AI Anarchy: Autonomous Racing Game Using Deep Q-Learning

Kendra Givens, Lorna Hedges, Makenna Owns, Jathan Lazala, Luwi Ogbaide

Project Objective

The objective of this project is to enhance a user's comprehension of reinforcement learning by allowing them to observe an agent learn to drive a car around a track in Unity3d in real time. This objective includes designing the track scene, implementing a drivable car, creating an interactive user interface, implementing deep Q-learning, incorporating sound, and using PlasticSCM source control.

Abstract

Almost all games using AI in Unity3d inherit from the Unity ML Agents class, complete with a pre-trained proximal policy neural network that the programmer can drop in (Juliani et al., 2021); an approach that sacrifices flexibility for ease. In contrast, our work is written entirely from scratch with a model that we have trained ourselves. It is not limited to using proximal policy such as the pretrained model, but can be adapted to any RL algorithm as long as the appropriate code is modified. Results show that while the agent is not entirely successful at completing the track, it is indeed capable of learning a rudimentary policy.

Introduction

Autonomous driving is quickly becoming a popular alternative to human driving as it has the potential to be safer, faster, and provide increased mobility (Fagnant and Kockelman, 2015). Self driving cars would eliminate distracted driving, have an increased reaction time, coordinate with other autonomous cars to get the fastest flow of traffic, and provide opportunities for people such as the legally blind, disabled, and elderly (Fagnant and Kockelman, 2015).

Deep Q-learning is a popular reinforcement learning algorithm used for autonomous driving. It takes in the agent's current state and predicts the next action to take. Eventually, the agent should begin to learn a policy that maximizes its reward (Mnih, 2015).

Background

Reinforcement learning is a class of machine learning methods that allows an agent to learn to maximize its reward function. Its knowledge is known as the policy which tells the agent which action to pick next in order to get the best reward. However, because the agent is tasked with getting the best possible reward, it might learn an action that results in a high reward, but is not entirely in line with the overall goal. In order to prevent this situation, we allow for exploration through random actions some probability of the time.

Deep Q-learning is a subset of these reinforcement learning algorithms that learns the best action to take to maximize its reward through a neural network. It takes in an agent's state and predicts a set of Q-values which represent the value of each action. Once the agent takes the highest action, the agent enters into a new state. The same process occurs where the agent takes an action. The previous state's value is then updated based on the value of the future state. A discount factor can be introduced to vary how much the agent takes into account the future states The algorithm can be summarized in this equation:

$$Q(s, A) = Q(s, A) + alpha[reward + gamma*max\_A(Q(s, A') - Q(s, A)]$$

During early training, the model's Q-value predictions should be random, but as more iterations occur, the model should begin to approximate the correct result of the equation resulting in more and more accurate predictions for the best action to take (Sutton & Barto, 2018).

Methods

We found a car and track prefab on the Unity Asset Store. The track came with a prebuilt mesh collider. We added a box collider and Rididbody component to the car to allow it to experience collisions and gravity.

We then wrote a car controller script which used wheel colliders and keyboard input to move the car, with the eventual goal to give control of this script over to the neural network to drive. A camera script was also created that adjusted to the car's position and reverses when the car drives backwards. Checkpoints were set around the entire track using empty game objects with box colliders on them.

Using the NuGet package manager in Visual Studio we configured the scripting environment to be compatible with Tensorflow.NET, Keras.NET, and Numpy.NET. However, NuGet is not compatible with Unity3d so we had to write a UDP server to connect the neural network to the rest of the application. The three layer neural network was implemented on the server's side, with a high level interface script handling byte conversion between Unity3d and the model. The Q-learning portion was done in its own agent script in Unity that connected to the car controller script and the high level server interface.

The state information for the car takes in its current speed, current turning angle, the distance to each object the car's raycasts are colliding with, and the distance to each checkpoint. The state information is passed into the neural network as its current state. The set of Q-values are then received which tells the car which action to take. In total, there are 9 actions the agent can take such as decelerate and don't turn, decelerate and turn left, decelerate and turn right, and so on for no acceleration and acceleration.

Additionally, the agent is rewarded continuously for moving by taking the absolute value of the square root of its velocity. The square root is taken to prevent possible large values from blowing up the model. If the agent passes a checkpoint, it gets an automatic addition of five to its reward. If it collides with a wall, the agent gets an automatic penalty of negative four, and the agent is also penalized for getting further from its next checkpoint by simply subtracting the total distance from the reward.

Once the action for its current state is taken, the agent enters into a new state and repeats the process. The values for both the current and previous state as well as the reward are used to calculate the Q-value equation. The result of which is used as the model's targets in training.

Additionally, all of the information exchanged from the model and the agent script is being converted into bytes. The entire byte packet passed into the server is summarized below:

[prepend byte, state length, Q-values length, state, Q-values]

The prepend byte tells the server whether the model should perform a fit or predict action. The length sections allow the server to split alone the correct index in order to get the training data (state) and the targets (Q-values).

We also implemented a countdown animation to signal to the AI at the start of the game to begin driving, this element is controlled through the use of a script that modified the animation, disabled car controls, and audio until the completion of the three seconds. A timer was implemented into the game through two scripts, one that managed the timer display through a series of conditional statements that ensured the time increments correctly, and another that handled the lap complete trigger to record the best time to the timer display.

The process of navigating throughout the game is accomplished with the help of UI, which provides a visually appealing and easy-to-use platform for players to interact with. When starting the game, regardless of any button selection, the user is presented with a countdown timer. After this timer is finished, the user or AI is able to play the game. Whenever the user wants to pause, they are able to with a simple keypress “P” on the keyboard. The pause menu will freeze all gameplay elements (movement, sounds, time). Finally, a “Quit to Menu” button will be visible on the screen. This button will seamlessly quit the game and return the user back to the main menu. Simplicity and effectiveness was our primary focus for these UIs. The final design of the main menu involved a simple logo with a button layout interface for user playing and AI controlling.

Our game is equipped with car sounds and background music. After the countdown, the user will hear the car start, and then an idle engine sound will play. As mentioned previously, if the user pauses the game, all audio will cease. But once the user resumes by pressing the “P” key again, the game sounds will continue where it left off. This is useful for an easy transition back into gameplay and helps prevent breaking immersion while within the game.

Both the finish line and starting line are designed from scratch. They were developed using three cube game objects. The three cubes that make up the finish line are green, and the three cubes that make up the start line are blue. Both of these lines were used as checkpoints in the AI training.

Results

For the most part, the car was not able to complete the track entirely, but it did learn how to avoid walls quite reliably. At one point, the car was able to pass the checkpoints and performed well. However, it did take a day of training and occasionally resetting the car back to its original position to accomplish. Interestingly, the car began to ride the wall in the proper direction. We hypothesize it is because the reward of getting closer to a checkpoint was more than the penalty of hitting the walls; so, it learned hugging the wall and accelerating was the easiest way to move forward and maximize its reward. A link to the successful run is in supplementary material.

Conclusion

Although the agent did not pass the goal of completing the entire track, it did show evidence of learning specifically in regards to avoiding walls. After a few training iterations, the agent was reliably able to back up from walls as soon as it got too close. It also learned that going forward toward the checkpoint resulted in a smaller penalty than driving backwards. This work shows it is possible to implement a learning agent in Unity3d without relying on their built in ML Agents class which opens up the doors for a much greater amount of flexibility and customization.

Further Work

In order for the car to perform better consistently, we believe more training time is needed. Also, episodic training where the car is reset every couple of iterations or when it collides with a wall instead of continuous training could help the agent to perform better (Mnih, 2015).

Another possible improvement could be adding a memory mechanism, eligibility traces, replay memory, or a recurrent architecture such as an RNN or GtrXL to allow the car to remember the past states and not overwrite what it has learned so far. (Sutton & Barto, 2018).

Trying out different reward functions could change the performance as well. It is currently being rewarded for movement in any direction and smaller distance to checkpoint, but accounting for other variables such as movement direction and turning stability could help as well.

Lastly, swapping out reinforcement learning algorithms could help too. Policy gradient is specifically designed for continuous action spaces and could possibly perform better than Q-learning in this case (Sutton & Barto, 2018).

References

Fagnant, Daniel J., and Kara Kockelman. “Preparing a Nation for Autonomous Vehicles: Opportunities, Barriers and Policy Recommendations.” Transportation Research Part A: Policy and Practice, vol. 77, July 2015, pp. 167–181, https://doi.org/10.1016/j.tra.2015.04.003.

Juliani, Arthur, et al. "Unity Machine Learning Agents Toolkit." GitHub, 2021, https://github.com/Unity-Technologies/ml-agents.

Mnih, Volodymyr, et al. “Human-Level Control through Deep Reinforcement Learning.” Nature, vol. 518, no. 7540, Feb. 2015, pp. 529–533, web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf, https://doi.org/10.1038/nature14236. Accessed 21 Mar. 2019.

Sutton, Richard S., and Andrew G. Barto. Reinforcement Learning: An Introduction. 2nd ed., MIT Press, 2018.

Supplementary material

Code: https://github.com/KendraGivens/Autonomous-Racing-with-Q-Learning

Late training with resetting to original position occasionally:

https://drive.google.com/file/d/1H-8_xKAzMcxwsz5zbRbSEiHNYJkLN09Y/view?usp=sharing