TD Learning to Find Car Control Policy Based on Local Information

Jameson Thies[1] , Kourosh Vali[1] , Yanda Chen[1]
{jsthies, kvali, ydachen} @ucdavis.edu

***Abstract*— In recent years, machine learning has become an increasingly common approach to solving problems conventional methods are unable to solve. Instead of a dataset, reinforcement learning uses strict problem formulation, including states and actions, to create an environment an agent can learn from. This paper presents a simplified model where an agent learns to drive a car around a track as many laps as it can in a predefined amount of time. In contrast to prior works, state is defined by the car's movement, and local information about how close the car is to a wall in multiple directions. *n*-step Temporal Difference (TD) learning is used by the agent to find a good policy in a limited number of episodes. Multiple values of *n* are tested, and the results are used to draw conclusions about which values of *n* are best for this specific problem.**

I. Introduction

With increases computational power and availability, has come in increase in the applications of machine learning. Machine learning can be broadly divided into three categories. (1) Supervised learning, which learns trends from labeled data. For example, training a neural network for digit recognition is a supervised learning task. A dataset of labeled digits is required, and based on the features of each digit a network can be trained for classification. (2) Unsupervised learning, which is used to learn trends in unlabeled data. For example, unsupervised learning could be used to roughly sort digits into ten groups, but cannot classify. (3) Reinforcement learning, which defines a system where an agent learns from its environment through a series of action, state transitions and rewards. For example, reinforcement learning can be used to find the optimal policy of traveling through a maze to get a reward at the end. In contrast to supervised learning and unsupervised learning, reinforcement learning has the unique advantage that no dataset is needed.

Clearly in autonomous driving a great deal of data about a cars surroundings and movement is captured, but something must control the driving. In this paper, we present a simplified car model represented as a reinforcement learning problem. The problem of learning a policy for controlling a simplified car model has been addressed in prior works. For instance, example 5.12 of *Reinforcement Learning: an Introduction* [1] uses a simplified car model where state is defined by the car's location and movement. The Monte Carlo algorithm is used to find a policy. Another paper [2] suggest defining state through extracted features of the graphical information associated with the environment. In contrast to the two examples, we propose defining state by car movement, and the distance to the nearest obstacle in a few directions. Section II describes the agent and environment of our reinforcement learning model. Section III continues with a brief description of our chosen learning method, *n*-step TD learning. Section IV details our testing and results. Lastly, Section V gives a few concluding remarks.

This paper offers two primary contributions. (1) We propose a new way of defining state in a car-track model which relies on only local information. Because the new state definition only uses local information, the agent performs well on new tracks where local information can be gathered using the same techniques even if the agent has not seen the track before. (2) We give a detailed and quantified description of how changing the step size affects the agent's ability to learn an effective policy in a limited number of episodes. By giving information comparing multiple step sizes in *n*-step TD learning, we show that larger step sizes increases the agent's ability to learn a good policy in limited episodes.

II. Agent and Environment

The two primary elements of problem formulation in reinforcement learning are the agent and the environment. It is the responsibility of the agent to make decisions, and to learn a policy by observing responses. In contrast, it is the responsibility of the environment to tell the agent the next state, and reward given the current state and the action taken by the agent. Using responses from the environment, the agent learns a value associated with each state-action pair. Using these value, a policy (π) can be determined.

This basic outline of an actor/environment model lays the foundation for our implementation. It all begins with a 2-dimensional grid of locations, similar to grid world or the track in Textbook example 5.12 [1]. Because we are using Python for this project and we want a graphical interface to show the simplified car model learn to navigate a track, we are using the Pygame Python library to create a visual representation of the agent and environment. The 640×400 pixel screen generated by Pygame naturally lends itself to a 2-dimensional grid environment. Within this two dimensional space, the track is defined by a list of lines segments.

The size of the gridded screen (640x400) leads to an issue in the modeling of this problem. If each state were to be defined by the location of the car as well as the multi-directional velocity as it is in example 5.12 of the Textbook [1], then the number of states can easily grow out of hand. In this example, the 640x400 pixel screen provides 256,000 possible locations. If there are also 2 velocity components with 5 possible values each, as there are in Example 5.12, then the number of states for a track of this size becomes

[1] Department of Electrical and Computer Engineering, University of California, Davis, CA, 95616

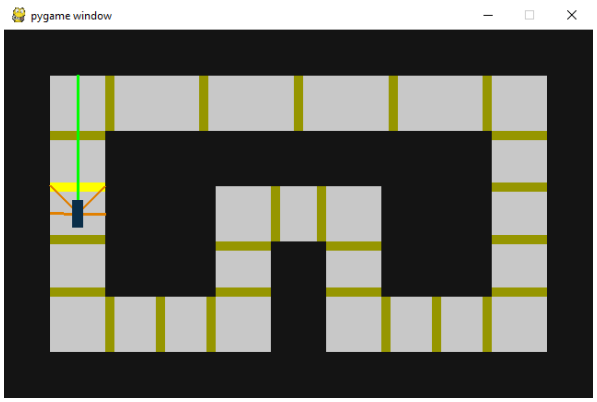

Fig. 1. Track 1 or training track. All learning by the agent is done on this track. The car is the blue box, lines protruding from the car are its sensors, and the color of the sensor indicates distance to the nearest track wall. Yellow boxes are checkpoints which return positive rewards. Only one checkpoint at a time is active, encouraging the car to go around the track. Note that the car is at its starting location and orientation.

Fig. 2. Track 2 or testing track. In addition to testing the performance of the agent on the track it learned from, the car is also tested on this track. By determining the performance of the agent on a track it was not trained on, we can draw conclusions about how generalizable the learned policy is. Note that the car is at its starting location and orientation.

6,400,000. Clearly using the car location to define state does not scale well. For this project, rather than define state by the location of the car, we will define state by the distance between the car and its surroundings. This is a more natural approach to this complex problem. People do not learn to drive by memorizing optimal actions based on their current location, instead they learn how to drive based on their general surroundings. They learn that if they are going fast, and there is something in front of them, then they should slow down. If there is something in front, and on the left, then they should turn right. By modeling the environment this way, two things are achieved. First, the state size is dramatically reduced. By limiting the resolution of the distance metric returned by the sensor, the number of states is limited. For example, if the state is partially defined by the distance to the nearest track barrier in 5 directions, and each distance falls into one of three categories (near, middle, far), then there are only 243 possible combinations of values returned by the sensors. This, in combination with two velocity values each between 0 and 4, results in 6,075 possible states. This method of representing the environment can lead to fewer states and faster training. Second, By defining the state by distance to the nearest obstacles, the training is conducted independent from the track. This not only allows the track be arbitrarily large without limiting the speed of training, but it also allows the policy to be used on a track which was not used for training. That is, the agent will be able to drive the car on a track it has never seen. We use the track in Figure 1 to define the environment that the car learns on. However, we test the performance of the agent by how well it an navigate both the track in Figure 1 and the track in Figure 2.

While there are certainly benefits for the proposed state representation, there are also drawbacks. There are multiple portions in the track the car could be which would return the same distances to the nearest track walls. It follows that the best action cannot be entirely determined by the state returned from the environment. While a learning algorithm will still converge to an optimal policy based on which action is most likely best, this leads to instability during training. Next, are formal definitions of the environment, agent, and system dynamics.

A. *Environment*

As mentioned earlier, the environment begins with a two dimensional 640x400 pixel grid space. The car starts at some initial location on the track defined by its position and orientation. Using the position and orientation of the car, the vertex and direction of each sensor can be found. There are sensors pointing out of the front, sides, and front diagonals from the car. Figure 1 shows the car, sensors and track rendered by Pygame. The distance returned by the sensor is quantized into one of three values: near (less than 25), middle (less than 50) and far (greater than 50). The distances returned by the five sensors constitute five of the seven dimensions of the state for this problem. The other two values which define state are the car's velocity and rotation. The velocity specifies whether the car is moving and can be two values: low if the car is still, and high if the car is moving forward. The rotation is the direction of change of the orientation and can be three values: left, right, and not turning. With 5 sensors, velocity and rotation defining a 7-dimensional state, there are 1,458 possible states. In our program, all elements of states are mapped to non-negative integers.

From the car's position and orientation, the location of the four corners of the car can be found. If any of the four corners are outside the track, then the environment returns the terminal state and a reward of -50. To give the agent an incentive to drive the car around the track, checkpoints are added. Each time a checkpoint is crossed by the car, a +10 reward is returned. The yellow bars in Figures 1 and 2 are checkpoints. Note that only one checkpoint is active at once. Passing one checkpoint disables it, and activates the next. These checkpoints loop, giving the car an incentive to complete as many laps around the track as possible. At

a point, the agent will learn to avoid collisions so to end episodes a maximum time of 8,000 steps is imposed. At this point, the terminal state is returned without a negative reward.

B. Agent

Given the environment, it is the agents responsibility to act. There are 2 actions with 3 choices each for a total of 9 possible action combinations. The actions control (1) acceleration and (2) steering. In the case of accelerations the three possible actions are slow down, maintain speed, and speed up. Because the velocity is either high or low, the value is saturated. For example, if velocity is high, then the increase speed action will not change the speed. Steering is similar. There are three possible actions steer left, maintain turn, steer right. Similar to the velocity portion of state, its limited range saturates the resulting rotation. A steer left action when the car is already rotating left is the same as a maintain steer action, as they will both result in rotation being left. Actions which change velocity or rotation will only have an effect when there is room in the state for a change. For example the car can only steer right if the car is either currently steering left, or not turning. This is similar to how a steering wheel prevents rotation after a certain angle of turning is reached. The combinations of these two actions are mapped to an integer between 0 and 8 in our program. In combination with the state, this gives an 8-dimensional $2 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 9$ matrix to represent the action-value function $Q(s, a)$ of our model.

C. System Dynamics

How is the next state determined from the current state and action? It all begins with an initial state. The velocity and rotation are initialized to low and not turning, respectively. The orientation, location and distance returned from the five sensors is dependent on the track and starting location. The agent then has the choice to take actions which change the velocity and rotation of the car. By changing the velocity and rotation of the car, the agent makes the car drive. If the agent sets velocity high by taking the accelerate action, then the cars location will move in the direction of its orientation every time step. Turning is a bit more complicated. If the action is taken to steer left, then the orientation (in degrees) is decremented every step until an action is taken to reverse the left steer. Conversely, if an action is taken to steer right, then the orientation is incremented every step until an action stops it. A rotation variable is used to keep track of the car's current turn. Zero is a turning left, one is not turning, and two is turning right. A steering action can be thought of as similar to turning a steering wheel; the car will continue to turn until the steering wheel is turned back. Because the car can only move forward, steering actions are used to turn the car. Given a combination of orientation, rotation,location, and velocity of the car, the next location, orientation and velocity are found with the following equations

$$o_{t+1} = o_t + r_t$$

$$x_{t+1} = x_t + v_t \times cos(o_t)$$

$$y_{t+1} = y_t + v_t \times sin(o_t)$$

where x, y, v, o, and r are the x coordinate of location, y coordinate f location, velocity, rotation and orientation of the car, respectively. In summary, the agent takes actions to change the cars velocity and velocity. Given the current velocity, rotation, orientation, location, and change in rotation and velocity, the next velocity, rotation, orientation, and location can be found. From this information, sensors find the nearest track walls to complete the new state. The agent then chooses to change the velocity or rotation based on the new state.

III. TD-Learning

The Monte Carlo method of reinforcement learning estimates value functions at the end of every episode. In contrast to Monte Carlo, Temporal Difference (TD) prediction uses the reward from an action, and the predicted value of the next state to update the predicted value of the state that the action was taken on. This process of using predicted values to update predicted values is known as bootstrapping.

TD prediction can be expanded to more closely resemble Monte Carlo. TD prediction as earlier described uses only the reward from a state and the predicted value of the next state to update the prediction of the current state. That is, only 1 step is taken before updating the value. If infinitive steps were taken, then the episode would always complete and the gain would be used to state value prediction. Therefore, when infinite steps are taken before updating, TD learning becomes Monte Carlo. In the space between 1-step TD and $\inf$-step TD (Monte Carlo) are *n*-step TD learning methods. In *n*-step TD learning methods, a state value estimate is updated *n* steps after it is visited. By increases the number of samples the update is based off of, the expected prediction is closer to the actual value. However, updating the state value long after it is visited significantly increases the number of terms required to find the new estimate and therefore increases the computational complexity. Clearly there is a trade-off between prediction accuracy and speed where Monte Carlo and one-step TD are at the extremes.

In this project we use *n*-step Sarsa to find $Q(s, a)$ where $Q(s, a)$ is an estimate of the true state-action value function $q(s, a)$. *n*-step Sarsa is an on-policy method which allows use to use TD-learning for prediction and control. First, state-action values in $Q(s, a)$ are randomly initialized. Then, the agent uses an ε-greedy policy with respect to $Q(s, a)$ to choose actions. As an episode progresses, the state-action value from n time steps ago is updated using the rewards over the last *n* steps. Because the policy is soft a random action is occasionally taken, ensuring exploration. By continually acting and receiving rewards, the agent learns a ε-greedy policy based on choosing which state-action pairs have the greatest expected return.

TABLE I

PERFORMANCE OF AGENT ON TRACK 1 (LEARNING TRACK)

Number of training Episodes	0	10	50	100	200	300	400	500	600	700	800	900	1000
$n = 1$	-35.31	-32.52	-31.30	-29.37	-23.06	-23.17	-22.88	-21.15	-21.91	-21.1	-21.71	-21.24	-21.45
$n = 4$	-35.99	-35.34	-19.90	8.07	56.83	83.14	105.81	146.16	172.03	167.68	171.25	166.82	185.94
$n = 16$	-38.24	-29.47	20.53	53.57	174.31	205.14	211.66	234.59	254.91	250.55	256.75	275.68	265.47
$n = 64$	-33.45	-15.25	210.92	392.02	446.15	479.38	476.08	449.00	420.57	415.91	513.84	466.17	522.76
$n = 256$	-34.04	-32.20	156.53	415.9	475.60	533.45	550.01	519.90	585.83	508.79	489.47	508.08	557.99
$n = 1024$	-37.93	-23.99	74.97	161.79	421.78	400.80	370.55	432.29	517.34	555.59	338.94	397.20	239.80

TABLE II

PERFORMANCE OF AGENT ON TRACK 2 (TESTING TRACK)

Number of training Episodes	0	10	50	100	200	300	400	500	600	700	800	900	1000
$n = 1$	-34.95	-31.87	-31.40	-29.79	-26.91	-26.78	-26.36	-25.08	-25.09	-24.20	-25.26	-24.33	-24.28
$n = 4$	-35.61	-34.75	-20.94	-8.00	50.35	78.55	100.68	145.99	166.56	167.60	182.92	172.95	177.77
$n = 16$	-37.77	-32.23	-8.68	27.63	141.22	175.83	169.56	193.91	230.55	222.37	223.98	230.6	224.49
$n = 64$	-36.68	-17.62	173.81	353.22	406.82	442.98	445.34	420.59	398.77	376.1	462.85	428.34	489.06
$n = 256$	-34.38	-34.04	109.67	384.48	457.51	478.57	488.20	453.76	521.20	426.18	423.46	419.29	481.10
$n = 1024$	-38.68	-29.53	52.28	126.32	373.53	366.25	311.6	409.17	461.99	475.91	335.89	310.58	220.52

IV. RESULTS AND ANALYSIS

To test the agent we differentiated between normal and testing episodes. In a normal episode of on-policy TD-Learning, the agent will use an ε-greedy policy with respect to the current action-value function to choose actions, then update the action-value function based on the rewards from the environment. A testing episode is similar to a normal episode in that it has the agent use the ε-greedy policy with respect to the action-value function to make decisions, however instead of using the rewards to update the action-value function, the action-value function is not updated and the sum of the rewards is returned as a metric of the policies performance. This method of freezing the action-value function, then running multiple test episodes allows us to see how effective each learned policy is.

A primary goal of this project was to see how the size of n in n-step TD-Learning influences how quickly the agent learns effective driving policies. We setup an agent with $\alpha = 0.01, \varepsilon = 0.01$, and $\gamma = 1$ to learn a policy using n-step TD-Learning. We allowed the agent to learn over 1000 episodes using the environment corresponding to Track 1 (Fig. 1). At episode 0, 10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 we stopped the normal TD-learning episodes and ran 100 test episodes on both tracks to see how effectively the agent had learned a policy for navigating the track. This was done ten times for $n \in$ [1,4,16,64,256,1024] and the results are averaged and given in Tables I and II. Training for 6 values of n 10 times with 1,000 training episodes and 2,600 test episodes each requires 216,000 total episodes. On a Dell Inspiron 15 laptop with an Intel i3-5005U CPU and 6 GB of RAM, running all episodes took 78.97 hours.

Tables I and II give two major takeaways. First, larger values of n help the agent to learn in fewer episodes. In one-step TD learning, the agent is largely incapable of learning to navigate the track in 1000 episodes. At best, the model with $n = 1$ learns to pass 3 checkpoints before hitting a wall. On the other hand, large values of n allow the agent to learn how to navigate the track rather quickly. When n is 64 or greater, the car is able to effectively navigate the track after 200 training episodes. An average performance over 400 means that either 40 checkpoints were passed and there was not a collision, or 45 checkpoints were passed and there was a collision. Second, the agent is able to learn a policy for how to control the car independent from the track. Performance on Track 2 increases along with performance on Track 1 although the agent has effectively never seen Track 2. This is because similar features in both tracks. From Track 1, the agent learns to navigate left turns, right turns and straightaways. The agent can then use the learned policy on any track involving those elements.

V. CONCLUSION

In this paper we expanded on a previous question in reinforcement learning. By re-framing how an agent defines state in a simplified car-track model, we created an environment in which an agent can learn a policy to control the car based only on its immediate surroundings. Because only local information is needed for the agent to chose an action, it is able to perform well on tracks it has never seen. We then test how changing the step size in n-step TD learning affects the agent's ability to learn an effective policy in a limited number of episodes. We conclude that a larger step size allows the agent to learn an effective policy in a few hundred episodes.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: an Introduction*. The MIT Press, 2018.

[2] J. S. Jan Koutnik and F. Gomez, "Evolving deep unsupervised convolutional networks for vision-based reinforcement learning," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '14, 2014, pp. 541–548.