Deep Q-Network Applied to Basketball Player's Action Value Analysis

Kyle Naddeo

***Abstract*—In this study an Off-Policy, Model Free Deep Q-Network is proposed to learn the optimal basketball strategy. The model will learn from NBA tracking and a event data, where the former describes the location of every player and ball for 25 frames per second, the latter describes the events that unfold in the game. The basic frame work for the data formatting and the DQN agent are described. Both codes are currently written yet need to be optimized and more precisely validated.**

***Index Terms*—DQN, Data Mining, Tracking Data**

I. Introduction

THIS study will investigate the value of players actions based off the quality determined by a deep Q-network (DQN). The data comes from the NBA during the 2015-2016 season where the player and ball locations were recorded at 25 frames per second.

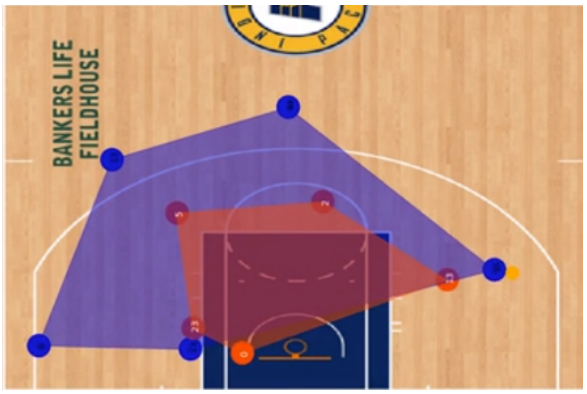

Fig. 1. An illustration of the data set used in this study.

The data set was found on Neil Sewards GitHub [1], who had code written for preliminary pre-processing such as: converting file types, converting full court to half court and a script to fix shot times. After still needed significant cleaning and reformatting in order to feed it to the DQN.

The value of a player is a very complex concept and cannot be analyzed with a single stat. For example, rating a player on total points scored will neglect the concept of a greedy player, who rather than passing to a teammate would rather take a low percentage shot. Although, through the course of the game this player will collect more points for the team, he/she may have reduced the possible total points of the team. The reason a player may act like this is either an inherit selfish trait or due to events taking place off the court. When a player is at the end of their contract they, will look to increase their stats at the expensive of the team to make themselves appear more valuable in their upcoming contract negotiations.

To further complicate the issue of determining the value of a player , in some different circumstances the same action can either be positive or negative. For example, when the shot clock is at twenty-four seconds the player should pass rather than taking a low percentage shot; however, when their are only a few seconds left the player should take the shot. The only way to truly determine the value of a players action is to compare it to the actions of a "perfect player". This is the motivation for developing the optimal agent through the DQN, once the agent is trained the actions of the players can be compared, similar to the concept used in a paper on NHL tracking data by Simon Fraser University [2].

A. Reinforcement Learning

Reinforcement Learning is a type of Machine Learning which is rooted in Classical Conditioning and Optimal Control. It excels in game play applications due to its ability to dynamically change its reactions based off observations from the environment. Classical conditioning is exactly how animals learn in nature and was formalized by Pavlov in an experiment with his dog. Pavlov would ring a bell then give his dog a treat; after many iterations Pavlov could ring the bell and the dog would salivate in expectation of a treat regardless if one was present or not. The bell is a neutral stimulus that the dog experiences while the treat is rewarded stimulus; the reaction to the treat is salivation of the dog. Through training, PavlovâĂŹs dog learned to react to the bell in the same way that it would react to the food. This concept is useful for training an reinforcement learning agent to react to a situation in a desired way by giving the agent rewards in training. After training the agent will react in the same way even without rewards. This learned reaction set is considered the optimal control given the current situation.

Reinforcement learning is a type of unsupervised learning because the training data does not include labeled values of the output. The agent understands its environment as being in a state which is a unique combination of its sensory inputs. In Q-learning the agent will refer to the Q-table which states the quality of an action based off the state; therefore, the dimensions of the Q-table are number of unique states by number of possible actions. The agent will take the action with the highest quality; based off this action the environment will change thus putting the agent in a new state and this new state will come with a penalty or a reward.

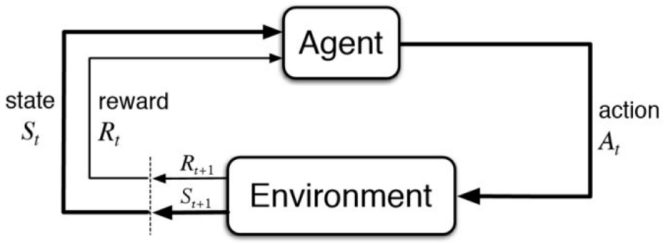

Fig. 2. An illustration of the training cycle of a reinforcement learning agent interacting with its environment. The agent will take an action based of its state, this action will effect the environment and produce a new state along with a reward (can be positive or negative).

The agents episode will end when it reaches a final state known as the terminal state. The route taken in a given episode is considered the agents deterministic policy from the initial state ($\pi(s)$). The cumulative reward of a policy must be maximized in order for that policy to be optimal, this is known as the reward hypothesis. The way to update the Q-function of a given policy is through Bellman's Equation:

$$Q_{\pi,new}(s,a) = Q_{\pi,old} - \alpha(R(s') + \gamma \max_a Q'_\pi(s',a') - Q_{\pi,old}(s,a)) \tag{1}$$

Where $Q_{\pi,new}(s,a)$ is the updated value of an action (a) in a state (s). The rate at which the the value will update is limited by the coefficient α, a value between 0 and 1. The reward comes from the next state and therefore is represented as the reward function value of s' rather than s as seen in R(s'). The reward will be added to the quality of the best action in the next state which is also limited by a coefficient called the discount rate (γ). These two terms combine to create the target for the update. It is important to note that the Q-function used to determine the quality of the next states best action is not updated during batches but rather at the end of each epoch. This is done to reduce variance.

Building a rewards function is identical to a business attempting to develop an incentives plan. The goal is to encourage certain behaviors and discourage others however the description is much simpler than the execution. The Cobra Effect is an anecdote which best describes the complexity of a good incentives plan. The anecdote is set in India under British rule; the government was concerned about the number of deadly cobras in Delhi and decided to offer a reward for every dead cobra brought into authorities. Unfortunately, the locals were clever and took this as an opportunity; they began breeding cobras to bring in. Once the government found out they stopped the program so the locals released all their useless cobras and thus in the end there were more cobras than ever. A similar effect could happen for the balancing agent if a small reward is given for passing the target and a higher reward is given to stop on the target the first instinct would be to stop on the target for the highest reward. However, a smart agent will learn to oscillate over the target for infinite reward instead of settling for one big reward. In some cases the rewards from the environment do not come immediately at each state rather they come at the end of the episode. This means that there will be no rewards to update the Q-table and therefore no learning will occur. To rectify this the rewards will be estimated by looking at the frequency and proximity of the states to the terminal state through an eligibility trace.

$$e_t(s) = \begin{cases} \lambda * e_t(s) + 1 & \text{if s} = s_t \\ \lambda * e_t(s) & \text{if } s \neq s_t \end{cases} \tag{2}$$

This states that the eligibility of a state (e(s)) will increase by 1 every time the agent occupies it in a time step. The eligibility of that state will also decrease at a rate of λ every time step.

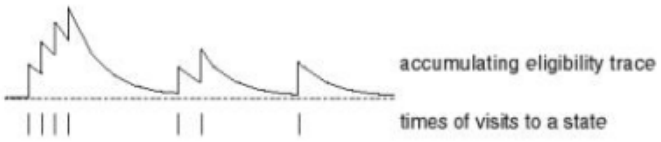

Fig. 3. An illustration of the eligibility of a single state thorough the course of an agents episode. Every time the state is visited the eligibility will increase then decrease at a given rate at every time step where the state is not visited.

Now the final reward of the episode can be multiplied by the eligibility of each state to assign a reward to every experience in the episode. Now the agent has a method to update its policy at the end of the episode.

The standard training method for most reinforcement learning algorithms is to build a virtual model of the environment and have the agent explore and train within this model. The advantages of this are the training iterations occur must faster than they would in real life allowing for the agent to train many more times in a given amount of time than a real world agent. The disadvantages of this are that for complex environments the modeling may be too difficult or too simplified to give an accurate representation. Sometimes an over simplified model of the environment may be acceptable enough to allow the agent to form a basic understanding of the environment to expedite its real world training. This type of training is called on-policy training because the agent is learning while following its current policy.

An alternative method to on-policy learning is off-policy learning. This is the case where the agent "looks over the shoulder" of another agent and learns from its successes and mistakes. In this type of learning no model is needed and the agent has no choice over the actions taken. The agent simply puts itself in the shoes of the other agent in an attempt to optimize its own policy.

B. Deep Q-Network

In the last section the basic frame work for a Q-Learning algorithm was described as an agent being in one of a finite number of states from which it could use a look up table to see which action had the best quality. The most glaring issue with such a strategy is concept of a state. By definition it is the unique combination of the sensory inputs. The majority of sensors in the real world output a signal on a continuous scale thus leaving an infinite amount of possibilities for one sensor; now multiply this with all the sensors and were left with an infinite amount of state. The knee jerk reaction to this is to place all the sensor readings into a finite number of bins. This leads to subjectivity into how granular can the

sensor be while still accurately representing its senses. This method does; however, work and given the proper binning the agent can learn to navigate high dimensional states.

A less recognized issue in Q-learning is the massive size of the Q-table which must be stored in the agents hardware leading to slower ability to even find the proper state to know the optimal action. If the agent is operating in the real world the goal would be to have a efficient, low cost and fast robotic. With a large look up table comes additional costly storage, extra material weight and decreased reaction speed due to state searching.

Enter the DQN which rather than using a look up table to find the quality of actions, it calculates the quality of actions. The calculations are performed by a neural network with many layers hence the Deep and Network in DQN. The inputs are the sensory points with or without the action; that is, the input can be the state space vector and output be the quality of each action or the input can be the state space vector and the action and the output be the quality of that one action.

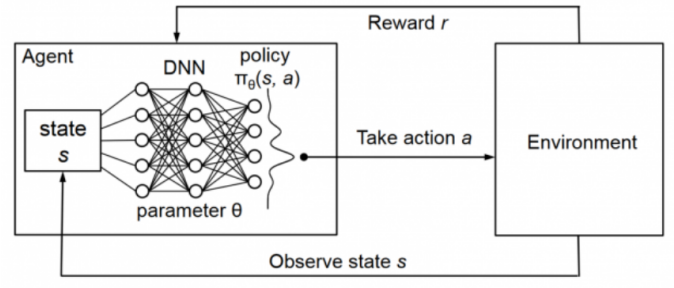

Fig. 4. An illustration of a Deep Q-Network which uses a Neural Network as a quality appreciator for a give state. The state is no longer a unique combination of sensory inputs but rather the vector containing all the inputs in their continuous form.

An extremely important nuance with the DQN is that states that have never been visited before now can have their action qualities approximated. The back propagation will now be performed with a loss function similar to the target in Bellman's Equation.

$$loss = (R(s') + \max_a \hat{Q}'(s', a', w) - \hat{Q}(s, a, w))^2 \qquad (3)$$

Notice that now the Q functions have hats on top of them to indicate that they are approximations or predictions. As stated before the quality used to determine the best action in the next state has weights that are only updated at the end of the epoch rather than at each batch to reduce variance.

II. Methods

A. Data Formatting

A great deal of effort was allocated to data formatting. The original data set was formatted in a way that every frame contained eleven rows, one for each player and one for the ball.

	team_id	player_id	x_loc	y_loc	radius	game_clock	shot_clock	quarter	game_id	event_id
1	-1	-1	11.4806	419.9969	6.76567	720	24	1	21500001	1
2	1610612737	2594	94.2130	409.5423	0.00000	720	24	1	21500001	1
3	1610612737	200794	-20.4121	323.7984	0.00000	720	24	1	21500001	1
4	1610612737	201143	-5.2384	419.8086	0.00000	720	24	1	21500001	1
5	1610612737	201952	12.6600	235.1728	0.00000	720	24	1	21500001	1
6	1610612737	203145	-67.4141	358.4368	0.00000	720	24	1	21500001	1
7	1610612765	101141	5.1460	329.7306	0.00000	720	24	1	21500001	1
8	1610612765	202704	4.8774	156.0893	0.00000	720	24	1	21500001	1
9	1610612765	202694	102.6915	406.0077	0.00000	720	24	1	21500001	1
10	1610612765	203484	-104.4564	407.8648	0.00000	720	24	1	21500001	1
11	1610612765	203083	-6.3552	417.3551	0.00000	720	24	1	21500001	1

Fig. 5. The form of the original data set after Neil Seward's pre-processing.

As seen in Figure 5 the useful information on the player ID, game time, shot clock time, player location and event ID are given. From here the data needed to be first cleaned of all incomplete instances (instances without 11 rows) and instances where the game or shot clock are not running (time outs or moments before jump ball). Next a play number was assigned to each sequence of instances with descending shot clocks, from there all plays under half a second were discarded. Now the final event for each play was found through a separate data set which also contained the event ID's of the game. Once this was accomplished all the plays which did not end in a shot or turnover were removed because these were the only events that were assigned rewards.

At this point all the instances are assigned play numbers and the end of the play results in a shot (either made or missed) or a turnover. The next step is to determine which team is in position of the ball. This was accomplished by finding which player was closest to the ball at each instance and and tallying up which team they were on. At the end of the play the team that was closest to the ball the longest was considered to be in possession.

Now the problem is to decide which player on that team is in possession of the ball [3]. A major issue with this is passing and shooting where there is no one in possession of the ball. Since the ball is known to be traveling at a much faster rate during a pass or shot the velocity of the ball was calculated to assist with determining possession. Now the player closest to the ball is temporarily assigned possession, then each change in possession was tracked forwards and backwards in time until it reduced to a threshold velocity. The time in this span of high velocity was marked as not possessed by anyone and the moment before the pass was labeled as an action of passing. The number of the action was determined by the players proximity to the ball before the pass i.e. if the ball was passed to the third closest player than the action of the player was 3. After this all the instances where the ball was possessed by no one was removed. This is because in the DQN the agent will always be the player with possession.

Now that the data set is slimmed to the least amount of players, each offensive player has their attributes added based off their shot percentages. This was done manually but in future work a code to generate the players attributes will be added. At this point the data is ready to be format ed into its final form of one row per instances. Each row will contain a columns dedicated to the game properties (game clock, shot clock, ball location, action taken and reward, although currently just a zero place holder), offensive and defensive properties. The final step is to make a temporary version of the data set and bin each row then run an eligibility trace for each play to assign a reward for each instance. Now the data

set is complete and can be normalized.

Game Properties						Offensive Properties						Defensive Properties
Game Clock	Shot Clock	Ball Location	Action	Reward	Team with Possession	Player 1				...	Player 5	...
						Distance to Ball	Location	Velocity	Attributes	...	...	...

Fig. 6. The form of the final data set after this studies pre-processing.

In this form the rows (without the reward and team with possession) can be given to the DQN as state space vectors.

B. DQN

In this application the there will be no model and the agent will never make its own decisions; therefore, the agent will learn in a model free off-policy fashion. In essence the agent will watch the basketball game and "act" as if it is the player with possession and then learn from that players failures and successes.

Algorithm 1 Model-Free, Off-Policy DQN Algorithm

Initialize action-value function Q with random weights Θ
while experiences not used $\neq 0$ **do**
  for episode = 1, N **do**
    for plays = 1, P **do**
      Choose a random expedience D(state,action, reward, next state) from play and save to memory
    end for
    Perform Experience Replay to Update Q function
  end for
end while Validate Model

Once the agent is fully trained the model can be validated by summing the values of every action of each team and ensuring that the team with the most valuable cumulative actions is the team that won the game.

III. Future Work

The current data formatting algorithm takes roughly two hours to run for one game, there are roughly 1300 games per season making the current algorithm insufficient. After performing the data formatting the first time some key insights were found such as a few redundancies which if they are rectified should decrease the number of computations needed. The current data formatting algorithm does work and is considered a first draft, now it must be optimized before final use.

The current DQN algorithm currently is not fully written due to time constraints of the semester. The original attempt was focused on building a custom environment in OpenAI's gym library; however, do to the nature of a model free, off policy agent it was deemed unnecessary to create an environment which is essentially just a model. In addition, an agent using such an environment only uses this to explore the space but for an off-policy learning agent there is no exploration. Rather the agent just follows the experiences of other players.

The DQN algorithm must also save the weights of the target network (Q'(s',a') constant through each episode while the base network will update with every experience replay.

IV. Conclusion

At this point in the study the basic frame work for both the data formatting and DQN agent have been written. The data formatting needs to be optimized as it has some redundancies and uses of for loops (computational expensive) that may be able to be replaced. In addition, the data formatting code only runs for a single game, a function to cycle through every game must also be written. The DQN agent needs to be finished and the proposed Q function frame work should also be optimized.

References

[1] Seward, N. (2019). sealneaward/nba-movement-data. [online] GitHub. Available at: https://github.com/sealneaward/nba-movement-data [Accessed 8 May 2019].

[2] Schulte, O., Khademi, M., Gholami, S., Zhao, Z., Javan, M. and Desaulniers, P. (2017). A Markov Game model for valuing actions, locations, and team performance in ice hockey. Data Mining and Knowledge Discovery, 31(6), pp.1735-1757.

[3] Squared Statistics: Understanding Basketball Analytics. (2019). Identifying Player Possession in Spatio-Temporal Data. [online] Available at: https://squared2020.com/2017/05/07/identifying-player-possession-in-spatio-temporal-data/ [Accessed 8 May 2019].