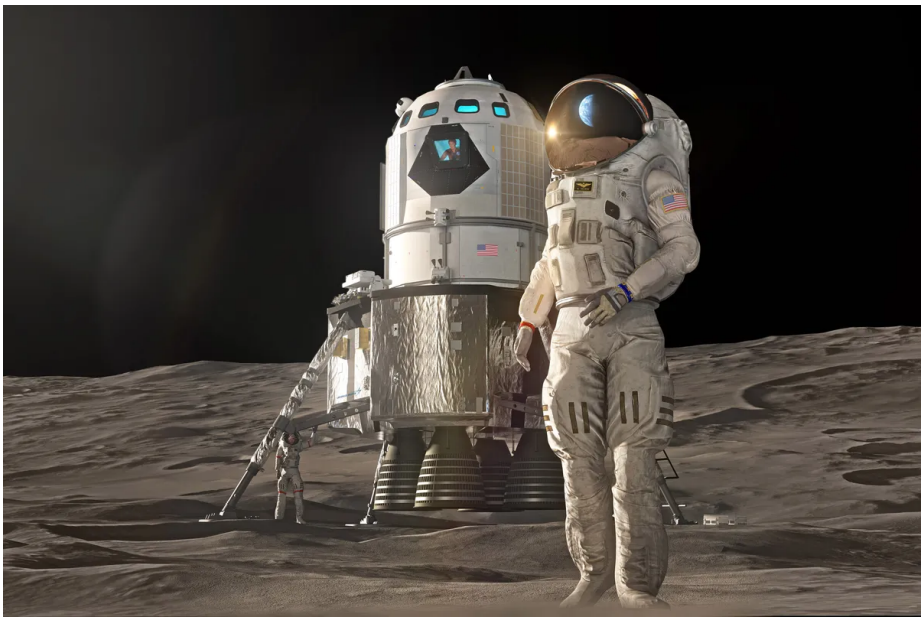
Beyond the Launchpad: Investigating the Benefits and Limitations of Curriculum Learning in Lunar Lander

Kevin Ortiz, Brandon Wilson
CS 138 Final Project

May 12, 2023



Abstract

This paper presents an investigation into curriculum learning techniques applied to reinforcement learning algorithms, namely Q-learning, SARSA, and n-step SARSA. The study utilizes the Lunar Lander environment from the OpenAI Gymnasium environment suite as a testbed for evaluating the effectiveness of curriculum approaches for enhancing the learning performance of these methods. The curriculum learning approach involves introducing gradually more complex tasks to the learning agent allowing it to progressively gain knowledge of the environment and to perform better on more complex tasks. In this study we designed a

curriculum based on external factors acting on the Lunar Lander in the environment ranging from simple initial stages to more challenging ones. The setup compares the performance of Q-learning, SARSA and n-step SARSA with and without using curriculum learning. Multiple trials are tested measuring the agent's ability to successfully land the Lunar Lander safely using a fixed number of episodes. The results demonstrate that the integration of a curriculum in these algorithms increased the agent's learning efficiency and overall learning. By leveraging curriculum learning in these algorithms, the agents were able to achieve more successful landings in the number of episodes than the baseline algorithms in a non-curriculum setting. These findings highlight the potential curriculum learning has on improving existing RL algorithms by efficiently using the training process to gain quicker learning.

1 Introduction

The Lunar Lander is a classic reinforcement learning problem that tasks an agent with landing a spacecraft safely on the moon within a given landing area. This environment provides an interesting and useful testbed for training and testing various reinforcement learning algorithms. One useful approach for performance comparison is curriculum learning, which trains an agent on incrementally increasing complexities. This environment lends itself well to this approach since there are several factors that can be incrementally adjusted to increase the external factors acting on the agent as it tries to land on the moon, such as wind and turbulence. In this project we explore this approach using the OpenAI Gymnasium environment package and compare curriculum learning verus non-curriculum learning over three separate RL algorithms; Q-learning, SARSA and n-step SARSA. By analyzing the benefits and limitations of this approach we are able to gain a better understanding of solving complex tasks within an environment using a reinforcement learning framework.

Reinforcement learning has already revolutionized the way we approach complex tasks, from real-world application such as autonomous driving and robotics to game playing. However, there remains a lot of work to be done and questions to be answered in this domain. Developing effective reinforcement learning methods remains a challenge for several reasons. Tasks often involve large state spaces and an incredible number of possible actions, which poses challenges to the exploration and learning of optimal policies. A lot of trial and error is involved in reinforcement learning and this can lead to unrealistic computational costs. Many real-world problems are so complex and dynamic that environments are difficult to model accurately, making it challenging to develop an optimal policy. These are just a few of the reasons for the need of innovative approaches to reinforcement learning that can accelerate learning rates and improve the quality of optimal policies. [4]

2 Background and Related Work

Curriculum learning is a training strategy commonly employed in RL to accelerate the learning process and improve performance. This method involves presenting an agent with a series of gradually increasing complex tasks or environments to facilitate a more effective learning process. By starting with simpler tasks and gradually progressing to more challenging tasks, the agent can build up its skills and knowledge progressively. When applied to RL, curriculum learning has been shown to help agents learn faster and achieve better overall performance. The idea is inspired by how humans learn, where we often start with simple and well-structured tasks before moving on to the more complex tasks.

In the context of the Lunar Lander environment, curriculum learning can be used to train RL agents to successfully land a spacecraft on the moon's surface. The Lunar Lander is a popular RL benchmark problem where an agent controls the thrusters of a spacecraft to navigate and land it safely on a designated landing pad. The agent receives sparse rewards based on the successful landing and has to learn a policy that maximizes the cumulative rewards. To incorporate curriculum learning in the Lunar Lander environment a curriculum can be designed to expose the agent to a sequence of increasingly difficult factors. In the initial stages, the agent can be trained with simpler environments where there is no wind and no turbulence. As the agent's performance improves, the difficulty of the environment can be gradually increased by introducing wind, introducing turbulence, and increasing the wind and turbulence strengths.

By gradually increasing the complexity of the environment, the agent can learn to tackle more difficult landing scenarios in windier and more turbulent environments by building upon the skills and knowledge acquired in the earlier stages. The curriculum provides a structured learning path that allows the agent to explore and exploit the environment more effectively, leading to improved learning efficiency and better final performance which results in more successful landings within a series of episodes.

Several research studies have explored the application of curriculum learning in RL. [1] [3] In the context of curriculum learning, researchers have proposed using reward shaping techniques to provide auxiliary rewards that encourage the agent to exhibit desired behaviors or achieve subgoals. By shaping the rewards, the agent can focus on specific aspects of the task, gradually learning more complex tasks. [2] [5]

These approaches are commonly used in RL research to improve learning efficiency and promote exploration in challenging tasks like Lunar Lander. By providing intermediate rewards that provide feedback on the agent's progress, the curriculum can guide the learning process, allowing the agent to build up its skills and knowledge in a step-by-step manner.

3 Experimental Domain

In this section we describe in detail the Lunar Lander environment and how we modified it to be suitable for evaluating our RL methods and implemented curriculum learning techniques. We discuss the algorithms and methods used to evaluate performance and the associated reward functions.

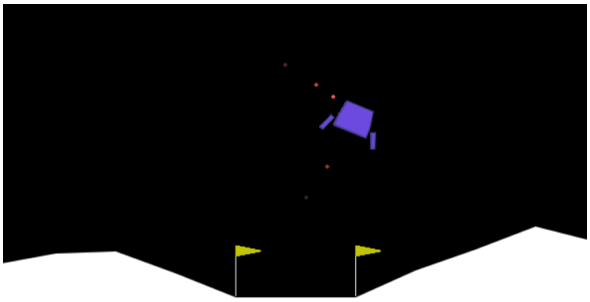


Figure 1: Still image of the Lunar Lander environment

3.1 Lunar Lander Environment

The Lunar Lander environment serves as a valuable testbed for exploring RL algorithms and comparing curriculum vs. non-curriculum approaches. The environment is challenging because it requires an agent to land a spacecraft on the moon's surface taking into account gravity, thrust, wind and turbulence. The task involves precise control, balancing forces, and avoiding crashes or sub-optimal landings. It also proves challenging for an agent to learn due to the sparse rewards, meaning the agent only receives significant feedback when it successfully lands safely. Sparse rewards can prove to be challenging for agents as they need to explore and learn from a large number of trial-and-error episodes to find optimal landing strategies. The environment also offers a natural gradient of difficulty making it somewhat straight forward to implement a curriculum. The wind can be turned on or off and the strength of the wind and turbulence can also be adjusted under the following parameters:

- Enable_wind = [True, False]. When wind is set to True, it is varied over time (t) by the function;

$$\tanh(\sin(2k(t+C)) + \sin(\pi k(t+C))) \tag{1}$$

  where k = 0.01 and C is chosen randomly from -9999 and 9999.

- Wind_power between 0.0 and 20.0. This dictates the maximum magnitude of linear wind applied to the spacecraft.

- Turbulence_power between 0.0 and 2.0 and dictates the maximum magnitude of rotational wind applied to the spacecraft.

The environment has either a continuous or discrete environment. In the continuous environment, the agent's actions can take any value between a set interval whereas in the discrete environment the agent's actions are binary, they either occur or do not.

As shown below, there are four discrete actions available:

- 0: do nothing
- 1: fire left orientation engine
- 2: fire main engine
- 3: fire right orientation engine

When continuous actions are enabled ('continuous=True' is passed to the environment variable), the action space becomes a Box space with a range of [+1, -1] and a shape of (2,) using a data type 'np.float32'. The first coordinate of the action represents the throttle of the main engine, while the second coordinate represents the throttle of the lateral boosters. For a given action represented as an np.array([main, lateral]), the behavior is as follows:

- If $main < 0$, the main engine is off.
- If $0 <= main <= 1$, the main engine's throttle scales linearly from 50% to 100% power. The main engine does not work with less than 50% power.
- If $-0.5 < lateral < 0.5$, the lateral boosters do not fire at all.
- If $lateral < -0.5$, the left booster fires.
- If $lateral > 0.5$, the right booster fires.
- The throttle of the lateral boosters scales linearly from 50% to 100% power between the ranges of -1 to -0.5 and 0.5 to 1 respectively.

We chose to explore the discrete environment for it's simplicity since discrete spaces are easier to work with and specific states are more likely to be visited more than once as well as the clearer interpertability of the discrete space.

Each episode starts with our lander at the top center of the simulated environment. At each time step, our agent will select one action and record what state it ends up in. Then the process repeats from the current state until termination. An episode terminates if one of the 3 conditions below are met:

1. The lander crashes (the lander body gets in contact with the moon)
2. The lander gets outside the environment (x coordinate is greater than 1)
3. The lander is not awake, meaning it is not moving or colliding with anything else (lands)

To define the state set, our environment uses a tuple-8 vector as such: [x-position, y-position, x-velocity, y-velocity, angle, angular-velocity, left-leg-down, right-leg-down].

All these features take on values that describe where our lander is and its orientation. Except the last two; left-leg-down and right-leg-down are Boolean values that determine whether the lander's respective legs are touching the ground, this plays into our reward. Below are the range of values that our features can take. These ranges are set by default and cannot be changed.

- x-position: -90 - 90
- y-position: -90 - 90
- x-velocity: -5 - 5
- y-velocity: -5 - 5
- angle: -3.14 - 3.14
- angular-velocity: -5 - 5

As previously mentioned, our environment has feature variables that could be used to make the environment more complex. The ones we will be working with are:

- Enabling wind, adjusting wind power
- Turbulence power

We decided to restrict a few of the environmental factors to simplify the environment for our experimental purposes. First, each state variable mentioned above has 10 decimal places of precision in the standard environment. Since this leaves us with an incredible amount of possible states the lander can be in at any given time, we chose to create a function that scaled each of these variables, therefore making it more likely that the lander will see a repeat state within our set of episodes and will learn a little quicker and making the policy updating more manageable. The second factor we decided to experiment with is the initial state the lander is placed in at the beginning of each run. The standard environment will put the lander in a random state at the beginning of each run, and so again to cut down on complexity and learning time. We experimented both with running with a seed and un-seeded and we ultimately chose to provide a seed at the beginning of each run so that the lander will start in the same position for every episode.

3.2 Rewards

At every step a reward is granted. The total reward of an episode is the sum of the rewards of all the steps in that episode. Depending on the action taken and subsequent state, a reward can take on a positive or negative value. According

to the documentation, an episode is considered a solution if it scores at least 200 reward points.

Below are the different manifestations of the reward function:

- Increased/decreased the closer/further the lander is to the landing pad.
- Increased/decreased the slower/faster the lander is moving.
- Decreased the more the lander is tilted (angle not horizontal).
- Increased by 10 points for each leg that is in contact with the ground.
- Decreased by 0.03 points each frame a side engine is firing.
- Decreased by 0.3 points each frame the main engine is firing
- -100 points for crashing
- 100 points for landing safely

4 Methodology

Throughout our experiment, there were a few concepts that were used by all our different methods. We define them here as they will be referred to only by name in the subsequent sections. [1]

Our baseline refers to running our algorithms without curriculum learning, in a Lunar Lander environment with specific parameter values. The values are illustrated below:

- Gravity: -10
- Wind Power: 15
- Turbulence Power: 2

To have reliability in our performance comparisons, we decided to use averaged number of landings as our metric. Each time we ran our algorithm on 10,000 episodes, we created an array that kept an aggregate count of the number of landings as each episode went on. We ran each algorithm ten times which generated 10 arrays of landing counts. Using all 10 arrays, we averaged the value of the first index, the second index, and so forth. In the end, we were left with a averaged array that depicted, on average, how many landings any given algorithm achieves in 10,000 episodes, starting with a random policy. This gave us confidence in our comparisons since the number of landings would fluctuate due to the inherent randomness in our algorithms.

Each of our methods also make use of the same scaling function. The size of our domain is intractably large, when we first ran our algorithms, our policies would learn nothing because each state would almost never be visited more

[1] All of the source code for this project was submitted with this report

than once. Since our state representation consists of an 8-tuple vector, with each variable having a range of values, we quickly realized that there were an intractable amount of combinations that could be encountered. We needed to reduce the size of our domain while still maintaining as much state information as we could. At each action step, the Lunar Lander environment returned a state representation. Before updating our policy on this state, we scaled the state vector to reduce the amount of possible combinations that our policy would encounter. We tried several different iterations of scaling functions and compared the number of landings given by each one. The final scaling function we went with is shown below.

- x-value rounded to whole number
- y-value rounded to whole number
- x-velocity rounded to first decimal place
- y-velocity rounded to first decimal place
- angle rounded to first decimal place
- angle-velocity rounded to whole number
- the last two Boolean variables were left untouched since they only had two possible values each could take.

4.1 Curriculum Learning

As mentioned in the background section, curriculum learning presents tasks to the agent in a distinct and structured order. The curriculum depended on the Lunar Lander's complexity features, this would be the gravity, wind power, and turbulence power.

Since our objective was to evaluate the concept of curriculum learning, we decided to only vary the wind power and turbulence power features and kept our gravity value constant. We figured manipulating the gravity value would add undo complexity. For the ceiling values of our two features, we decided on wind power equaling 15, and turbulence power equaling 2. Moving forward, we will consider these feature values to be our baseline lunar lander environment–the environment that we will use to compare the results of our different algorithms.

We created different curricula by varying our chosen feature values, ordered by easiness[1]. Starting with a simplified environment and increasing the environment's complexity throughout the training, we could measure what performance effect this would have on generating the optimal policy. We tried several different curricula but decided to evaluate our algorithms on the three choices below.

Curriculum 1

- Phase 1 - Wind Power: 0, Turbulence Power: 0

- Phase 2 - Wind Power: 5, Turbulence Power: 1
- Phase 3 - Wind Power: 10, Turbulence Power: 1.5
- Phase 4 - Wind Power: 15, Turbulence Power: 1.5
- Phase 5 - Wind Power: 15, Turbulence Power: 2

Curriculum 2

- Phase 1 - Wind Power: 0, Turbulence Power: 0
- Phase 2 - Wind Power: 0, Turbulence Power: 1
- Phase 3 - Wind Power: 5, Turbulence Power: 2
- Phase 4 - Wind Power: 10, Turbulence Power: 2
- Phase 5 - Wind Power: 15, Turbulence Power: 2

Curriculum 3

- Phase 1 - Wind Power: 0, Turbulence Power: 0
- Phase 2 - Wind Power: 5, Turbulence Power: 0
- Phase 3 - Wind Power: 10, Turbulence Power: 0
- Phase 4 - Wind Power: 15, Turbulence Power: 1
- Phase 5 - Wind Power: 15, Turbulence Power: 2

In creating our curricula and deciding how much to vary each feature and in what order, we went through 3 main routes. To vary both Turbulence Power and Wind Power at the same time, to vary Turbulence Power first, or to vary Wind Power first. We also wanted to equally divide our 10,000 episodes of training amongst the five phases. This dictated the delta of the feature values among each phase. We made the increments equally distributed between the lowest value and our ceiling (baseline) value.

In our experiment, we use asymptotic performance to compare the efficacy of the policies after training [3]. This is the performance of the trained policy on the target task. A comparison of one trained policy having gone through the curriculum and another policy trained directly on the target task from the start.

4.2 Q-Learning

Q-Learning is a model-free RL algorithm that is well-suited for solving Markov Decision Processes in which an agent interacts with an environment. It is known for its simplicity, effectiveness and ability to handle large state and action spaces. This method has a learned action-value function, Q, that directly approximates

q_*, the optimal action-value function independent of the policy being followed. This method is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_T)] \quad (2)$$

Q-learning aims to find this optimal action-value function, called Q-function, by estimating the expected cumulative reward an agent will receive by taking a specific action in a given state. The Q-function is iteratively updated based on the agent's experience using the Bellman equation to gradually improve the policy. Since the Lunar Lander environment typically provides sparse rewards where the agent receives the majority of it's feedback on successful landings, Q-learning combined with an epsilon-greedy strategy enables the agent to explore different actions and discover landing strategies that lead to higher overall rewards. This allows the agent to learn from a large number of trail-and-error episodes leading to the optimal policy.

We began exploring the Lunar Lander environment with the Q-learning algorithm by running it on a baseline environment and adjusting the learning rate (α) parameter to determine which α returned the better results. The baseline environmental parameters that these were set to was wind_power = 15 and turbulence_power = 1.5. This is the default environmental parameters provided by the documentation. The other parameters we used were 0.99 discount factor, and a 0.1 epsilon for the epsilon-greedy implementation. The below plot shows the average number of landings over 10 iterations through a series of 10,000 episodes per iteration.

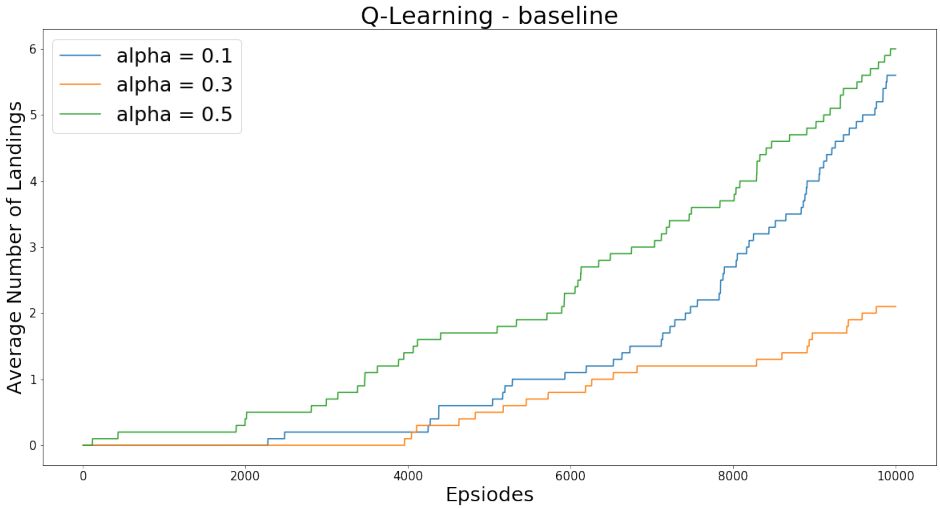


Figure 2: Average number of landings for different learning rates using Q-learning

As we can see from Figure 2, a learning rate of 0.5 achieved the highest number of average landings over the series of episodes. We didn't want to go much

higher than that because lower learning rates allow for more stable and gradual updates to the Q-values and ensuring a balanced exploration-exploitation trade-off. This is how we chose to use $\alpha = 0.5$ for the Q-learning algorithm for the rest of the curriculum testing.

4.3 SARSA

We decided to employ the Sarsa (on-policy TD control) algorithm to iterate through the episodes and deduce the optimal policy. We're planning to run Sarsa with the same number of episodes for each RL method. That way, each method gets the same amount of training and we can compare how the output policies perform. They may not be the actual optimal policies but since we're imposing a limit on the number of episodes, we are considering the output as the optimal policy.

To evaluate our SARSA algorithm, we first needed to find its optimal alpha parameter which can be thought of as its learning rate. A constant in the algorithm's equation that denotes how big of an update we allow for every action taken. In this algorithm, every time we take an action, we update the value of our previous state, depending on what reward is received from the action. If it's a good reward, the previous state is encouraged to take the same action, if it's a negative reward, the previous state is discouraged from taking the same action. The policy update equation that enables this learning is shown below.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_T)] \qquad (3)$$

By running our SARSA algorithm in a non-seeded, baseline environment, and trying several different values of alpha, we were able to find the one that yielded the most averaged landings.

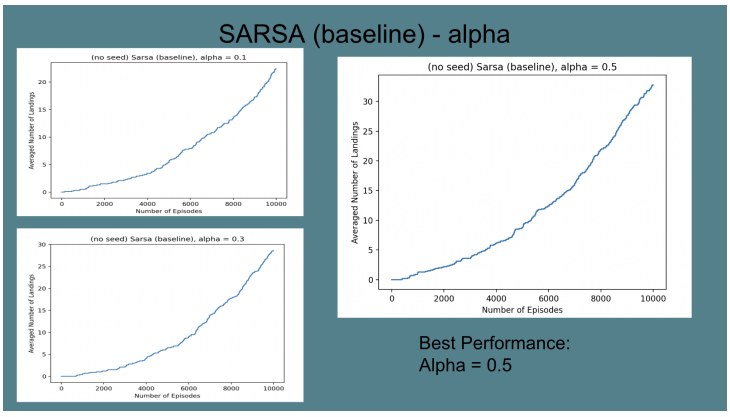


Figure 3: Average number of landings for different learning rates using SARSA

Following this, we wanted to more clearly visualize the learning rate of our algorithm so we compared a non-seeded environment with a seeded one. Since

a seeded environment uses the same start state for our Lander, we were able to get more landings. As we were trying ways to optimize our algorithm, the differences in our choices were made more clear due to the seeded environment allowing more data points.

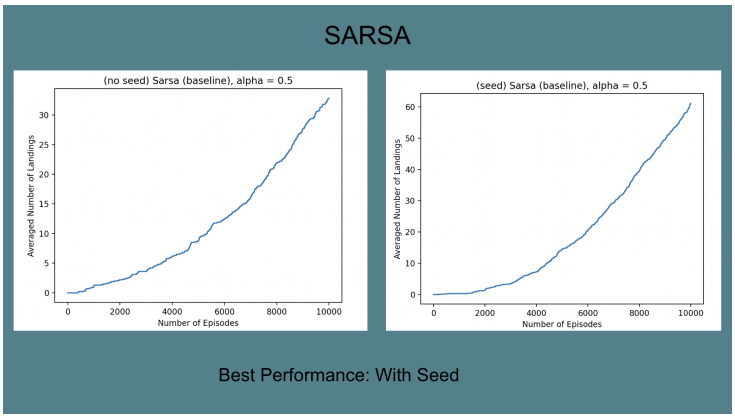


Figure 4: Average number of landings for a seed vs non-seeded environment using SARSA

Our pre-curriculum, final measure for SARSA's performance comes from 10 runs of our algorithm, each with 10,000 episodes. We aggregated the landing count per episode in each run and then averaged them by all 10 runs. By plotting this out, averaged number of landings vs episode number, we were able to analyze the slope of the graph which denotes its learning rate.

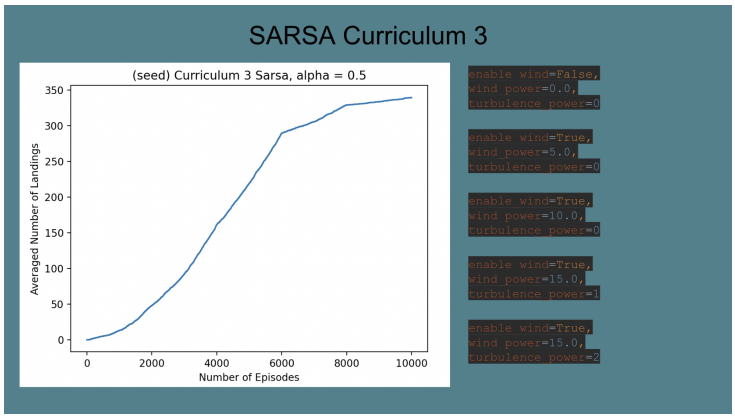


Figure 5: Average number of landings for Curriculum 3 using SARSA

As shown above, using the optimal parameters, we ran our SARSA algorithm on our curriculum choices and plotted the learning performance. By looking at

our plots, we were able to see how well the algorithm was learning during each curriculum phase and we able to deduce which curriculum provided the best results.

Lastly, each time our algorithms trained on 10,000 episodes, they returned the policy they learned, which we saved. To compare which generated policy provides better performance, we ran each "optimal" policy on another 5,000 episodes, this time not updating the policy at all. This allowed us to come to a conclusion on which variables allow our SARSA algorithm to perform the best in this Lunar Lander environment. This is discussed in our results section.

4.4 n-step SARSA

Conceptually, one can say that SARSA is just 1-step SARSA. With n-step SARSA, after every action, instead of just updating your previous state, you update your previous n states. This allows for more of your states to be updated after every action which could result in faster learning. There are pitfalls one must watch out for since you may also be updating too much which could make your policy learn nonsense. The policy update equation which enables this learning is shown below.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G - Q(S_t, A_t)] \tag{4}$$

Where G is the discounted sum of rewards of the previous n states.

With this in mind, we follow the same process as with SARSA to find the optimal alpha parameter. We use a consistent value for n so we can compare the performance based on alpha. Once we found our optimal alpha, we tried several different values for n and discovered which one performed best.

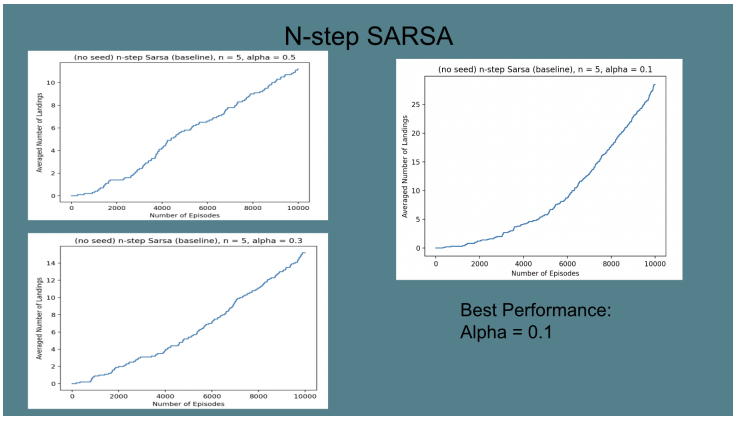


Figure 6: Average number of landings for different learning rates using N-Step SARSA

Having these two optimal parameters, we again do 10, 10,000 episode runs to compare our seeded environment with non-seeded and found that seeded

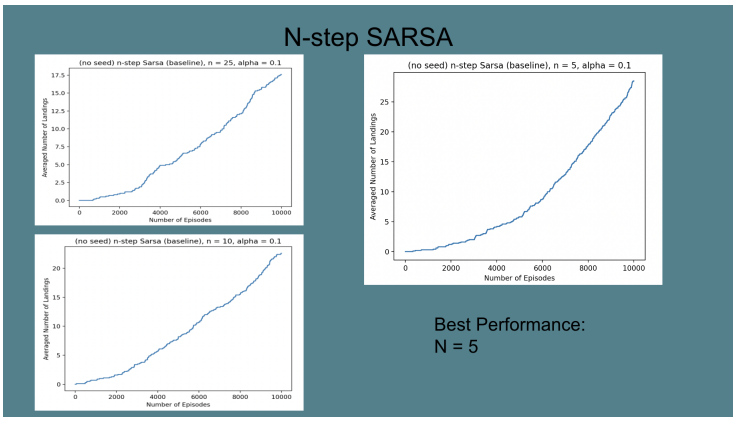


Figure 7: Average number of landings for different values of N using N-Step SARSA

environments allow for better visualization of performance and learning rate. This was our final performance measure for N-step SARSA, pre-curriculum.

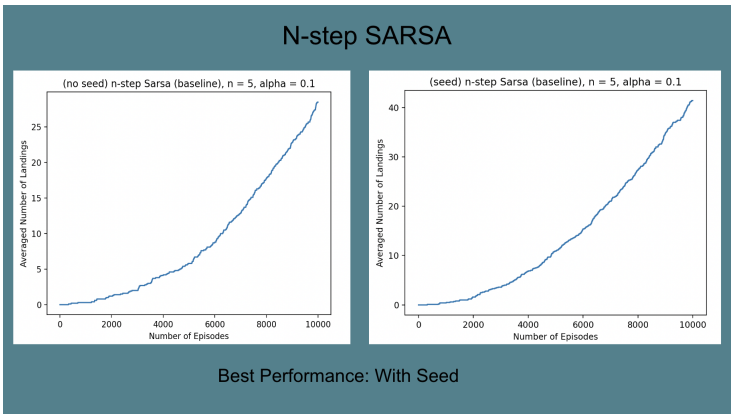


Figure 8: Average number of landings for a seed vs non-seeded environment using N-Step SARSA

Following this, we ran our N-step algorithm through our curriculum choices and discerned which curriculum managed to land the lander the most. It turned out to be curriculum 3.

Lastly, by saving the returned policies of our N-step algorithm and our curriculum N-step algorithm, we ran each on an additional 5,000 episodes, without any policy updating, and compared which algorithm generated more landings. The allowed us to compare the performance effects of curriculum learning and is discussed in our results section.

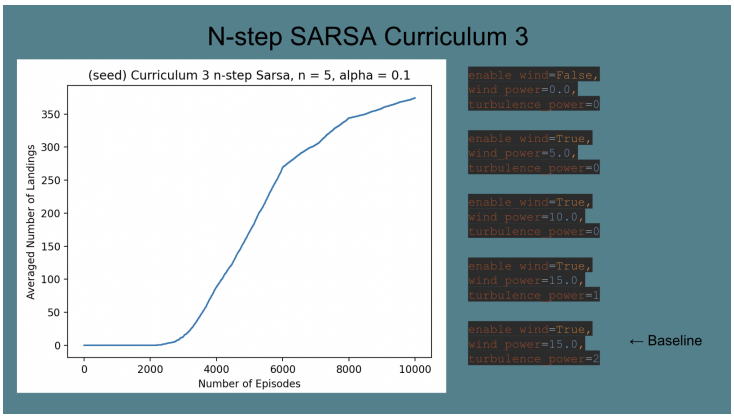


Figure 9: Average number of landings for Curriculum 3 using N-Step SARSA

5 Results

Using the same Lunar Lander environment, with complexities turned on to their baseline values, we will evaluate the performance of our RL methods and compare them to discern benefits and limitations of the respective RL methods.

Our hypothesis is that the Curriculum Learning method will perform better than our baselines. We are hypothesizing this due to the orderly structure of Curriculum Learning. This method builds upon itself with the only change being the movement from the simplest source task to the increasingly complex tasks. By initially learning on simpler tasks, our lander is able to land more often, teaching it that landing is what gives it the most reward. This helps when transitioning to a complex environment because the lander already has more experience with landing and realizing how much reward it generates. Compared with initially learning on a complex environment, the lander does not land as often so it has less experience with the goal of its task.

In our Q-learning experiments we compared the performance of the algorithm with and without using a curriculum and employing an $\alpha = 0.5$ learning rate. This process was tested on all three of the outlined curricula and the performance was measured by the average number of landings over 10 total iterations of 10,000 episodes each iteration. The difficulty in tasks was introduced in equal steps every 2000 episodes. As shown in Figure 9 the Q-learning curriculum using curriculum 1 only slightly outperformed the base case and curriculum 2 did just about as well as the base case shown in Figure 10. Finally, curriculum 3 out performed the baseline experiment the best, although none of these were obviously substantially better than the baseline. We believe that the reason the curricula performed differently from each other is that they all increased the difficulty of the tasks at different rates. Curriculum 3 eased the agent into the most difficult environment with smaller increments of wind and turbulence power, therefore not shocking the agent as much and setting it up to

learn a little more efficiently. Aside from curriculum design, another factor to the poorer performance of the Q-learning algorithm could be the timing of when the difficulty was increased through the runs of episodes. Some of the harder tasks could have been introduced too early when the agent hadn't sufficiently mastered the foundational skills. Some methods that could have resulted in a little better performance would be to increase the number of episodes, giving the agent more time to learn in the environment at each curriculum step, and to better optimize the timing and difficulty rates at which the agent is exposed to. The timing of curriculum introduction should be refined as introducing tasks too late could result in the agent having already converged on an optimal policy making it difficult for the agent to improve performance.

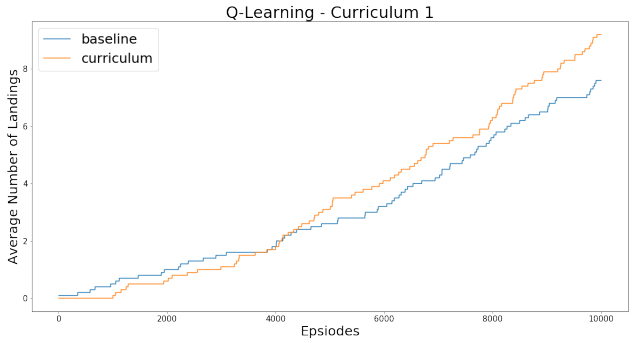


Figure 10: Average number of landings for Curriculum 1 using Q-learning compared to the base case without the curriculum implementation

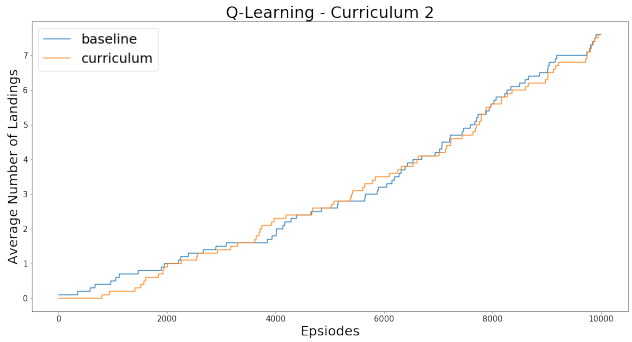


Figure 11: Average number of landings for Curriculum 2 using Q-learning compared to the base case without the curriculum implementation

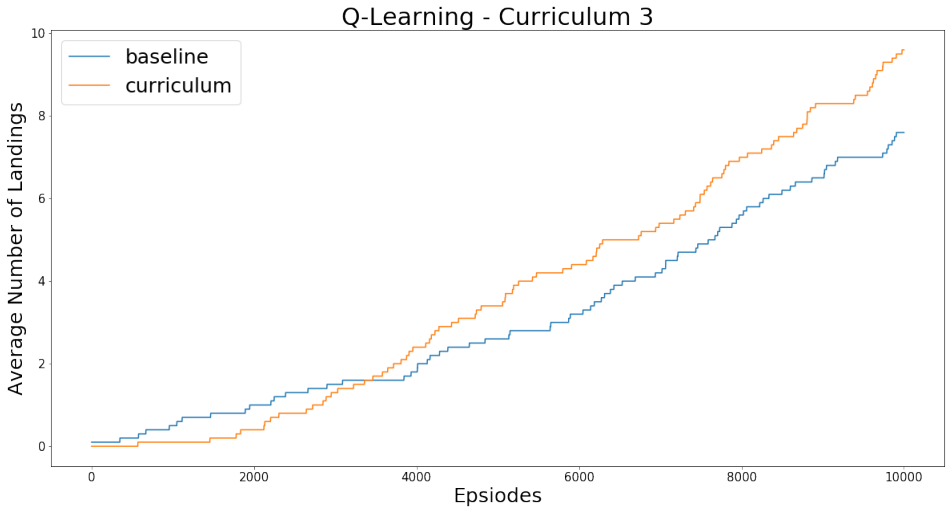


Figure 12: Average number of landings for Curriculum 3 using Q-learning compared to the base case without the curriculum implementation

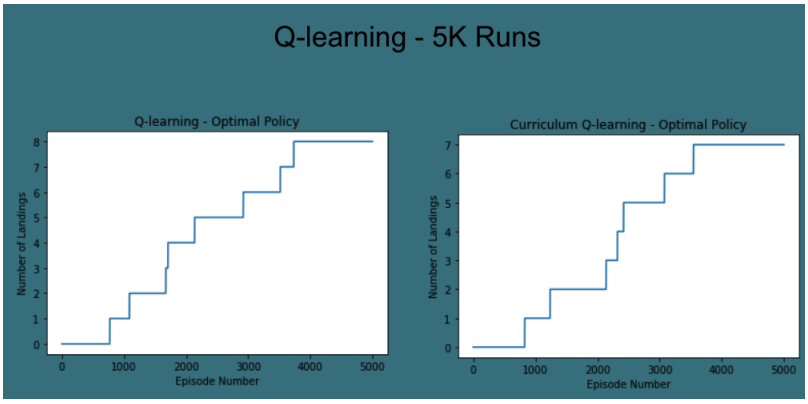


Figure 13: Number of landings for optimal policy using Q-learning

With our SARSA method, we compare the performance of regular SARSA with SARSA using a curriculum. In both scenarios, the optimal parameter values that were found were employed. The slopes of these graphs define how well the agent is learning and should ultimately converge to 1, which would mean the lander lands on every episode. We must be careful with using number of landings as our metric because it may not be completely indicative of how well trained the final policy is. Curriculum learning had a much higher number of landings but that is because it initially trained in simpler environments so it accumulated landing experiences early on. You can see in our curriculum plot that the slope starts to decrease every 2000 episodes which is when our algorithm

advances the Lunar Lander environment to the next level of complexity. Our baseline environment begins at episode 8,000 and you can see that the agent lands the least number of times during the episode interval.

On the other hand, our SARSA plot indicates an exponential slope. This makes sense because since we started with our baseline environment, the initial learning may have been slower but started to catch up quick towards the end of the training. If we trained on greater than 10,000 episodes, SARSA may have reached its maximum learning limit, where it is able to land on every episode. Further analysis can be done to determine how quickly each method reaches it maximum optimal policy. Due to time restraints, we capped our trainings at 10,000 episodes and denoted that as our optimal policy.

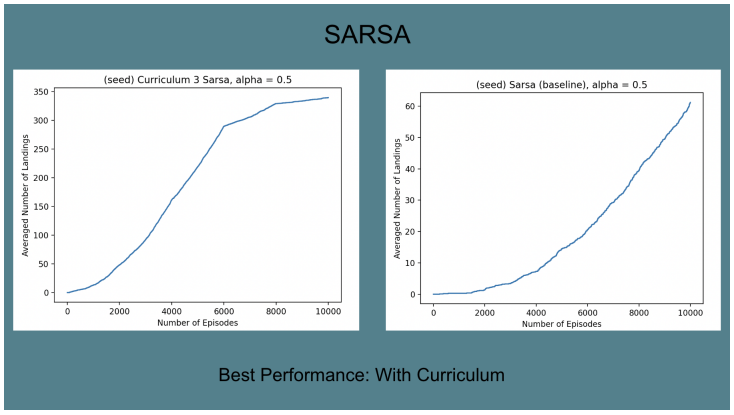


Figure 14: SARSA vs Curriculum SARSA

By saving the optimal policies generated by training, we can compare which policy performs best. This was a better measure of which training produced the most learning. As shown below, SARSA with a curriculum still performs better but their performances are not as distinct as our previous graphs. Curriculum SARSA is only showing marginal improvement over regular SARSA. This shows a clearer picture of the efficacy of our training methods. Our results indicate that learning on a curriculum does generate a better policy, given a restricted training horizon of 10,000 episodes.

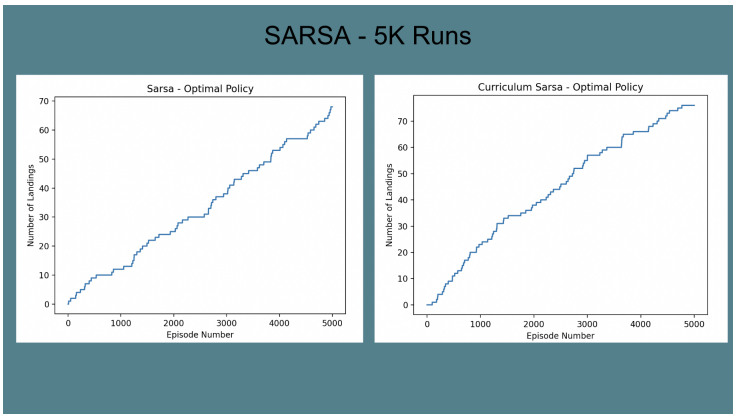


Figure 15: Number of landings for optimal policy using SARSA

A similar story plays out with N-Step SARSA. Below illustrates our final measurements for N-Step SARSA vs curriculum N-Step SARSA. On the curriculum plot, at every 2,000 episode interval, we increase the environment complexity and we can see the slope decreases, which indicates that our agent is landing less often. The sharpest decreases come during the intervals when our Turbulence feature was turned on. This indicates that Turbulence affects the success of our Lander more than Wind Power.

One distinction to point out in the curriculum plot is that there seems to be no learning occurring in the first interval of 2,000 episodes due to 0 landings. Regular N-Step SARSA does show learning in this interval, as well as our SARSA. It's unclear why this is but we did notice our algorithms tended to learn two behaviors. One to land and the other to hover, which also gave it positive rewards, the slower and more stable it hovered. Our policy could have been learning this hovering behavior first and then discerned that landing gives it the most rewards. This distinction may be more prominent in a curriculum because it is easier to hover in an environment with no Wind or Turbulence which was the first phase in our curriculum.

Our regular N-Step SARSA also displays an exponential learning curve. It displays the same behavior as our SARSA plot but does not land as many times. Indicating that SARSA has better performance than N-Step SARSA.

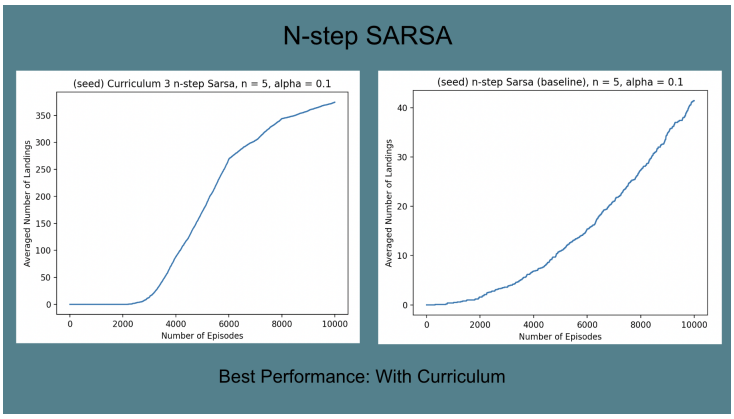


Figure 16: N-Step SARSA vs Curriculum N-Step SARSA

After saving our policies and running them on 5,000 episodes, we received the below. Our curriculum shows improved performance but only by about 10 landings. Again, we are able to see a clearer picture as to which training method generates a more effective policy. Since we are keeping the policy constant during these episodes, we can see that the plots are more linear than our learning curves. This makes sense since the same policy should display consistent behavior.

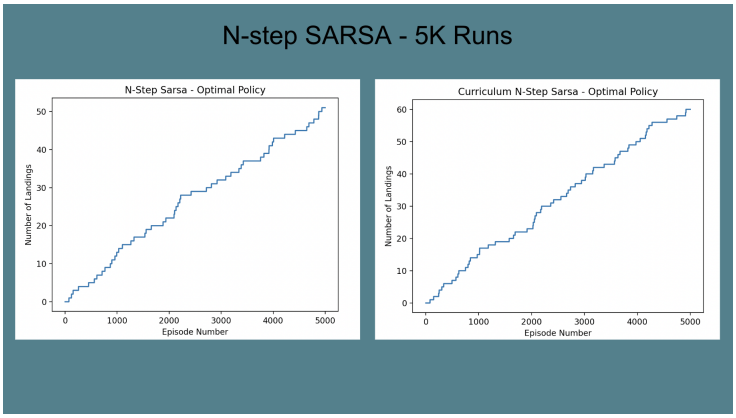


Figure 17: Number of landings for optimal policy using N-Step SARSA

Lastly, we display our end results in table format. In all cases, our curriculum helped learning which provides evidence that training agents on increasingly complex environments helps reduce the time to reaching an optimal policy. Moreover, SARSA proved to be the best algorithm for this environment as we saw the most landings come from this optimal policy in our 5K runs.

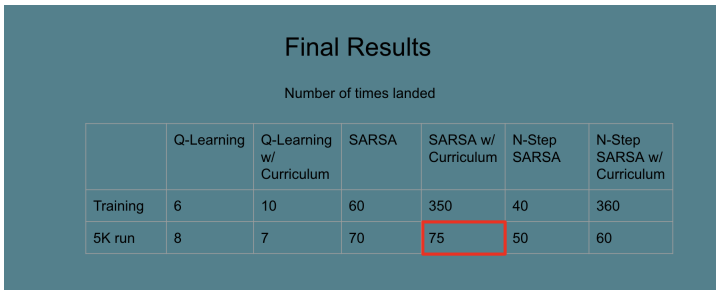


Figure 18: Number of landings for each of our methods

6 Conclusion and Future Work

6.1 Conclusion

Overall, we tried several different algorithms so we could discern the performance effects of adding a curriculum to those algorithms. Our results indicate that in all scenarios, adding a curriculum benefited our performance and resulted in the Lander achieving more landings. Henceforth, our hypothesis was supported in that, applying a curriculum would allow the agent to learn a more optimal policy, given the same training time as a method not using a curriculum. Faster training is an advantage of curriculum learning. In some cases, due to the curriculum, the agent spends less time on noisy/harder examples when it is not ready to incorporate them [1]. Our case is a bit of the converse, due to the Curriculum, the agent spends more time achieving its landing goal and learns the actions that encourage a successful landing. Therefore, it is more adept at landing once it enters the more complex (baseline) environment.

Our experiment also uncovered that our SARSA algorithm performs the best in this Lunar Lander environment. Looks like only updating the previous state helps the Lander learn better actions to take. Versus propagating those updates further back at each time step like in N-Step SARSA. This did not make as much intuitive sense since one would think when you fly, you plot out a trajectory and you would want to update your whole trajectory which is more in line with our N-step method. Alas, our results showed otherwise.

One limitation we encountered was the size of our domain. Through our scaling function, we were able to greatly reduce the size where it made our learning methods tractable. Although, that does not mean the way we reduced the domain was the optimal way. Further work could be done in modifying the state representations that are returned by the Lunar Lander environment.

6.2 Future Work

There are several different tweaks that could be made to our algorithms and environmental domain which could produce further improved performance. To start, our scaling function was not guaranteed to be optimal. There could have

been other variations that decreased the size of the overall domain but kept more useful state information. Ideally, the scaling function should modify our state representation so our policy is able to see recurrent states more often.

We also tried different methods of keeping the lander centered, that way it was more likely to interact with the landing pad. We must remember that the Lander can land outside of the landing pad but it is less likely to successfully land there due to the Lunar terrain. We tried modifying the reward function by adding extra reward if it landed on the landing pad; coordinate (0, 0). This showed no signs of improvement. We also added extra reward as the Lander got close to the landing pad and added negative reward if it went farther away. We even tried a simplified version where we give the lander extra reward just for being near x-coordinate: 0. It may not be rewarded extra for landing on the landing pad but it was encouraged to stay centered. None of these modifications showed improvement but I think their concepts are sound. I believe the way we modified our reward function could be better, we modified it using our state representation, post scaling. I think it's worth looking into using the pre-scaled representation as it has more information. This should work well for generating reward since we're comparing inequalities instead of saving specific state representations.

References

[1] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA, 2009. Association for Computing Machinery.

[2] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *J. Mach. Learn. Res.*, 21(1), jan 2020.

[3] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. pages 566–574, 2016.

[4] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[5] Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):4555–4576, 2022.