Deep Reinforcement Learning Applied to a Racing Game

Charvak Kondapalli, Debraj Roy, and Nishan Srishankar

***Abstract*—This is an outline of the approach taken to implement the project for the Artificial Intelligence course. Autonomous driving has recently become an active area of research, with the advances in robotics and Artificial Intelligence technology. Therefore, the aim of the project was to apply a Deep Reinforcement Learning technique to a racing game to investigate the performance on autonomous driving tasks. Our goal is to understand if reinforcement learning is a viable algorithm genre for self-driving cars in addition to deep learning through the use of the Outrun simulator as a first step. Several algorithms were considered, and we include an extensive survey of the related work; but only one algorithm could be trained due to time and GPU-resource constraints.**

The Deep Deterministic Policy Gradient algorithm was chosen for the final implementation on account of actor-critic algorithms generally outperforming value-based algorithms, both in terms of the time and the resources required for training the algorithm. The game used was the CannonBall OutRun, which is a reimplementation of the classical SEGA Out Run without any timing constraints or the presence of traffic or opposing racers. The algorithm is evaluated based on the improvements in the rewards received during the training period. For our implementation, we used the open source neural networks API, Keras, running TensorFlow in the backend. Quantitative results are provided along with several screenshots of the game during training.

A video of the algorithm training on the game at around the 500 episode mark is available on YouTube. It was seen that the ability of the agent to find a policy to maximize expected rewards increases over episodes. This is noted visually as the car travels further without crashing into obstacles and obtains a higher score from the game.

I. Introduction

This project is part of the requirements for the Artificial Intelligence course at WPI (Fall 2017), and deals with Deep Reinforcement Learning which is a subset of Artificial Intelligence. Reinforcement Learning deals with scenarios where agents interact with the environment and attempts to use trial and error to learn an optimum policy for sequential decisions which in turn maximizes cumulative reward. Deep Neural Networks have been integrated with Reinforcement Learning where the value function, model, or policy can be replaced with a deep neural net.

In this project we aim to see if a Deep Reinforcement Learning algorithm can be trained to successfully play the Out Run arcade game. The metric for success is explained further in Section III-C.

While this project is implemented on an Outrun racing simulator, the motivation and significance for this project is to know if deep reinforcement learning algorithms can be utilized to obtain proper and safe driving policies in actual self-driving vehicles. In actual self-driving cars with more comprehensive sets of states, observations, and actions better tuning of rewards and policies would naturally be needed. Should the Reinforcement learning be apt on simulation, with further training and better implementation it can be utilized with other deep neural networks as part of an ensemble in hardware. In such a system, the agent would maintain autonomous driving if there is a consensus between at least two of the algorithms in the ensemble, and if there is a discrepancy, hand over control to a human driver.

Fig. 1: Types of Autonomy levels in a Self-Driving car[1]

This report is divided into five sections. We introduce the project and describe the problem statement in Section I. Section II provides a detailed discussion of the related and useful work to the present project. The initial plan for the implementation and evaluation of the algorithm is presented in Section III. Section IV describes the methodology employed for the execution of this project and the results of the performance evaluation are given in Section V. Finally, Section VI provides the conclusions and final remarks on the project and speculates probable directions for future work.

II. Literature Review

Autonomous driving has been an extremely active research area in the fields of Robotics, AI, and Controls since Stanford's Stanley won the 2005 DARPA Grand Challenge using a machine-learning based obstacle avoidance software and focusing on improving algorithms than implemented hardware. In the real-world, there are four levels of autonomy as seen in Figure 1.

To create an autonomous agent, there are three main tasks that need to be completed, namely Recognition, Prediction, and Planning. Recognition deals with identifying components in the surrounding environment such as pedestrians, traffic

signs, surrounding cars that will be of interest for our agent. Prediction uses information obtained to predict the future state of the environment such as building a map or object tracking. The planning stage deals with creating an efficient model to plan a series of driving actions for successful navigation. The combination of Reinforcement Learning and Deep Learning helps the agent achieve human-level control.

[2] proposes a lightweight framework that uses asynchronous gradient descent for the optimization. The authors analyze and present asynchronous versions of four standard reinforcement learning algorithms - Single step Q-Learning, Single step SARSA, n-step Q-Learning, and the A3C. They found that using parallel actor learners to update a shared model had a stabilizing effect on the learning process of value-based methods. [3] is arguably the most similar to the present work. They implement [2] for end-to-end driving in a car game, and rely only on images and car speed from the screen. The main difference between this implementation and the implementation provided in [2] is that they use the World Rally Championship 6 (WRC6) simulator which they argue to be more realistic.

An iterative procedure for optimizing policies is proposed in [4] and guarantees a monotonic improvement. The authors unify policy gradient and policy iteration methods, and show them to be special cases of optimizing a certain objective - subject to a trust region constraint. The issue with applying TRPO or its equivalent PPO, is that these algorithms are very data hungry and require thousands of images to train.

One of the recent developments for end-to-end driving was proposed by researchers in NVIDIA[5]. They utilized a single camera mounted on the hood of the car to obtain road images with steering angle labels. An end-to-end Convolutional Neural network was trained to map pixels to steering information. It was found that the CNN learns to implicitly detect road outlines etc, and also to output an appropriate steering angle given an input road image. However, most algorithms used in real-world driving cars utilize just deep learning. This end-to-end network could be implemented on a expert-driven policy i.e. where a human demonstrates actions to take on a lap as a form of imitation learning/behavioral cloning. The issue with imitation learning is that the data is not IID, and has no failure cases, such that the algorithm diverges by a small error every time creating compounding mistakes. This can be corrected in a costly manner using a DAgger policy (Data Aggregation) where data is obtained from run policies, and is added to the dataset and used to retrain a new policy.

Implementations of reinforcement learning were performed in simulation[6], as is what will be done with our project.

The main challenges of implementing a DRL algorithm on a simulator would be training it on a GPU cluster until the normalized reward vs. steps/episodes reaches steady-state. Furthermore, choosing a proper actor-network, a critic-network, as well as a proper reward function are the natural challenges of this project. It involves having to properly understand the observations that can be utilized, as well as understanding the scenario. There are few immediate benefits from solving this problem, however, it will show that as a proof of concept, that reinforcement learning algorithms can be utilized for solving simulation games for driving behavior. With proper tuning of reward functions, perhaps, the same approach could be used as one possible algorithm (among an ensemble of algorithms) in actual hardware.

TABLE I: Evaluations of different DRL training algorithms on 57 Atari 2600 games

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9 %	47.5 %
Gorila	4 days, 100 machines	215.2 %	71.3 %
D-DQN	8 days on GPU	332.9 %	110.9 %
Dueling D-DQN	8 days on GPU	343.8 %	117.1 %
Prioritized DQN	8 days on GPU	463.6 %	127.6 %
A3C, FF	1 day on CPU	344.1 %	68.2 %
A3C, FF	4 days on CPU	496.8 %	116.6 %
A3C, LSTM	4 days on CPU	623.0 %	112.6 %

Table I is a survey of training speeds of different algorithms on 57 Atari 2600 games. Due to time constraints as well as resource issues, one of the main considerations was to choose an algorithm that provided a relatively high mean score/accuracy with a low(er) training time. It was seen that overall, policy-based advantage actor-critic methods outperforms value-based methods that were used.

III. Methodology

In this section, we lay down possible algorithms and simulators that were looked into, and could be used for our project. Following further research, Deep Deterministic Policy Gradient (DDPG) algorithm was selected to be trained on the CannonBall OutRun environment. OutRun was chosen because of the popularity of the game and the simplicity of the interface.

A. Algorithms

There are a number of existing algorithms[7], [8] that can be used to solve this problem such as the Adaptive Heuristic Critic, TD(λ) algorithm, Q-Learning algorithm, Asynchronous Actor-Critic (A3C), or Trust-Region Policy Optimization (TRPO).

Figure 2 shows the basic reinforcement learning structure that the other algorithms build upon. Actor-critic methods (Figure 3) implement generalized policy iteration and work toward policy evaluation and improvement. The actor aims towards improving the current policy, while the critic performs an evaluation of the policy. This is better than critic-only policies (e.g. Q-learning, which uses a state-action value function without a reliable guarantee of convergence or optimality of policy), or actor-only policies (e.g. REINFORCE algorithms which optimize cost over the policy parameter space, and while has a strong convergence, the estimated gradient has large variances).

In actor-critic algorithms, the critic approximates and updates a value function using samples. This is then used to update an actor's policy to improve performance, while ensuring convergence. The actor is responsible for generating a control

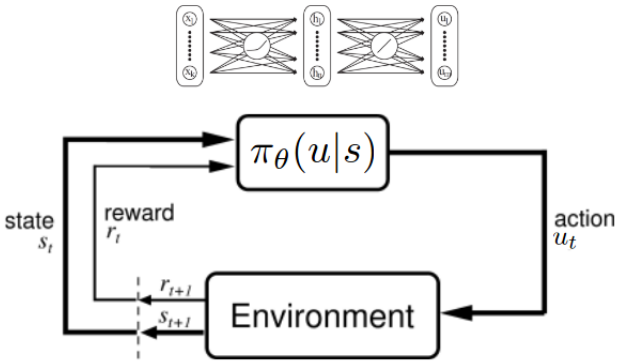

Fig. 2: Basic Reinforcement Learning approach to iterate on policies[2]

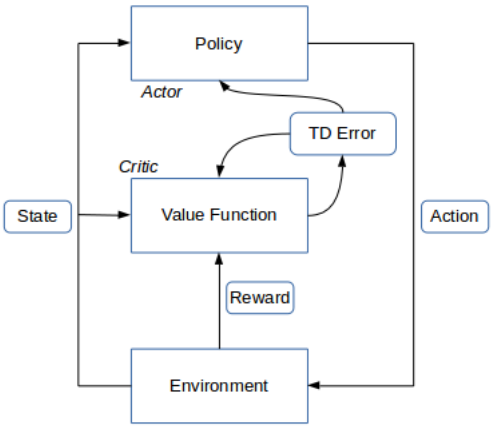

Fig. 3: Architecture of Actor-Critic

input given a current state, while the critic is responsible to evaluate the quality of the policy by modifying the value function estimate. The evolution of actor-critic methodology is illustrated in Figure 4.

Our implementation is based on David Silver's [9] paper, Continuous Control with Deep Reinforcement Learning. The implementation of Deep Q-Network (DQN) has been found to have high performance on Atari video games with pixel-input. However, while DQN works in high-dimensional observational space, it works decently only in low-dimensional action space as maximizing action-value function is iterative at every timestep and computationally expensive.

B. Software Tools

The software tools that we plan to use for this project are TensorFlow, Keras, and an environment simulator for the driving/racing game.

TensorFlow is an open source software library for numerical computation using data flow graphs. It was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research. Keras is a high-level neural networks API that runs on top of TensorFlow. It

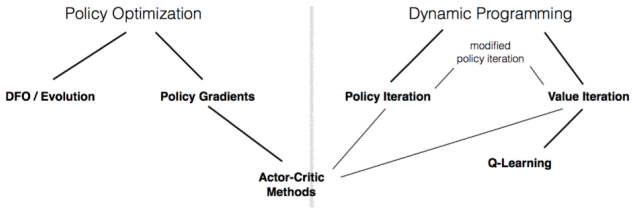

Fig. 4: Actor-Critic Methodologies[9]

is an user-friendly, modular, easily extensible API that can be worked on using Python.

The choice of environment is of vital importance for this project. OpenAI gym's driving environment which shows a simplistic top-down view of a track could be used as a test or even final simulator. Racing games such as TORCS (The Open Racing Car Simulator) or World Rally Championship which are 3D racing games with physics engines and Out-Run or Dusk-Drive which are the 2D versions of a racing game can be chosen due to ease of implementation. An additional selection would be GTA V which provides the most realistic simulator with other agents/vehicles as well as pedestrians, however combining and feeding in the outputs of an RL algorithm into GTA is complex.

C. Evaluation

There are multiple ways to evaluate the algorithm, depending on the exact environment of the game. If it's a racing game, the metric can be the total time taken by the agent to drive around the track or the average speed of the car throughout the lap, as well as whether the agent wins the first place. The metric for a driving game will be whether a lap could be successfully completed by the agent or not, without crashing into obstacles or performing any dangerous driving actions. Another possible metric would be training-testing losses, and accuracy during training the neural networks within the DRL algorithm.

The project was finally carried out on Out Run, the 1986 arcade game by SEGA, using the Deep Deterministic Policy Gradient (DDPG) algorithm [9]. We also implemented the Asynchronous Advantage Actor-Critic (A3C) algorithm for training on the CarRacing-v0 Box2D environment of OpenAI Gym. However, the Box2D environments were discovered to be not thread-safe, which prompted the use of other simulators like TORCS and CannonBall. CannonBall was chosen based off of the popularity and success of the game.

D. CannonBall OutRun

OutRun is a 3D video driving game released by SEGA in 1986. The objective of the game is to reach the destination before a set timer counts down to zero. CannonBall is an open source OutRun implementation that boasts 60 FPS gameplay, widescreen and high resolution sprite scaling along with other enhancements. Illustrated in Figure 5, it is a complete reimplementation of the original game, in C++.

Fig. 5: CannonBall OutRun- Algorithm input state

A reward function and the appropriate environment structure was implemented from scratch to enable the deep reinforcement algorithm to learn to play the game. At each step, unless the car was in one of the penalty states described below, a positive reward equal to the log speed of the car was awarded to the algorithm (the log speed results in a normalized reward based on the speed that doesn't outweigh any other terms in the reward policy). A penalty of -0.04 was given for states in which the car was on track but was not moving, -0.6 when the car went off-road (Figure 6) and -2 for crashes (Figure 7).

Fig. 6: The car wheels going off-road. Our DDPG algorithm outputs a penalty of -0.6 for this state.

E. Deep Deterministic Policy Gradient Algorithm

The Deep Deterministic Policy Gradient (DDPG) algorithm [9] is an off-policy, model-free technique that builds on the previous work on Deterministic Policy Gradients. It is a policy gradient algorithm that uses a stochastic behavior policy for exploration and estimates a deterministic target policy. DDPG uses two neural networks for the actor and the critic, predicting actions for the current state and generating temporal-difference error signals at each step as seen in Figures 3 and 4, the DDPG implementation uses an stochastic exploration policy (seen in Equation 1) to implement a deterministic target policy. Furthermore, the policy-gradient optimizes a policy end-to-end by using noisy estimates of the gradients of the expected

Fig. 7: A crash state on the simulator. The DDPG algorithm returns a penalty of -2.0 and resets the car during this state.

Algorithm 1 Deep Deterministic Policy Gradient Algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer *R*
for *episode=1:M* **do**
  Initialize a random process $\aleph$ for action exploration
  Receive initial observation state s_1
  for *t=1:T* **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \aleph_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in *R*
    Sample a random minibatch of *N* transitions (s_i, a_i, r_i, s_{i+1}) from *R*
    Set $y_i = r_i +' (s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{(s_i,\mu(s_i))} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{(s_i)}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end
end

reward in a policy to update the policy in the direction of the gradient, which in turn updates weights of the actor neural network.

$$dX_t = \theta(\mu - x_t) + (\sigma dW_t) \quad (1)$$

Equation 1 is the Ornstein-Uhlenbeck exploration process

(opposed to the Epsilon-Greedy approach) to obtain temporarily correlated exploration independent from the learning algorithm and tries to mimic a particle under Brownian motion (random motion W_t) with friction. In this case θ, μ, and σ are parameters of the DDPG algorithm, while W_t is obtained from the standard normal distribution.

One of the issues when using Neural Networks for DRL is that the policy assumes that samples are IID (Independent, Identically Distributed), however this is not so when samples are obtained sequentially. For this reason, DDPG uses replay buffers from DQN where transitions are sampled from the environment according to the policy, and the tuples of state, action, reward, next-state are stored in the replay buffer (which is periodically renewed). The actor and critic are updated by sampling from this replay buffer.

The input to the network is the current state RGB pixel values and the output is a set of Steering/Acceleration/Brake values. The steering can take the values left/right/none, and the acceleration and brake values are either on or off. The pseudocode for the DDPG algorithm from [9] is presented in Algorithm 1 for reference.

F. Experiments

The training was carried out in a virtual python environment that was created using Conda. Conda is an open source package management and environment management system that enables the user to easily run different versions of software in different virtual environments. The environment was first setup with all the tools required for running the algorithms - Python, TensorFlow, Keras, etc. The performance of the algorithm was evaluated by training it on the simulator interface for 8 hours, and the results are discussed in the next section.

IV. Results

A screen capture of the algorithm training on CannonBall OutRun is presented in Figure 8. Short videos of the preliminary training of the algorithm are also available on YouTube - here and here. After training for 500 episodes, the resulting policy outputs the video here on YouTube. During the initial stages of training, a random policy was much more effective at obtaining higher rewards than the DDPG algorithm.

Fig. 8: DDPG training-in-progress on CannonBall OutRun showing an output velocity of 114km/h

The algorithm was trained for nearly 1000 episodes on a single Intel i3 CPU. The rewards for each episode were plotted against the number of episodes and this plot is shown in Figure 9.

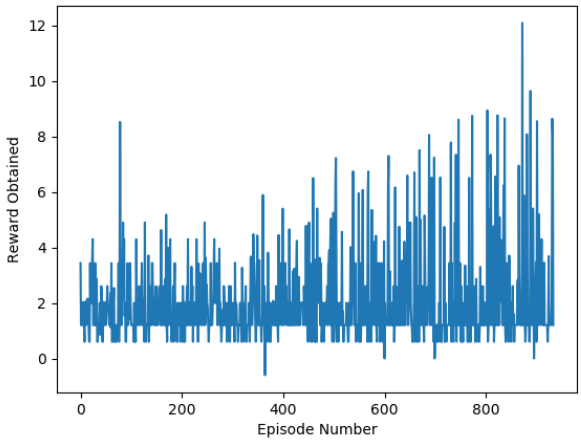

Fig. 9: Reward per episode over 1000 episodes

In the first few episodes of training (around 300 episodes), the rewards obtained by the algorithm are around a total of 2 on average. As the training progresses, and the number of episodes increases, it can be seen that the rewards accumulated by the algorithm at the end of each episode range from about 12 to 0, and the average reward increases to about 6 per episode. Due to time and computing power constraints, we could not train the algorithm for more than 1000 episodes.

Although there is no significantly apparent improvement in the rewards over 1000 episodes, there is a slight increase in the trend of the rewards which indicates that a longer training time over higher number of episodes may yield better results. Furthermore, the latest DDPG policy shows that the algorithm learns an interesting trend- at some sections of the track, the agent goes off-road to reach a different track in the game. This means that it accepts slight negative penalty obtained from the wheels going off-road at the current state, to be compensated by a much higher reward obtained from higher speeds in subsequent timesteps from simpler laps. This means the reward function needs to be more finely tuned to penalize tires off-road (e.g. number of time-steps off road multiplied by the number of tires off-road and a higher constant penalty).

Additionally, our DDPG implementation was done on a game without time limits or opposing traffic/racers. This was done as Cannonball Outrun only outputs crashes with the environment (and not opposing vehicles) as observations which means that our implemented reward function cannot consider these crashes leading to an incorrect policy. The issue with not having time limits in the game is two-edged; the game is open-ended with different track choices leading to different environments opposed to just racing around a similar lap which would mean that the algorithm needs to be continuously trained. However, if timings were imposed, the reward function needs to be better tuned such that the

car goes as fast, accounts for the time remaining, and also account for the possibility that the car needs to brake/reduce speed at certain sections of the road to avoid crashing in subsequent sections (and hence not be penalized for this foresight).

V. Conclusions

This project acted as the culmination of the knowledge acquired during the course CS 534, Artificial Intelligence, at WPI.

Although the algorithm was tested on a game simulation, the results strongly suggest that Deep Reinforcement Learning techniques are very much capable of competing with the current best end-to-end control systems for autonomous driving. The main bottlenecks with this algorithm were the proper choice of Actor, Critic networks given training states and testing observations/sensor inputs as well as reward functions. Additionally, the choice of algorithm can be better evaluated by comparing different algorithms such as Asynchronous Actor Critic or even Proximal Policy Optimization as well as random policies and evaluating the mean score or completion time that was obtained over multiple trials. In addition to this, for simplicity, our implementation assumed no traffic and an unlimited time trial- subsequent iterations of this project can have opposing, adversarial cars and time trials.

Future work can be dedicated to implementations dealing with more realistic simulators that do not allow a constrained action space. Additionally, real world implementations prove to be exponentially more difficult to execute and since the current work is only in simulations, the implementation of the algorithm on real world systems is a very important direction of research. An additional future work could be to use this DDPG algorithm that outputs actions for the trained policy at every timestep in conjunction with an end-to-end deep neural network that also takes in images as an input and extracts a single action. In such an algorithm, the action will only be performed if there is a consensus between the two, else no action is taken.

References

[1] Brian Buntz. Online-what are the five levels of autonomous driving?, 2016.

[2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[3] Etienne Perot, Maximilian Jaritz, Marin Toromanoff, and Raoul de Charette. End-to-end driving in a realistic racing game with deep reinforcement learning. *CVPRW*, 2017.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[5] Bojarski, Del Testa, Dworakowski, Firner, Flepp, Goyal, Jackel, Monfort, Muller, Zhang, Zhang, Zhao, and Ziebao. End to end learning for self-driving cars. *2017 IEEE Intelligent Vehicles Symposium*, 2017.

[6] El Sallab, Abdou, Perot, and Yogamani. Deep reinforcement learning framework for autonomous driving. *Neural Information Processing Systems*, 2016.

[7] Kaelbling, Littman, and Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 1996.

[8] Arulkumaran, Deisenroth, Brundage, and Bharath. A brief survey of deep reinforcement learning. *IEEE Signal Processing*, 2017.

[9] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *ICLR*, 2016.