**Using an Artificial Neural Network to Predict MLB Hitting Salaries**

**Introduction:**

Since the advent of contemporary Major League Baseball free agency, which began with the Collective Bargaining Agreement between players and owners in July 1976, Major League Baseball players have been able to test their value on the free market after reaching at least six years of service time at the Major League level. Prior to 1976, players were bound to the team they originally signed to through the Reserve Clause, severely limiting their earning potential by giving players no choice of which team they played for.

With this contemporary free agency system, along with other economic structural changes to the game since in recent years, players have come closer to earning what they merit based on their performance. This paper will use an aritificial neural network to predict salary for a subgroup of Major Baseball Players, hitters, from 1985-2012 based on their hitting statistics. The full experimental plan, which took roughly 10 hours to execute, involved exploring different optimizers, activation functions, unit number of layers, learning and decay rates, and epoch numbers.

**Objective:**

The objective of this project is to use the data provided from the Lahman Historical Baseball Database to develop a model to use offensive performance and age to be able to predict a player’s future salary. This could theoretically be used by player agents or teams to calculate the amount that an offensive player should be paid based on their production at the plate, which is a timely topic given the current labor dispute between MLB players and MLB owners in negotiating the next Collective Bargaining Agreement (CBA). The weights of this model could be updated if the fundamental economics of how MLB players are paid changes, but if the economic structure remains fundamentally the same then this model could continue to have predictive accuracy. I used a regression neural network to make a prediction of player salary based on offensive performance and age.

**Data Link and Description:**

***Link to data source:***

The link below leads to Kaggle.com, which contains the Lahman History of Baseball Database. This link leads to the batting\_csv file used in the paper, but a salary\_csv file and a player\_csv file are also used and are intuitively accessible from the link below.

<https://www.kaggle.com/seanlahman/the-history-of-baseball?select=batting.csv>

***Data description:***

The data is a combination of 3 files from the Lahman History of Baseball Database: a Batting file containing batting statistics for each season of batting for every batter between 1871-2012, a Player File containing demographic information, including birth month and birth year, for every player between 1871-2012, and a Salary file containing salary information for every player between 1985-2015.

The three files were merged so that they all contained the same years of shared data (from 1985-2012), with unnecessary columns dropped and a custom Age column being created by subtracting their “Effective Age” (which was created because players born after June 30th in a season are considered to be the age that they were for the majority of the season by Baseball Reference) from the year of the season in question. Additionally, players with less than 100 ABs in a season were filtered out for the purposes of removing most pitchers from the dataset (as their salaries largely do not depend on batting performance) and other player seasons with insufficient data.

The final dataset was 10554 Rows x 22 Columns with offensive statistics per year, the player’s position, and player salary with the following metrics: Games Played (G), At-Bats (AB), Runs Scored(R), Hits (H), Doubles (2B), Triples (3B), Home Runs (HR), Runs Batted In (RBI), Stolen Bases (SB), Caught Stealing (CS), Walks (BB), Strikeouts (SO), Intentional Walks (IBB), Hit by Pitches (HBP), Sacrifice Hits (SH), Sacrifice Flys (SF), Grounded Into Double Plays (GIDP), bats (batting side of the plate), throws (throwing hand), Age, and salary.

Of these variables, all of them are numerical except for two: bats and throws. These variables will be encoded within the data pre-processing section.

**Example of Data (before pre-processing for clarity):**

A picture containing text, orange

Description automatically generated

**Data Pre-Processing**

The categorical predictive variables (throws and bats) were encoded and tranformed into dummy variables using the Label Encoder. The data was then split into train (64%), test (20%) and validation (16%) sets, split into X (predictive variables, which are all variables but the salary) and Y (salary, our dependent variable), and preprocessed to be scaled between 0-1.

**Metrics**

The goal of this neural network is to predict a numeric response. Therefore, I used a regressor ANN model. I will use Mean Squared Error (MSE) as my primary performance metric, which is labeled in the graphs as “loss”. I also tracked Mean Absolute Error (MAE) as an alternative metric so that I am not simply overfitting to a single performance metric.

**Experimental Plan**

1. Get a working model and measure the MSE of that model for both the training and validation sets. This model is not optimized in any way and all subsequent steps will be taken in order to improve upon the performance of this base model.

2. Optimize the optimizer by using grid search, using “Adam”, “SGD” and “RMSprop” as possible options. Likely use the optimizer that yields the best MSE, also considering the standard deviation (SD) and other potential factors in making the choice if the differences in MSE are minimal. Modify and run the model from Step 1 and plot new results.

3. Optimize the activation function by using grid search, using “relu”, “tanh”, “sigmoid” and “linear” as possible options. Likely use the activation function that yields the best MSE, also considering the standard deviation (SD) and other potential factors in making the choice if the differences in MSE are minimal. Modify and run the model from Step 2 and plot new results.

4. Optimize the unit layers and layer sizes by using grid search. Likely use the unit layers and layer sizes that yield the best MSE, also considering the standard deviation (SD) and other potential factors in making the choices if the differences in MSE are minimal. Modify and run the model from Step 3 and plot new results.

5. Optimize the learning and decay rates by using grid search. Likely use the learning and decay rates that yield the best MSE, also considering the standard deviation (SD) and other potential factors in making the choices if the differences in MSE are minimal. Modify and run the model from Step 4 and plot new results.

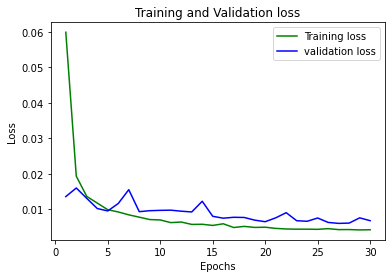
6. Experiment with different batch sizes, starting at the size of 64 from the original model and trying sizes of 32 and 16. Pick the batch size that achieves the best balance between reducing the MSE and keeping the model running quickly. Modify and run the model from Step 5 and plot new results.

7. Experiment with epoch sizes, using the knowledge that you have gained from Steps 1-6 to pick appropriate epoch sizes to try. Pick the epoch size that achieves the best balance between reducing the MSE and keeping the model running quickly. Modify and run the model from Step 6 and plot new results.

8. Run the final model completed in Step 7, swapping the validation data we used to train the model for the test data to test the validity of the model for predicting future outside data.

**Step 1: Get a Working Model**

The working base model was created usinga sequential model with 2 dense layers with 128 and 64 units respectively, a relu activation function, Adam optimizer, batch normalization, a final dense output layer with 1 output, metrics of MSE and MAE, MSE as the loss function, 30 epochs, a batch size of 64, and validated using validation data.



***Results:***

loss: 0.0042 - mean\_squared\_error: 0.0042 - mean\_absolute\_error: 0.0444 - val\_loss: 0.0068 - val\_mean\_squared\_error: 0.0068 - val\_mean\_absolute\_error: 0.0574

Time required for training: 0:00:06.019980

**Takeaways:**

Training loss begins to decrease very quickly while validation loss fluctuates over time. This model is a good start, even though it has a long way to go and is currently overtrained.

**Step 2: Optimize the Optimizer**

Graphical user interface, text, application

Description automatically generated

***Results:***

*the best fit is: -0.004943 using {'optimizer': 'SGD'}*

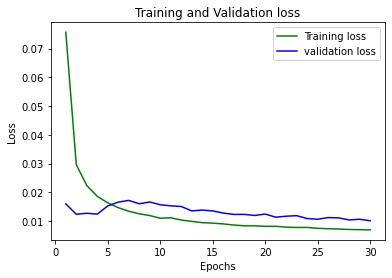
-0.004943 (0.000566) with: {'optimizer': 'SGD'}

-0.006091 (0.002039) with: {'optimizer': 'RMSprop'}

-0.006660 (0.002565) with: {'optimizer': 'Adam'}

Time required for training: 0:04:39.410048

***Model Now:***



loss: 0.0076 - mean\_squared\_error: 0.0076 - mean\_absolute\_error: 0.0634 - val\_loss: 0.0103 - val\_mean\_squared\_error: 0.0103 - val\_mean\_absolute\_error: 0.0744

***Takeaways:***

The model now performs better using the SGD optimizer, which performed better than the Adam optimizer I started with. The validation data loss now decreases at a steady rate after 6 epochs.

**Step 3: Activation Functions:**

**Graphical user interface, text

Description automatically generated**

***Results:***

The best accuracy is: -0.005338 using {'activation': 'linear'}

-0.005613 (0.000659) with: {'activation': 'relu'}

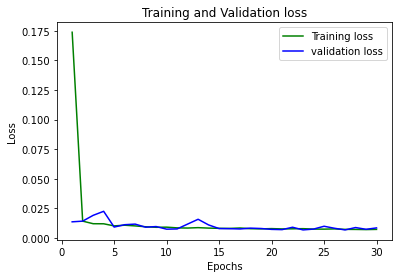
-0.005741 (0.000801) with: {'activation': 'sigmoid'}

-0.005338 (0.000684) with: {'activation': 'linear'}

-0.005940 (0.000763) with: {'activation': 'tanh'}

Time required for training: 0:05:51.489711

***New Model:***



loss: 0.0067 - mean\_squared\_error: 0.0067 - mean\_absolute\_error: 0.0555 - val\_loss: 0.0067 - val\_mean\_squared\_error: 0.0067 - val\_mean\_absolute\_error: 0.0571

time for training: 0:00:04.978977

***Takeaways:***

This step saw a big increase in performance in both the training and validation loss with the change to using linear activation functions from the original relu activation functions, with the levels of loss essentially becoming the same after 15 epochs. This model is still overtrained but is performing well.

**Step 4: Number of unit layers**

Graphical user interface, text, application

Description automatically generated

**The best accuracy is: -0.006170 using {'num\_nodes\_first': 200, 'num\_nodes\_second': 20}**

-0.006555 (0.000338) with: {'num\_nodes\_first': 20, 'num\_nodes\_second': 20}

-0.006812 (0.000700) with: {'num\_nodes\_first': 20, 'num\_nodes\_second': 50}

-0.006852 (0.000605) with: {'num\_nodes\_first': 20, 'num\_nodes\_second': 100}

-0.007780 (0.001827) with: {'num\_nodes\_first': 20, 'num\_nodes\_second': 200}

-0.006220 (0.000384) with: {'num\_nodes\_first': 50, 'num\_nodes\_second': 20}

-0.007317 (0.000852) with: {'num\_nodes\_first': 50, 'num\_nodes\_second': 50}

-0.007062 (0.000941) with: {'num\_nodes\_first': 50, 'num\_nodes\_second': 100}

-0.007344 (0.000918) with: {'num\_nodes\_first': 50, 'num\_nodes\_second': 200}

-0.006851 (0.001066) with: {'num\_nodes\_first': 100, 'num\_nodes\_second': 20}

-0.007003 (0.000842) with: {'num\_nodes\_first': 100, 'num\_nodes\_second': 50}

-0.008503 (0.001339) with: {'num\_nodes\_first': 100, 'num\_nodes\_second': 100}

-0.008493 (0.001492) with: {'num\_nodes\_first': 100, 'num\_nodes\_second': 200}

-0.006170 (0.000404) with: {'num\_nodes\_first': 200, 'num\_nodes\_second': 20}

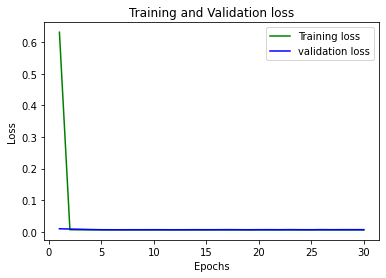
nan (nan) with: {'num\_nodes\_first': 200, 'num\_nodes\_second': 50}

nan (nan) with: {'num\_nodes\_first': 200, 'num\_nodes\_second': 100}

nan (nan) with: {'num\_nodes\_first': 200, 'num\_nodes\_second': 200}

Time required for training: 0:21:40.481453

***New Model:***



loss: 0.0062 - mean\_squared\_error: 0.0062 - mean\_absolute\_error: 0.0526 - val\_loss: 0.0070 - val\_mean\_squared\_error: 0.0070 - val\_mean\_absolute\_error: 0.0602

Time required for training: 0:00:06.342912

***Takeaways:***

The model now trains very quickly, with most of the training and validation loss being achieved before the 5th epoch. The unit layer sizes that performed best were 200 nodes for the first dense layer and 20 nodes for the second dense layer.

**Step 5: Learning and Decay Rates**

**Text, application

Description automatically generated**

The best accuracy is: -0.006729 using {'decay': 0.1, 'learn\_rate': 0.01}

-0.008148 (0.001352) with: {'decay': 0, 'learn\_rate': 0.001}

-0.009157 (0.001660) with: {'decay': 0, 'learn\_rate': 0.05}

-0.013483 (0.009999) with: {'decay': 0, 'learn\_rate': 0.01}

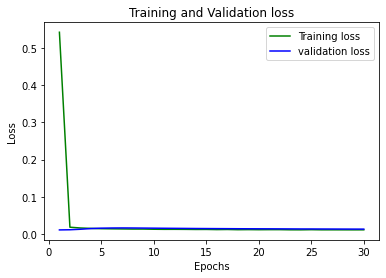
-0.008002 (0.001828) with: {'decay': 0.1, 'learn\_rate': 0.001}

-0.007671 (0.000607) with: {'decay': 0.1, 'learn\_rate': 0.05}

-0.006729 (0.000281) with: {'decay': 0.1, 'learn\_rate': 0.01}

Time required for training: 0:08:00.796697

***New model:***



loss: 0.0111 - mean\_squared\_error: 0.0111 - mean\_absolute\_error: 0.0751 - val\_loss: 0.0130 - val\_mean\_squared\_error: 0.0130 - val\_mean\_absolute\_error: 0.0836

Time required for training: 0:00:04.906611

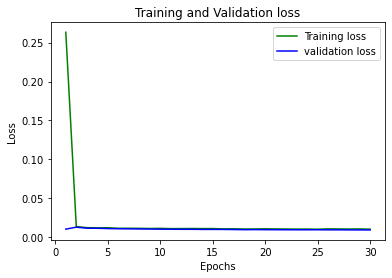
***Takeaways:***

The model performance slightly improved using a decay rate of 0.1 and a learning rate of 0.01, both of which are different from the defaults. The current model performs well, but is very overtrained.

**Step 6: Batch Size**

We have been using a batch size of 64, so I decreased the batch size first to 32 and then to 16. Here are the results:

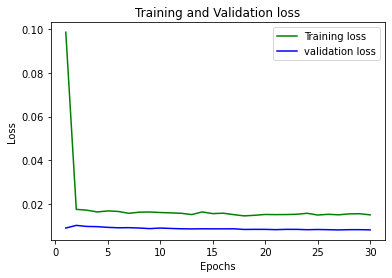
***Batch Size of 32:***



loss: 0.0104 - mean\_squared\_error: 0.0104 - mean\_absolute\_error: 0.0741 - val\_loss: 0.0095 - val\_mean\_squared\_error: 0.0095 - val\_mean\_absolute\_error: 0.0690

Time required for training: 0:00:08.818419

***Batch Size of 16:***



loss: 0.0151 - mean\_squared\_error: 0.0151 - mean\_absolute\_error: 0.0917 - val\_loss: 0.0083 - val\_mean\_squared\_error: 0.0083 - val\_mean\_absolute\_error: 0.0659

Time required for training: 0:00:15.303323

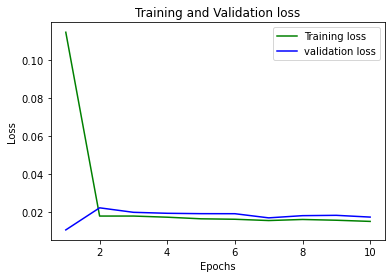
***Takeaways:***

The ideal batch size appears to be approximately 32, as the validation loss improves without a significant increase in run time when adjusting from 64 to 32, but the model performs worse and takes almost twice as long to run when batch size is decreased to 16.

**Step 7: Number of Epochs**

Based on what I have learned from steps 1-6, I know I should try smaller epoch sizes to decrease overfitting. I decided to try epoch sizes of 10 and 20 and compare the results.

***Epoch Size: 10***

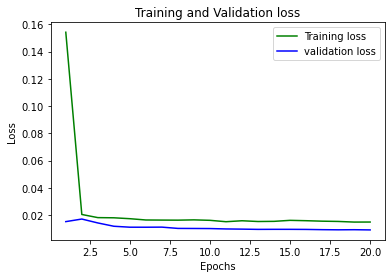


loss: 0.0152 - mean\_squared\_error: 0.0152 - mean\_absolute\_error: 0.0915 - val\_loss: 0.0175 - val\_mean\_squared\_error: 0.0175 - val\_mean\_absolute\_error: 0.1010

Time required for training: 0:00:03.594306

**Validation loss is still decreasing**

***Epoch Size of 20:***



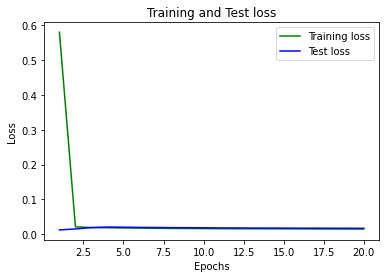
loss: 0.0150 - mean\_squared\_error: 0.0150 - mean\_absolute\_error: 0.0915 - val\_loss: 0.0092 - val\_mean\_squared\_error: 0.0092 - val\_mean\_absolute\_error: 0.0697

Time required for training: 0:00:06.116719

***Takeaways:***

At 10 epochs the validation loss appears to still be significantly improving but this is not the case at 20 epochs. This causes me to believe that 20 epochs is approximately the right number to use for the final model.

**Step 8: Test Final Model (against test data):**



loss: 0.0149 - mean\_squared\_error: 0.0149 - mean\_absolute\_error: 0.0884 - val\_loss: 0.0168 - val\_mean\_squared\_error: 0.0168 - val\_mean\_absolute\_error: 0.0929

Time required for training: 0:00:05.978157

***Structure of Final Model:***

The final model was created using a sequential model with 2 dense layers with 200 and 20 units respectively, a linear activation function, a SGD optimizer, batch normalization, a final dense output layer with 1 output, metrics of MSE and MAE, MSE as the loss function, 20 epochs, a batch size of 32, and validated using test data.

***Takeaways:***

This final model performs well on both the training and test data, which validates the parameters that I have tweaked and improved throughout the experimental process. We achieve a final training MSE of 0.0149 and a test MSE of 0.0168.

**Final Code:**

Graphical user interface, text, application, email

Description automatically generated

**Text

Description automatically generated**

**Text, application

Description automatically generated**

**Text

Description automatically generated**

**Validation of Final Model:**

The final model was validated in multiple ways:

1. We tested the model on both training and validation data and then used an independent test set that we set aside to test the final model. The model performed extremely well on this test set, giving confidence that is generalizable.

2. We used hyperparameter optimization through the grid search algorithm to optimize many of the parameters. This gives us confidence that we set each parameter appropriately.

3. We did a very thorough testing of parameters within the model, with very little left untested.

**Discussion of Methods**

The final model uses Stochastic Gradient Descent (SGD), which is a type of feedforward network that performs a parameter update for each training example x and label y. This likely works best for this data because it is organized in many cases sequentially, with players earning less at the beginning of their careers and then gradually earning more as they get older. Backpropagation is not as strong of a fit here because applying future errors to past observations will likely result in larger errors in this case. A recurrent structure is not a good fit here because it would likely take too long given that feedforward is effective and because we are only trying to predict a single value, as opposed to predicting the shape of an image with possibly hundreds or thousands of numbers.

Because the data is labeled and we know the response variable that we want i.e. salary, we only use supervised learning in this model. Unsupervised would be a better fit if we did not know the structure of the data, but most traditional baseball data is highly structured, with biomechanical data being a possible fit for unsupervised learning.

The input unit number was determined by the structure of our final dataset, which was 21 predictors, and was therefore inflexible unless we wanted to add or subtract data columns. The output data structure is also set at 1, which was set by the objective (predict salary numerically).

The proportion of the training, testing and validation sets were pre-determined at logical parameters but were not tested. This could be done in the future, although it would likely only be done in the direction to increase the training set size and decrease the test set size given the size of our original dataset.

I settled on 2 hidden layers to model the data given that the data is relatively simple in structure. More testing on the number of hidden layers could be conducted in the future.

**Discussion of Final Results and Future Modifications**

This model could be tested upon future years of baseball hitting data to see how well the weights and parameters perform within a changing financial and analytical climate within Major League Baseball. In the years since 2012 we have seen a major shift in how players are evaluated, partially due to technological innovations such as Statcast, which is biometric tracking technology, and partially due to innovations in baseball research within the public and internal (MLB club employees) spheres. This would likely lead to the model’s weights needing to be adjusted to better understand why contemporary hitters are being paid certain amounts.

This model did not adjust for the different statuses of players, as players with less service time are systemically paid less, regardless of performance. This is somewhat accounted for with the age predictor, but future analysis could separate pre salary arbitration players from arbitration eligible players from free agent eligible players, although this process would be time intensive.

Even in using grid search, we did not account for every possible value for every parameter to save time, as we instead picked a logical range of possible parameter values for each parameter and then tested. This was done to save time, as a full comprehensive grid search would have been practically infeasible. A larger range of parameter values could be tested in the future, however.

A model of this structure could also easily be applied to pitching performance, which was intentionally excluded from this paper for simplicity purposes. The pitching model would need to be separate from the hitting model created here, as pitchers are compensated based on different statistical performance marks that do not cross over with hitting performance. Additionally, pitchers have different aging curves and inherent injury risk rates that cause their compensation levels to be systemically different from hitters.