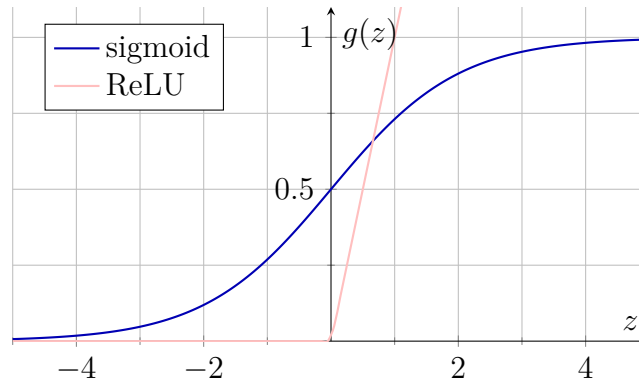


Figure 2: Sigmoid and ReLU activation functions

So in words, the model depicted in Figure 1 derives five new features by computing five different linear combinations of $\mathbf{x}$, and then squashes each through an activation function $g(\cdot)$ to transform it. The final model is linear in these derived variables.

The name *neural network* originally derived from thinking of these hidden units as analogous to neurons in the brain — values of the activations $A_k = h_k(\mathbf{x})$ close to one are *firing*, while those close to zero are *silent* (using the sigmoid activation function).

The nonlinearity in the activation function $g(\cdot)$ is essential, since without it, the model $f(\mathbf{x})$ in (2.1) would collapse into a simple linear model in $x_1, \dots, x_p$. Moreover, having a nonlinear activation function allows the model to capture complex nonlinearities and interaction effects.

Fitting a neural network requires estimating the unknown parameters in (2.1). For a quantitative response, typically squared-error loss is used, so that the parameters are chosen to minimize

$$\sum_{i=1}^n (y_i - f(x_i))^2. \quad (2.6)$$

2.2 Multilayer Neural Networks

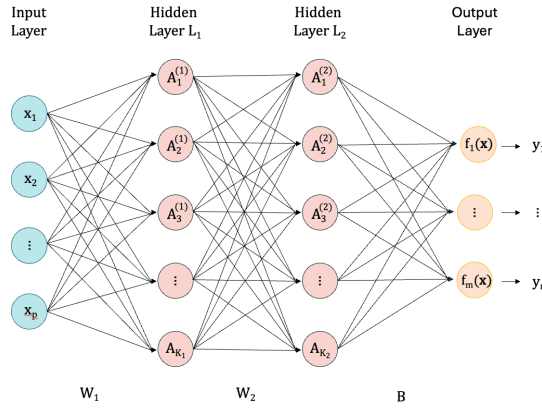


Figure 3: Neural network with two hidden layer

Modern neural networks typically have more than one hidden layer, and often many units per layer. In theory a single hidden layer with a large number of units has the ability to approximate most functions. However, the learning task of discovering a good solution is made much easier with multiple layers each of modest size.

Analogously to the singlelayer neural network the first hidden layer is defined as

$$A_k^{(1)} = h_k^{(1)}(\mathbf{x}) = g \left(w_{k0}^{(1)} + \sum_{j=1}^p w_{kj}^{(1)} x_j \right) \quad (2.7)$$

for $k = 1, \dots, K_1$. The second hidden layer treats the activations $A_k^{(1)}$ of the first hidden layer as inputs and computes new activations

$$A_\ell^{(2)} = h_\ell^{(2)}(\mathbf{x}) = g \left(w_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)} \right) \quad (2.8)$$

for $\ell = 1, \dots, K_2$. Notice that each of the activations in the second layer $A_\ell^{(2)} = h_\ell^{(2)}(\mathbf{x})$ is a function of the input vector $\mathbf{x}$. This is the case because while they are explicitly a function of the activations $A_k^{(1)}$ from layer L_1 , these in turn are functions of $\mathbf{x}$. This would also be the case with more hidden layers. Thus, through a chain of transformations, the

network is able to build up fairly complex transformations of $\mathbf{x}$ that ultimately feed into the output layer as features.

Lastly the output layer. The first step is to compute m different linear models similar to the single model in 2.1

$$Z_m = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} h_{\ell}^{(2)}(\mathbf{x}) = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_{\ell}^{(2)}, \quad (2.8)$$

Please note that also a single layer neural network is able to compute m outputs. Now if we were interested in a quantitative response we would just set each

$$y_m = f_m(\mathbf{x}) = Z_m$$

If instead we were interested for each estimate to represent a class probability (e.g for some classification task) we would apply a special softmax activation function and use a cross-entropy cost function to train the network but that is outside the scope of this paper.

2.3 Fitting a Neural Network

Let's consider the simple network depicted in Figure 1. In that model the parameters are $\beta = (\beta_0, \beta_1, \dots, \beta_K)$, as well as each of the $w_k = (w_{k0}, w_{k1}, \dots, w_{kp})$, $k = 1, \dots, K$. Given observations (x_i, y_i) , $i = 1, \dots, n$, we could fit the model by solving a nonlinear least squares problem:

$$\min_{\{w_k\}_{k=1}^K, \beta} \frac{1}{2} \sum_{i=1}^n (y_i - f(x_i))^2, \quad (2.9)$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^K \beta_k g \left(w_{k0} + \sum_{j=1}^p w_{kj} x_{ij} \right). \quad (2.10)$$

The main difficulty of the problem arises from its nonconvexity in the parameters (which implies there are multiple solutions).

To overcome some of this issue and to protect from overfitting, two general strategies are employed when fitting neural networks:

- **Slow Learning:** The model is fit in a somewhat slow iterative fashion, using gradient descent. The fitting process is then stopped when overfitting is detected.
- **Regularization:** Penalties are imposed on the parameters, usually LASSO or Ridge.

Let θ be a vector containing all the parameters. Then the objective in (2.9) can be rewritten as

$$R(\theta) = \frac{1}{2} \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2. \quad (2.11)$$

2.3.1 Gradient descent

The idea behind gradient decent is straightforward:

1. Start with a guess θ^0 for all the parameters in θ , and set $t = 0$.

2. Iterate until the objective (2.11) fails to decrease:

- (a) Find a vector δ that reflects a small change in θ , such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective; i.e., such that $R(\theta^{t+1}) < R(\theta^t)$.
- (b) Set $t \leftarrow t + 1$.

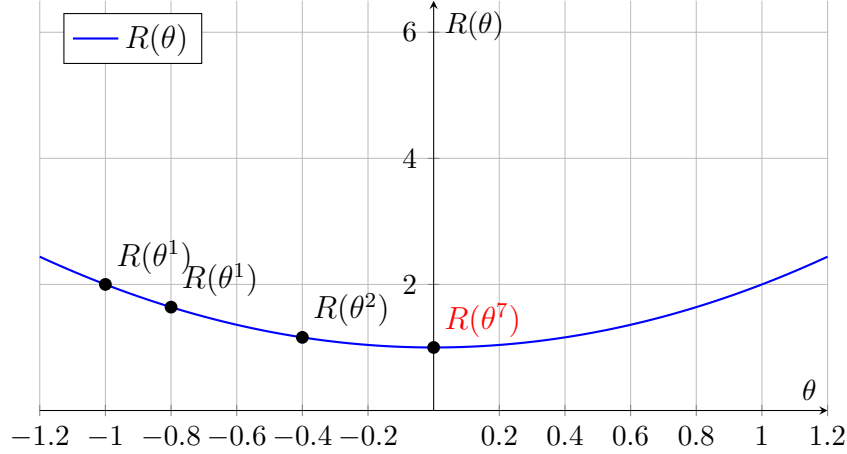


Figure 4: Illustration of a simple gradient descent of a one-dimensional θ towards the minimum of $R(\theta)$ found after 7 steps.

2.3.2 Backpropagation

The model basically "learns" how to tweak the parameters (move θ) so to find the minimum of the loss function we set.

The *gradient* of $R(\theta)$, evaluated at some current value $\theta = \theta^m$, is the vector of partial derivatives at that point:

$$\nabla R(\theta^m) = \left. \frac{\partial R(\theta)}{\partial \theta} \right|_{\theta=\theta^m}. \quad (10.26)$$

The subscript $\theta = \theta^m$ means that after computing the vector of derivatives, we evaluate it at the current guess, θ^m . This gives the direction in θ -space in which $R(\theta)$ *increases* most rapidly. The idea of gradient descent is to move θ marginally in the *opposite* direction (since we wish to go downhill):

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m). \quad (10.27)$$

Where ρ is the *learning rate* which is an *hyperparameter* that defines how much the model can change the weights at each iteration. Notice that choosing a:

- **Small learning rate** ($\rho \ll 1$)
Will guarantee stable convergence but will be computationally expensive as the gradient descent algorithm may get stuck at a local minima for many epochs
- **Large learning rate** ($\rho \gg 1$)
Will guarantee a rapid initial progress but may overshoot the optimal solution

The algorithm of choice will iterate this process over the entire dataset multiple times until it finds a θ^* such that $\nabla R(\theta^*) = 0$. In the machine learning environment each iteration over the whole data set is called an *epoch*. Notice that to find such θ^* one of the key algorithms used (together with gradient descent) is *backpropagation*. It computes how the network’s error (or residual) is distributed across its parameters, enabling the model to learn by adjusting these values to minimize the error.

The process involves calculating the gradient of the error with respect to each parameter in the network. This is achieved by applying the chain rule of calculus, which breaks the gradient computation into smaller, manageable pieces. The *residual error* at the output layer is propagated backward through the network, layer by layer, to assign a fraction of the error to each parameter. This ensures that each weight and bias is adjusted in proportion to its contribution to the overall error.

3 Methodology

3.1 Data Processing

The dataset used in this study contains information about housing values in various regions in California, with 20,640 entries. Each entry represents a unique district, with features describing the geographic location, demographics, and housing characteristics of that area. Key variables include *longitude*, *latitude*, *housing age*, *number of rooms*, *population*, *number of households*, *median income*, and *proximity to the ocean*, as well as the target variable, *median.house.value*, which indicates the median value of houses in each district.

Preprocessing Step	Description
Handling Missing Values	Removed rows with missing values, especially in <i>total_bedrooms</i> , to retain data integrity.
Feature Scaling	Standardized numeric values to a normal of mean 0 and standard deviation 1
Categorical Encoding	One-hot encoded <i>ocean_proximity</i> , creating binary features for categorical values.
Stratified and shuffled Train-Validation Split	Split data into 80% training and 20% validation to enable independent assessment of model performance and stratify it using equally distributed bins of <i>median_income</i> .

Table 1: Summary of Data Preprocessing Steps

3.2 Model initialization and setup

3.2.1 Basic Structure of the Neural Network

For now we do not know how the structure of the model will look in its entirety. Keeping in mind our task, we are sure that the model will take as input a vector $\mathbf{x} \in \mathbb{R}^{15}$ (as the resulting data frame from the transformations now has 15 features) and will have a scalar output y (the model's best guess of the log transformation of `median_house_value`).

3.2.2 Fine Tuning

Once the data preparation part has been handled initializing the model is actually pretty straight forward.

The real challenge is to find the optimal *hyperparameters* for the specific task at hand. To do this we use `keras_tuner`. The first step is to select the hyperparameters to optimize and choose a set of values among which `keras_tuner` randomized search algorithm will iterate the model's training.

We create the following space

Hyperparameter	Allowed range
<code>num_layers</code>	Allows the number of layers to vary between 1 and 5
<code>units</code>	Allows the number of neurons in each layer to vary between 16 and 256 with a step of 32
<code>dropout_rate</code>	Introduces a dropout layer for each hidden layer allowing values between 0 and 0.5 for the dropout rate with a step of 0.1
<code>learning_rate</code>	Allows for the learning rate to vary between 0.0001 and 0.01

Table 2: Summary of hyperparameters space

Where `dropout_rate` is the probability that each node in a layer is temporarily "turned off". This is called *dropout learning* and is a common method to avoid *over-fitting*. Intuitively, this technique prevents the network from relying too much on a single neuron, forcing it to explore more sophisticated nuances in the data.

On top of that we implement `EarlyStopping` which monitors `val_loss` with a `patience` of 15. This is a `callback` method that interrupts *training* if the loss function does not improve over the *validation set* for more than 15 epochs to prevent the model from over-fitting.

3.2.3 The final structure

What `keras_tuner` is doing is training the model randomly assigning values in the allowed range to the specified hyperparameters at each training instance and returning the one that minimizes the MAE (the loss function we chose for this analysis). The final model presents 5 hidden layers with 112, 208, 144, 176 and 144 neurons where a dropout rate of 0.4 is applied to the second to last layer and of 0.3 to the last one.

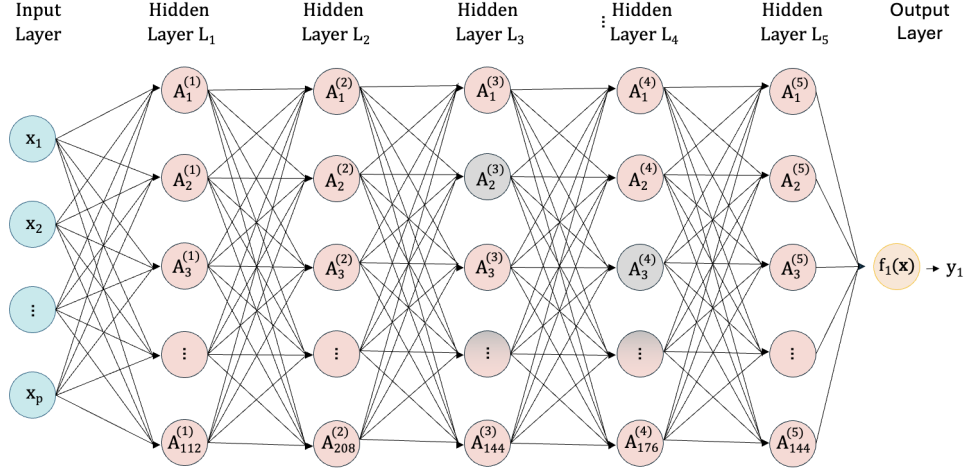


Figure 5: Representation of the final model used for our analysis. The gray scale symbolizes the neurons silenced by the dropout learning layer

4 Results

The scatter plot in Figure 6 shows the predicted values (y-axis) against the true values (x-axis) of the target variable after applying the Neural Network. The red dashed line represents the line of perfect prediction, where the predicted values would equal the true values.

The spread of points around the red line indicates the accuracy of the predictions. Ideally, the points should align closely along the perfect fit line if the model accurately predicts the true values.

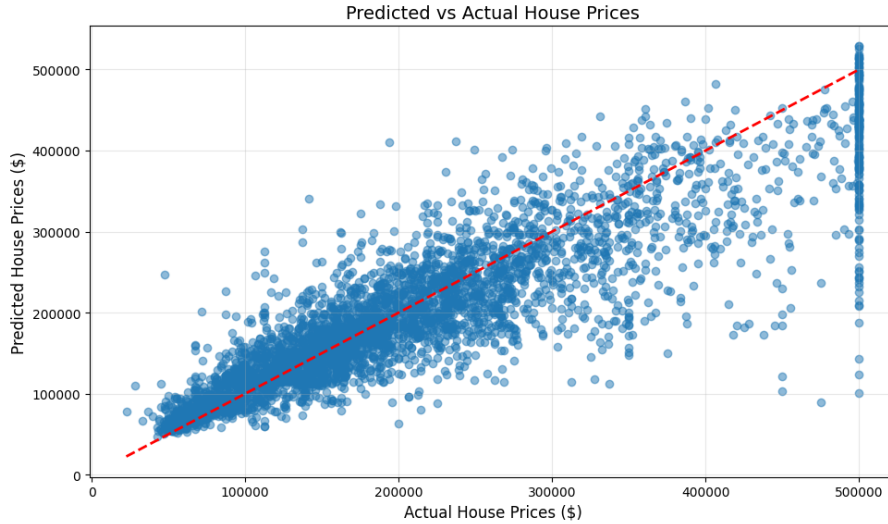


Figure 6: True vs. Predicted values for the Neural Network

Overfitting has been successfully avoided in the neural network model as shown in Fig. 7. The plot presents the model's performance across training epochs, evaluated using the Mean Absolute Error (MAE).

In the Training and Validation Loss (MAE) plot, we observe that the training loss decreases rapidly in the initial epochs, reflecting the model's ability to learn from the data. Simultaneously, the validation loss follows a similar downward trajectory, stabilizing at a level comparable to the training loss. The absence of a significant gap between the two curves indicates strong generalization, meaning the model performs well on unseen data. A divergent trend where validation loss increases while training loss decreases—would have signaled overfitting.



Figure 7: Training and Validation MAE over successive epochs

5 Analysis

5.1 Predictive performance

The model's predictions were evaluated on the test set, assessing both the predictive accuracy and the associated uncertainty of predictions.

- **Mean Squared Error (MSE):** 4236408691,29
- **R-squared (R^2):** 0.6694

These metrics indicate that the model explains approximately 66.9% of the variance in housing prices and achieves a mean squared error of 4236408691,29, demonstrating a reasonably accurate fit.

5.2 Performance comparison

The following analysis compares the Neural Network with the Bayesian Linear Regression and traditional OLS, LASSO, and Ridge models across several metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R^2). Table 4 summarizes the performance of each model.

Model	MAE	MSE	RMSE	R^2
LASSO	50500.43	4784523113.39	69170.25	0.6466
Ridge	50491.27	4783140549.35	69160.25	0.6467
OLS	50493.54	4782679151.64	69156.92	0.6467
Bayesian Linear Regression	30960.90	1787904818.80	42283.60	0.7183
Neural Network	45730.87	4236408691.29	65087.70	0.6694

Table 3: Performance comparison of different models.

- **Mean Absolute Error (MAE):** The Neural Network model achieves an MAE of 45,730.87, which is lower than the MAE values of the OLS, LASSO, and Ridge models, all of which are around 50,500. However, it is higher than the MAE of the Bayesian Linear Regression model, which is 30,960.90. A lower MAE indicates that the Neural Network model provides predictions that are closer to the actual values on average compared to the traditional linear models, but it does not outperform the Bayesian Linear Regression in this metric.
- **Root Mean Squared Error (RMSE):** The RMSE for the Neural Network model is 65,087.70, which is lower than the RMSE of the OLS, LASSO, and Ridge models, all of which exceed 69,000. However, it is higher than the RMSE of the Bayesian Linear Regression model, which is 42,283.60. Since RMSE penalizes larger errors more heavily, the Neural Network demonstrates improved performance over traditional linear models in handling extreme prediction errors, but it does not surpass the Bayesian Linear Regression.
- **R-squared (R^2):** The Neural Network model achieves an R^2 value of 0.6694, outperforming the OLS, LASSO, and Ridge models, which have R^2 values around 0.6467. However, it falls short of the Bayesian Linear Regression model's R^2 of 0.7183. The Neural Network's performance reflects its ability to capture more complex patterns and relationships in the data compared to the linear models.

6 Conclusion

In this series of papers, we conducted a comprehensive analysis of several predictive models applied to the housing dataset, including Ordinary Least Squares (OLS), Ridge Regression, LASSO Regression, Bayesian Linear Regression, and Neural Networks (NN). Each model was evaluated on key performance metrics, including Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R^2), providing insights into their strengths and weaknesses.

- The **Neural Network** demonstrated its ability to capture complex, non-linear relationships within the data, outperforming traditional regression methods (OLS, Ridge, and LASSO) in most metrics.
- The **Bayesian Linear Regression** excelled in controlling errors, achieving the lowest MAE of 30,960.90 and the highest R^2 score of 0.7183 among all models evaluated. This performance difference reflects the Bayesian framework's inherent regularization and its ability to incorporate prior information, effectively minimizing variance and reducing the impact of outliers. The Bayesian model's lower RMSE of 42,283.60 further indicates its effectiveness in handling extreme prediction errors. The probabilistic nature of Bayesian Linear Regression also provided valuable insights into prediction uncertainty, which is critical for applications requiring reliability and confidence intervals.
- **Traditional linear models** such as OLS, Ridge, and LASSO demonstrated comparable but inferior performance across all metrics, with MAE values around 50,500 and R^2 scores of approximately 0.6467. These models, while simpler and computationally efficient, struggled to capture the intricate patterns and relationships in the dataset, which likely limited their predictive power. The similarity in performance among these models suggests that regularization techniques in Ridge and LASSO did not significantly enhance predictive accuracy over OLS in this context.

This comparison highlights the importance of selecting a model based on the specific requirements of the task at hand. While Neural Networks are powerful for achieving high accuracy, Bayesian Linear Regression may be more suitable for applications requiring stability, interpretability, and robustness to outliers. These findings demonstrate how a thoughtful application of machine learning and statistical models can yield tailored solutions for predictive tasks.