Use Reinforcement Learning to Teach a TurtleBot to Drive in Circles (in a Simulator)

Mayur Joseph Bency *
mbency@eng.ucsd.edu

July 27, 2018

1 Introduction

Reinforcement Learning (RL) is a class of problems designed to teach an agent through trial-and-error interactions with a dynamic environment [3]. Although many kinds of approaches exist for learning this optimal behavior, current literature relies heavily on statistical techniques and dynamic programming methods to estimate the utility of taking actions in states of the world. The rest of this section establishes notation and a brief summary of the RL model.

1.1 The Reinforcement Learning Model

In the standard RL model, an agent is connected to its environment via perception and action. The environment is presumed to have a way for communicating its current state at time-step t, s_t. Each step of the agent results in the agent interacting with the environment by taking action a_{t+1}. The action changes the state of the environment to s_{t+1}, and the value of this state transition is communicated to the agent through a scalar "reward" signal, R_t. The objective of the agent is to find a behavior, or a "policy", that chooses actions which tend to increase the long-run sum of values of the reward signal. This can be done by systematic trial and error, guided by a wide variety of algorithms, one of which is used to solve the problem presented here.

The most important difference between RL and the more widely studied problem of supervised learning is the lack of explicit teaching signals for given input states. Instead, after choosing an action the agent receives an immediate reward in that time-step and the value of the next state, but is not told which action would have been in its best long-term interests. As will be discussed in more detail in the coming sections, the agent's ability to learn an optimal policy often directly depends on its ability to explore the possible system states, actions, transitions and rewards.

1.2 Reward Shaping

A significant challenge in RL problems is crafting the right reward function to get the desired behavior. Since the agent can only follow a trail of scalar values and try to maximize that over a sequence of states, poorly designed reward functions can often lead to unintended behaviors or even trivial solutions. To put it in succinct terms: you get what you incentivize, not what you intend. A few of the common challenges that come up when designing reward functions are:

- Rewards are too sparse. Sparse rewards indicate that the agent doesn't receive feedback very often. This makes it difficult for the agent to identify valuable states to be in and may thus end up exploring vast amounts of the state-space before converging to a reasonable policy.
- Rewards are too dense. This is simply the other end of the sparse rewards problem. If the agent receives too much reward too often, it may be falsely led into believing that it is close to an optimal policy when it really isn't. Once again, may lead to exploring unnecessary amounts of the state space before the agent learns the desired behavior.

*Submitted as a test assignment for OffWorld Reinforcement Learning and Robotics

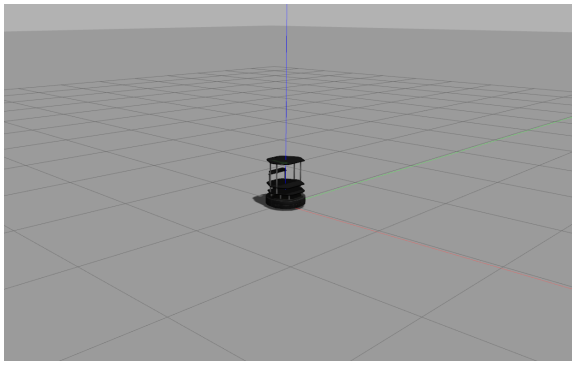


Figure 1: The environment used in this assignment with the Turtlebot present in it. Note that the current position of the robot is at the origin.

- Selecting the numeric structure of the reward. Positive rewards indicate that the agent will be wanting to accumulate as much reward as possible. On the other hand, negative rewards incentivize reaching a terminal state as quickly as possible to avoid accumulating penalties.

As brought up in the previous subsection, an unsupervised approach to learning such as RL is supposed to avoid having to deal with the teacher-critic dynamic that is hard to produce, either through obtaining datasets or by hand-engineering explicit targets. However, reward shaping might lead one to believe that RL continues to have that problem. There is a less circular way (pun fully intended) to solve the problem: that is, to infer the best reward function.

Abbeel and Ng [1] proposed an approach for apprenticeship learning through inverse reinforcement learning where the expert is trying (without necessarily succeeding) to optimize an unknown reward function that can be expressed as a linear combination of known "features". Inverse Reinforcement Learning, while stepping over the problem of designing a reward, generally cannot guarantee a correct recovery of the expert's true reward function and is thus significantly harder to train. For this reason and others, this work does not explore Inverse Reinforcement Learning options.

The goal of this assignment is to get a robot to move around perpetually in a circle. Since this is to be solved using an RL model, the behavior required for driving around in a circle can only be inferred through state transitions and obtained rewards. In other words, the problem here is to design a reward function that will incentivize the robot to move around in circles.

2 Setting up the Environment

The agent used in this problem is a Turtlebot [1] simulator that operates in an environment hosted by the Gazebo physics engine. OpenAI Gym is a Python library developed to provide a framework for designing environments that can record states and take actions. For this problem, gym-gazebo [2], an extension that works with ROS to bring Gym's RL framework to Gazebo. Figure 1 shows a snapshot of the environment with the Turtlebot agent, simulated in Gazebo.

Refer `/gym-gazebo/gym_gazebo/envs/turtlebot/gazebo_empty_turtlebot_lidar_nn.py` in the provided Docker image for the python script that is used to generate this custom gym environment. Note that the gym environment this is based off of (`gazebo_circuit2_turtlebot_lidar_nn.py`) measures its state by using LIDAR information, while the new environment uses additional custom methods to sense its state, as described next.

2.1 States and Actions

To formulate this problem as an RL model, the state transition model needs to be defined first. The states are chosen as the cartesian coordinates of the base of the robot in the world frame. That is, each state s is represented as $\{x, y\}$. Note that the the state space belongs to the continuous domain here. To get the location of the agent in real time, a ROS service request to

[1] https://www.turtlebot.com/
[2] https://github.com/erlerobot/gym-gazebo

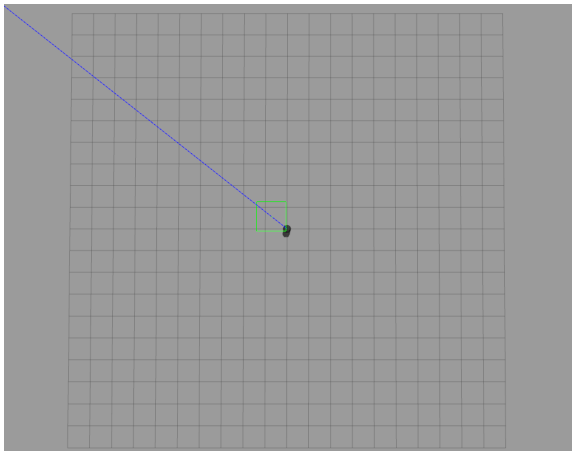


Figure 2: A birds-eye view of the environment. The grid represents the visitable area of the world. When the robot attempts to cross over the edge, the environment is reset and the robot is placed at origin.

`/gazebo/get_model_state` is made that returns the position and orientation of the desired model in the environment.

The agent is allowed to move through linear velocity and angular velocity of its base. To simplify the problem, two assumptions for the action space are made:

- Linear velocity of the agent is fixed at a constant value. This forces the agent to be always be in motion and thus allows us to maintain a simple reward function that doesn't have to worry about trivial solutions.
- Angular velocity is chosen as the action variable, a. Further, angular velocity is discretized into 21 uniformly spaced values ranging from -1 to 1 radians/sec.

In summary, the state space is a 2D dimensional vector representing the X and Y coordinates of the base of the robot in the environment while the action space is a scalar range representing the angular velocity of the base.

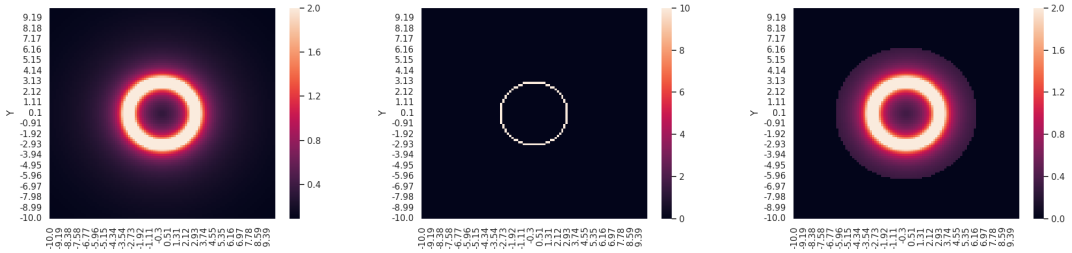
To ensure that the agent has enough time to explore its state space during episodes, a generous step limit of 5000 steps per episode is set, after which the episode terminates. The episode also terminates when the agent attempts to go beyond the boundaries of the workspace, which is set as [-10, 10] for both axes (see Figure 2). This workspace limit is set so that the agent doesn't spend large amounts of time in its initial phase exploring far away states that have little values for the task at hand.

2.2 Reward Function Design

Keeping in mind the pitfalls of reward shaping described in 1.2, an iterative procedure for developing an appropriate reward function for moving around in a circle was developed. The radius of the circle was fixed at $r = 3$ units, centered at the origin. A clear and obvious intuition is to assign high rewards if current state $s = x, y$ lies on the circle and low rewards otherwise.

A point $P(x, y)$ lies on a circle with radius r centered around the origin if it satisfies the condition $x^2 + y^2 = r^2$. Since in our environment the state s of the agent is simply the cartesian coordinates of the agent in the world frame, let $\hat{r}$ represent the distance of s from the origin. The goal then of the agent should be to be in states such that error $e = |r - \hat{r}|$ is close to zero. The constant linear velocity of the robot ensures that the agent never "cheats" by staying in one spot and maximizing its return that way.

One way to design a reward function that fits this criteria is to make the reward of being in state $s = x, y$ be $R(s) = 1/e$. To make the error function continuous, a maximum reward cap can be installed. Figure 3 shows three different versions of the reward function with different caps. After experimenting with all three variants. 3c was selected as the final reward function due to observed faster convergence.



(a) Maximum reward is capped at 2

(b) Maximum reward is capped at 10. Note the different scales.

(c) Maximum reward capped at 2 while reward below 0.3 is set to 0.

Figure 3: Different variations of the reward function design strategy discussed in 2.2. Setting the caps at different values can cause significant differences in episodic returns and thus, affect convergence.

3 Training the Agent

This section deals with the model used to train an RL agent to drive in circles. Since the state space here is continuous and the action space is discrete, tabular Q-learning methods to learn the state-action values cannot be used. Instead, a neural network based function approximation algorithm known as Deep Q Network (DQN) was used to train the agent.

3.1 Deep Q Network

Deep Q Networks (DQN) is a deep learning model that was shown to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning [4]. This was the first work that proposed to alleviate the problems of correlated data and non-stationary distributions that normally cause problems for deep learning models by using an experience replay mechanism which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors. Refer [4] for a detailed theoretical explanation of how the value-function estimation happens.

The DQN model takes in the state and returns the action-state value function estimates for all actions available for that state. For our problem, since the state space is two dimensional and the action space is discretized into 21 values, there are 2 input nodes and 21 output nodes. The model is designed to contain 2 hidden layers having 100 units each, both followed by ReLU non-linear activation functions. As described in the paper, a replay buffer that randomly samples experiences is used to generate batches of 64 experiences for training the model. Learning rate is set at 0.01 with default initialization schemes for weights and biases. Adam optimizer, a popular extension to stochastic gradient descent, is used for fast convergence.

3.2 Exploration versus Exploitation

The concept of exploration versus exploitation is important for successfully training policies for agents. Without sufficient exploration of the state-action space, an agent may never come across state-action pairs that contribute the most to the episodic return and therefore take an unnecessarily long term to converge to a solution.

DQN is an off-policy algorithm: it learns about the greedy strategy $a = max_a Q(s, a; \theta)$, ($\theta$ represents the parameters of the neural network) while following a behavior distribution that ensures adequate exploration of the state space. In practice, the behavior distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with probability $1 - \epsilon$ and selects a random action with probability ϵ. Figure 4 shows the ϵ-greedy strategy used here. Note that a higher ϵ in the beginning forces the agent to rely more on random explorations to cover more of the state-action space and as the network starts learning from the buffer of past experiences, the agent gradually relies more on the network's value estimates to take an action.

Apart from the ϵ-greedy strategy, another exploration strategy was also employed as an experiment.

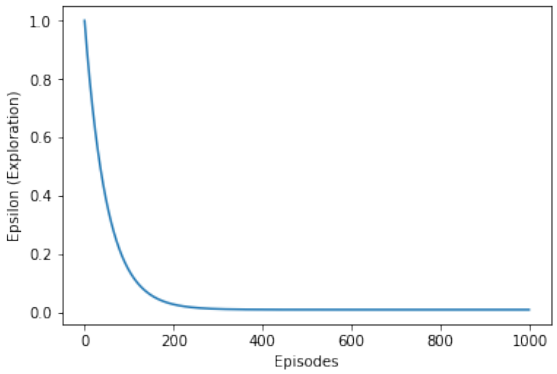


Figure 4: ϵ profile getting annealed as a function of episode.

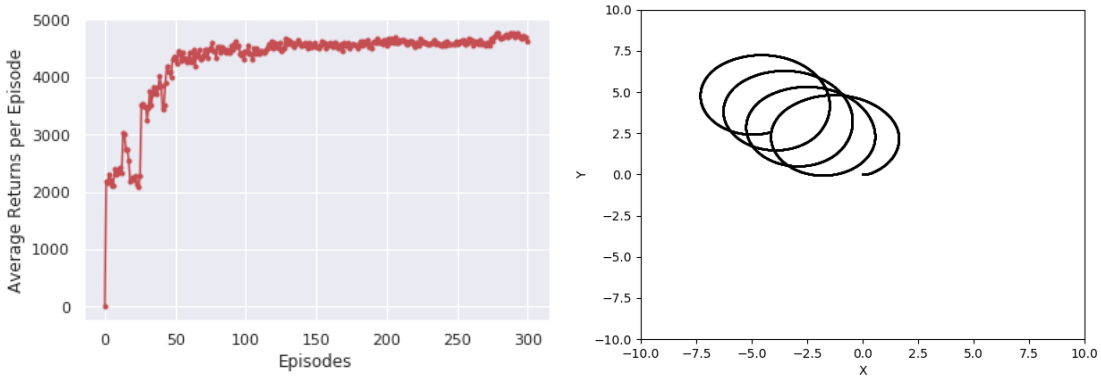


Figure 5: Training to drive around in circles when exploration is set to have directional bias. (a) shows the returns per episode of the training process, (b) shows the trajectory taken by the robot after the model has reached convergence. Note that the motions are actually circular, the warped aspect ratio makes it look more like an ellipse.

3.2.1 Exploration with Directional Bias

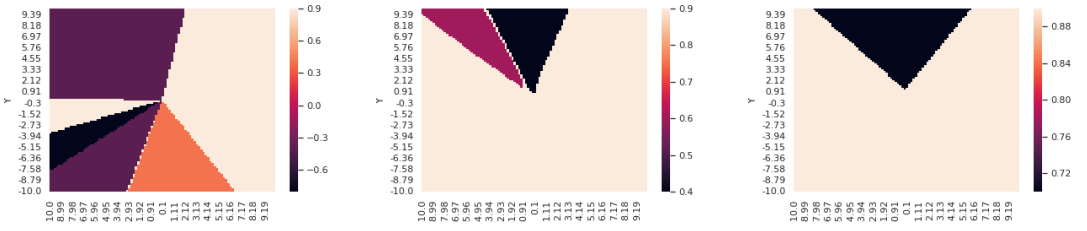
Directional Bias as used here is defined as sampling the action (i.e, the angular) in only one direction when the ϵ for ϵ-greedy kicks in. The idea being that if the agent is forced to pick one direction for an entire episode and train on these associated episodes as well, it's more likely the agent will complete a circle in that episode and notice the increased return. Since the angular velocity is discretized between [-1, 1] rad/sec, directional bias would dictate angular velocity be sampled between [0, 1] rad/sec.

This results in faster convergence to a circular motion, as evidenced by the average episodic return plot in Figure 5. At the same time, the policy learned by the DQN, while being able to generate smooth uniform circular motions, also seems to have a drift associated with it. Figure 6c may have an explanation to that. While most of the state seems to have learned to maintain a constant unidirectional angular velocity (essential for uniform circular motion), the black patch represents a lower angular velocity that distorts uniform circular motion and generates a drift whenever the agent enters that region of the state space.

The law of unintended consequences catches up with everyone eventually.

3.2.2 Exploration without Directional Bias

As opposed to 3.2.1, no directional bias is simply running the ϵ-greedy strategy without biasing the ϵ generated actions in any way. Refer 7 for average returns during the training process and the final policy learned by the network. Note the difference in convergence speeds between the two approaches. A uniform circular trajectory is generated almost perfectly in this case, with no



(a) (b) (c)

Figure 6: (a), (b), and (c) are representations of the policies learned by the network after 1, 10, and 100 epochs (near convergence) respectively. The scalar values are the angular velocities learned for a uniformly sampled distribution of states in the environment. A positive value means motion is directed towards the left.

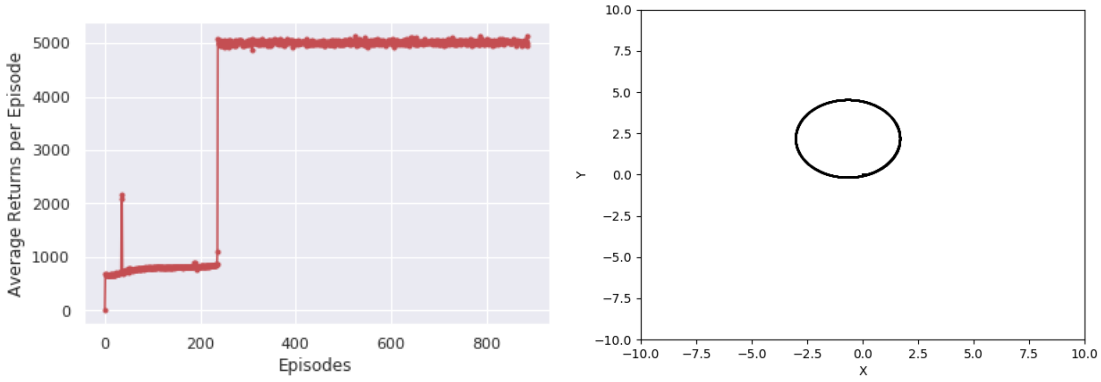


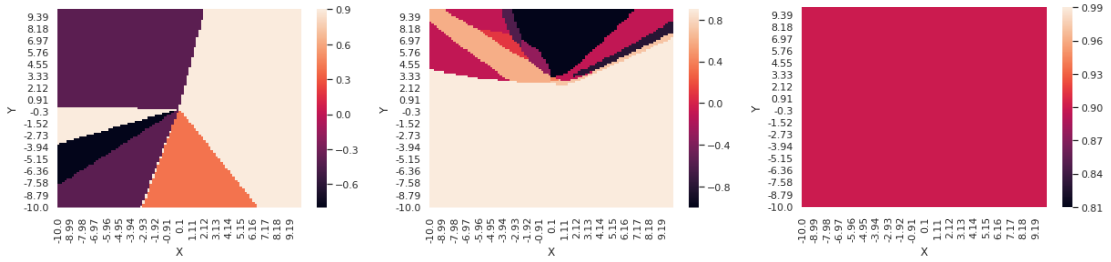
Figure 7: Training to drive around in circles when exploration is set to sample uniformly with no bias. (a) shows the returns per episode of the training process, (b) shows the trajectory taken by the robot after the model has reached convergence. Note that the motions are actually circular, the warped aspect ratio makes it look more like an ellipse.

drift as observed in the previous case. However, the circle seems to have shifted its center from the desired origin to a location slightly offset. We can speculate that a more robust and carefully crafted exploration strategy might make the agent realize that there are even higher rewards to obtain.

Figure 8c confirms our findings. The policy learned by the final fully trained model simply tells the agent to maintain a constant angular velocity no matter what state it is in for uniform circular motion. This also shows the change in center, since the agent begins at the origin and moves with constant angular velocity thereafter.

4 Concluding Remarks

Designing a good reward function is no walk in the park, even for a problem as deceptively simple as this. Optimizing reward for intended behavior can take a fair amount of fine-tuning and perhaps even generous amounts of sheer luck. Of course, it does take some of the sheen off of RL if tiresome amounts of hand engineering have to go into it, and that is one reason the literature tends to focus on ways to improve learning speeds for systems having really sparse rewards. For example, in our case, if reward were to be given only to a narrow band of states around the circle and zero otherwise, it might take a naive DQN forever to converge to a solution, but something a little more sophisticated like Hindsight Experience Replay [2] might be able to perform without having to resort to complicated reward engineering.



(a) (b) (c)

Figure 8: (a), (b), and (c) are representations of the policies learned by the network after 1, 100, and 200 epochs (near convergence) respectively. The scalar values are the angular velocities learned for a uniformly sampled distribution of states in the environment. A positive value means motion is directed towards the left.

References

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 1–, New York, NY, USA, 2004. ACM. ISBN 1-58113-838-5. doi: 10.1145/1015330.1015430. URL http://doi.acm.org/10.1145/1015330.1015430.

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[3] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.