Our multilayer perceptron model outperforms all others we've tested, yet there's potential for further refinement. Initially, we used default settings of hyperparameters for simplicity.

Now we take a closer look at the model, we can optimize the model by adjusting some of the hyperparameters: Number of layers, number of neurons inside each hidden layer, learning rate, batch size and number of epochs.

We employ k-fold cross-validation for a robust evaluation. The original dataset is randomly partitioned into k equal-sized subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining k-1 subsamples are used as training data. The process is then repeated k times, with each of the k subsamples used exactly once as the validation data. The k pairs of MSE and R-squared calculated each time can then be averaged to produce a single estimation as the metric to assess the model.

Cross validation not only provides a more accurate performance measure but also mitigates the risk of overfitting by ensuring that our hyperparameter choices are effective and robust across various data samples.

We began with the number of hidden layers and the number of neurons inside each hidden layer, which determine the structure of the neural network model. Our initial configuration included three hidden layers with 100, 200, and 100 neurons. Adjustments to this structure showed diminishing returns and reduced training efficiency with larger sizes. While smaller configurations underperformed. Consequently, we've maintained our original layer and neuron settings.

Regarding the number of epochs. Empirically and intuitively, the more epochs the better. But there is a tradeoff between model performance and training time. The graph on the top is the curve of R-Squared as epochs increases, and the graph at the bottom is the curve of MSE. We noticed that both curves converge before the 100th epoch, making 100 the optimal number of epochs.

Learning rate is also a crucial hyperparameter. It controls the size of the steps that the algorithm takes during the optimization process. A learning rate that's too large can cause the model to converge too quickly to a suboptimal solution. On the other hand, a learning rate that's too small not only increases the computational time but also is prone to be stuck in a suboptimal solution. We chose three different learning rates and as you can see from the graphs, 0.001 works the best with biggest R-squared, striking a balance between speed and convergence.

Lastly, we examined batch size, which defines the number of samples for the model to work through each time. larger batch sizes can improve training speed while have poor converge ability. And smaller batch sizes make it slower to converge and there’s more risk of overfitting. Note that larger batch size requires larger learning rate for the model to converge. So, we selected three sets of batch size and learning rate. Clearly a batch size of 64 coupled with a learning rate 0.003 is optimal.