Deep Learning Assignment:

Defining a Relationship for Player Efficiency Rating from John Hollinger's NBA Statistics – Part 2

Objective

The purpose of this report, and overall study, is to attempt to determine a both a predictive and explanatory relationship between John Hollinger's individual advanced statistics and his Player Efficiency Rating (PER) statistic. It is known that there is a relationship between the 2 variables – however, to this point, the formula for the relationship between PER and the other advanced statistics has not been published. PER is presented as a single advanced statistic, that considers all of a players positive and negative contributions to a game and returns them in a single, weighted measure.

This analysis (Part 1) was undertaken earlier in the course, during the regression course. Three different types of regression (vanilla linear, Ridge, Lasso) were modeled, and it was found that, when considering model complexity, computational time (amongst other things) that the model that best met the joint criteria of explain-ability and accuracy as the vanilla linear regression model – the model was accurate on a holdout/testing set (86% accuracy) with great explain-ability (6 linearly combined variables).

The objective of this exercise will be to attempt to model the situation using a standard feedforward neural network, to see if we can improve the analysis over and above the vanilla linear regression.

The Data

The data used for the analysis is taken from a Kaggle dataset. It is a singe csv file that summarizes Hollinger's advanced NBA statistics from the 2002 – 03 season to the 2017 – 18 season. The data set is already cleaned, containing no zero, N/A etc data and contains 5404 row entries. Column entries include:

- Rank: that player's PER rank for that given season
- ts%: True Shooting Percentage - what a player's shooting percentage would be if we accounted for free throws and 3-pointers. True Shooting Percentage = Total points / [(FGA + (0.44 x FTA)]
- ast: Assist Ratio - the percentage of a player's possessions that ends in an assist. Assist Ratio = (Assists x 100) divided by [(FGA + (FTA x 0.44) + Assists + Turnovers]
- to: Turnover Ratio - the percentage of a player's possessions that end in a turnover. Turnover Ratio = (Turnover x 100) divided by [(FGA + (FTA x 0.44) + Assists + Turnovers]
- usg Usage Rate - the number of possessions a player uses per 40 minutes. Usage Rate = {[FGA + (FT Att. x 0.44) + (Ast x 0.33) + TO] x 40 x League Pace} divided by (Minutes x Team Pace)
- orr: Offensive rebound rate
- drr: Defensive rebound rate
- rebr: Rebound Rate - the percentage of missed shots that a player rebounds. Rebound Rate = (100 x (Rebounds x Team Minutes)) divided by [Player Minutes x (Team Rebounds + Opponent Rebounds)]
- per: Player Efficiency Rating is the overall rating of a player's per-minute statistical production. The league average is 15.00 every season.
- va: Value Added - the estimated number of points a player adds to a teammates season total above what a 'replacement player' (for instance, the 12th man on the roster) would produce. Value Added = ([Minutes * (PER - PRL)] / 67). PRL (Position Replacement Level) = 11.5 for power forwards, 11.0 for point guards, 10.6 for centers, 10.5 for shooting guards and small forwards

- ewa: Estimated Wins Added - Value Added divided by 30, giving the estimated number of wins a player adds to a team's season total above what a 'replacement player' would produce.

Outlier data was not removed from the set/each feature, as we did not want to remove important data on shooting % (and other things) that we know are important in determining PER.

Given that the features per, va, and ewa are quite similar in that they describe a players contribution, the va and ewa features were dropped for the analysis. The following were considered for preliminary analysis:

<u>Features</u>

1. ts% True Shooting Percentage

2. ast Assist Ratio

3. to Turnover Ratio

4. usg Usage Rate

5. orr Offensive rebound rate

6. drr Defensive rebound rate

7. rebr Rebound Rate

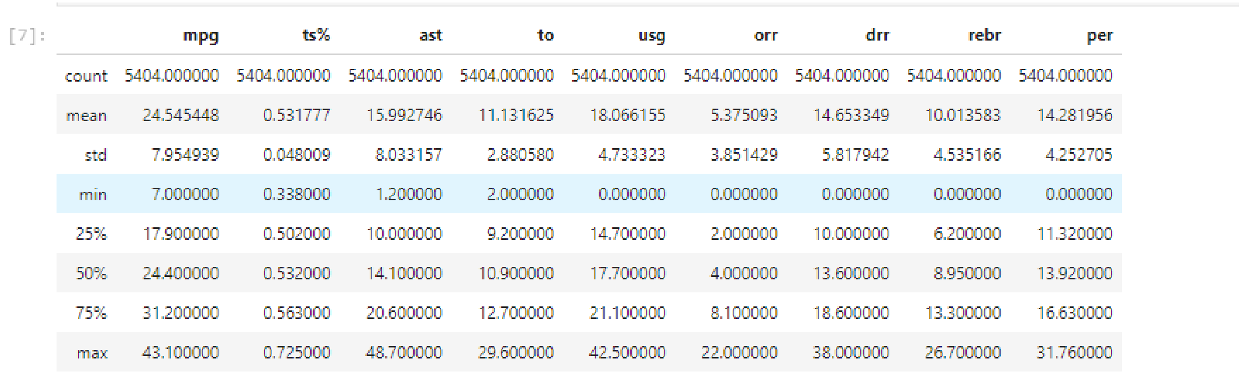
8. Minutes per game (mpg)

<u>Target</u>

1. PER

<u>Exploratory Analysis</u>

The first thing done was to get a description of the data. Since all the data was numeric, it is nice to look at the mean, median and different quartiles of the data, to see how it is spread out.

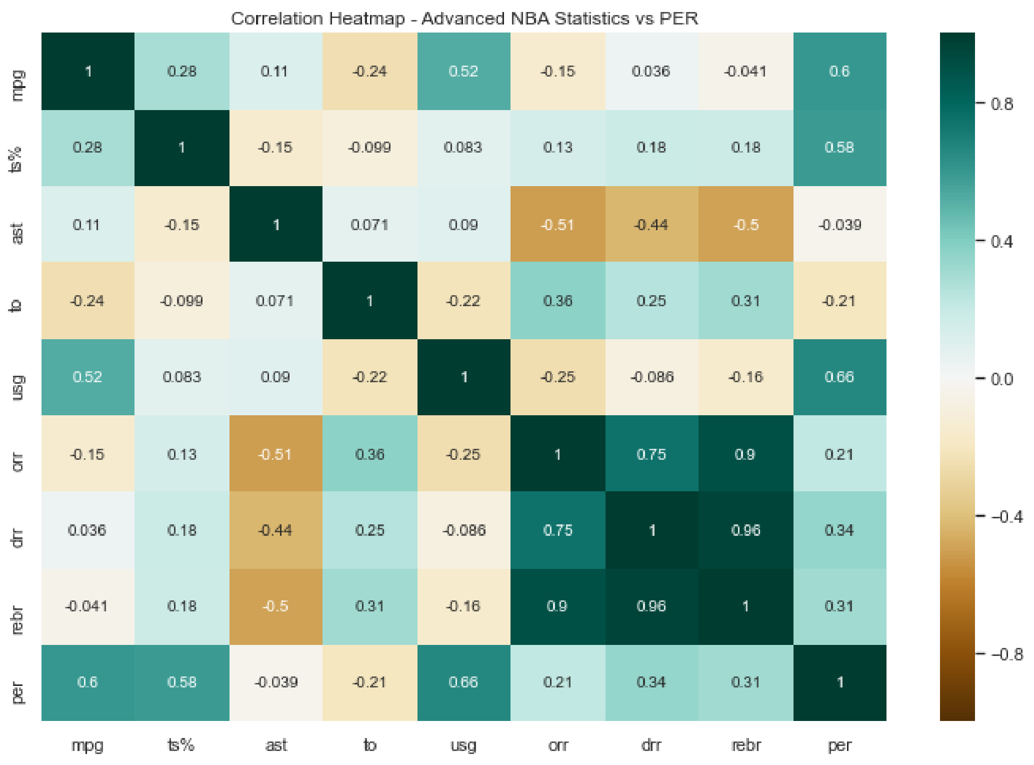


[7]:

	mpg	ts%	ast	to	usg	orr	drr	rebr	per
count	5404.000000	5404.000000	5404.000000	5404.000000	5404.000000	5404.000000	5404.000000	5404.000000	5404.000000
mean	24.545448	0.531777	15.992746	11.131625	18.066155	5.375093	14.653349	10.013583	14.281956
std	7.954939	0.048009	8.033157	2.880580	4.733323	3.851429	5.817942	4.535166	4.252705
min	7.000000	0.338000	1.200000	2.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	17.900000	0.502000	10.000000	9.200000	14.700000	2.000000	10.000000	6.200000	11.320000
50%	24.400000	0.532000	14.100000	10.900000	17.700000	4.000000	13.600000	8.950000	13.920000
75%	31.200000	0.563000	20.600000	12.700000	21.100000	8.100000	18.600000	13.300000	16.630000
max	43.100000	0.725000	48.700000	29.600000	42.500000	22.000000	38.000000	26.700000	31.760000

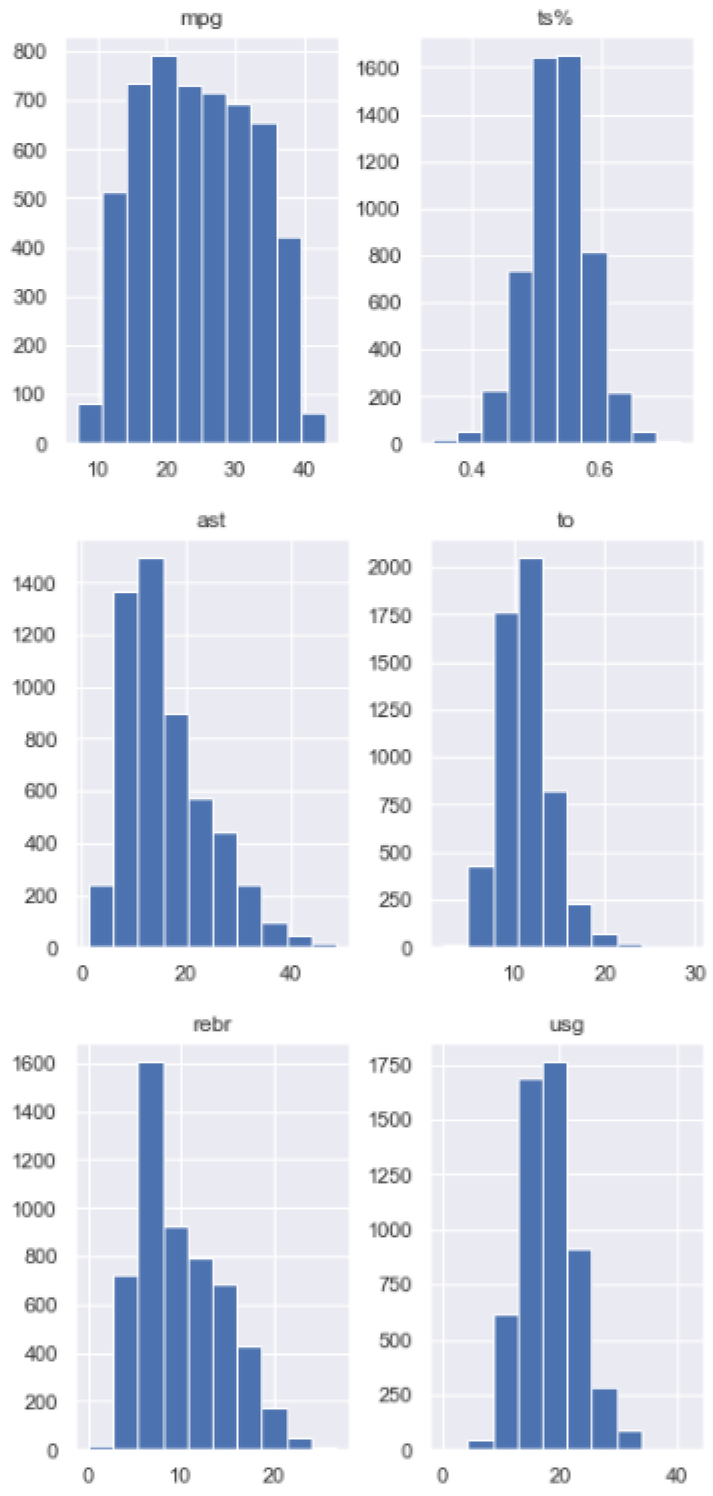
Note that the PER mean is 14.28, very close to Hollinger's' definition of 'league average PER' which is 15.0.

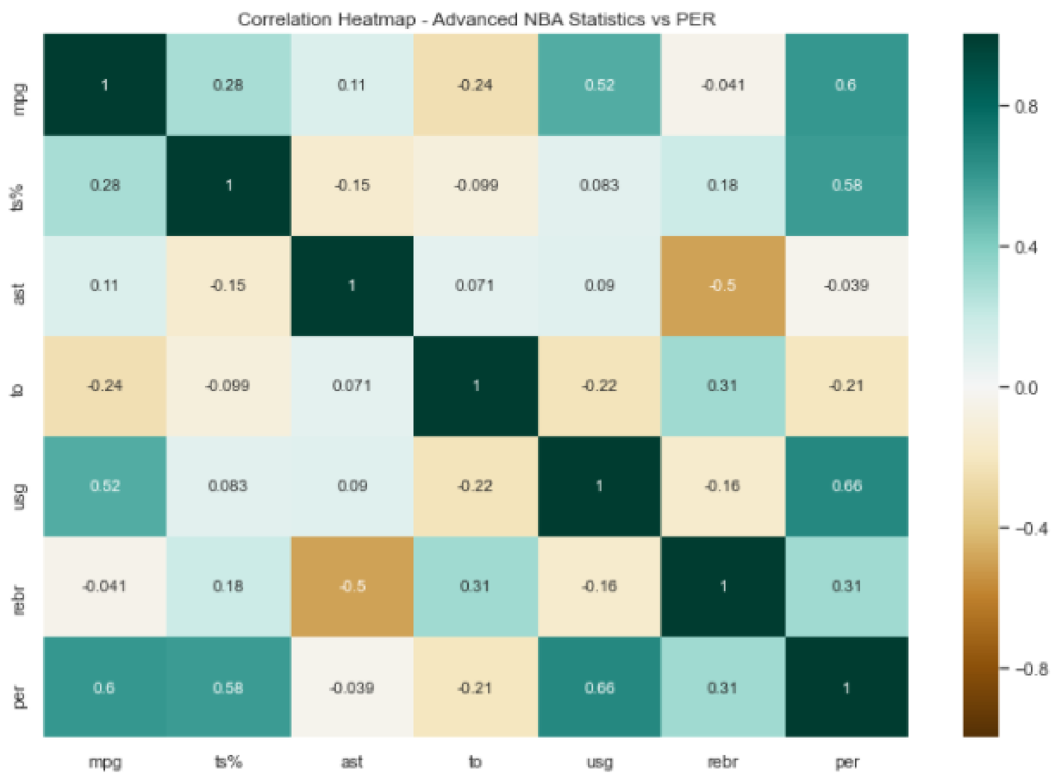
Next, a correlation heatmap was built, to show the correlations between features and the correlations between features and the target variable per:



When evaluating the heatmap, it was determined that there were very strong correlations between the features orr, drr and rebr – makes sense since all are measures of rebounding rate. In an attempt to minimize multicollinearity, the decision was made to remove orr and drr, leaving rebr as the only measure of rebounding rate included in analysis. It is understood tat it may cost some explain ability, but it will help in making the regression analysis more 'clean'.

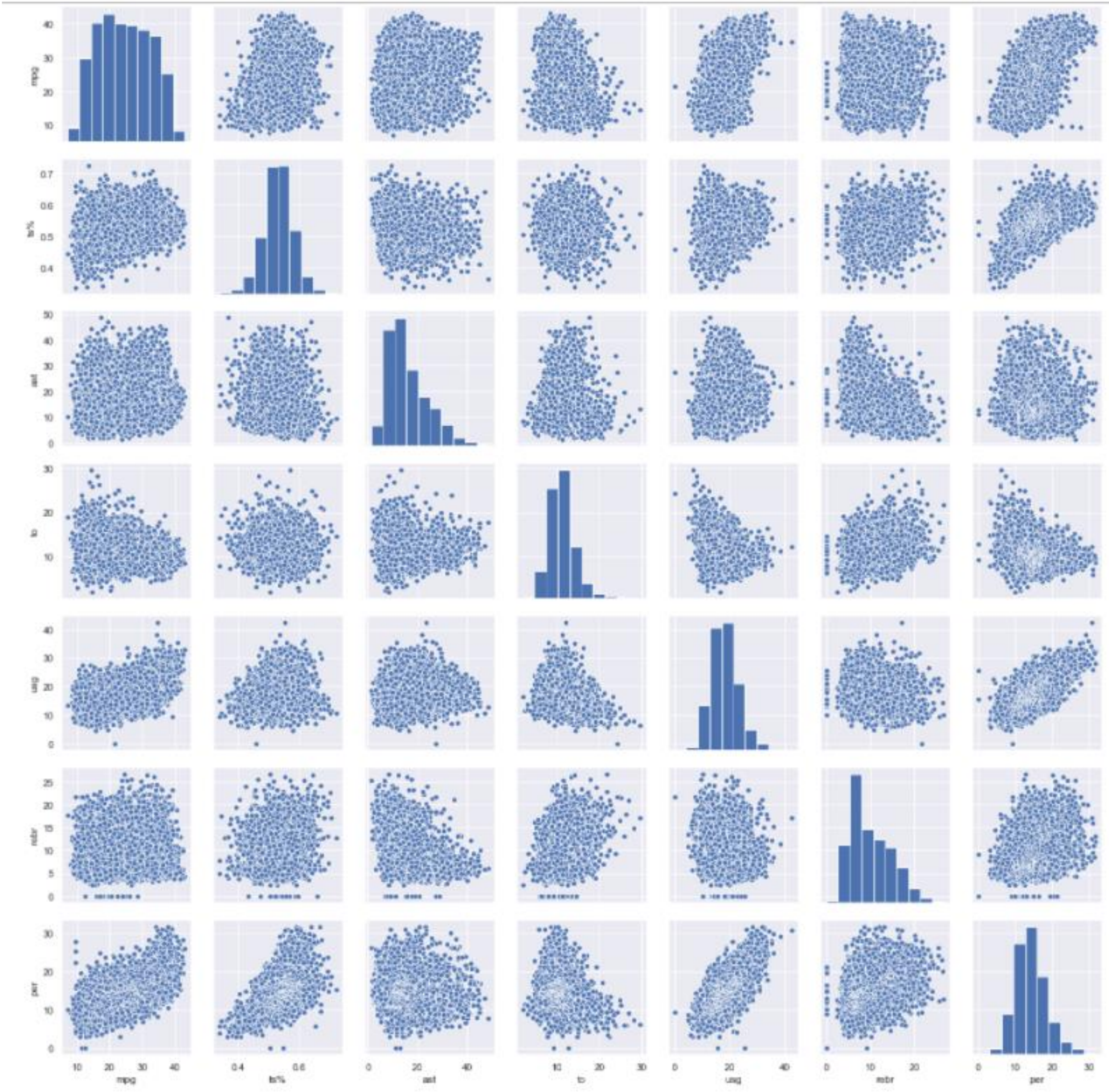
The finalized features were reasonably distributed and had a lower amount of inter feature correlation, as you can see below:


mpg
ts%
ast
to
rebr
usg



<u>**Data Analysis – Regression**</u>

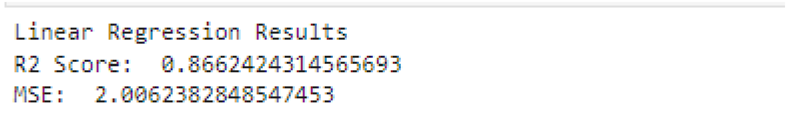
See below a seaborn pair plot of the selected features vs per. It is noted that there is some noisy and probably non-linear relationship between the features and per (see the final column of the pair plot



In accordance with the performance from the previous Regression project, vanilla Linear Regression was carried out on the data, as it showed the best combination of accuracy and explain-ability.

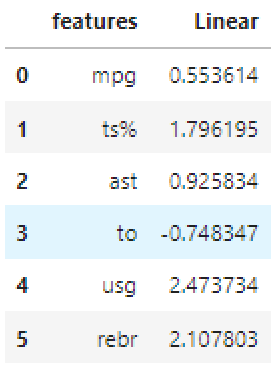
The dataset was initially split into a 70/30 train/test split (3782 training records, 1622 testing records) and scaled using the sklearn StandardScaler object.

The results of the regression, showed (to the 4th decimal point)



```
Linear Regression Results
R2 Score:  0.8662424314565693
MSE:  2.0062382848547453
```

It is the authors opinion that, with limited data, this is excellent model performance (86% accuracy as measured by R2 Score, a MSE of ~ 2.0, and great explain-ability). The coefficients of the regression were as follows – note that the only feature negatively correlated (negative linear coefficient) with PER is turnover rate. This makes perfect sense as it is also the only feature that would be called a "negative contribution" in a game situation (turning the ball over to the other team – good players tend to do this less, as a % of their possession/usage).



	features	Linear
0	mpg	0.553614
1	ts%	1.796195
2	ast	0.925834
3	to	-0.748347
4	usg	2.473734
5	rebr	2.107803

Next, we will see if we can account for non-linearities in the data's relationships through a feedforward neural network model.

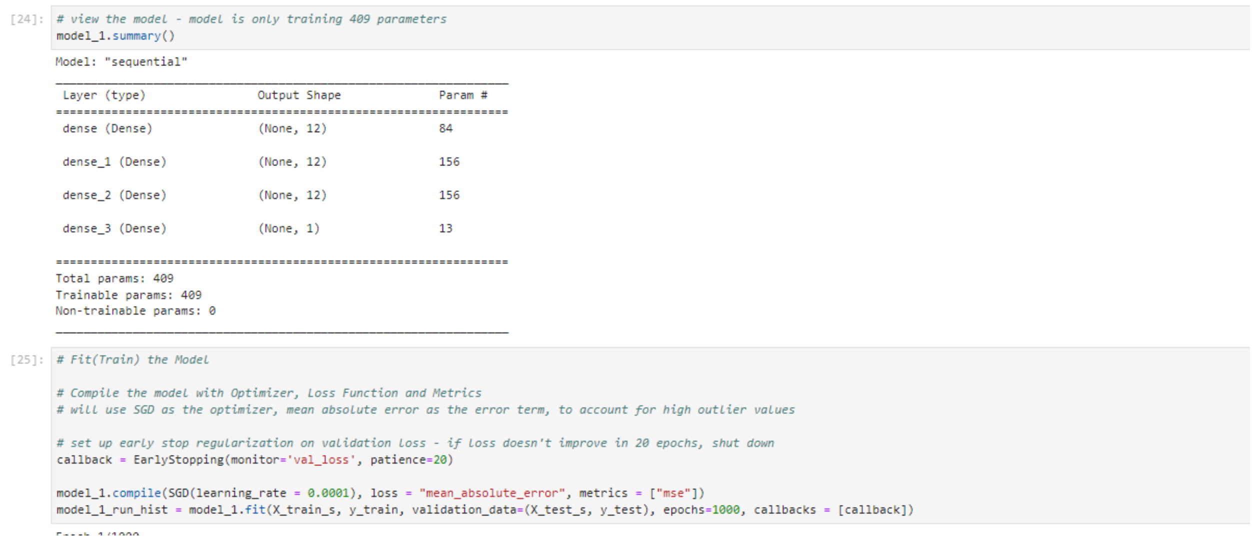
Deep Learning – Feedforward Neural Network

Model 1 – Feedforward Network (3 Hidden Layers, Width = 12 nodes) with SGD Optimizer

For the first model, it was decided to build a 3-layer deep feedforward neural net (multi-layer perceptron), with the following parameters:

- Input layer width = 6 units (6 variables go into determining PER in this study)
- Hidden layer width = 12 units
- Activation functions (hidden layers): ReLU
- Output Layer: 1 unit (PER estimate)
- Activation function at output layer: Linear
- Learning Rate = 0.0001
- Epochs = 1000 (maximum)
- Early Stopping: through research, found an appropriate number of epochs to wait before early stopping (no longer a decrease in the validation loss function) was 20, so patience was set to 20
- Optimizer: for simplicity, chose a standard Stochastic Gradient Descent with no momentum component
- Loss Function: due to the presence of high outliers in the data, the mean absolute error was chosen as the loss function of choice. Again, this was due to findings in literature

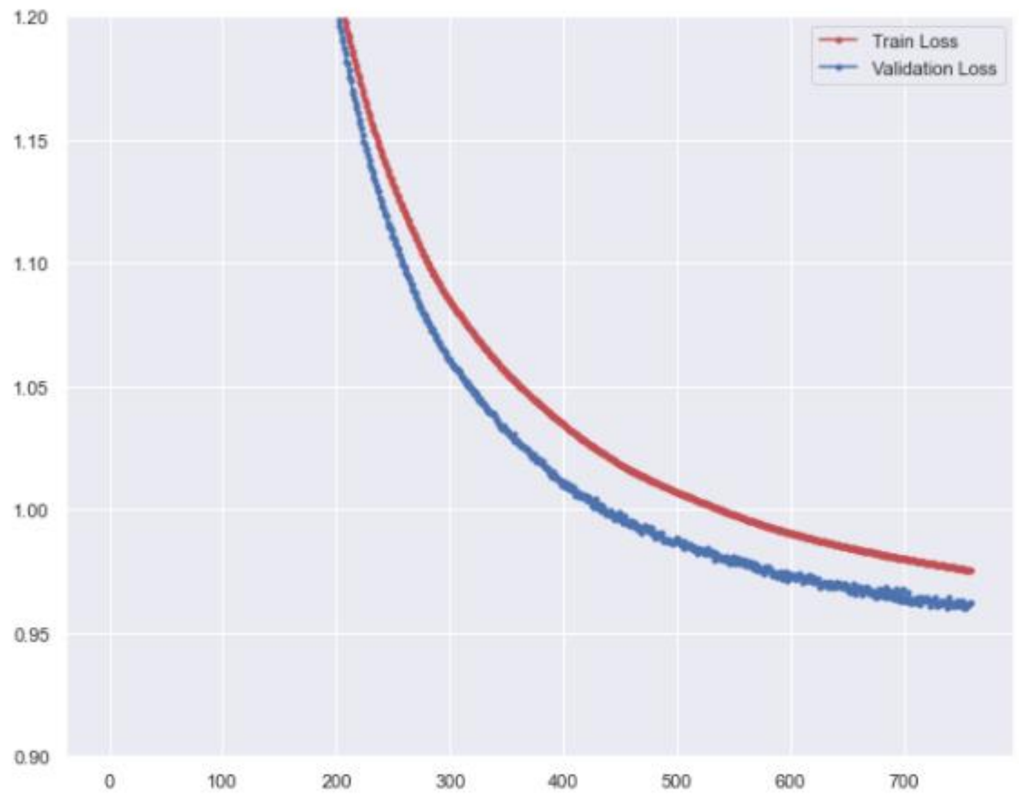
The model as set up would have to train 409 parameters, and did so relative quickly:



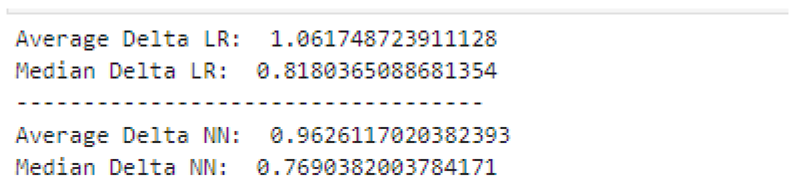
```
[24]: # view the model - model is only training 409 parameters
      model_1.summary()

      Model: "sequential"
      _________________________________________________________________
       Layer (type)                Output Shape              Param #
      =================================================================
       dense (Dense)               (None, 12)                84

       dense_1 (Dense)             (None, 12)                156

       dense_2 (Dense)             (None, 12)                156

       dense_3 (Dense)             (None, 1)                 13

      =================================================================
      Total params: 409
      Trainable params: 409
      Non-trainable params: 0
      _________________________________________________________________

[25]: # Fit(Train) the Model

      # Compile the model with Optimizer, Loss Function and Metrics
      # will use SGD as the optimizer, mean absolute error as the error term, to account for high outlier values

      # set up early stop regularization on validation loss - if loss doesn't improve in 20 epochs, shut down
      callback = EarlyStopping(monitor='val_loss', patience=20)

      model_1.compile(SGD(learning_rate = 0.0001), loss = "mean_absolute_error", metrics = ["mse"])
      model_1_run_hist = model_1.fit(X_train_s, y_train, validation_data=(X_test_s, y_test), epochs=1000, callbacks = [callback])
```

The model ran for 759 epochs before it stopped early. The curve below shows that convergence was relatively smooth, likely owing to the low learning rate:

```
[26]: <matplotlib.legend.Legend at 0x1d9451a4348>
```

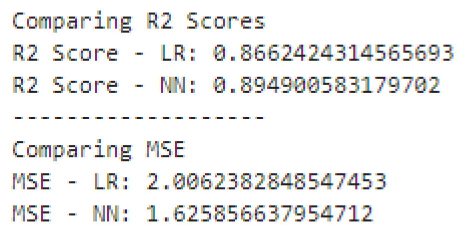


Although not a perfect measure, we did calculate a R2 Score for the Neural Network Regression, for ease of comparison to the Linear Regression previously modeled. Additionally, we also measured the differences in predicted values for both the neural net and linear regression versus the test data, a parameter we were calling delta, as seen below. As seen, the median and average deltas for the neural net we're about 10% better than that of the linear regression:



```
Average Delta LR:  1.061748723911128
Median Delta LR:  0.8180365088681354
-----------------------------------
Average Delta NN:  0.9626117020382393
Median Delta NN:  0.7690382003784171
```

Finally, we compared both the R2 score and Mean Squared Error (we calculated MSE for the neural network as its performance metric along with MAE in the loss function). As seen below, the R2 score of the neural network is slightly higher (again, an imperfect measure) and the MSE is ~ 20% below that of the linear regression.

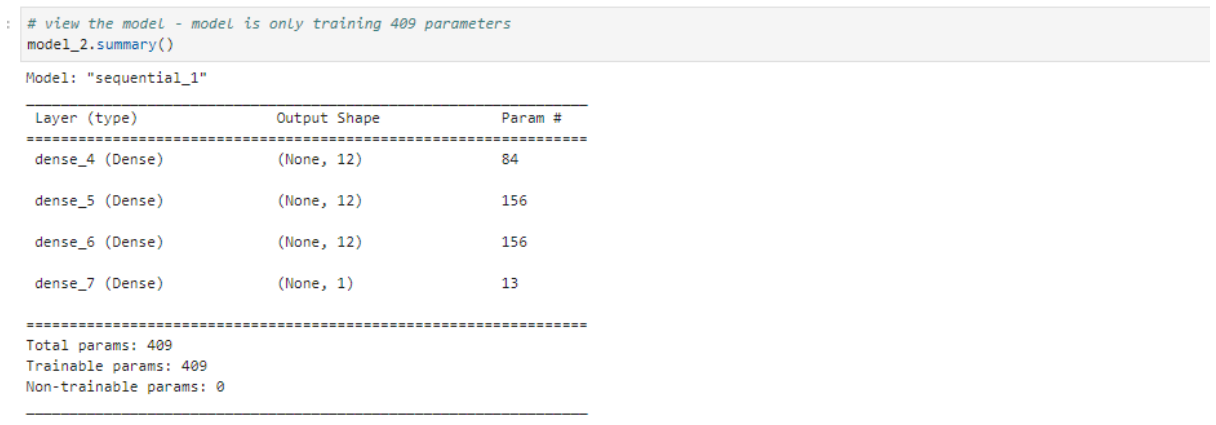


```
Comparing R2 Scores
R2 Score - LR: 0.8662424314565693
R2 Score - NN: 0.894900583179702
-------------------
Comparing MSE
MSE - LR: 2.0062382848547453
MSE - NN: 1.625856637954712
```

Model 2 – Feedforward Network (3 Hidden Layers, Width = 12 nodes) with Adam Optimizer

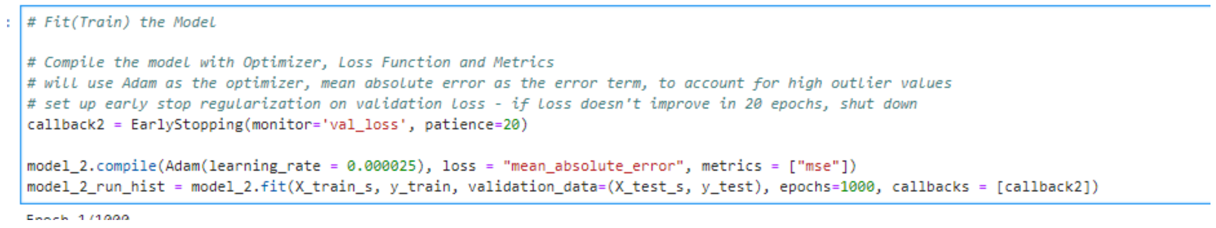
For the second model, the exact same 3-layer deep feedforward neural net (multi-layer perceptron) was built. The only difference here was the optimizer – to gauge the difference, the Adam optimizer, with learning rate tuned to 0.000025 (1/4 of the SGD optimizer learning rate) and default beta parameters was selected.

The model as set up would have to train 409 parameters, and did so relative quickly:



```
# view the model - model is only training 409 parameters
model_2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 12)                84

 dense_5 (Dense)             (None, 12)                156

 dense_6 (Dense)             (None, 12)                156

 dense_7 (Dense)             (None, 1)                 13

=================================================================
Total params: 409
Trainable params: 409
Non-trainable params: 0
_________________________________________________________________
```

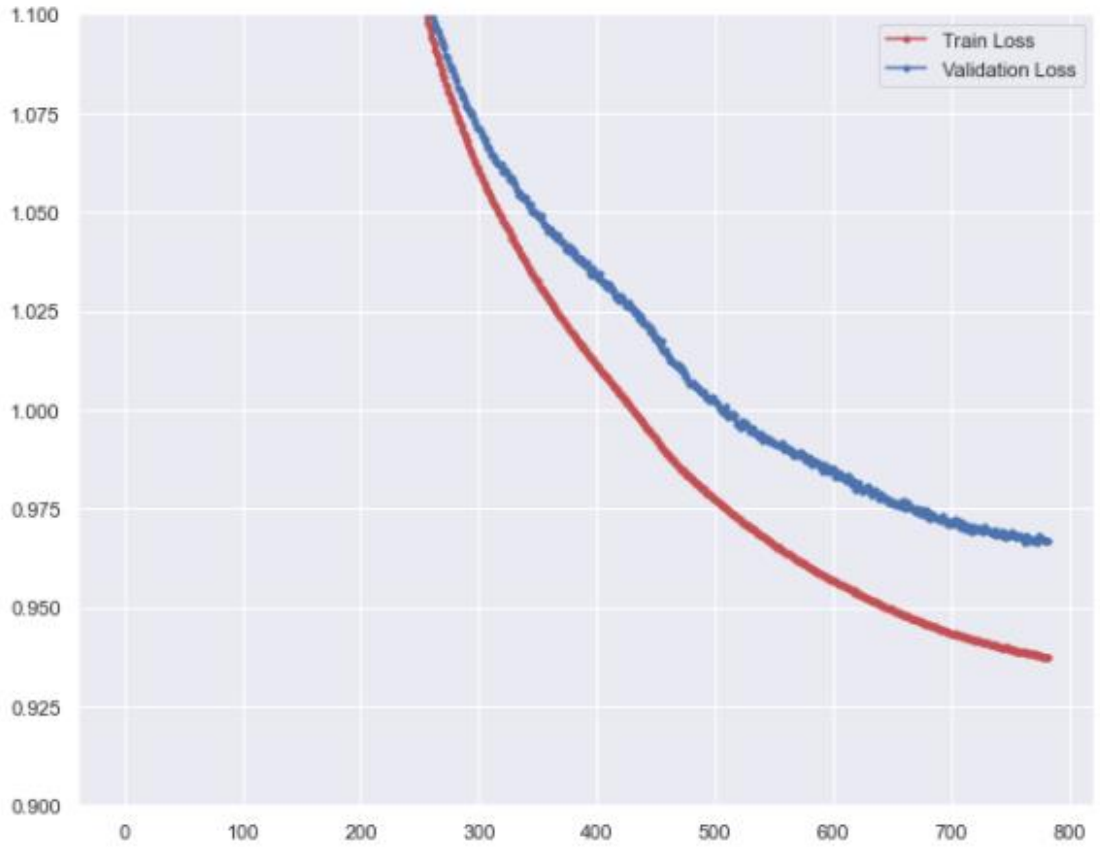


```
# Fit(Train) the Model

# Compile the model with Optimizer, Loss Function and Metrics
# will use Adam as the optimizer, mean absolute error as the error term, to account for high outlier values
# set up early stop regularization on validation loss - if loss doesn't improve in 20 epochs, shut down
callback2 = EarlyStopping(monitor='val_loss', patience=20)

model_2.compile(Adam(learning_rate = 0.000025), loss = "mean_absolute_error", metrics = ["mse"])
model_2_run_hist = model_2.fit(X_train_s, y_train, validation_data=(X_test_s, y_test), epochs=1000, callbacks = [callback2])
```

```
Epoch 1/1000
```

This time, the model ran for 783 epochs before it stopped early. The curve below shows that convergence was relatively smooth, likely owing to the low learning rate:

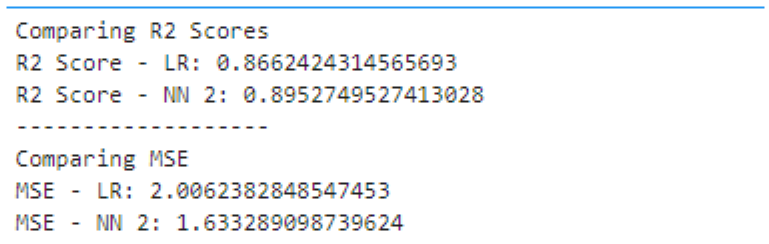
```
<matplotlib.legend.Legend at 0x1d9474fc288>
```



Again, although not a perfect measure, we did calculate a R2 Score for the Neural Network Regression, for ease of comparison to the Linear Regression previously modeled. Additionally, we also measured the differences in predicted values for both the neural net and linear regression versus the test data, a parameter we were calling delta, as seen below. Again, the median and average deltas for the neural net we're about 10% better than that of the linear regression:

```
Average Delta LR:  1.061748723911128
Median Delta LR:  0.8180365088681354
-----------------------------------
Average Delta NN:  0.9668021772939688
Median Delta NN:  0.769151096343994
```

Finally, we compared both the R2 score and Mean Squared Error (we calculated MSE for the neural network as its performance metric along with MAE in the loss function). As seen below, the R2 score of the neural network is slightly higher (again, an imperfect measure) and the MSE is ~ 20% below that of the linear regression.



```
Comparing R2 Scores
R2 Score - LR: 0.8662424314565693
R2 Score - NN 2: 0.8952749527413028
-------------------
Comparing MSE
MSE - LR: 2.0062382848547453
MSE - NN 2: 1.633289098739624
```

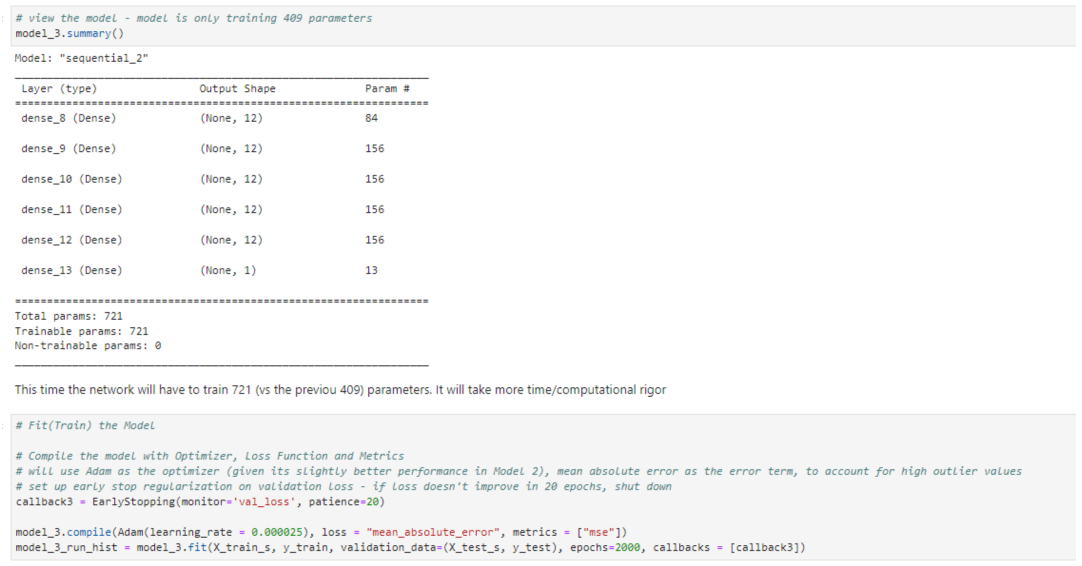
As a note, there was very little in the way of performance difference between the 2 models. You could make the case to use either of these and not be wrong. Going forward, we will use the Adam optimizer for the third model, which will be a deeper network.

<u>Model 3 – Feedforward Network (5 Hidden Layers, Width = 12 nodes) with Adam Optimizer</u>

For the third and final model, it was decided to build a 5-layer deep feedforward neural net (multi-layer perceptron), with the following parameters:

- Input layer width = 6 units (6 variables go into determining PER in this study)
- Hidden layer width = 12 units
- Activation functions (hidden layers): ReLU
- Output Layer: 1 unit (PER estimate)
- Activation function at output layer: Linear
- Learning Rate = 0.000025
- Epochs = 2000 (maximum, assuming it would take longer to train a deeper net)
- Early Stopping: through research, found an appropriate number of epochs to wait before early stopping (no longer a decrease in the validation loss function) was 20, so patience was set to 20
- Optimizer: As in the second model, we chose the same Adam optimizer
- Loss Function: due to the presence of high outliers in the data, the mean absolute error was chosen as the loss function of choice. Again, this was due to findings in literature

The model as set up would have to train 721 parameters, requiring a little more in the way of computational rigor:



```
# view the model - model is only training 409 parameters
model_3.summary()
```

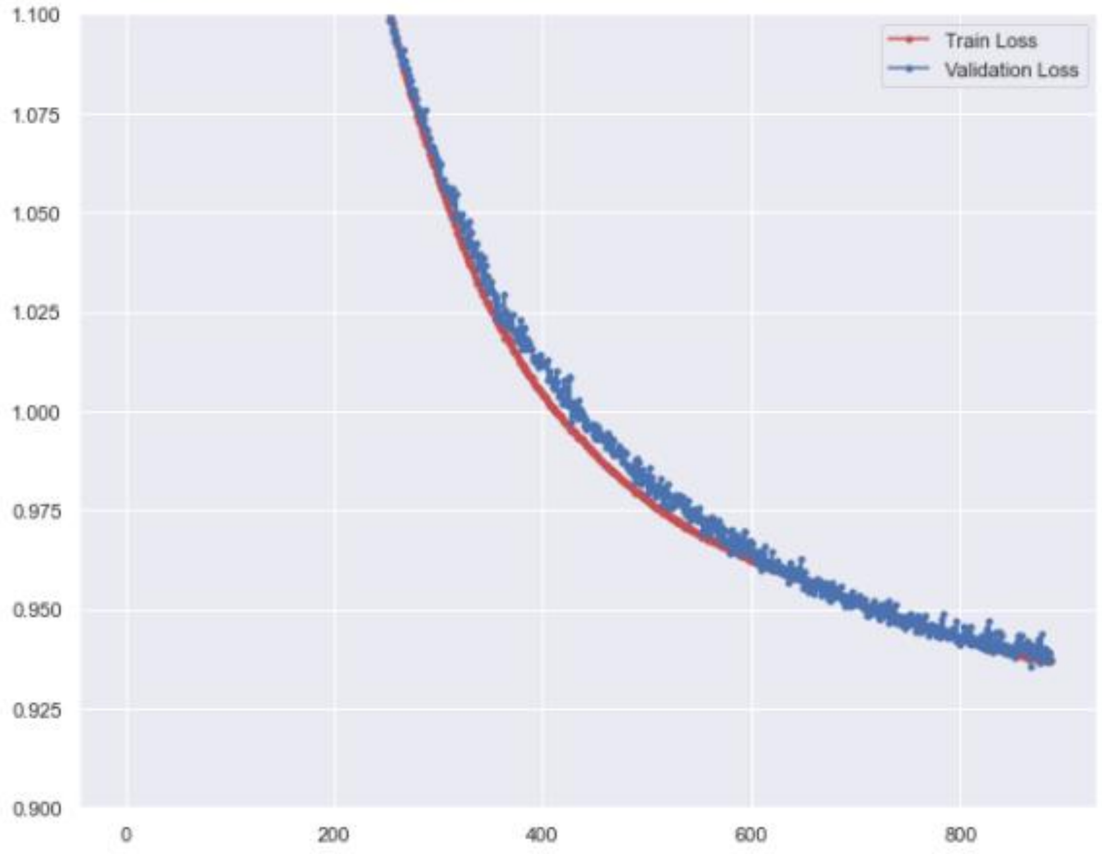
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_8 (Dense)             (None, 12)                84

 dense_9 (Dense)             (None, 12)                156

 dense_10 (Dense)            (None, 12)                156

 dense_11 (Dense)            (None, 12)                156

 dense_12 (Dense)            (None, 12)                156

 dense_13 (Dense)            (None, 1)                 13

=================================================================
Total params: 721
Trainable params: 721
Non-trainable params: 0
_________________________________________________________________
```

This time the network will have to train 721 (vs the previou 409) parameters. It will take more time/computational rigor

```
# Fit(Train) the Model

# Compile the model with Optimizer, Loss Function and Metrics
# will use Adam as the optimizer (given its slightly better performance in Model 2), mean absolute error as the error term, to account for high outlier values
# set up early stop regularization on validation loss - if loss doesn't improve in 20 epochs, shut down
callback3 = EarlyStopping(monitor='val_loss', patience=20)

model_3.compile(Adam(learning_rate = 0.000025), loss = "mean_absolute_error", metrics = ["mse"])
model_3_run_hist = model_3.fit(X_train_s, y_train, validation_data=(X_test_s, y_test), epochs=2000, callbacks = [callback3])
```

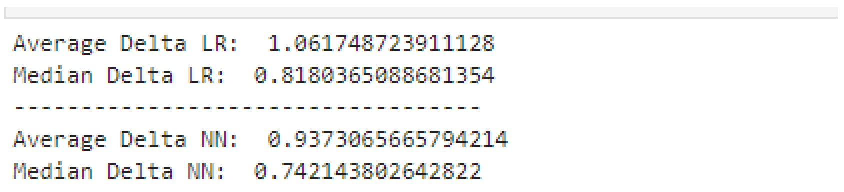
A little surprisingly, the model ran for only 888 epochs (~ 100 additional epochs) before it stopped early. The curve below shows that convergence of the validation loss function was a little less smooth than that of the shallower models. I have read, that depending on the problem, and the sweet spot of network depth that sometimes a deeper network tends to overfit/memorize the data, rather than generalizing, which may be happening here. With that being said, the validation set loss for this network was lower than it was for any of the other trialed networks:

```
<matplotlib.legend.Legend at 0x1d948a86588>
```



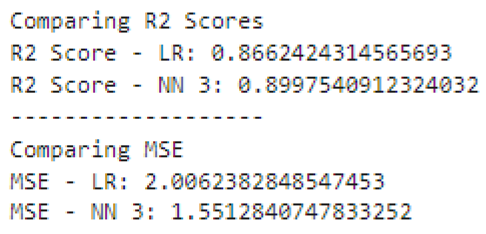
```
Model 2 - 3 Layer NN
Validation Loss:  0.9668021202087402
--------------------------------------------------------
Model 3 - 5 Layer NN
Validation Loss:  0.9373064041137695
```

Again, although not a perfect measure, we did calculate a R2 Score for the Neural Network Regression, for ease of comparison to the Linear Regression previously modeled. Additionally, we also measured the differences in predicted values for both the neural net and linear regression versus the test data, a parameter we were calling delta, as seen below. As seen, the median and average deltas for the neural net were a little better than those of the previous neural nets:



```
Average Delta LR:  1.061748723911128
Median Delta LR:  0.8180365088681354
-----------------------------------
Average Delta NN:  0.9373065665794214
Median Delta NN:  0.742143802642822
```

Finally, we compared both the R2 score and Mean Squared Error (we calculated MSE for the neural network as its performance metric along with MAE in the loss function). As seen below, the R2 score of the neural network is slightly higher (again, an imperfect measure) and the MSE is ~ 25% below that of the linear regression.



```
Comparing R2 Scores
R2 Score - LR: 0.8662424314565693
R2 Score - NN 3: 0.8997540912324032
-------------------
Comparing MSE
MSE - LR: 2.0062382848547453
MSE - NN 3: 1.5512840747833252
```

Key Findings / Take Aways

Tuning the various neural networks was not a straightforward task – namely, deciding upon depth/width, as well as learning rates for loss function convergence. Specifically, it took considerably more time to tune in the 5-layer networks than the 3-layer networks, with only slight performance benefits.

In general, all 3 neural networks performed at least 20% better on a mean squared error basis than the vanilla linear regression. Given the small size of the dataset, it is likely worth it to move forward with any of the tested networks over the linear regression. However, given the computational requirements, if the dataset was going to grow (ie: in 20 years' time, doing a much larger study, or another study incorporating data from different leagues) a pause for thought, considering the business implications of the slight gain in predictive power would be warranted.

There are likely many improvements that can be made to the analysis contained within, simulating further to understand how differences in width/depth affect performance (the simulations practiced within the scope of this project, widening the neural nets up to 18 nodes, did nothing to improve performance).

From the scope of this study however, it can be said that deeper is better, with respect to neural nets, although that will only last to a point, where overfitting will begin to dominate.

Further Analysis

As described above, plain linear regression offers quit a good 'bang for the buck' analysis – it provides acceptable accuracy, with a relatively low error and great explain-ability.

Further analysis here would center on the business objective – is the customer/sponsor looking for:

- Accuracy, in which case the neural nets (deeper) provide about 20% improved accuracy
- Explain-ability, where the linear regression and its corresponding weights rule the day

In general, given the relatively small data set (advanced stats/PER data only goes back so far…), it may be worth it to attempt to put together a data set with longer history (going back to the 1980s if available) or to incorporate data from different leagues (feeder leagues, such as the CBA/NBA G-League, although we're unsure if that would help or hinder the analysis).

Finally, some attempt could be made to expand the data set currently used, to include more advanced statistics. However, without subject matter expert assistance it would be difficult to tie the data to a single (and little mis-understood) measure such as PER.