Predicting NBA Point-Differentials using Recurrent Neural Networks

Jordan Corbett-Frank
12 May 2020

1. Introduction

Recurrent neural networks (RNNs) are a popular type of model to perform natural language processing and time-series prediction tasks. Unlike conventional feed-forward networks, RNNs assume samples in a data-set are dependent on one another. As a RNN processes input samples, it maintains a state – the "hidden" state – and combines this state with the input samples to produce output samples [1]. Figure 1, taken from [1], illustrates this concept. Our goal was to apply an RNN-based network to the problem of predicting the point differential of various NBA games based on each team's performance over the last N games. This problem was chosen primarily due to the rich data-set of available NBA statistics.

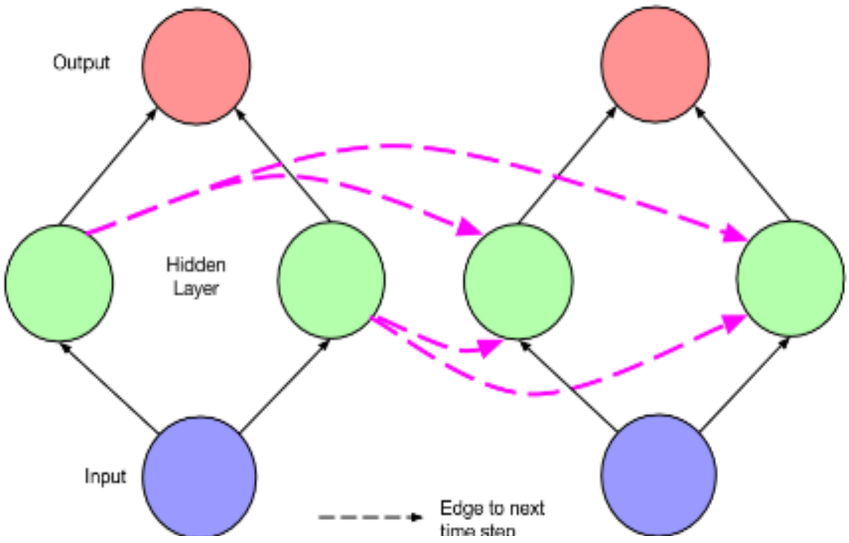

Figure 1: Recurrent Neural Network Architecture

2. Data

The data was retrieved using *nba_api*, a Python API client, meant to facilitate the use of data from www.nba.com [2]. Using this package, records for roughly 47,000 games from 1983-2020 were downloaded. The data was formatted such that there are two rows for each game – one for each team. Figure 2, below, shows an example of the records for one game. One can see that each row contains statistics on that team's performance for that game, as well as metadata, like the game ID, team ID, game date, etc.

SEASON_ID	TEAM_ID	TEAM_ABBREVIATION	TEAM_NAME	GAME_ID	GAME_DATE	MATCHUP	...	DREB	REB	AST	STL	BLK	TOV	PF
22019	1610612737	ATL	Atlanta Hawks	21900969	2020-03-11	ATL vs. NYK	...	38.0	53.0	26	6.0	3	17	25

Figure 2: Raw NBA game records

Data Cleaning and Pre-Processing

While data cleaning and pre-processing is not necessarily exciting, we spent a significant amount of time doing it and, so, we would feel remiss if we did not provide the details.

While, overall, the data was very clean a small number of games had to be thrown away because of missing values or because there were instances in which there were not two rows with the same game ID. About 0.3% of the data was removed, so the effect on our prediction should be negligible. A link to the code that downloads the data, throws out the invalid rows, and saves the cleaned version can be seen in section A.

With the data sufficiently clean, we started the pre-processing steps. First, home and away columns were added to the data-set; the values were determined by the presence of a "vs." or an "@" in the "match-up" column as seen in Figure 2. While only one column is needed to completely represent the information (home could be represented by 0 and away could be represented by 1), the value is one-hot encoded into two columns to represent the categorical value. If this was not done, it is possible that our model would interpret being the home team as intrinsically better or worse than the away team – which could or could not be true.

Next, the season IDs were refactored. The NBA season ID consists of an integer α followed by the year that the season began. The first integer, α, can be one, two, or four. Based on the dates where we saw different values for α, we believe that an α of one or two signifies that the game occurred in the regular season before and after the New Year, respectively, while an α of four signifies that the game is a playoff game. Based on this information we converted the season ID values to the year that the season began and whether it was a playoff game (1) or regular game (0). Again, we one-hot encoded the playoff information. With this done, we moved onto engineering the time features.

For each game, we converted the game date into the following: the day of the week that the game was played, the number of days into the season, and the number of games into the season (for each team). While two teams that play on the same date will obviously play on the same day of the week and the same number of days into that season, they do not necessarily have to have played the same number of games in that season. One team may have had more prior games that season than the other team. The day of the week was one-hot encoded into seven features, as they are categorical, while the other two features were not as they are numerical.

Next, the team categories were one-hot encoded. This expanded each team category into a vector of length 42 teams as there have been 42 NBA teams from 1983-2020. Additionally, each column was scaled so that its range was between zero and one. The transformation was done using scikit-learn's MinMaxScaler, so that it could be stored for future use [3]. Finally rows with the same game ID, i.e. teams playing against each other, were merged so that each game consisted of a single row and contained the stats for each team's performance during that game. Figure 3, below shows one row. The *_H* suffix refers to the home team that game and the *_A* suffix refers to the away team.

SEASON_ID	TEAM_ID_H	TEAM_ABBREVIATION_H	GAME_ID	GAME_DATE	MINUTES_H	PTS_H	...	STL_A	BLK_A	TOV_A	PF_A
21983	1610612746	SDC	28300252	1983-12-11	240	118	...	6.0	7	9	30

Figure 3: Merged NBA records

With the feature engineering and scaling complete, we moved onto organizing the data so that it could be fed into our model. For each game, we stored that game's home team's N prior games and that game's away team's N prior games. The game count resets at the beginning of each season, so only games that were each team's $N+1^{th}$ game were eligible for our model. Additionally for each game, we stored the engineered time features and the point differential. Figure 4 shows what this data for a single game would look like. Tables 1 and 2 list all of the features used.

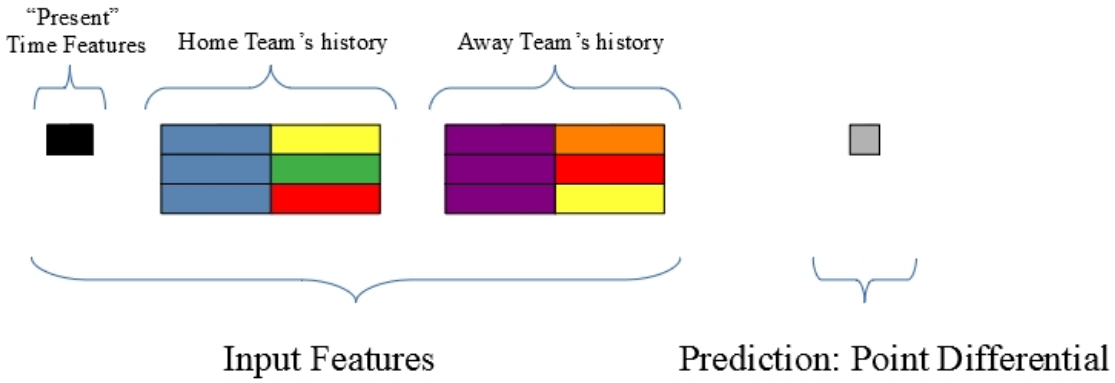

Figure 4: One sample from the organized data-set

The colors in the histories are meant to represent unique teams. It may be useful to give a concrete example. If Team A was playing Team H on March 3, 2008 at Team H's arena, the "present" time features would the engineered time features for March 3, 2008 and the point differential would be Team H's score minus Team A's score. The home team's history would contain the data from Team H's previous N match-ups, including their opponents stats. The away team's history would contain the data from Team A's previous N match-ups, including their opponents stats. A link to the pre-processing code can be seen in section A.

3. The Model

To perform the prediction we concatenated the histories, fed them into a gated recurrent unit (GRU) [4], concatenated the last hidden state with the "present" time features and fed that into a single layer. Figure 5, below, illustrates this. The idea is that the GRU should learn the function that best "summarizes" the difference in team skill based on each team's previous performances. For example, if

a human was asked what this function might be, he or she might say it is the average over the past N games – that the average performance of each team over their last N games is the best indicator of the outcome of the "present" game. We then take the output of the GRU and concatenate the "present" time information, so the input to the final layer is akin to both the home team's and away team's performance over the last N games in addition to the time information of the current game. The final layer then performs the regression using this feature vector.

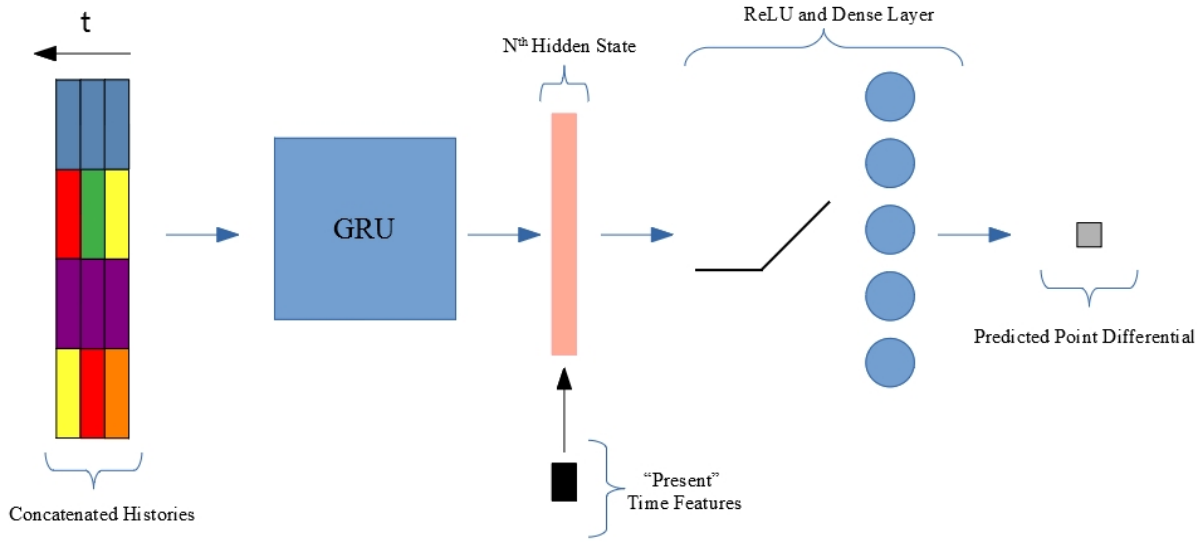

Figure 5: Model architecture

4. Model Implementation and Results

PyTorch was used to implement the model and train it [5]. A link to the PyTorch code can be seen in section A. Additionally we used Google Colab to leverage GPU acceleration for free. We initially tried to train the network locally on a CPU but it was prohibitively slow.

As Table 1 shows, the number of input features fed into the GRU was 256. The majority of these features came from one-hot encoding the teams. There were 42 teams, and two histories, each with two sets of teams, so 156 of the 256 features contained information on which teams were playing. We set the hidden dimension to 100 in an attempt to compress some of these features. We also tried setting the hidden dimension to 256, but no noticeable change was noticed. There were 13 "present" time features – as Table 2 shows – so the final dense layer consisted of 113 nodes. The number of games used to construct the histories, N, was set to 20. Because this eliminates games in which both teams do not have 20 previous games played that season, this left us with 35,493 games to train and test on.

Mean squared error was used as the loss function while the Adam algorithm was chosen to optimize the network's weights [6]. The sets of data formed by the "present" match-up, histories of the teams to play, and the point differential were randomly split into training and testing data-sets with 80% of the data used for training and 20% of the data used for testing.

The training and testing losses as functions of the number of epochs used to train the data can be seen in Figure 6. One can see that the network starts to over-fit to the training data at around eight epochs, as indicated by the plateau/rise of the test loss. The best test-loss of 150.7 occurs at epoch eight, which corresponds to an average point differential error of 12.3. Additionally, we can use the predicted point differential to determine the predicted winner. A positive point differential indicates the home team won while a negative differential indicates the away team won. This predicted win/loss accuracy as a function of epoch can be seen in Figure 7. The best accuracy achieved is 78.9% at epoch seven and drops down to 73.7% at epoch eight – where the best test loss is found. One can see that there is less variation in this win/loss accuracy, which makes sense. A slight change to the weights in the network will result in a different regression value, but the value is less likely to change sign unless it is hovering around zero. Overall we are satisfied with the results. While it would be nice to reduce the test loss further, a win/loss accuracy of 70-80% seems quite good.

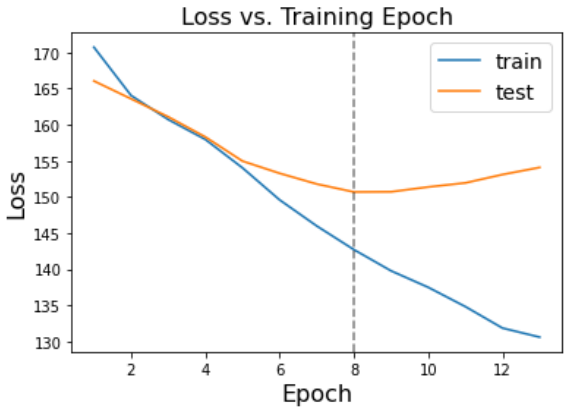

Figure 6: Training and test loss vs. epochs trained

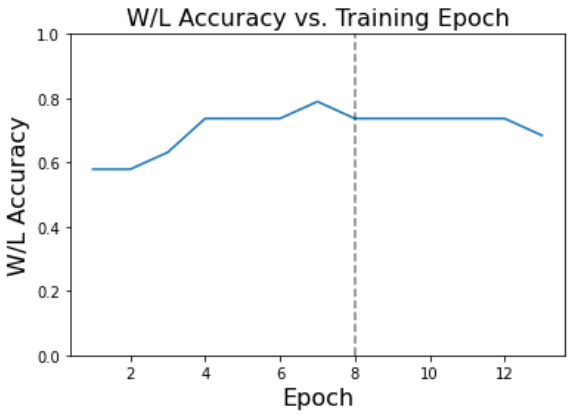

Figure 7: Win-loss accuracy vs. epochs trained

5. Conclusions and Future Work

Using an RNN-based network we predicted point differential and win/loss accuracies for NBA games using data from over 30 years of play. At best, our predicted point differentials were 12 points off from the true point differentials.

During this project we stepped through the entire machine-learning engineering process, from data retrieval and cleaning to model design and evaluation. By far, the most time-intensive portion was the data cleaning and organization. Because of the long execution times of various steps in the pipeline, Jupyter Notebooks were used. While they are nice for quick data exploration and visualizing results, they quickly become unwieldy. If we were to do this again, we would focus on maintaining better organized code so that we could focus more on the interesting portion of the work: the machine learning.

Future work would include exploring different hyper-parameters, such as learning rate, the hidden dimension, etc. Again the organization and storage of these different experiments would need to be a priority. Additionally we could structure our PyTorch data-set to automatically pad the sequence lengths of data in the same mini-batch. This would allow us to pass variable length sequences into our model, which would mean instead of looking back a fixed N games, we could pass both team's entire season histories. In other words if we were predicting the point differential for each team's 60^{th} game of the season, the history would be 59 games long, or if we were predicting each team's second game the history would be one game long. This would allow us to feed more data into the model. Finally to improve further upon these results, we would like to look towards Graph Neural Networks [7]. In each team's history we stored the team's performance as well as their opponents performance in each game in the history. However we could go further than this, storing the opponents' histories as well and their opponent's histories and so on. In other words we would be looking at the entire time-dependent graph of the NBA season. We initially started down this path, but quickly realized we would not be able to finish the project in time.

6. Tables

Table 1: Features used per team history; there were two histories for each game: one for the home team and one for the away team

Category	One-hot encoded	Columns used	Team-Dependent	Total Columns Used
Home/Away	Y	2	N	2
Season	N	1	N	1
Playoff	Y	2	N	2
Day of Week	Y	7	N	7
Days into Season	N	1	N	1
MIN	N	1	N	1
PTS	N	1	Y	2
FGM	N	1	Y	2
FGA	N	1	Y	2
FG3A	N	1	Y	2
FG3M	N	1	Y	2
FTM	N	1	Y	2
FTA	N	1	Y	2
OREB	N	1	Y	2
DREB	N	1	Y	2
AST	N	1	Y	2
STL	N	1	Y	2
BLK	N	1	Y	2
TOV	N	1	Y	2
PF	N	1	Y	2
Games into Season	N	1	Y	2
Team	Y	42	Y	84
			Total Columns:	128

Table 2: "Present" game's time features

Category	One-hot encoded	Columns used	Team-Dependent	Total Columns Used
Season	N	1	N	1
Playoff	Y	2	N	2
Day of Week	Y	7	N	7
Days into Season	N	1	N	1
Games into Season	N	1	Y	2
			Total Columns:	13

7. References

1. Lipton, Z., Berkowitz, J., Elkan, C. A Critical Review of Recurrent Neural Networks for Sequence Learning. *ArXiv preprint arXiv:1506.00019, 2015*

2. *https://github.com/swar/nba_api*

3. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*.

4. K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the properties of neural machinetranslation: Encoder-decoder approaches.*arXiv preprint arXiv:1409.1259, 2014*

5. *Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., Facebook, Z. D., Research, A. I., Lin, Z., Desmaison, A., Antiga, L., Srl, O., & Lerer, A. (2019). Automatic differentiation in PyTorch. Advances in Neural Information Processing Systems 32.*

6. Kingma, D. P., & Ba, J. L. (2015). Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*.

7. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2020). A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*. https://doi.org/10.1109/tnnls.2020.2978386

A. Appendix

The code can be found at https://github.com/jcorbettfrank/mit180651. There are two files there: *getData.ipynb* downloads the data, cleans it, and stores a cleaned version. The other file, *rnn3_colab.ipynb* does everything else: loads the cleaned data, pre-processes it, defines the model, trains the model, and tests the model. Below is the PyTorch code for the model. We list it so that there is some code in this write-up.

```
import torch.nn as nn

class RNNregressor(nn.Module):
        def __init__(self, input_dim, hidden_dim, output_dim, currFeats  n_layers=1,
                dropP=0,batch_first=False):
                super(RNNregressor, self).__init__()
                self.hidden_dim = hidden_dim
                self.n_layers = n_layers
                self.input_dim = input_dim
                self.batch_first = batch_first
                self.gru = nn.GRU(input_dim, hidden_dim, n_layers, dropout=dropP,
                                                        batch_first=batch_first)
                self.dense = nn.Linear(hidden_dim+currFeats, output_dim)
                self.relu = nn.ReLU()

                if self.batch_first:
                   self.indexFN = self.indexBatchFirst
                else:
                   self.indexFN = self.indexSeqFirst

        def indexBatchFirst(self, out):
                return out[:,-1,:]

        def indexSeqFirst(self, out):
                return out[-1,:,:]

         def forward(self, x, curr):
                out, _ = self.gru(x) #out will be [N, L, output_dim]
                outVec = self.indexFN(out)
                l_in = torch.cat((curr, outVec),axis=1) #last sequence, last layer, all in batch
                return self.dense(self.relu(l_in))
```