

NBA Machine Learning Game Predictor

By: Christian Miller, Jack Lambert, Joe Gullo, and Gabe Elbling

Our project involves using Python to implement a machine learning algorithm to predict the outcome of NBA games. Because basketball is one of the most unpredictable sports in the world, we use automated web-scraping to get data in a usable, csv format and then feed it to our machine learning algorithm. By utilizing the Pytorch library, we were able to create a constantly updating model which has the ability to accurately predict the win-loss outcome of NBA games with upwards of 75% accuracy. Given a date, the model predicts Blowout Metrics¹ for each game and stores which will be the closest in a modified minimum heap structure to support class structuring and storage abilities. Lastly, as more data becomes available, our model continues to become more and more accurate.

Github Repository:

<https://github.com/cmille47/NBA-ML-Outcome-Predictor>

Script Descriptions:

Before the model and data structures behind it can fully be explored, it is important to first understand what each script does and how they build on each other to implement the model

DATA-SCRAPING:

In order for any machine learning algorithm to work, we must first have data which is cleaned and processed into a non-biased format. Anything that holds an implicit bias such as rankings is not recommended as it will decrease the objectivity of the algorithm. To acquire the data we first attempted to get data directly from the NBA website, however, this became increasingly difficult as our access was blocked after too many requests. We then turned to basketball-reference.com – a website dedicated to working with NBA data that allowed for many requests.

The way we pulled data was taking individual date's box scores. The user inputs the start date and end date of a season, then that range's data is scraped from basketball-reference.com, and manipulated in a way that each box score contains a team's average stats from the last 20 games and their opponent of that day's average stats from the last 20 games. The final output is two CSVs: one with full data, and another with reduced data of differences between team and

¹ Blowout Metric: Value between 0-100 created from output of model indicating how close a game will be. A value close to 100 indicates a "blowout" contest where one team will have a dominating win. A value closer to 0 implies a competitive game where either team may win

opponent averages. The advantage of the former is that the machine learning algorithm will output slightly more accurate results. The advantage of the latter is that the machine learning algorithm can run quicker with fewer columns.

Below are the important variables and functions from the program.

NAME	TYPE	DESCRIPTION
box_scores	dataframe	Initial dataframe made by scraping raw box score data from each date in season range containing important stats from game
teams_data	dictionary	Dictionary with keys being the 30 NBA teams and values being a dataframe of each individual team's box scores from the chosen year
final_data	dataframe	Final dataframe of all eligible games (both teams playing have at least 20 games played in that season of data to predict outcome of that game) that contains important game outcome stats, as well as both team's average stats in their last 20 games
reduced_data	dataframe	Dataframe made from final_data that cuts down the number of columns by taking a difference between team average and opponent average to speed up machine learning run time
scrape_data()	function	Takes input of start date and end date and scrapes all box scores within the range, saving each as a row in box_scores
revise_box_scores()	function	Adds a game_ID value to each box score, while aggregating a variety of stats about how teams have played against each team in the last 20 games
set_team_values()	function	Uses unique team values in box_scores to make a dictionary of key team name and value dataframe of team's box_scores
find_averages()	function	Iterates through all each team's individual box scores and finds the team's average stats in the 20 games leading up to this game (to use as predictors for outcomes) and stores data in final_data dataframe
find_opp_averages()	function	Adjusts final_data by adding the last 20 game average stats of the opponent into each row
clean_data()	function	Deletes repeat games (i.e. if the Celtics play the Nets that

		game will be in the final_data dataset twice: once with Celtics as team and Nets as opponent, and once with Nets as team and Celtics as opponent. The data in this row is repetitive, though, so only one of these rows is kept), and unnecessary columns from final_data
reduce_data()	function	Creates dataframe reduced_data from the final_data dataframe by taking the differences of team averages and opponent averages

MACHINE-LEARNING MODELING

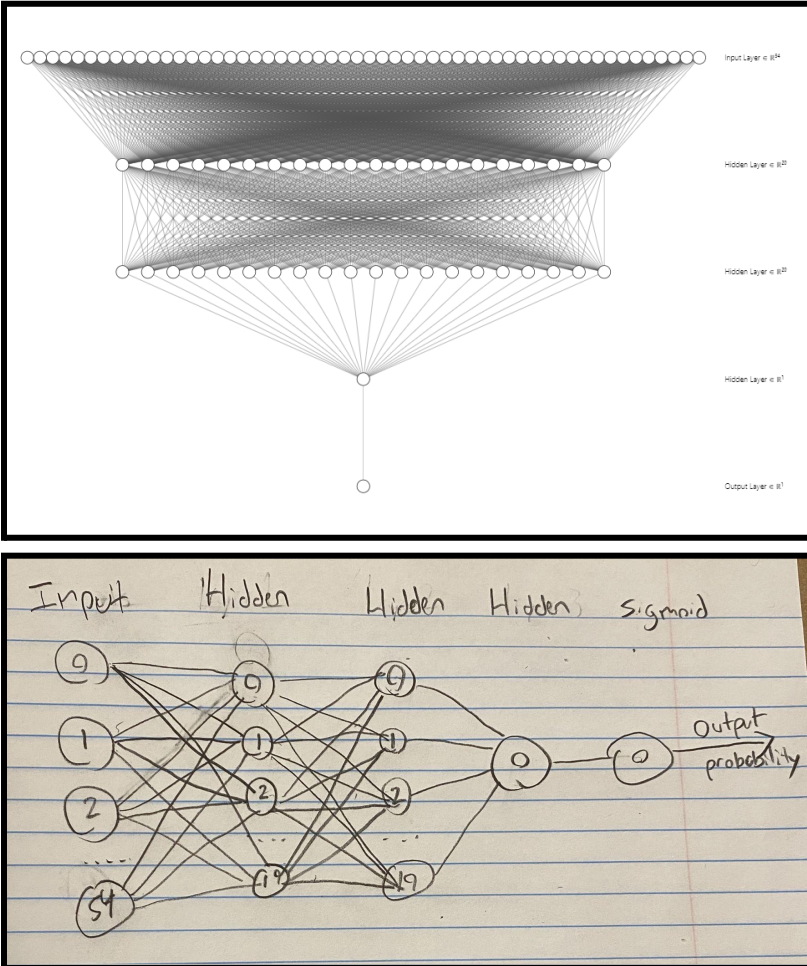
After the data has been acquired and updated into a csv file, it is ready to be assessed by the model. In Pytorch, there are numerous models to analyze and predict outcomes. There is no set method for picking a model. It is mainly trial and error, but there are some general guidelines that we used that are explained below. Models can be created and modified in order to be more efficient in terms of training time error/loss.

The issue we aimed to tackle with our machine learning algorithm is generally referred to as a binary classification problem, which is a problem that requires a probability that one of two things occur. In our program, we had to predict whether or not a team won or lost based on their stats the last 20 games, and their opponents stats the last 20 games.

After a lot of trial and error, we ended up creating and modifying our own model. Some of the key modifications we made were in the topology of the neural network, learning rate and optimizer type (all of which are explained in further detail below in the “Bringing it all Together” section). Another key implementation that we made was making our output layer a sigmoid function. This method squishes any real number into a range between 0 and 1.² We used it, more specifically, to

NAME	TYPE	DESCRIPTION
load_data()	function	The input of this function is the path to the CSV file that we are pulling our data from. This function uses pandas to load in the data from the CSV. It then calls the normalize_data() function (referenced below) to convert all inputs into a value between 0 and 1. It then randomizes the order of the data to prevent bias. Then, it splits the data into our desired inputs and outputs and stores them in numpy arrays. It returns both of the numpy arrays.

² <https://sparrow.dev/pytorch-sigmoid/>

normalize_data()	function	<p>This function iterates through the columns of the inputs, finds the max value in each column, and divides each column by 10, 100 or 1000 depending on the size of the max value. For example if the max value in a given column is 150, it will divide all the data points in that column by 1000. Similarly, if the max value is 9, it divides all the data points in that column by 10.</p>
accuracy()	function	<p>This function takes in the test_input and test_output datasets, and uses our trained model to predict the blowout factor for that game. It then rounds all the prediction outputs to 0 or 1 and checks whether or not our guess is correct. It returns the percentage of the correct data we predicted.</p>
Net()	class	<p>This is our neural net model structure. When creating a model of this class, the inputs are input_size (input layer size) and hidden_size (hidden layer size). Below are two illustrations of our neural network:</p> 

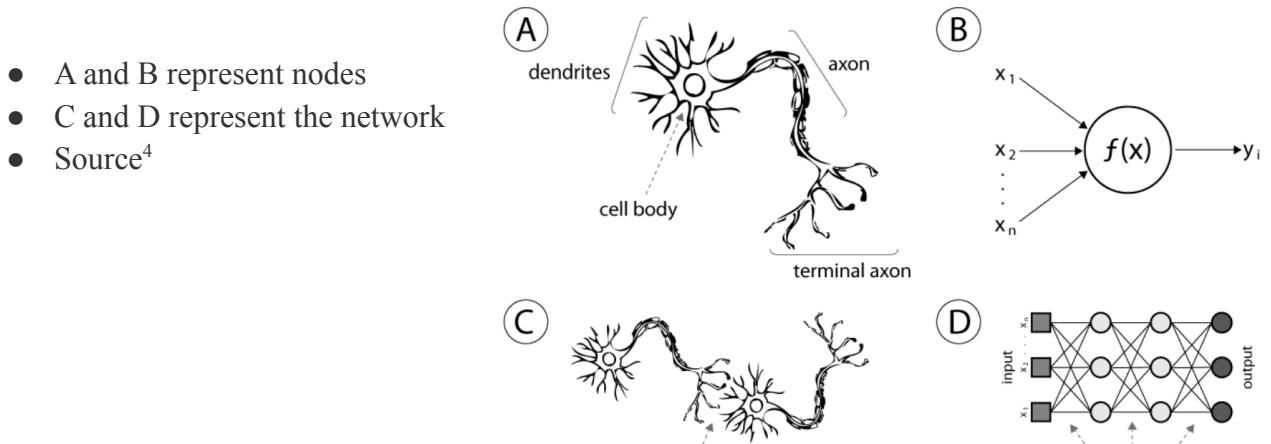
MODIFIED MINIMUM HEAP

The last major component of our machine learning model is the storage of our results in a minimum heap structure. The minimum heap structure stores classes with integer values serving as the organizing factor. As such, larger sets of data can be organized and stored using the data structure with quick and efficient access. Unique to our approach, the following are the structures, functions, and significant variables found within this script:

NAME	TYPE	DESCRIPTION
basic_info	class	Holds three descriptive categories for each game: the home team name, away team name, and predicted Blowout Metric which is obtained from the model
min_Heap	class	Defines minimum heap functions and creates minimum heap from each analyzed game. The following variables/functions are from this class
heap	array	Stores basic_info structures for each game in minimum heap format of parent with children. Left child index = $\text{parent_index} * 2 + 1$. Right child index = $\text{parent_index} * 2 + 2$
left()	function	Returns left child position from a given parent node index
right()	function	Returns right child position from a given parent node index
min_heapify()	function	Recursive based function that takes in position value, setting parent as node at position. Accesses parent and children Blowout Metric values, swapping if children value is smaller than the parent. Passes in new parent position in recursive call, swapping until no more can occur
build_min_heap()	function	Iterates through heap, calling min_heapify with each position valid position to organize minimum heap
insert()	function	Takes in a basic_info object and appends it to heap. Calls build_min_heap to organize heap.
pop_top()	function	Removes current minimum element and, similar to insert(), calls build_min_heap to reorganize minimum heap

Bringing it All Together:

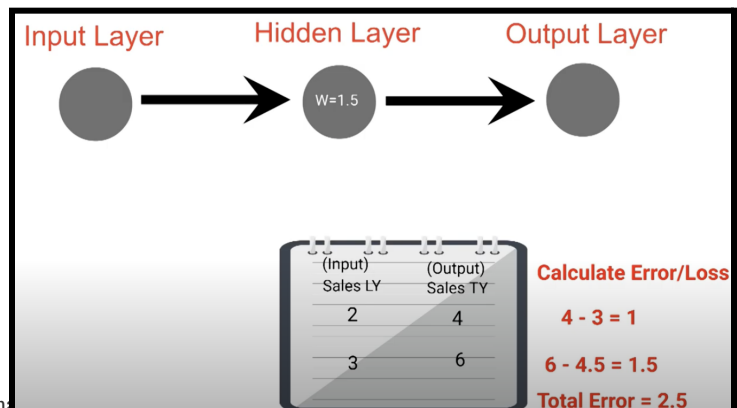
The human brain is made up of neurons (nerve cells) arranged in a network. This network processes and transmits information which comes from our senses (smell, touch, etc.). Machine learning algorithms are designed to work as the brain does. This is done by forming artificial neural networks. Nerve cells and their interconnections are extremely similar to the neural network design as seen below³:



The network is composed of layers (input, hidden and output layers respectively) which are made up of nodes (“neurons”). The input nodes take in n number of inputs and are fed into hidden layer nodes which attempt to find commonalities in the data through weights. It then compares its weighted output estimate with the data in the output layer nodes. During the training process, the neural network starts with a random weight and then adjusts its weights with each epoch (one iteration forward and backward through the data) in order to better match the desired training data output. The network corrects itself through calculating its error/loss by comparing its weighted estimate with the output in each epoch.

Below is a simple single input and output example of this process:

- FIRST EPOCH
- initial random weight is 1.5
- output - (weight * input) = error
- total error = 2.5
- Source⁵

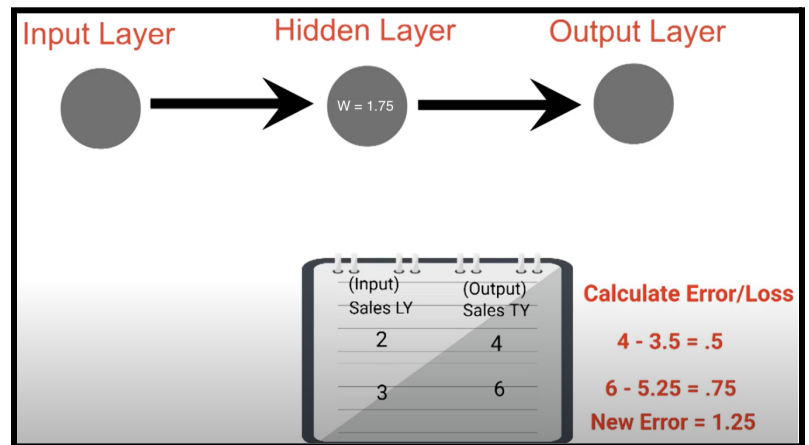


³ <https://medium.com/predict/artificial-neural-networks-m>

⁴ <https://wp.nyu.edu/yungjurick/2020/03/15/debate-on-the-relationship-between-neural-network-and-the-brain/>

⁵ https://www.youtube.com/watch?v=oFTpK_v9llw

- SECOND EPOCH
- Adjusted weight is 1.75
- $\text{output} - (\text{weight} * \text{input}) = \text{error}$
- total error = 1.75

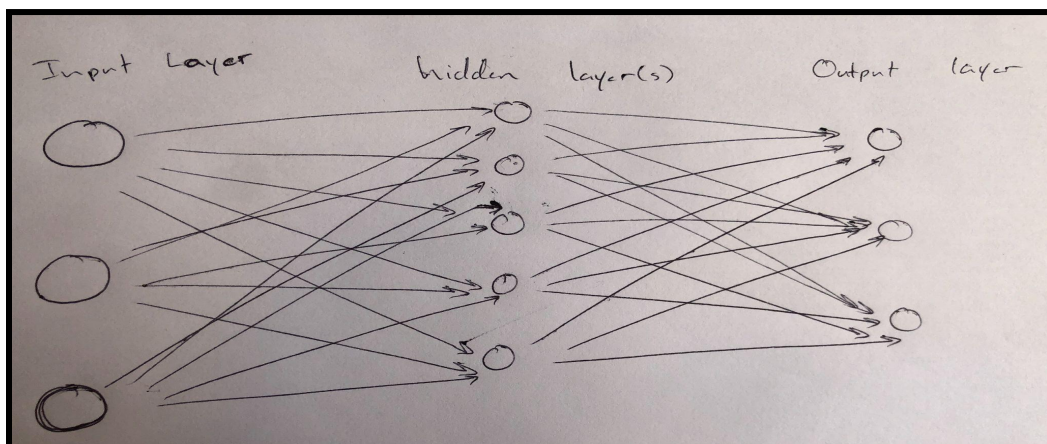


As seen above, with every training epoch, the algorithm gets better at predicting the outcome. The efficiency of a model is directly correlated to the time the and the accuracy (loss) it yields. Models can be modified and created in different ways to tackle specific problems more efficiently. There are many ways of doing this, the ones we used to optimize our neural network are:

- Learning Rate:
 - How much to change a model according to the calculated error each time the wrights are updated. For our pytorch model, we ended up settling on a fairly high learning rate of .9. A high learning rate allows the training algorithm to learn quickly at the risk of learning incorrect patterns.
- Topology:
 - Amount and organization of nodes in the hidden layer.
- Optimizer Type:
 - Algorithm that “optimizes”/changes characteristics of the neural network network such as weights and learning rate. This is done in order to improve the loss.

Once an efficient model is selected and trained, this model can be saved and used endlessly. Its model receives inputs and predicts a previously unknown output.

Below is an example of a full and complex neural network:



Now with all the fundamentals established and each piece of code understood, it is time to bring it all together in our main driver script. The first thing we do is get a valid date on which NBA games are played. Next, following the principles of machine learning, we get what is the most important: the data. For each game, we utilize our data-scraping script to extract each team's relevant statistics from the last 20 games played. We do 20 games because the NBA is constantly evolving and a team which may have been struggling earlier can completely turn around and become a dominant force (i.e. Celtics 2021-2022 season⁶). Before these are inputted into the machine-learning predictor algorithm, each team's name and away/home status is placed into the `basic_info` class for later use. After, the data is inputted into the predictor which returns the theorized Blowout Metric value for the game. Here, our algorithm differs from many others already created as it does not return a win/loss value for each team but rather how close the game will be, giving unique insight in which games will be competitive and fun to watch.

With the Blowout Metric obtained for a game, the next step is to input it into the `basic_info` class and then insert that class into the modified minimum heap which will constantly update by looking at the Blowout Metric to place the best/closest game on the top. Last, the program will output the proposed best game and offer the user an option to get the Nth best game or quit.

Analyzing the output of Blowout Metric, a few unique insights are garnered compared to similar machine learning algorithms. First, instead of a mere win/loss, the output gives a relative scale of how certain a game result is. For example, while another algorithm will say Team A will win, our algorithm gives an idea of certainty by displaying if it will be a blowout win (Above 75 Blowout Metric) or a 50/50 win (near 0 Blowout Metric). Second, the Blowout Metric metric gives a scale of how competitive a game will be. Thus, the user can better choose which games to watch or attend. Third, the metric gives an objective view of how good/bad each team is. Instead of merely following records or ranking to see matchup information, the model gives an unbiased and focused view of how well each team is truly performing up to the game. This model can be used to evaluate the current performance of a team. Overall, the model offers more in-depth insights than a typical identification or win/loss model.

INSIGHTS FROM DATA STRUCTURES

The goal of our project was to develop a machine learning algorithm to predict NBA games and output the results in an insightful format. Through the use of hash-tables, classes, a modified minimum heap, and Tensors, we succeeded in our goal of accurately predicting the results of NBA games. In creating this unique combination of data structures and specific modifications We hope to implement the algorithm on the remaining playoff games to see the implementation in a real-time scenario.

To begin, the hash-tables were instrumental in our web scraping algorithm. The primary hash table being used was `team_data`. This python dictionary utilized team names as keys and a

⁶<https://www.si.com/nba/2022/02/24/boston-celtics-jayson-tatum-jaylen-brown-turned-season-around>

dataframe of box scores as values. The $O(1)$ lookup time significantly reduced runtime in the web scraping algorithm, because for any given season there were over 2000 box scores to iterate through and organize. Being able to access each box score quickly was crucial in creating the data testing sets to train the model with.

Next, the creation of classes was necessary in organizing our retrieved data and building the modified minimum heap. The classes developed in the modified minimum heap script allow for the easy storage, retrieval, and organization of our output data to allow the user to get the most use out of our model.

Moving on, the modified minimum heap is one of the focal points of our project in presenting the data to the user. In order to maximize efficiency and reduce looping through arrays and comparing values, we redesigned a minimum heap to take in a class instead of an integer value. The class contained game identification information such as team names and the predicted Blowout Metric value. Through accessing the Blowout Metric, the minimum heap would then organize each class sending the one with the lowest value to the top as the best-predicted game. Thus, the user receives the top game along with basic information about that game in an efficient and constantly updating format without further looping and sorting. Looking at the modified minimum heap from a bird's eye view, the most interesting part is that this process can be emulated in other similar data structures, meaning similar storage and organization of larger heaps of data through integer "pointer" values is possible and extremely useful.

Putting it all together, the multi-faceted approach toward the data structures enabled us to maximize their benefits and reduce any side-effects. Their overall integration in the program enabled for efficient and manageable data scraping, model predicting, and outputting of results to the user to create overall a user-friendly model. Additionally, the separate data-structure usage promotes more modularity, allowing for ease of future edits and experimentation of other structures/approaches.

MOVING FORWARD

We are very proud of the product we have produced, but there is still plenty of room for this project to be further developed. This was all of our first experience using machine learning. In this project we did significant research into the topic, the PyTorch package, and the Pytorch package, and learned a lot. We were able to test multiple models and compare our results between them, but moving forward with more knowledge and experience we could implement even more to find the most accurate model.

We also used data from the past 6 NBA seasons to train and test our algorithm (roughly 5000 data points) – more data to train the model may be useful in the future.

In terms of the functionality of user experience in connection with the algorithm, a future iteration of the project could consist of a very user-friendly execution of our code. For example, automating box scores to be scraped and added to the master dataset the day after they occur could be very useful. Additionally, an automation that determines the current date and returns our

algorithms prediction for the closest games of the day could be very useful in achieving one of our initial goals: help fans decide which games to watch.

ACKNOWLEDGEMENTS

First, we wanted to thank basketball-reference.com for access to their data. It goes without saying that this project couldn't be completed without data. The free access to data they offer made our project possible.

This project would also not have been possible without the help of many different people. We want to thank Professor Morrison for how helpful he has been all semester in teaching us about so many different data structures. This project definitely would not have been possible without the knowledge we have acquired through his Data Structures course. Specifically, our utilization of hash tables and a minimum heap in this project stem directly from his teachings. Additionally, Professor Morrison was incredibly helpful allowing us to utilize the Center for Research Computing machines to implement our project.

We also wanted to thank our Project Manager, Deeksha Arun. Deeksha has supported us in her willingness to meet with us, provide code feedback, and share her machine learning expertise with us. This project would not have run as smoothly as it did without her guidance.