Baseball Win Predictions

Juliette Love
Department of Computer Science
Stanford University
jullove@stanford.edu

1 Introduction

There are countless statistics in baseball—sites like Baseball Reference(2) and FanGraphs(1) track over 100 different metrics for each player. There is also a high volume of game data available—each team plays at least 162 games in a season (with some exceptions, such as this year). However, baseball game results are highly variable; on any given day, the best team in baseball could lose to the worst, due to factors like home field advantage, the pitchers in each game, and the teams' travel schedules. Because of this variability, human-level performance is not a particularly good benchmark for prediction accuracy, as even experts routinely make incorrect predictions.

This project aims to build a deep learning model that can effectively predict baseball game results. The high volume of data but unpredictability in outcomes offers a unique opportunity for deep learning to provide value over existing methods for this task. A model that can provide baseball game win predictions could be used in a number of ways. Many baseball game information sites, fantasy baseball sites, and betting platforms all provide some form of outcome prediction (ESPN's Gamecast is a common example), but the models they use are often rudimentary and ineffective. This project explores the ability of neural models to make accurate game outcome predictions.

2 Prior Work

Some work has been done applying Deep Learning and other Machine Learning techniques to the problem of baseball game outcome prediction. Elfrink and Bhulai(6) used a fully-connected network architecture on basic game metrics and achieved 55% prediction accuracy. However, the most comprehensive, and likely the most successful, attempt was from Soto-Valero(8), who introduced some player-specific features. The study explored various types of architectures, achieving 56% accuracy with a fully-connected network.

Other work has been done on related problems, such as predicting other features of baseball games given related data. Calzada(4), the paper that began the DeepBall project(3), applies a recurrent architecture to predict player outcomes, such as hitting and pitching statistics, in a particular season based on his results from previous seasons. They first feed vectors of previous-season statistics from the player in question into a unidirectional LSTM. They then combine the output of the LSTM with two static inputs, the results of feeding stats particular to that season and the player's biographical data through separate fully-connected layers. These fully-connected layers form the inputs to a single fully-connected layer which makes a numerical prediction for a given stat for that player (for example, RBIs) for the upcoming year.

3 Task Definition

Given an input vector x of data about a team playing in a game, the problem is framed as a binary classification into "win" ($y = 0$) and "loss" ($y = 1$), representing whether or not the team in question will win the game. Over the course of the project, the specific information contained in the vector x

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

has evolved (as is explained in later sections of this report). Binary classification accuracy is used to evaluate the classification methods employed. Accuracy is a viable metric for this task because the underlying data is equally distributed - the classes "win" and "loss" appear with equal probability in the data.

4 Fully-Connected Model

As a baseline, I implemented a fully-connected network and achieved comparable results to Elfrink and Bhulai(6) and Soto-Valero(8) (56% accuracy).

4.1 Dataset

I generated the data for the fully-connected model by pulling and parsing batting and pitching statistics for each MLB player by year using a combination of different baseball statistics databases.

I employed the baseball-scraper API(9) to compile a player lookup dictionary containing around 2000 batters per year and 1500 pitchers for each year since 1990, with 31 batting metrics and 44 pitching metrics reported for each player. I then pulled the results from, and players that appeared in, each game from the past ten years by scraping baseballsavant.com (the home of MLB's Statcast). I processed Statcast's raw HTML into concrete game-level statistics, corrected errors, and standardized across naming conventions. For each game, I combined the two datasets to get statistics for the players that appeared in the game, using each player's team data to identify in which lineup that player appeared. Finally, I concatenated the features from each player into a full feature vector for each game. This gave me an input matrix $M \in R^{n \times m}$ where each of the m columns corresponds to one game and each of the n entries in a datapoint m_i is a particular statistic corresponding to a player in the game. This yielded a dataset of $m = 12692$ training examples with $n = 794$ features. However, this method of player feature vector concatenation is flawed, as it concatenates the batters from each team in a random order. I thus augmented this data by including multiple input entries for each game, where the batters on a given team are ordered in different ways. Including 10 possible orderings yielded a more reasonable dataset with 126920 training examples.

The output $\hat{y}_g \in 0, 1$ is a prediction of whether or not the team won the game. This is then compared to the true outcome of the game y_g to train and evaluate the classifier.

4.2 Results

I tested different combinations of hyperparameters and layer sizes to find the best accuracy I could with the fully-connected model (Figure 1). With a dropout rate of 0.5, an L2 regularization strength of 0.0001, and three hidden layers of 200 neurons each, I achieved a peak validation accuracy of 0.561; this is comparable to the results achieved by Soto-Valero with a fully-connected model. The network uses cross-entropy loss and an Adam optimizer.

Figure 1 shows that these hyperparameters are able to strike a balance between underfitting and overfitting. The models without dropout applied (0, 2) produced a large gap between the training and validation accuracy, due to overfitting. The models with a lower regularization strength of 0.00001 (0, 1) also yielded a large gap in training and validation accuracy. The models with a regularization strength of 0.001 (3, 5) had little gap between the training and validation accuracy, but both accuracies for those models were lower, suggesting that the high regularization penalty is leading to high model bias. Although a deeper, smaller network with four layers with 100 neurons each has fewer parameters than the larger three-layer network, the various models I trained with this configuration had varying accuracies and losses, suggesting that the other hyperparameters (dropout rate and regularization strength) had a more significant effect on the results.

However, the nature of baseball games means that predicting a game outcome is very difficult when the input data about the players is aggregated across the previous year. Game-to-game results are highly variable; for example, results in a series of two to four games between two teams are often split (with both teams winning at least one game). While the outputs $y_{g1} = 1$ and $y_{g2} = 0$, for example, the input vectors x_{g1} and x_{g2} would be nearly identical. As such, the results of a predictor based on data aggregated across the previous year is likely limited by unavoidable error. Although the cumulative metrics it uses for predictions are likely better for measuring a player's overall production

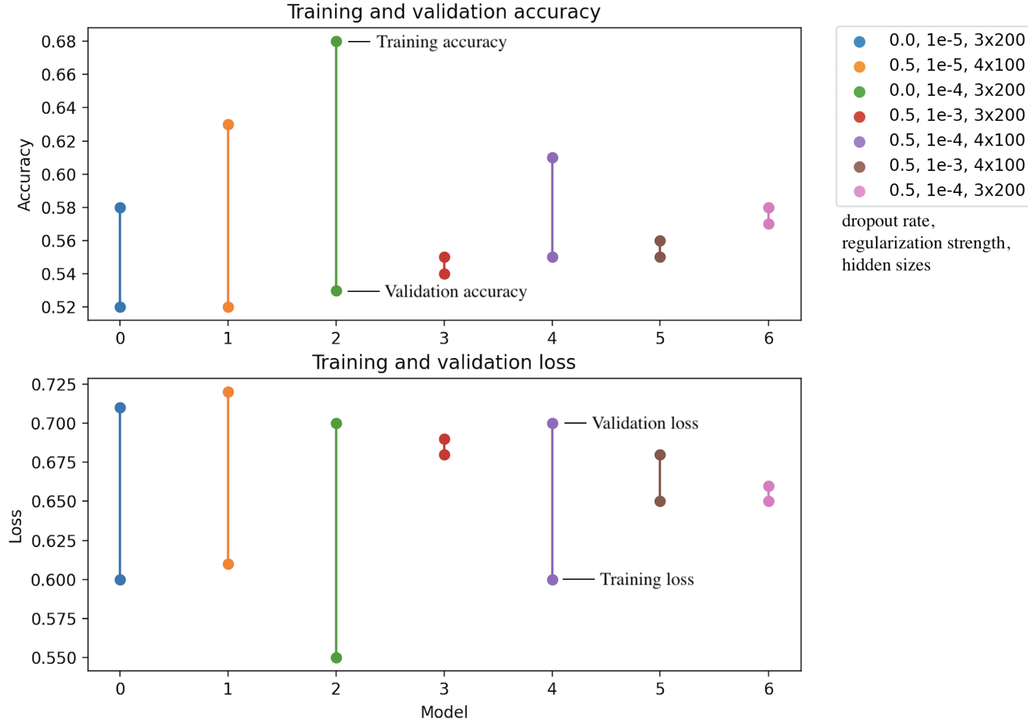

Figure 1: Comparison of different fully-connected architectures and hyperparameters

or expected value over the course of the next season, they may not be the best for predicting a single game outcome. The metric of human-level performance suggests this might be the case as well—due to this variability, it seems unlikely that even experts could significantly outperform a classifier with this accuracy based only on inputs of that nature.

5 Recurrent Model

Due to these limitations, I then reformulated the input to the problem as a sequence of game results from the previous days, instead of aggregate results from the past season, under the hypothesis that a game outcome may be easier to predict given more fine-grained, recent information.

5.1 Dataset and Architecture

Using a recurrent model required generating a time series dataset. To do so, I used baseball-scraper's Baseball Reference modules to generate datasets of both yearly team summary statistics as well as each team's schedule in each year. I also pulled statistics about each game in the team's schedule, such as the number of innings, the opponent, the game time, and the team's ranking at that time the game was played. I reformatted the sequential data (game results) into a time-series with the corresponding result labels for each game; this yielded a total of 114,981 training examples with 11 features in each one. I reformatted the yearly summary statistics into a matrix with 114,981 examples (each row for a given team in a given year is identical) and 26 normalized features (one for each team).

The model then takes two inputs for each training example - a series of vectors $x_g^1, \ldots, x_g^{t-1}$, where g is the tth game in the season, and a static vector x_g^s, representing the team-level information that is constant across the entire sequence. Thus, the predictor can utilize information about the players and results of each individual game from the current season that had been completed before game g took place.

The sequence of vectors $X_g \in \mathbb{R}^{(11 \times t-1)}$ is then fed into an LSTM to produce an output representation vector $z_g^t \in R^{11}$. In some versions I explored, this vector is then fed into a single fully connected layer

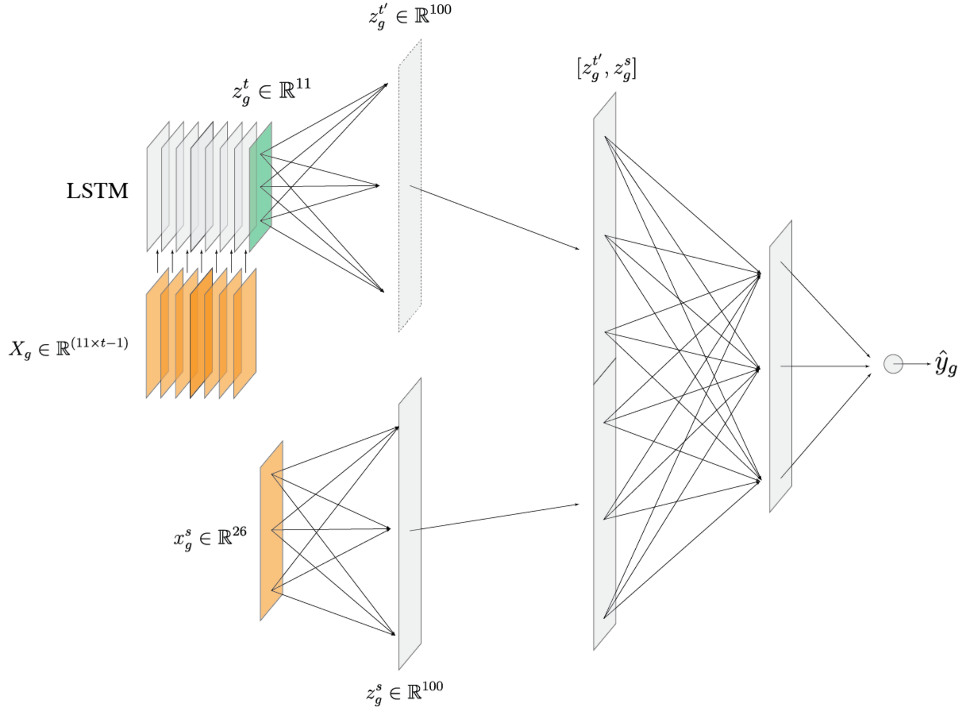

Figure 2: Diagram of recurrent architecture

to create the vector $z_g^{t\prime} \in R^{100}$. Concurrently, the static vector $x_g^s \in \mathbb{R}^{26}$ is fed through a single fully-connected layer with 100 neurons and a ReLU activation to produce an intermediate representation $z_g^s \in \mathbb{R}^{100}$. The two vectors z_g^t (or $z_g^{t\prime}$) and z^s are then fed into classifier, which consists of a fully-connected layer with 128 neurons and ReLU activation followed by another fully-connected layer with a single output neuron to produce a binary label $\hat{y}_g \in \{0, 1\}$.

The recurrent architecture allows the network to utilize the order and information about individual games in order to make a prediction; the fully-connected architecture allows it to include static features as well. Thus, the inputs X_{g1} and X_{g2} to the LSTM for successive games g_1 and g_2 against the same team are no longer as similar, allowing the network to be more robust to game-specific variations.

5.2 Results

With the recurrent architecture, I was able to achieve a maximum validation accuracy of 0.586 (Appendix 1) and finally a test accuracy of 0.585. I trained the model for 50 epochs and with a batch size of 20. I tested a number of hyperparameters and architecture choices for this model as well—Figure 3 shows six different models where the parameters that had the largest impact on results were varied. The particular decisions noted are:

- Inserting a dropout layer between the final fully connected layers: this was not an effective strategy, and led to lower training and validation accuracies on the models in which I used it (models 0 and 1). In general, it led to a smaller gap in performance between the training and test sets, but at the expense of the overall performance. This suggests that the dropout layer was too strong a regularizer for the model, leading to high bias and an inability to classify even the training set well.
- Including a fully connected layer between the output of the LSTM and the concatenation layer: this was generally an effective strategy, which improved both training and validation accuracy when the other hyperparameters were held constant. For example, we can compare model 2 (with the fully-connected layer) with model 4 (which has the same hyperparameters as model 2, but without the fully connected layer)—model 2 has higher training and

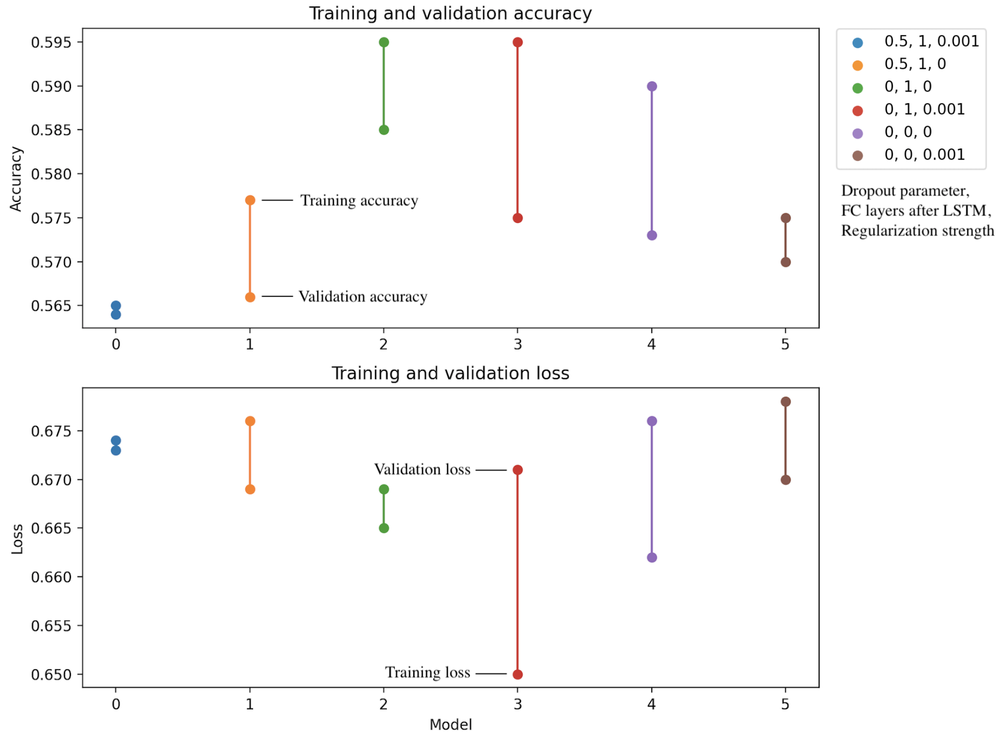

Figure 3: Recurrent model - comparison of different architecture choices and hyperparameters

validation accuracies than model 4. Similarly, we can compare models 3 and 5; as with models 2 and 4, model 3 (which has the fully-connected layer) has higher training and validation accuracies. This suggests that using the fully-connected layer increases the capacity of the network in a way that enables it to make better predictions.

- Including a regularization penalty on the 100-node fully-connected classification layer: this had the effect of decreasing the gap between the training and validation accuracy. However, this was achieved mostly at the expense of training accuracy, and not through an improvement in validation accuracy. Comparing models 0 (without regularization) and 1 (the same hyperparameters as model 0, with regularization), we can see that model 0 has very little gap between test and training accuracy, but much lower training accuracy than model 1 (and slightly lower validation accuracy as well). Similarly, model 5 (with regularization) has a smaller training-validation gap, but lower accuracy and higher loss than model 6 (without regularization). This suggests that like the dropout layer, the regularization penalty is reducing overfitting at the cost of increasing underfitting, leading to worse overall results.

6 Conclusion and Future Work

The combination of an LSTM and fully-connected classifier was able to achieve XX% test accuracy on the classification task. This accuracy exceeds that achieved by Elfrink et al?? (cite) and Soto-Valero (cite) in their work on this task. However, there may be additional room for improving this accuracy. Instead of using the previous results of one team when making the prediction, one could employ two separate recurrent networks on the previous games of both teams involved. To generate the two independent recurrent sequences that ended on the same day would require a dataset with game identification numbers or some other way of matching the data; this may be possible to generate, but is not yet within the scope of this project. I hope to be able to explore this in the future, because I believe it could yield non-trivial improvement over the methodology detailed above.

A Appendix

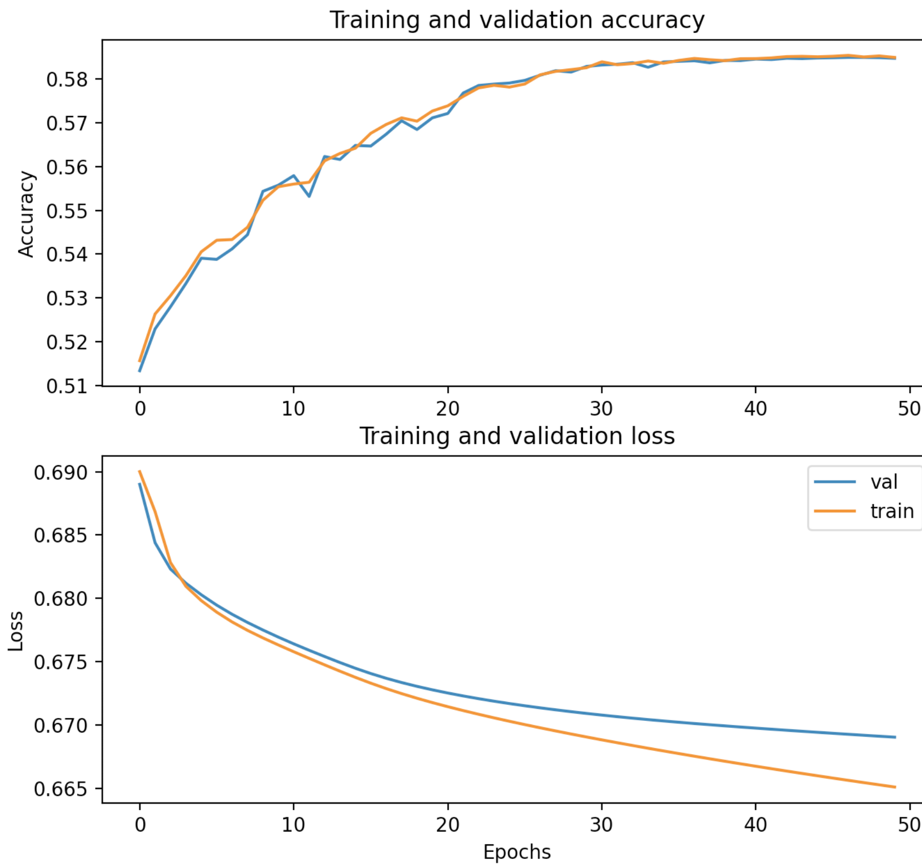

Figure 4: Loss and accuracy for best model

References

[1] Baseball statistics and analysis, 2020.

[2] Mlb stats, scores, history, and records, 2020.

[3] D. Calzada. Deepball data - your home for predictive baseball statistics, 2019.

[4] D. Calzada. Deepball: Modeling expectation and uncertainty with recurrent neural networks. *SABR Analytics Conference*, 2019.

[5] N. Danisik, P. Lacko, and M. Farkas. Football match prediction using players attributes. *2018 World Symposium on Digital Intelligence for Systems and Machines (DISA)*, pp. 201–206, 2018.

[6] T. Elfrink and S. Bhulai. Predicting the outcomes of mlb games with a machine learning approach. 2018.

[7] Y. Pi. Predicting next baseball pitch type with rnn. *CS230: Deep Learning, Spring 2018*, 2018.

[8] C. Soto-Valero. Predicting win-loss outcomes in mlb regular season games – a comparative study using data mining methods. *International Journal of Computer Science in Sport*, 15:91–112, 12 2016. doi: 10.1515/ijcss-2016-0007

[9] M. Spilchen. baseball-scraper, 2020.