SMU Data Science Review

Volume 5 | Number 2 Article 12

2021

Rocket Learn

Daanesh Ibrahim
Southern Methodist University, daaneshi@mail.smu.edu

Jules Stacy
Southern Methodist University, jstacy@smu.edu

David Stroud
Southern Methodist University, david@davidstroud.me

Yusi Zhang
Southern Methodist University, yusiz@mail.smu.edu

Rocket Learn

Daanesh Ibrahim, Jules Stacy, David Stroud, Yusi Zhang
[1] Master of Science in Data Science, Southern Methodist University,
Dallas, TX 75275 USA

Acknowledgments. Jacquelyn Cheun, PhD. – Capstone Professor

Abstract. This paper covers the development, testing, and implementation of Reinforcement Learning methods designed to autonomously learn and optimize Rocket League play. This study aims to analyze and benchmark model frameworks commonly used in Reinforcement Learning applications. These models can be applied to tasks ranging in difficulty from simple to superhumanly complex, and this study will begin with and build upon simple models performing simple tasks. It will result in complex models performing difficult tasks. Models will be allowed to train autonomously on the game using mass parallelization to expedite training times with the goal of maximizing reward function scores. This research constructs a framework to identify the best performing Reinforcement Learning model to complete this task. Multiple Reinforcement Learning methods were attempted and it was found that a Proximal Policy Optimization (PPO) model was able to learn how to play the game and consistently increase its reward function scores over time. Of all the models attempted for this game, PPO did the best job of learning how to play and it is recommended for future tasks in similar spaces.

1 Introduction

Reinforcement Learning is still in its infancy. However, even though it is not widely implemented, it is indisputable how important the concept is with regards to technological advancement and how great of an impact it will have on day-to-day life. Society is at the forefront of the era of automation, and Reinforcement Learning is the cutting edge.

Digital twins and toy models are an important explorative stage of automation and Reinforcement Learning models. Training a Reinforcement Learning model in a digital environment before deploying it in a real-world environment enables production teams to avoid risks and save untold amounts in development value regarding capital, development time, and human work. Enabling Reinforcement Learning models to train the same agent through trial-and-error through mass parallelization allows them to develop optimal strategies and paths to achieve desired outcomes while even exhibiting human traits such as logical reasoning and intuition. These outcomes are reinforced through reward functions, which reward desired behavior and punish undesired behavior. These methods allow models to solve complex problems and to develop strategies to solve future, never-encountered problems.

Reinforcement Learning is a type of Machine Learning, that allows machines and software agents to automatically determine the ideal behavior within a specific context. Markov decision processes model decision making in discrete, stochastic, sequential environments. Reinforcement Learning has its origins in Markov decision processes and learning by trial and error. As early as 1983, neuronlike models were being built to solve difficult learning problems, and Barton et. al. benchmarked such a model using the pole balancing problem [1]. More recently, genealogically evolutionary methods were developed by Stanley et al. in 2002 with the NEAT algorithm [2]. This was expanded upon with HyperNEAT, which was trained to play video games as recently as 2012 by Hausknecht et al. [3]. Reinforcement Learning entered the public view when AlphaGo learned how to play Go well enough to defeat a professional champion Go player in 2016 [4]. Silver et al. created AlphaZero in 2017, which learned how to play Chess well enough to beat other world-champion programs Stockfish and Elmo [5]. And in 2019, Berner et al. introduced OpenAI Five, which became the first AI system to defeat professional world champions at an esports game [6], Dota 2, where it won 2-0 in a best-of-3 competition. Footage of the first match is cited in the sources cited [7].
Reinforcement Learning algorithms currently in use by the industry include NEAT, QLearning, LSTM, ε-greedy, optimistic initial values, upper confidence bound, and soft actorcritic.

Reinforcement Learning is an important algorithm in that it deals with large and complex problems but may only have partial information regarding it. Most importantly, it is the closest algorithm in Machine Learning that resembles human learning. A Reinforcement Learning model can be trained to respond to unforeseen environments by receiving a reward or punishment based on every action taken. This is critical for industries such as supply chain management and robotics, where the problems that arise are highly dynamic —solutions to problems such as these need to be highly adaptive. Let the algorithm make the mistakes and learn in a simulated environment so fewer mistakes are made in the natural environment, all while optimizing the process. Future uses of Reinforcement Learning will include self-driving cars, fulfillment center robots, city traffic grids, environment controls, and military applications.

Reinforcement Learning models have a wide range of applications in Machine Learning and real-world applications. Products such as self-driving cars, automated fulfillment centers, and self-monitored coolant systems are just a few examples of Reinforcement Learning models' applications. Our theories will utilize multiple Reinforcement Learning models in a similar manner for alternative applications, namely training a Reinforcement Learning model to play Rocket League by analyzing our reward function's performance, with the ultimate goal of creating an AI player that can play and win a match against an opponent. This study aims to be able to compare model performances directly, with broader goals of both improving existing Reinforcement Learning methods as well as educating a wider audience on novel approaches to Reinforcement Learning.

Rocket League is a video game that draws similarities between hockey, soccer, rally car racing, and aerial dogfighting. Players are put into an enclosed square arena with rounded corners with goals on either end. Points are scored when the player’s team knocks a ball into the opponent's goal, with the object of the game is to score more points than the opposing team. Teams consist of up to three players per team. Players are able to accelerate, brake, turn left and right, and "drift" or power-slide their car. Players also have a boost meter which fills up when the player drives over pads on the ground, and the boost resource allows players to accelerate their car faster than what is achievable without it. Players are able to jump, double jump, control the aerial rotation of their car, and "air dodge" with their vehicle; an air dodge provides instantaneous velocity in midair and then attempts to rotate the car 360 degrees along an axis. The arena is fully 3-dimensional, and boost can be applied in midair, allowing cars to be propelled through the air. The team-focused nature of the game and the freedom of movement constrained by the arena have given rise to many strategy patterns present within the community, affecting ball control, shots on goal, offense, and defense in a very similar manner to real-world sports. The game has a dedicated audience and is considered an "esports game, with yearly global world championships. Related to Rocket League, RLBot is a community dedicated to developing superior artificial intelligence which play Rocket League and are pitted against each other in regularly scheduled tournaments. This environment provides ample opportunity for Reinforcement Learning applications, especially with regard to the development of artificial intelligence capable of playing the game.

2 Literature Review

2.1 Foundations of Reinforcement Learning

Reinforcement Learning methods originated with the concept of the Markov Decision Process. Previous work on Markov decision processes has been done by Oguzhan et al. (2010) on the construction and evaluation of MDPs used in medical decision making like the timing of liver transplantation in a patient who has a living donor available, compare Markov decision processes and standard Markov process [8]. The authors use the policy iteration algorithm in solving the illustrative MDP model by applying the backward induction algorithm while ensuring the value functions for any 2 subsequent steps are identical. MPM model is able to evaluate only one set of decision rule at a time so it calculates the total expected life years when a decision rule specifying threshold MELD scores. Authors found by selecting the threshold patient health that results in optimal policy with the largest total life expectancy. But in the MDP model, a patient's dynamic behavior complicates the decisions further, so authors developed an infinite-horizon discounted stationary MDP model with a total expected discounted reward criterion. The resulting set of actions provide the maximum values that give the optimal policies. After comparing these two models with liver transplant problems, MDP are able to model sequential decision problems in embedded decision mode at every stage, the computational time is much smaller. This could be useful in a fast-paced learning environment such as Rocket League, where algorithm completion must occur multiple times per second. However, MDP also requires the data needed to estimate a transition probability function and a reward function for each possible action, and there is no software for MDP so extra programming will be needed. Alberto et al., (2018) proposed another framework called

Configurable Markov Decision Processes (Conf-MDP), also a new learning algorithm Safe Policy -Model Iteration (SPMI) to combine and optimize the policy and environment configuration [9]. Conf-MDP's principle is to restrict attention to the transition model and focus on the problem of identifying the environment that will achieve the highest performance possible even though any of the Conf-MDP's parameters can be tuned.

Foundations for Reinforcement Learning also stem from Monte Carlo problems, where random exploration can lead to possible solutions for deterministic problems [10]. The Monte Carlo method for Reinforcement Learning learns from experience with no prior knowledge of the actions. The random aspect is whether there is a reward or not. Therefore, when applying the Monte Carlo method to Reinforcement Learning, it must be in an episodical situation. This is because an episode must "terminate" before any returns can be calculated [11]. There is no update after every action, only after every episode. Using an incremental means to update the average reward with each episode will allow the algorithm to see how much progress is made with every episode [12]. This will monitor each instance of playing Rocket League and judge if it is improving exponentially or not. Given the number of episodes this will require, anything that helps monitor the progress in a summarized way will be extremely beneficial.

2.2 Explore vs. Exploit

Many Reinforcement Learning algorithms vary their approach based on the concept of exploration versus exploitation. One of the simplest implementations of an explore vs exploit algorithm is the Multi-Armed Bandit approach.. An agent chooses an action, and that action yields a reward based on whatever the probability distribution is for that action. The agent does this over many episodes in an effort to maximize its reward. Take a slot machine for example. This slot machine possesses many levers with different rewards given for each one. Every pull of one of the levers is a "turn" and there are in total a 100 turns. Therefore, it must find a strategy to get the most reward that can be obtained in 100 turns. An option could be to pull each lever once and keep track of the reward given for each one, and then simply go back to the lever that gave the most reward. While it is a valid strategy, the issue with this method is that each lever has its own probability distribution associated with it [13]. It may take multiple turns to properly sample which levers give the most reward (or the best chance for it).

The downside to many Reinforcement Learning algorithms is that every action spent trying to gather information is an action taken away that could have been used to maximize the reward. This is the explore vs. exploit dilemma, and it represents how to balance the issue of sampling vs maximizing reward. So how does one determine the probability distributions for each action while maximizing the reward? One approach is Epsilon-Greedy [14]. EpsilonGreedy initializes by taking random actions and determining the starting average reward for each one. A value is picked for epsilon – this value represents how much of the remaining time or turns will be spent exploring the other levers and is usually quite low (5-10%). The rest of the time or turns will be spent on the action that gives the maximum reward in the first step. Another approach is Optimistic Initial Values [14]. This algorithm does not find the best action initially, but instead sets the average reward to an unattainable value. Then it goes through an option and updates the average reward for that option, and picks whatever action has the next highest average. Since the first value of the reward was so high, the average will therefore fall with every turn. Therefore, the top option will always be changing and over time the average reward for each action will reach its true value. The benefit over Epsilon-Greedy is that once the optimal choice is found, the algorithm no longer needs to explore any other options. A third approach is Upper Confidence Bound and is similar to the Central Limit Theorem in that it states as the number of observations increases, the distribution of the sample mean will approach normality centered around the true population mean [15]. So, a sample mean of 10 observations is less accurate than a sample mean derived from 100 observations. This algorithm puts an upper bound on the sample mean according to the number of observations. It no longer chooses the action with the highest reward, instead it chooses the action with the highest upper confidence bound. As the algorithm progresses the confidence bound gets tighter, allowing it to sample each action enough to know the distribution confidently and move to the next action once it is no longer competitive.

2.3 Off-Policy Learning

Soft Actor-Critic is an off-policy algorithm which is used for Reinforcement Learning tasks that contain continuous actions and which functions independently from Epsilon-Greedy, Optimistic Initial Values, and Upper Confidence Bound methods. Instead of only trying to gain the maximum reward, Soft Actor-Critic also attempts to maximize the entropy of the action. Essentially this means it wants to increase the randomness and unpredictability of the Reinforcement Learning algorithm to combat the “brittleness” of the attempted task it wants to accomplish [16]. Higher entropy encourages exploration and can create a more stable process overall. It also helps to prevent the algorithm from finding an inconsistency and exploiting it to get the maximum reward, something that would not be sustainable over time [17]. Using Soft Actor-Critic could even train an algorithm on things that it has never seen before by applying actions based on generalized conditions it faced during training. The idea is to use Soft Actor-Critic to see if what the algorithm learns in one aspect can be applied to other aspects, such as having the agent take shortcuts on the track or finding a boost on the path to maximize speed and efficiency. Deep neural networks and Q-Learning algorithms are also off-policy algorithms. Arthur et al. (2015) explored double Q-Learning in an Atari 2600 environment [18]. A deep Q network is a multi-layered neural network but double Q-Learning uses the same values to select and evaluate the action. In this paper, the results show that overestimations are really common and severe. This model was run on Atari games. With Double-Q learning one can successfully reduce the over optimistic results in a more stable and reliable learning environment.

Deterministic policy gradient algorithms are most useful in environments with continuous actions. Policy gradient algorithms are normally used by sampling this random policy and adjusting the policy parameters in the direction of greater cumulative rewards. In 2013, David et al. introduced an off-policy learning algorithm to choose actions according to a random behavior policy, then learned about a deterministic target policy to ensure deterministic policy gradient algorithms continue to find satisfaction [19]. In the experiments, authors first focused on a direct comparison between the stochastic policy gradient and the deterministic policy gradient, the results shows a significant better performance to the deterministic update; second experiments focused on the stochastic actor-critic and used the same fixed variance as the deterministic actor-critic. Compatible off-policy deterministic actor-critic Q learning had slightly better performance than stochastic actor-critic and off-policy actor-critic. Under policy gradient algorithms, Mohammad et al. (2006) proposed a Bayesian framework that models the policy gradient as a Gaussian process [20]. Early stage, both conventional and natural policy gradient methods depend on Monte Carlo techniques to estimate the gradient of the performance measure, however they tend to produce high variance estimates. Bayesian alternative is to treat the first term in the integrand as a random function, the randomness of which reflects the subjective uncertainty concerning the true identity, by modeling the gradient as a Gaussian Process. In this article, authors ran experiments in continuous-action bandit problem and a continuous state and action linear quadratic regulation problem. Bayesian quadrature has lower bias than Monte Carlo in a bandit problem, the BQ gradient estimate has lower variance than its MC counterpart in a linear quadratic regulator. In conclusion, even the experimental results are as we expected, but authors assume that there are even higher gains using this approach.

2.4 NeuroEvolution

There also exists the concept of a model whose topology (and therefore behavior) is randomly evolved from a simple starting model to a complex, application-specific model. NeuroEvolution of Augmenting Topologies (NEAT) was introduced in a 2002 article by Kenneth O. Stanley et. al. [2]. NEAT was able to outperform the best fixed-topology method; the authors claimed that this was primarily due to evolution and speciation of networks and starting with and then building upon a lightweight modeling framework. NEAT outperforms other Reinforcement Learning methods such as Adaptive Heuristic Critic and Q-Learning by searching for behaviors instead of a value function, and thus is well-adapted to handling tasks in continuous and high-dimensional state spaces. It can solve Non-Markovian pathing by trialing two models and then creating an offspring of the two models by mixing each model's "genetics." The authors benchmarked the algorithm using the classic "double pole balancing" test, and the results showed that NEAT was "several times more efficient than [other] neuroevolution methods." More recently, Helmuth et al. (2020) explored many methods for parent selection in genetic programming with the goal of benchmarking "recent and common parent selection methods" [21]. Algorithm performances were evaluated in the program synthesis domain. Established benchmarking methods were used to determine optimal parent selection methods, and the authors found that lexicase-selection-derived methods were consistently the top-performing selection methods. Ranking averages were listed, and the topperforming selection methods were Down-sampled lexicase and MADCAP ε-lexicase. Other selection methods such as Tournament or novelty search performed poorly.

HyperNEAT was a further development of NEAT explored by Van den Berg et al. in 2013. The authors examined success factors for HyperNEAT and found that performance decayed for more advanced tasks, and that there was underperformance for tasks with a fracture in the problem space when compared to NEAT [22]. They further found that HyperNEAT's performance decreased on irregular tasks, and suggested that "irregularity is an extreme form of fracture." They state that though the premise of genotype-phenotype encodings used in

HyperNEAT are promising for evolutionary computation, employing such methods in practice has proved challenging. Hausknecht et. al. (2012) developed a HyperNEAT-based agent to play Atari games *Asterix* and *Freeway* with minimized domain-specific knowledge [3]. Their ultimate goal was to generate an "agent capable of learning and seamlessly transitioning between many different tasks." The way that the HyperNEAT algorithm works is outlined in four stages within the article: 1. Compositional Pattern Producing Network (CPPN) topology evolution, 2. Using the CPPN to develop input/output node weights for a neural network, 3. Applying the neural network to a problem, and 4. Evolving CPPN populations using NEAT. Visual processing, agent detection, API interfacing, and experimental design are discussed. Test results showed that the HyperNEAT agents outperformed Sarsa(λ) agents.

Though the application of such algorithms to a game as complex as Rocket League will be a challenge, they are good starting points to properly train an agent to maximize reward efficiently.

2.5 Parallelization

In order to maximize the efficiency of training an agent, training will be parallelized to hasten training time. Eldridge et. al. (2014) proposed the utilization of neural network-based accelerators to approximate functions used by the GNU C Library which commonly appear in modeling benchmarks [23]. The authors were able to achieve math functions that were 68x lower in average energy-delay than the traditional library functions. Whiteson et. al. (2006) explored evolutionary function approximation as an automatic approach to function approximation methods for facilitation Reinforcement Learning models. Their approach combined NEAT with Q-Learning to create a NEAT+Q model, which "automatically discovers effective representations for neural network function approximators." To benchmark these methods they utilize the mountain car task and server job scheduling. Using temporal difference methods allows agents to learn over the course of its lifetime; however, cases where multi-dimensionality results in a value function that is impractically big ultimately require function approximation for value functions. The authors set out to automate the search for optimal hyperparameters that would allow for function approximation to occur in an implementable manner, as poor hyperparameter selection often results in an unusable model. NEAT methods are used to select function approximators for Q-Learning models to be applied to "on-line" scenarios, or scenarios where the agent not only tries to quickly learn an optimal policy, it also attempts to maximize the rewards for the policy. Kosiachenko et. al. (2019) developed a CUDA library known as Multi-Agent Spatial Simulation (MASS) [24]. The library attempts to address problems with automating parallelization at the GPU level, but the authors admit that their method does not improve parallel performance.

2.6 Popular Projects

Reinforcement Learning algorithms are what people think of when they imagine Artificial Intelligence or Machine Learning. This is in large part thanks to exhibitions where

Reinforcement Learning algorithms learned to play popular games such as Chess or Go, and later to a variety of video games. In 2018, Siljebråt et al. used Starcraft 2 as an environment to test both human and artificial agents on the same task [25]. One of their goals was to extend the capabilities of Reinforcement Learning to capture abstract concepts such as situational insight. They attempted to relate the field of cognitive sciences (e.g. neuroscience, psychology, behavioral economics) to general video game playing (GVGP) in such a way that it impacted their Reinforcement Learning model, where the actions taken by the agent could be reasonably taken by a human and would have logical rationale. The authors make mention of prior models such as LetaBot which uses Monte Carlo Tree Search and text mining, however they make note of the fact that prior models are not able to play an entire match as they are trained on certain portions of the game; as such prior models are difficult to benchmark and not as comparable to human behavior. The authors developed a StarCraft II Learning Environment (SC2LE) to provide an environment that mimics the environment a

human would play within. The authors conclude that instead of focusing on training a model to play an entire game, they propose that the study will focus on moving an individual unit within the game onto a beacon. They set up a framework by which they will collect human data and apply it in future work.

Berner et. al. (2019) developed a Reinforcement Learning model that learned to play Dota 2, a video game with critical acclaim [6]. Their model, named OpenAI 5, was scaled to train on approximately 2 million frames every 2 seconds, and continually trained for 10 months. The authors then trained the AI against human players; it trained against 3,193 teams and achieved a win rate of 99.4%. The model ultimately defeated a professional, world champion Dota 2 team on April 13th 2019. The researchers state that "OpenAI Five demonstrates that self-play Reinforcement Learning can achieve superhuman performance on a difficult task." The researchers outline aspects of the game of Dota 2; in particular the long time horizons, partially observed states, and high dimensionality. Regarding time horizons, the authors compare Dota 2 to both Chess and Go. Dota 2 has 20,000 steps per episode whereas Chess has 80 and Go has 150. Regarding partially observed states, Dota 2 has an active team-level fog-of-war during play, and thus the model must infer and predict enemy positioning and react accordingly. Regarding high-dimensionality, the model observes 16,000 total values and chooses between 8,000 and 80,000 actions per time step; Chess requires approximately 1,000 values per observation and Go approximately 6,000 values per observation. The authors define one time step as acting on every 4th frame, and the game runs at 30 frames per second. The authors state that certain game decisions are pre-scripted such as item purchase order; they believe that the agent could perform better if these decisions were not scripted, and that superior performance was achieved prior to scripting. The neural network built by the authors contains approximately 159 million parameters, and is passed through a Long Short Term Memory (LSTM) recurrent neural network architecture. The authors acknowledge the fact that while the model does not see all the information a human can gain access to, the model observes all information instantaneously and simultaneously. The model does not see each frame, as this would greatly impact computational complexity in a negative manner as well as impede the goal of studying strategic long-term planning. Reward functions were decided upon at the start of the project based on the team's game familiarity, and found that the initial reward choices worked well and that only minor adjustments were needed. The model was trained using self-play experience on a pool of up to 1,536 GPUs. The Adam optimizer was applied to samples of length 16 timesteps. Games were run at approximately half the speed of real time as more than double the number of parallel game instances could be run. The most recent agent iteration trained against itself 80% of the time and against old policies 20% of the time. The authors discuss computing architecture and game/model communication, and how model data is updated while games are in-progress rather than at the end of each game. Because training the model was capital-intensive with long time frames, and because game updates would change the structure of the model, the authors would update the model using a process they termed “surgery,” where new models would be created from updated older models. The authors continue to discuss training parameters such as batch size, long term credit assignment, and adjusting for long time horizons.

Given the various methods of Reinforcement Learning that have been described, a technique will be developed to produce the best performing agent for a continuous deterministic environment such as Rocket League.

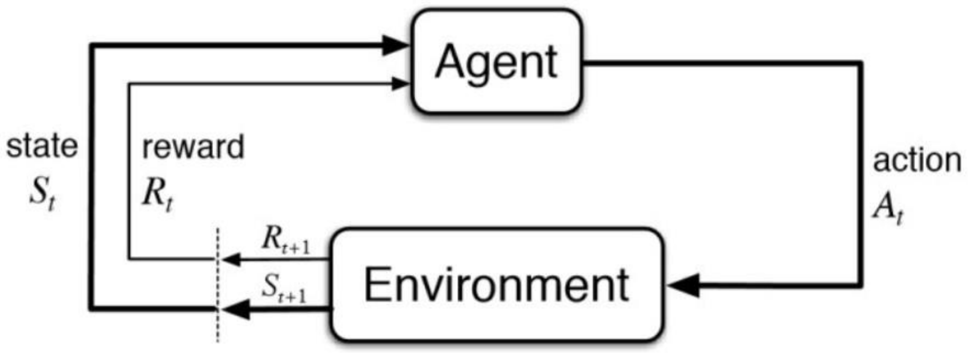
3 Methods

3.1 Data Collection and Model Utilization

A Reinforcement Learning model is trained using the Rocket League client as the training environment, which serves as the source of data. The algorithm is developed within RLBot, an extension of the Rocket League client. RLBot is a development environment for building automated Rocket League players, or “bots.” RLBot was developed by the RLBot community, and consists of a Rocket League client, a menu for adjusting match settings, and a python interpreter to read custom code for developing bots. The end goal is to build a bot capable of playing and winning a match against an opponent. RLBot allows raw gamestate monitoring in real time, including the position, velocity, and orientation of all cars and the ball within the arena. In addition it allows for the monitoring of the time remaining in the match, the score, what gamemode is currently being played, and much more. With this raw input available to the AI, a deep learning model could apply it’s filters to this raw data and generate a strong understanding of what it means. Thus, with the use of RLBot a deep learning AI for Rocket League is possible. The efficacy of the trained bot can be tested in live matches against opponents (both bot and human) and overall win rates can be used as scoring metrics.

Data is collected from successive training sessions within the Rocket League client in live exhibition matches against active opponents. The environment will be mass-parallelized, such that the algorithm will train a multitude of agents at the same time; this will allow for more rapid learning which will ultimately speed the training process. Reinforcement Learning models that will be explored include Multi-Arm Bandits, NEAT, Policy Gradient Algorithms, Deep Q Learning, and the Soft Actor Critic Model.

General Overview of Reinforcement Learning



[26] Fig. i: The Reinforcement Learning loop

Reinforcement Learning is an unsupervised Machine Learning method by which an algorithm learns to perform a task. The general method flow is illustrated in Figure i. The agent is the actor, and represents the portion of the model that the algorithm can control. This agent takes actions within the environment, which is the general setting and which the algorithm can interact with through actions taken by the agent. The agent takes an action, and obtains updated data from the environment. Once the algorithm obtains and processes the new environment information, it updates the agent's state. The state of the agent is a concept relating to the agent's current action, and could include variables such as velocity, physical and metaphysical properties, structural properties, locational data, and many more. During this state update, the algorithm also evaluates and updates its reward function. Another loop is then primed for the agent to take another action based on the updated state and reward functions.

The reward function is a separate measure that determines the agent's overall performance and ultimately drives its actions. The reward function is in essence a list of desired and undesired behaviors or outcomes that are scored and applied to the agent as it performs actions. If an outcome is desirable it is assigned a positive score, and if an outcome is undesirable it is assigned a negative score. In this way, the agent is rewarded and punished for the outcomes of its behaviors. Rewards and punishments can be very closely related to agent actions, such as rewarding speed increases, or indirectly related, such as a reward being applied when a ball goes into a goal. The algorithm is then directed to maximize the score from the reward function, and in this way the reward function directly drives the behavior of the agent.

Similar to other Machine Learning methods, there are training and test phases. During the training phase, the algorithm is given the opportunity to take explorative or exploitative behavior (this is referred to "explore vs. exploit"). At the initial stages of training, the agent takes purely explorative behavior as it learns how to interact with the environment, how to avoid punishment, and how to obtain rewards. As the agent's pathing becomes more defined and as its actions become more optimized, the algorithm changes from exploring to exploiting as it attempts to maximize its reward score. During the test phase, the agent undergoes a trial in order to benchmark its performance.

With regards to Rocket League, the car is the agent. The arena, ball, and opponents are the environment. Actions the agent can take include turning, jumping, air dodging, boosting, drifting, accelerating, and aerial rotation. And the greatest reward will be when the agent wins a match, and the second greatest reward will be when the agent scores a goal.

3.2 Markov Decision Process

The Markov Property is a theory of probability named after the Russian mathematician Andrey Markov, in which the values of future observations depend only on the series' most recent values. As a result, the Markov Decision Process (MPD) will not memorize the past if the present state is given; thus it is a memoryless random process. In general, the relationship between Reinforcement Learning(RL) and MDP is that RL is a framework for solving problems that can be expressed as MDPs. One key factor that affects how well RL will work is that the states should have the Markov property which the value of the current state is enough knowledge to fix immediate transition probabilities and immediate rewards following an action choice.

A Markov Process (or Markov Chain) can be represented by a tuple {S,P}: S is (finite) set of states, and P is the state transition probability matrix Pss' = P [St+1 = s ' | St = s]. The Markov reward process is a Markov chain with values{S, P, R, γ}. R is a reward function and γ is a discount factor, where γ ∈ [0, 1]. The decision process is a Markov reward process with decisions {S,A, P, R, γ}. S is a finite set of states, A is a finite set of actions, P is a state transition probability matrix where P a ss'= P [St+1 = s ' | St = s, At = a], R is a reward function where Ra/s = E [Rt+1 | St = s, At = a], and y is a discount factor where γ ∈ [0, 1]. In the two functions below, an MDP is a 5-length tuple {S, A, P, R, y} where [27]:

- S is a set of states
- A is a set of actions
- P(s, a, s') is the probability that action a in state s at time t will lead to state s' at time t+1
- R(s, a, s') is the immediate reward received after a transition from stat s to s', due to action a
- y is the discounted factor which is used to generate a discounted reward.
- π is the policy, a solution to the MDP, and is defined in each possibility of state

$$\mathcal{P}^{\pi}_{s,s'} = \sum_{a \in \mathcal{A}} \pi(a|s)\mathcal{P}^{a}_{ss'}$$

This function gives the probability that a given action will change from state S to state S' [28]. This probability function can be explained as the probability of state change from S to S' under state S when executing policy π, and should equal the sum of the probability of executing all the actions under state S times the corresponding action's probability.

$$\mathcal{R}_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s)\mathcal{R}_s^a$$

This reward function can be explained as the reward under state S when executing policy π, and should equal the sum of the probability of executing all the actions under state S times the corresponding action's reward.

The goal of MDP is to frame the problem of learning from interaction to achieve a goal. This directly applies to Reinforcement Learning which is founded upon MDP and operates in a very similar manner.

3.3 Deep Learning

Deep learning is a part of the broader topic of Machine Learning, specifically those based on artificial neural networks with representation learning. This learning can be further categorized as supervised, semi-supervised, or unsupervised. The "deep" aspect of it refers to the use of multiple layers within the network to extract higher level features from raw input. These higher-level features can be abstract and composite representations of the original raw data, such as a matrix of pixels or separating individual profiles from a group photo in image recognition. The caveat of deep learning is that the algorithm figures out these higher-level features completely on its own. After initializing and tuning the algorithm, the design ultimately lets the algorithm read inputs and separate out noise while applying layers to learn the important features effectively. The number of layers present in the algorithm is usually a direct indication of how complex it is as well. While this could be interpreted as "the more layers the better", this is not always the case. For example, a deep learning network which contains numerous layers can be prone to overfitting and long computation times. Overfitting can be defined as taking non-important variables which would normally be classified as noise and attempting to extract meaningful data for the algorithm to use. This results in a situation where finding additional features is more difficult. The same is true for the inverse: underfitting a model results in a poor representation of the entire dataset. Any predictions or outputs created by an under fitted or overfitted model would be unreliable. If the size of the training data contains too many features or observations the deep learning model runs the risk of extensive computation times. For modern GPUs and CPUs computation time is usually a non-factor, however it could be an issue for older hardware or be indicative of a larger issue (such as uncleaned data).

With regards to Rocket League, an AI using deep learning could theoretically improve over time and achieve superhuman performance. Currently, the AI in the Rocket League client are used as placeholders until a human player can connect to the game. This allows the rest of the players on the server to start the match while waiting for additional human players to take the place of the AI. The AI is also used to replace a player who suddenly left the game in the middle of the match so that the game can continue for everyone else. The AI in Rocket League are “basic representations” of players, and often pale in comparison to human performance, such that they can barely defend the goal or score on an open net. Often in casual online matches, the AI are viewed as liabilities on both ends of the field. If an AI could be successfully trained using deep learning, then there would be no need for “placeholder” AI. Instead, a competitive AI could be present that would hold its own against a multitude of opponents. Over time, the AI could learn optimized strategies to be dominant against any competitor in real time, including changing strategies based on the current situation in order to best ensure a victory.

3.4 Proximal Policy Optimization

Proximal Policy Optimization (PPO) belongs to the policy gradient algorithm family. This means that the policy is updated in such a way that the probability of actions taken now are much more likely to provide a larger reward in the future. The agent will explore the state and the algorithm will track the agent’s actions and how the state changes as a result. These interactions are referred to as trajectories. When the algorithm has a collection of trajectories it will examine them to see which actions performed by the agent resulted in a positive or negative reward [29]. The policy will then be updated based on these results.

With regards to weights, these are adjusted to make favorable results more likely and make bad results less likely. Within the PPO, the advantage function does this by predicting the benefit of the action the agent is about to perform. The actual result of the action is then recorded and compared to the advantage estimate. If it is better than the estimate, the gradients are updated with increased weights that make that action more likely to occur again. If it is worse than the estimate then the result will be weights that are decreased making that action less likely to occur [29].

What sets PPO apart from other algorithms is the probability ratio that factors into updating the policy. This is the difference between the original action log probability and the new model’s action log probability. It prevents large updates to the policy from happening. Why would this be beneficial?

Loss function using ratios from the new and old policy action log probability, referred to as Conservative Policy Iteration (CPI) [29]

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right].$$

With normal policy gradient algorithms, any update to the policy can potentially be so drastic that a potential reward peak window could be missed. Without that knowledge of the large reward, the policy will not know which way to update the gradient to improve the policy loss. With this confusion the policy could potentially be eliminated without being able to be recovered. A PPO using the ratio of the original action log probability to the new model's action log probability will prevent the risk of missing reward windows and hurting the policy. It can't miss reward windows because the policy is not allowed to update too much after every step - the updates are clipped according to a set amount (the default is 0.2). The tradeoff for this is that training agents can take quite a long time on even the most basic functions.

PPO is the perfect algorithm for an environment as complex and continuous as Rocket League. There are so many different ways to achieve the goal-state. Many cannot be coded into proper rewards. The agent must explore and try things incrementally in order to make proper connections from actions to desired state changes. The goal is to have the agent try different avenues and see what works. Rocket league is a game of unpredictability - no two games are alike. PPO helps the agent to do that as well. It is much more difficult to predict the actions of an agent trained using PPO than it is with any other policy gradient algorithm. It is expected that over the course of many timesteps, an agent worthy of the ever changing environment of Rocket League will be produced.

4 Results

4.1 Scope

The scope of this study is to compare the performance of a Reinforcement Learning model with its past performance and show evidence of improvement. Further, the study aims to compare a successful Reinforcement Learning model with other models; in this case a Proximal Policy Optimization model and a Twin Delayed DDPG model.

4.2 Model

The optimal model for this application was found to be Proximal Policy Optimization. This model was successful in increasing its average reward score over time. These results are shown in Fig. 2. Average scores approximate a natural log curve. Additionally, it exhibited the behavior of intentionally hitting or carrying the ball into the goal.

A TD3 and an A2C model were both attempted, but neither model was able to intentionally hit or carry the ball into the goal. These models would begin training in a similar manner as the PPO model, then would relatively quickly get stuck in a forward/reverse behavioral pattern in the middle of the field, unable to move.

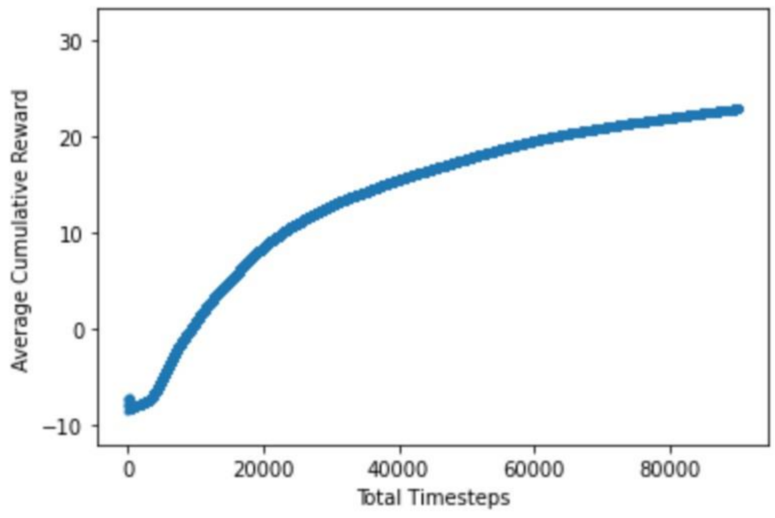


Fig. 2: Average Cumulative Rewards vs. Timesteps Trained, PPO

5 Discussion

5.1 Reward Function

The overall objective of Rocket League is to score more goals than the opponent in the allotted time. While this is the most important aspect of the game, rewards must be formulated to help the agent achieve this task as efficiently as possible. Below is the breakdown of the rewards used to train the agent.

- *Event Reward*
- Reward given for every goal the agent scores.
- Punishment given for every goal scored against the agent.
- Reward for every time the agent touches the ball.
- Reward given for every shot the agent takes at the goal.
- Reward given for every save the agent makes.
- Reward given for every time the agent demolishes the opponent.
- *Velocity of agent to ball*
- Finding the distance between the agent and the ball multiplied by the velocity the agent was traveling at. The closer the agent is to the ball the larger the value for “distance” will be (ratio using normalization).
- This is an attempt to train the agent to get to the ball as fast as it can. It should make shot attempts easier to come by if it can beat the opponent to the ball.
- *Velocity of ball to goal*

- Finding the distance between the ball and the agent's target goal multiplied by the velocity the ball was traveling at. The closer the ball is to the goal the larger the value for "distance" will be (ratio using normalization). This value is then multiplied by the ball's height with respect to the height of the goal (1+[height of ball/height of goal]).
- The agent will learn to hit the ball towards the goal consistently and with speed while giving it a bonus multiplier if the ball has height on it. Shots that have height and speed are significantly harder to save than just a slow aerial shot or fast ground shot.
- *Reward for agent being behind ball*
- Take the Y-coordinate position of the agent and the ball. If the ball's Ycoordinate is greater than the Y-coordinate of the agent, the reward is given.
- Helps to train the agent to keep the ball in front of it at all times. This is beneficial to learn for offensive and defensive purposes.
- *Save Boost*
- The boost amount the agent currently holds from 0 to 1 (50 boost being equal to 0.5). The square root of this number is then taken for the reward.
- Taking the square root allows the bot to learn that even holding 20 boost is beneficial than holding 0 boost. Helps to train the agent not to waste boost unnecessarily.

5.2 Exploitation of reward function

The agent's primary goal is to maximize the score it achieves from the reward function. In other words, given enough training time the agent will attempt to display only those behaviors which will increase its reward score. This behavior is called exploitation, and it is pivotal to an agent's performance. With this in mind, behaviors are observed after training that demonstrate that the agent exhibits exploitative behavior.

Shots made by the agent tend to go into the air. Rewards obtained for a shot on goal are multiplied by a coefficient that increases with the height of the ball. It is possible for the exploitation of this behavior to become problematic as the reward happens even when the shot is made inaccurately; this is balanced by the fact that a reward is given if the ball goes into the goal.

Other behaviors the agent expresses are demolishing cars and not expending boost immediately. These are consistent with exploitative behavior. The agent is also consistently observed aligning the ball in-between the car and the opponent's goal; this increases the likelihood that the ball will go in the goal, and is a form of exploitative behavior.

5.3 Subjective comparison to human play

As a subjective analysis of the agent's level of play at 175 million timesteps of training, the agent takes common actions that a human player would take during play. Humans tend to develop strategies that take into account the positioning of their car, the ball, and the goal in such a way that the three are aligned in order to maximize the likelihood that a shot on the goal would go in. Additionally, skilled human players are likely to shoot on the goal in an unpredictable manner in order to make goalkeeping difficult, and save boost for pivotal moments in order to tip the advantage in their favor. After training, the agent is consistently observed aligning the car, ball, and goal: if the ball is to the right of the goal, the agent will drive further right until the ball is between the car and the goal before taking a shot. The agent is also consistently observed shooting the ball into the air rather than on ground level as well as saving boost for big moments, both of which are consistent with rewards given by the reward function.

Shortcomings are observed in the agent's playstyle. Agents consistently remain oriented towards the opponent's goal, and if both the ball and the opponent's goal are not in front of the car the agent will drive backwards until they are. In this situation a human player typically orients towards the ball rather than the opponent's goal, and will turn around to position themselves between their own goal and the ball. Additionally, agents are not observed jumping into the air with any frequency. Jumping and aerial flipping are commonly observed actions of human players as these actions impart immediate change to the momentum of the car, allowing for control of the car in midair.

As discussed, these behaviors are consistent with exploitative behavior by the agent in order to maximize its reward score. Desired behaviors not yet demonstrated can still be learned through additional training and reward function adjustment.

5.4 Proximal Policy Optimization vs. Other Models

The Proximal Policy Optimization method performed best in this space because it is designed for continuous action spaces, long-term goal achievement, and gradual policy updates [29]. These properties allow agents to develop beneficial behaviors while also allowing for the development of fundamental behaviors. Other models attempted were a Deep Q Learning model and a Twin Delayed DDPG (TD3) model. These were found to be ineffectual for learning how to play Rocket League. Deep Q Learning was developed for discrete action spaces such as in the game of Go, whereas Rocket League is a continuous space. TD3 is an update to Deep Deterministic Gradient Policy (DDPG), which suffers from brittleness with respect to tuning hyperparameters as well as overestimation of reward probabilities. These shortcomings are addressed in TD3, which learns two reward functions instead of one, delayed updates to its policy, and attempts to smooth actions by adding random noise. However, this method is still problematic when compared to PPO. TD3's policy is still prone to large updates resulting in potentially drastic behavior changes, making it more prone to non-discovery of fundamental behaviors; this is contrasted with PPO's much more gradual policy updates. And the noise that TD3 adds to actions is problematic for games with high amounts of physics precision such as Rocket League, since it interferes with object control (namely, steering the car and dribbling the ball).

Subjectively, the TD3 function does not behave in an optimal manner. It does not exhibit strategic behavior, and play appears to be chaotic. The gaussian noise added to actions causes the car to jump and flip constantly, which interferes with the agent's ability to approach the ball. The agent does not touch the ball and goals scored are more often attributed to random chance than to exploitative behavior. Early iterations of PPO exhibit this same behavior, which is chaotic; however PPO is able to actively increase its reward scores unlike TD3.

5.5 Real-World Applications

Though Reinforcement Learning is likely to one day be used for video gaming applications, there are other industries more likely to make use of it before then. The most direct real-world applications for Reinforcement Learning related to Rocket League are self-driving cars and rocket trajectories, since agents trained in Rocket League have control over a car that can fly through the air with a rocket booster. Besides these and other automated trajectories systems, Reinforcement Learning is actively used in cooling systems for computer systems, trading and finance, natural language processing, and others.

5.6 Challenges

Reinforcement Learning is a novel, broad, and convoluted topic. This presents a formidable challenge to new practitioners and learners, as it proves to be a difficult topic to learn and apply. Because of its breadth and the application diversity, it is not feasible to test the full list of models in a single environment. Model selection and hyperparameter tuning was particularly challenging.
Reinforcement Learning is not the only novelty relating to this project. The environment used, RLGym, was released shortly after the inception of the project. This added yet another degree of abstraction, and learning how to interact with environment proved to be a formidable task in its own right.

5.7 Additional Insights

Since Reinforcement Learning is such a broad topic, the potential for further research is obvious. Areas for further research span from hyperparameter tuning to model selection, and even include the development of new Reinforcement Learning models. In fact, the topic was chosen for its broad nature and this project carried out in an exploratory fashion. To that end, one of the primary reasons this problem was chosen was for fun! Another reason was to showcase the possibilities of Reinforcement Learning to a broader audience, which is expected to conduct further independent research on the topic. Ultimately, the project proved to be a success: a Reinforcement Learning agent was trained to play Rocket League, and the method used was compared to others. But beyond that, the project was successful in other ways; namely, it served as an excellent learning experience.

5.8 Ethics

Artificial intelligence has become part of daily life; dealing with the simulation of intelligent behavior using computers concerning the capacity to copy and ideally improve human behavior. It is aiming to improve efficiency, lower the costs and accelerate development but at the same time there are a lot of worries that those complex models may do more societal damage than economic benefit. Reinforcement learning is well used in a big variety of fields like autonomous driving, trading and finance, reinforcement learning in Natural Language Processing, healthcare, gaming, news recommendation, marketing and advertising, etc. In autonomous driving, there are some considerations such as driable zones, traffic rules and traffic conditions, avoiding human and other objects, speed limits in different road choices, etc. The core of reinforcement learning is that the optimal behaviour or actions is reinforced by a positive reward. In order to get the positive rewards, the model will need more and more driving data captured from the cameras and LiDar and radar around the car to maximize the reward which is to make predictions about the surroundings and take actions based on those predictions.. The confidentiality of the customer's driving data is a big concern in this scenario. Companies will be able to collect the driving habit, driving path, real-time location information, etc. There is a lot of discussion around data privacy for self-driving car use, the data can be valuable to various government and private organizations for various uses that it does not matter if it benefits the driver or not. The fact that the user's location, on- road behaviour and inside or outside footage of the car is personal privacy, there is no party who should ever use that without the driver or the passenger's consent. The potential data security risks come from a variety of sources, both internal and external to the self-driving car, eventually there should be laws to cover data usage and privacy.

Reinforcement Learning Algorithms used in the self-driving vehicle's decision making process might lead to some ethical problems, the risk to the safety of the passengers and human beings on the road. There is no guarantee that accidents will be avoidable for selfdriving cars. How should the algorithm minimize the risk of harm like loss of life is a bid discussion. When human driving, there are moral values to consider while making the decision on how this accident would happen. A self-driving car will make real-time decisions while driving with various inputs from the sensor data and the result of code developed by a programmer ahead of time. It is really difficult to code the decision in a lot of scenarios. For instance there is a unpreditic driving car straight towards the self-driving car, to avoid that, the car needs to hit the wall on the left side. It might cause a serious crash and bring death to the passenger. Or more ethically, a self-driving car has the option to go straight to hit a person or to avoid hitting that one person doing a sharp turn might hit the other person on the other side. There is no right or wrong answer, both are human beings. But there will be some grey answer if it is being discussed in ethical perspective, for example if one pedestrian is criminal and the other pedestrian is a pregnant lady, with all these information like gender, age, criminal background, will this make the car change the decision when the accident has to happen especially with Reinforcement Learning, it tend to let the car to make the decision on its own. There should be an ethical discussion behind the development of the self-driving car.

6 Conclusion

Training an AI to play Rocket League shows what Reinforcement Learning is capable of. Rocket League is a complex game that even humans have trouble learning effectively in. This is because of the information overload that can occur - too many options to potentially perform. Mistakes are going to occur, but how well do humans learn from their mistakes? That answer varies, but for a Reinforcement Learning algorithm the answer is clear - yes, but extremely slowly.

The optimal model for this application was found to be a PPO model. This model is able to consistently improve its reward score over time, indicating that it actively learns desired behaviors and performs exploitative behavior in attempts to maximize its reward scores.

Ultimately this model is a good choice for applications with continuous action spaces and long term goal setting and achieving, and could be a good candidate in self-driving vehicles, trajectory pathing, and industry automation.

References

1. A. G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," in IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, no. 5, pp. 834-846, Sept.-Oct. 1983, doi: 10.1109/TSMC.1983.6313077.

2. Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. Evol. Comput. 10, 2 (Summer 2002), 99–127. DOI:https://doi-org.proxy.libraries.smu.edu/10.1162/106365602320169811

3. Matthew Hausknecht, Piyush Khandelwal, Risto Miikkulainen, and Peter Stone. 2012. HyperNEAT-GGP: a hyperNEAT-based atari general game player. In Proceedings of the 14th annual conference on Genetic and evolutionary computation (GECCO '12). Association for Computing Machinery, New York, NY, USA, 217–224. DOI:https://doiorg.proxy.libraries.smu.edu/10.1145/2330163.2330195

4. Silver, D., & Hassabis, D. (2016, January 27). *AlphaGo: Mastering the ancient game of Go with Machine Learning*. Google AI Blog. https://ai.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html.

5. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv, abs/1712.01815*.

6. Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J.W., Petrov, M., Pinto, H.P., Raiman, J., Salimans, T., Schlatter, J.,

Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., & Zhang, S. (2019). Dota 2 with Large Scale Deep Reinforcement Learning. *ArXiv*, *abs/1912.06680*.

7. TI9 CHAMPION OG vs OpenAI Final Version 2019 - Game 1. (2019, September 3). [Video]. YouTube. https://www.youtube.com/watch?v=t4il-QagP5w

8. Oguzhan A., Heather H., Andrew J. S., Mark S. R., Markov Decision Processes: A Tool for Sequential Decision Making under Uncertainty, University of Pittsburgh, 2010

9. Alberto M.M., Mirco. M., Marcello M..(2018)Configurable Markov Decision Processes. https://arxiv.org/pdf/1806.05415.pdf.

10. Medium. (n.d.). Retrieved April 12, 2021, from https://towardsdatascience.com/an-overview-of-monte-carlo-methods-675384eb1694

11. Choudhary, A. (2020, April 30). Reinforcement Learning: Introduction to Monte Carlo Learning using the OpenAI Gym Toolkit. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2018/11/reinforcement-learningintroduction-monte-carlo-learning-openai-gym/

12. Brownlee, J. (2019, September 25). A Gentle Introduction to Monte Carlo Sampling for Probability. Machine Learning Mastery. https://machinelearningmastery.com/monte-carlo-sampling-for-probability/

13. Choudhary, A. (2020b, May 24). Reinforcement Learning Guide: Solving the Multi-Armed Bandit Problem from Scratch in Python. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-banditscratch-python/

14. Hubbs, C. (2020, January 8). Multi-Armed Bandits and Reinforcement Learning - Towards Data Science. Medium. https://towardsdatascience.com/multi-armedbandits-and-reinforcement-learning-dc9001dcb8da

15. Dar, E. E., Mannor, S., & Mansour, Y. (2006). Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems. Journal of Machine Learning Research 7, 1079(1105), 3–4. https://www.jmlr.org/papers/volume7/evendar06a/evendar06a

16. V.Kumar, V. (2019, January 9). Soft Actor-Critic Demystified - Towards Data Science. Medium. https://towardsdatascience.com/soft-actor-critic-demystifiedb8427df61665

17. Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., & Levine, S. (2019). Soft Actor-Critic Algorithms and Applications. ArXiv Preprint ArXiv:1812.05905, 1–6. https://arxiv.org/pdf/1812.05905.pdf

18. van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1). Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/10295

19. David S., Guy L., Nicolas H., Thomas D., Daan W., Martin R.. (2013) Deterministic Policy Gradient Algorithms

20. Mohammad G., Yaakov E.. Bayesian Policy Gradient Algorithms, Department of Computing Science, University of Alberta, 2006

21. Thomas Helmuth and Amr Abdelhady. 2020. Benchmarking parent selection for program synthesis by genetic programming. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20). Association for Computing Machinery, New York, NY, USA, 237–238. DOI:https://doiorg.proxy.libraries.smu.edu/10.1145/3377929.3389987

22. Thomas G. van den Berg and Shimon Whiteson. 2013. Critical factors in the performance of hyperNEAT. In Proceedings of the 15th annual conference on Genetic and evolutionary computation (GECCO '13). Association for Computing Machinery, New York, NY, USA, 759–766. DOI:https://doiorg.proxy.libraries.smu.edu/10.1145/2463372.2463460

23. Schuyler Eldridge, Florian Raudies, David Zou, and Ajay Joshi. 2014. Neural network-based accelerators for transcendental function approximation. In Proceedings of the 24th edition of the great lakes symposium on VLSI (GLSVLSI '14). Association for Computing Machinery, New York, NY, USA, 169–174. DOI:https://doi-org.proxy.libraries.smu.edu/10.1145/2591513.2591534

24. Kosiachenko L., Hart N., Fukuda M. (2019) MASS CUDA: A General GPU Parallelization Framework for Agent-Based Models. In: Demazeau Y., Matson E.,

Corchado J., De la Prieta F. (eds) Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection. PAAMS 2019. Lecture Notes in Computer Science, vol 11523. Springer, Cham. https://doi.org/10.1007/978-3-030-24209-1_12

25. Henrik Siljebråt, Caspar Addyman, and Alan Pickering. 2018. Towards human-like artificial intelligence using StarCraft 2. Proceedings of the 13th International Conference on the Foundations of Digital Games. Association for Computing Machinery, New York, NY, USA, Article 45, 1–4. DOI:https://doiorg.proxy.libraries.smu.edu/10.1145/3235765.3235811

26. Bhatt, Shweta. "5 Things You Need to Know about Reinforcement Learning." *KDnuggets*, 2021, www.kdnuggets.com/2018/03/5-things-reinforcementlearning.html

27. Blackburn. "Reinforcement Learning : Markov-Decision Process (Part 1)." *Medium*, 23 Aug. 2020, towardsdatascience.com/introduction-to-reinforcementlearning-markov-decision-process-44c533ebf8da.

28. 范叶亮. "马尔可夫决策过程 (Markov Decision Process)." Leo Van | 范叶亮, 23 May 2020, leovan.me/cn/2020/05/markov-decision-process.

29. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.