A Framework for Formal Verification to Correct Actions in Reinforcement Learning

ETHAN HOBBS and VIKAS NATARAJA, University of Colorado Boulder

In reinforcement learning, proving the possibility of a safe state is inherently difficult due to the range of methods available to find a new safe state. One of the most common approaches for finding a new state when the agent reaches an unsafe one is reverting to an initial state. In many cases, these methods are inefficient or might not actually produce a safe state transition. In this paper, we propose a new method for verifying an agent's policy for transitioning out of an unsafe state in the immediate future by using backtracking to guarantee safe state transition.

ACM Reference Format:
Ethan Hobbs and Vikas Nataraja. 2020. A Framework for Formal Verification to Correct Actions in Reinforcement Learning. *ACM Forthcoming* CSCI 5535, Spring 2020 (May 2020), 11 pages.

1 INTRODUCTION

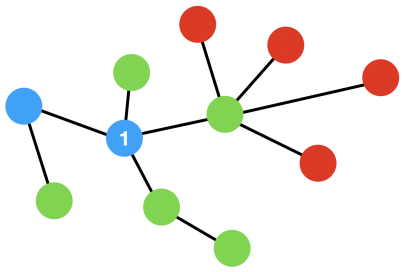
Reinforcement Learning (RL) has gained immense momentum in recent years particularly in the field of robotics where tasks are repetitive and RL can make an instant impact. This is because the agent can learn a policy that maximizes the reward function much quicker in repetitive tasks because the reward environment is denser and guarantees near-continuous rewards for every action that the agent takes. With RL gaining popularity, verification of such systems is an active area of research in Computer Science. The core problem of any software verification is to verify that a given system satisfies its specification. A conventional way to verify validity is to establish safety rules before the agent is deployed in the environment which requires extensive data and pre-computation [4]. It is not always possible to predict the states or the changes in the environment beforehand particularly if it is dynamically changing. Another common approach that is more widely deployed in Machine Learning is for the system itself to verify its own progress [9, 12]. While it is difficult to characterize such a verification, it affords better safety which is essential in RL (and relational verification of RL). Formal state verification then becomes essential so that the agent can monitor its progress by checking the validity of states. We propose *backtracking* through past states to avoid transitioning to an invalid or unsafe state similar to the method proposed by Goyal et al. [5]

In this paper, we address the problem of overcoming invalid or unsafe states by presenting an extension to [12]. First, we introduce a toy problem explaining our process, the terminology that will be used in the rest of the paper, describing and defining the basis functions followed by a technical section explaining the theoretical background behind our implementation. We then present an evaluation scenario and round out by discussing the conclusion and future work and improvements.

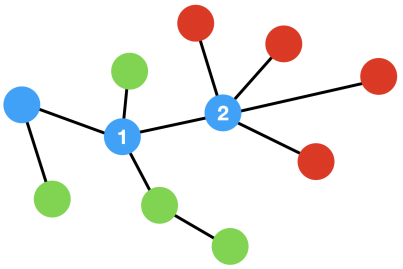
Authors' address: Ethan Hobbs, ethan.hobbs@colorado.edu; Vikas Nataraja, viha4393@colorado.edu, University of Colorado Boulder.

2 OVERVIEW

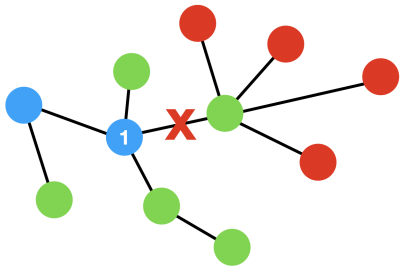
In traditional reinforcement learning systems, an agent can get stuck in a situation where there are no safe actions that allow the system to progress. This is especially apparent in physical systems like a robotic arm where to get to a new state, the robot might cause damage to itself. An example situation can be seen in Figure 1a and 1b. Here the agent transitions from state 1 to 2 which is a valid action that is safe. However there are no actions at this location that are safe so the learning process would stop leaving the agent untrained.



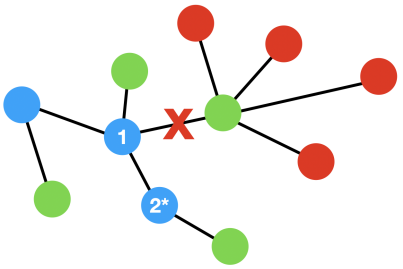
(a) Initial state labeled state 1



(b) New state at 2. Notice there are no safe states to transition to.



(c) Backtracking step to state 1 avoiding being stuck. The red x indicates that the transition to former state 2 is no longer possible



(d) Repeating the process at state 1 leads us to state 2*.

Fig. 1. A toy example of backtracking. The blue indicates states that exist in memory. The green indicate safe states while the red indicate unsafe states.

To make the system more intuitive, we consider a cartpole system which is an extension of the classical inverted pendulum problem. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. A visual reference can be seen in Figure 2. The system is controlled by applying a discrete force measure to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. Assuming a *fully observable* system, the observation space then becomes the cart velocity, cart position, pole angle and the pole velocity at the tip. Using these parameters, again, under the assumption that these are fully observable, we can describe the entire system. The action space is discrete and binary - moving the cart to the left or the right but not both. More details about the dynamics and specifics of the cartpole environment are explained in Section 5. Using intuition, an unsafe state in this environment could potentially be if the proposed action caused damage to the cart such as large force to the cart causing it to leave the designated

track. We will be using the example of the cartpole system throughout the paper to explain our methodology.

One method to get around this problem of overcoming unsafe states was developed by Zhu et al. [12]. In this method they synthesize a deterministic program from the policy and verify the action based on that approximation. If they encounter an invalid state they perform a mathematical operation on the state space reducing its size. This method works very well, however since most reinforcement learning systems are physical systems like robots, it would be preferable to apply a physical based operation to help the verification instead of an abstract mathematical operation.

If an agent had some knowledge of its previous states, then, when it encountered an unsafe one, it could revert to a previous state and try a different path. We call this idea *backtracking*. There are two components to this idea: a look ahead phase, and then a possible backwards step if there are no safe states. These components can be seen in Figure 1c and 1d. In Figure 1a, the agent is stuck at state 2 after completing the look ahead stage, but it has a memory of the current and previous states shown in blue. We revert back to state 1 and eliminate the possibility of taking the same transition. The agent then makes a new choice of the possible states and enters into state 2^* thus avoiding the invalid states.

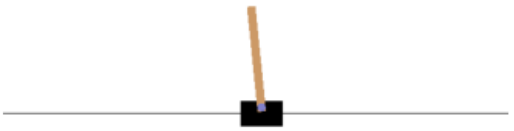


Fig. 2. Screenshot of the Cartpole-v1 environment

3 REINFORCEMENT LEARNING BACKGROUND AND TERMINOLOGY

This section will detail some basic concepts in reinforcement learning such as Q-learning, policies, rewards and other common terminology used in reinforcement learning that will be used in subsequent sections.

3.1 Q function, policy and reward

In reinforcement learning, the objective is to maximize a certain *reward* in a given situation. In our example of a cartpole, the objective is to balance the pole without having it fall over and for every timestep that is achieved, a positive reward is received. It is important to consider immediate rewards and future rewards simultaneously because optimizing for one means sacrificing the other and this concept of exploration vs exploitation is a core concept of RL and we use γ to indicate the *discount factor* which balances these two types of policies. To quantify reward, formally we can write the cumulative discounted reward as:

$$R_t = \sum_{t=0}^{\infty} \gamma^k r_t \tag{1}$$

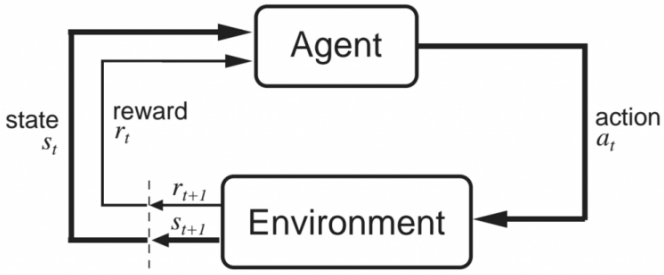


Fig. 3. Schematic representation of an RL agent and environment [10].

where t indicates timestep, r_t is the expected reward at timestep t.

Next, we introduce *policy* as describing the way of taking an action or more specifically, it describes the probability of taking an action in a given state. It takes in an action and a state and return the probability of taking that action with the goal being to maximize over some parameter(s) θ. It is denoted by $\pi_\theta(s, a)$ or sometimes $\pi_\theta(a|s)$ as a way of indicating conditional probability. We will use the former notation in this text.

Using these terms, we can formalize an action value function called Q which indicates the value of taking action a in state s when acting under a policy π as the expectation of receiving reward R_t when in state s and taking action a:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_t|S_t = s, A_t = a] \tag{2}$$

The expectation takes into account the randomness in future actions according to the policy, as well as the randomness of the returned state from the environment. The randomness arises because both the policy and the transition function to go from one state to the other may be stochastic. Figure 3 describes the schematic relationship between an agent and its environment. At a time step t, the agent is in state s_t and takes an action a_t. The environment then responds with a new state s_{t+1} and a reward r_{t+1}. The reason that the reward is at $t + 1$ is because it is returned with the environment with the state at $t + 1$.

3.2 Q-learning

Q-learning was first introduced in 1992 by Watkins et al. as a way for an RL agent to learn an optimal policy in a Markov Decision Process (MDP) [11]. Since then, it has gained immense popularity and is one of the most common ways of learning a policy. It uses the Q function to learn a policy for a state-action pair. In this context, Q function is the action value function that indicates the value of taking a particular in some state under a policy π. At the start of the episode or session, all Q values are either initialized with zeros or randomly initialized. For each state, every action possible at that state has an associated reward for taking that action and it should be noted that rewards can be negative as well to indicate that the agent chose the "wrong" action and learned a sub-optimal policy and these negative rewards are cumulatively used to learn a better policy at each timestep depending on the reward. After every action is taken and an outcome is observed, the Q value is updated. At each timestep, the maximum expected future reward is balanced against optimizing current state-action policy by using γ. To update the Q value from the previous Q value we use the expected reward (of taking action a at state s) and the maximum of the predicted Q' value over a possible future state s' and action a' influenced by the discount factor γ. We also introduce *learning*

rate α which balances the extent to which the new Q value is considered over the previous one. Putting this together, we arrive at this equation:

$$\underbrace{\text{New}Q^{\pi}(s,a)}_{\text{New Q-Value}} = Q^{\pi}(s,a) + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount factor}} \overbrace{\max Q'(s',a')}^{\text{Maximum predicted reward, given new state and all possible actions}} - Q(s,a)]$$

3.3 Deep Q-learning

Deep Q-learning is an extension of the Q-learning scheme and improves upon it by speeding up the learning process using deep neural networks and reduces memory consumption. Classical Q-learning works well for environments with a small number of states and actions because the Q-table can be easily built. But it cannot handle computation for environments having hundreds or thousands of states and actions because saving the Q-table in memory is simply unrealistic. Deep Q-learning overcomes this problem by approximating the Q-value using neural networks trained as a regression problem and evaluated under appropriate loss functions like mean squared error. For each state, it approximates the actions and chooses the *best* action for that state and records progress in the experience replay buffer. In our proposed method, we employ a deep Q-network to solve the evaluation scenario, more details about the implementation are available in Section 5.

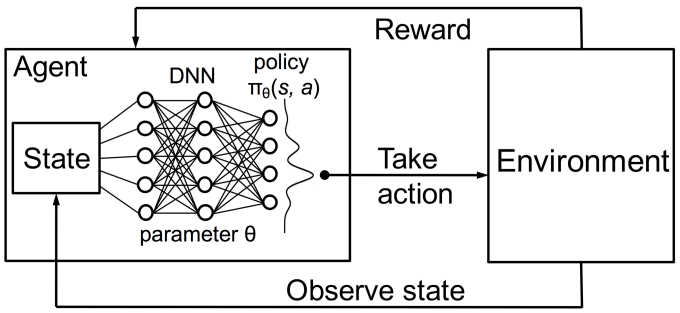


Fig. 4. Schematic representation of a Deep Q-network [7].

3.4 Replay Buffer

The replay buffer, also called experience replay, is a technique used to allow the agent to learn from previous experiences and thereby improving learning by making better informed decisions about the action to be taken at a state. The experiences are stored in a buffer with a preset memory size and the size of the memory can affect the speed of computation. Liu et al. explored the effects of changing memory sizes and found that too much or too little memory slow down the learning process [6]. In our proposed work, we use the replay buffer to traverse through the history of saved states to find a safe state. The states in the buffer are guaranteed to be safe because the agent has continued learning and has updated its policies after passing those states.

3.5 Unsafe or Invalid States

Following the notation explained by Garcia and Fernandez [3] we can define safe and unsafe states (also called error and non-error states) in RL as follows:

Let S be a set of states and $\phi \subset S$ the set of error states. A state $s \in \phi$ is an undesirable terminal state where the control of the agent ends when s is reached with damage or injury to the agent, the learning system or any external entities. The set $\Gamma \subset S$ is considered a set of non-error terminal states with $\Gamma \cap \phi = \varnothing$ and where the control of the agent ends normally without damage or injury.

In the context of this paper, we describe an error state as an undesirable state that if entered, the episode ends abruptly and could cause harm to the program. Our aim with this paper is to show that when an unsafe state is encountered (or is about to be encountered), instead of reverting to a set of initial states, we can backtrack through the agent's recorded states and history in the replay buffer.

4 VERIFICATION PROCESS

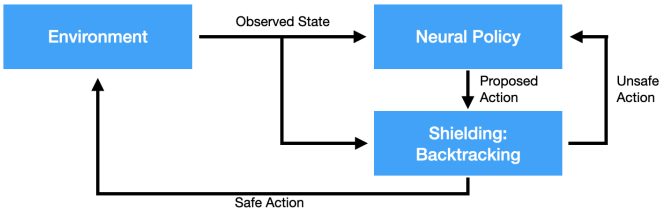


Fig. 5. Modified flow of the reinforcement learning algorithm. Notice the inclusion of the sheilding step with two possible outputs: a safe action that updates the environment or an unsafe action which updates the neural policy.

We chose to do a run time shielding method that will be assessed at each action taken by the agent. We follow the flow of the diagram in Figure 5 which is similar to the methods proposed in Zhu et al. [12], but without the deterministic synthesis. We chose to do this since the more physical idea of backtracking was incredibly hard to abstract into the mathematical domain. There are several important factors to this verification step that can be tuned by the user as hyper parameters in the RL method: number of steps that the method looks ahead n, the number of states kept in the memory m, the number of steps previously that we start looking for a new state $thresh$ where $thresh \leq m$.

Given the current state of the system s, and the developed policy π, we can begin the algorithm. First we must find all possible states n-steps ahead of the current state. We approach this by developing a tree with the number of leaves as the number of possible actions at each step. The set of each possible path of states is called $S_{possible}$. This method works for discrete systems with finite number of actions at each time step like the cartpole. We then must identify the safety of the states in $S_{possible}$ creating a new set of the safe states called $S_{safe} \subseteq S_{possible}$. Identifying the safety of individual states is a hard problem on its own and outside of the scope of this paper. We chose an unrealistic method of manually inserting known invalid states while assuming all other states are safe states. Then we enter the shielding portion of the algorithm. If we have any possible states that are safe, we transition to them depending on the Q-function and add it to the replay buffer memory. Otherwise, we find a state s_k in the replay buffer R_b which we chose uniformly at random.

We must then clear the replay buffer for every step after s_k and repeat the algorithm until we have as safe action available to take. This method is summarized in Algorithm 1.

Algorithm 1: Shielding Algorithm (state s, policy π, replay buffer R_b)

```
Set threshold for backtracking timestep thresh
do
    S_possible ← Possible states from state s n-steps ahead
    S_safe ← Determine safe states in S_possible dependent on γ
    if S_safe is not ∅ then
        s' ∈ S_safe, a' ← that maximizes NewQ^π(s', a')
        R_b ← R_b ∪ {s'}
    else
        {s_k} ∈ R_b Chosen uniformly at random with k ≥ thresh
        remove all elements in R_b before the updated s_k and a_k
        Recalculate Q^π(s_k, a_k) for the selected previous state
    end
while S_safe is ∅;
return state s, action a
```

5 EMPIRICAL EVALUATION

This section details our evaluation procedure and the experiment setup. We explain the inner workings of the dynamics of the chosen evaluation environment to tie it back to Algorithm 1.

Our proposed method was implemented in OpenAI Gym's cartpole environment [1]. Specifically, we implemented our model in the cartpole scenario, an extension of the inverted pendulum environment. The goal is to balance the cartpole without it falling over. The environment offers a discrete force unit of +1 and -1 using which the pole can be balanced. For every timestep that the pole doesn't fall over, a +1 reward is provided. Note that this can be changed to suit different positive discrete value. The state space, which is now fully observable because of dense rewards, becomes our observation space which can be seen in Table 1. The agent's episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center after which the environment resets and the subsequent episode starts.

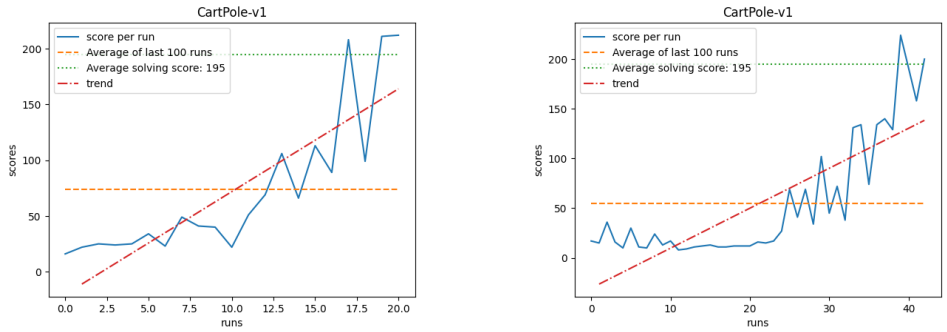
Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	$\sim -41.8^\circ$	$\sim 41.8^\circ$
3	Pole Velocity at Tip	$-\infty$	∞

Table 1. Observation Space constraints for Cartpole environment

To compare against, we chose the baseline to be a deep Q-network with 4 fully-connected or dense layers, each with non-linear ReLU (Rectified Linear Action Unit) activation except for the final layer which uses a linear activation because the output is a discrete value to be regressed over (the output of the network is an approximation of the action space which is discrete in our evaluation environment). Our proposed model implements a deep Q-network with state backtracking (not to

be confused with neural network back propagation which uses gradients and derivatives to learn). Both models were trained with mean squared error loss function to learn the control policy of the environment. In the proposed model, we leverage the agent's state history which is already cached in the replay buffer (also called experience replay and memory replay). This also enables the agent to learn from off-policy experiences to better judge future state-action pairs.

To get a fair comparison, we decided to record the scores for a total of 4 cases: baseline model evaluated under all valid states, proposed model evaluated under all valid states, baseline model evaluated under one random invalid state, and the proposed model evaluated under one random invalid state. We compared the scores in all 4 cases and fixed a threshold of 190 as recommended by OpenAI to mean "scenario solved" meaning when the model crosses a score of 190, it is considered solved. Additionally, we also fixed a pseudo-threshold of 2 for the number of times the model achieves a score of 190 meaning we consider the scenario to be solved when the model crosses 190 twice.



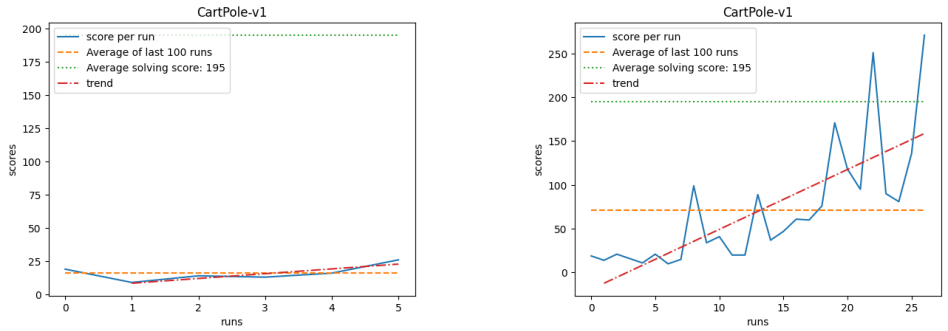
(a) Baseline model evaluated with all valid states (b) Proposed model evaluated with all valid states

Fig. 6. Baseline and proposed models evaluated under all valid states. Blue line shows the score per run/episode, orange line shows the average score of the past runs, green line shows the threshold of 190, red line tracks the trend of the scores

We observed that our model required more time to solve the cartpole environment when compared with a model that doesn't incorporate our changes in the absence of an invalid state. This is due to a combination of stochasticity of the environment and the fact that our look-ahead factor (the number of timesteps in the future to monitor for an invalid state) was quite large and thus resulted in computational speed. However, the total score achieved and the total reward remain largely the same. Figure 6 shows the comparison between the two models' performances. It tracks the episodes or runs on the X-axis and the recorded score for that run on the Y-axis. The baseline model takes about 19 episodes to cross the 190 threshold for the second time while our proposed model takes 42 episodes.

To test the model's performance in the presence of an invalid state, we forced one random element (with the same seed for both the baseline and the proposed models) of the observation space to go into an invalid form by replacing that element with a non-numerical state. Under these conditions, we noticed that the standard baseline model stopped as soon as it hit that state whenever that may be (due to the inherent stochasticity in the environment). Our model was able to detect the presence of the invalid state before entering it ahead of time and backtracked k timesteps through its previous states to find an alternative albeit a temporarily sub-optimal policy. This resulted in the model overcoming the invalid state and thus finding a secondary derived policy

which ultimately solved the scenario. Figure 7 shows the comparison between the two models. The baseline model stops after 5 episodes which is when it encounters the invalid state. The proposed model's score at episode 5 noticeably goes down for 3 episodes which is when it is computing an alternative policy. After episode 8, it recovers and the score starts increasing again and the model ultimately solves the scenario by crossing the 195 threshold for the second time at episode 26.



(a) Baseline model evaluated with one invalid state

(b) Proposed model evaluated with one invalid state

Fig. 7. Baseline and proposed models evaluated under one random invalid state.

6 RELATED WORK

In recent years, significant work has gone into verification of reinforcement learning and more specifically, the verification of states. A two-pronged system was implemented by Zhu et al. where the concept of safety and safety verification were both baked into the synthesis of a deterministic program with an inductive invariant alongside the neural policy [12]. However, when an invalid or incorrect state is observed, the agent reverts to one of initial safe states. In our approach, the agent back propagates to an already traversed safe state. A similar formal verification approach was taken by Sun et al. but to verify the actions of an autonomous robot by constructing a finite state abstraction and performing reachability analysis over the search space [9]. However, if the initial set of safe states are not within a certain limit, there is a high risk of state space explosion. At the same time, it does not apply to unbounded states and does not capture the relationships between the state variables. These shortcomings are bypassed in our approach because we use the existing state space to find a safe state. Singh et al. introduced a method to certify deep neural networks by using a novel abstract interpreter which relies on a combination of floating-point polyhedra and activation functions such as ReLU (Rectified Linear Unit) and the sigmoid function [8]. While this approach works to prevent exploding gradients during training, it works by transforming activation functions which inherently means that the neural network, in its current form, cannot be made robust to deeper architectures. In our proposed framework, we eliminate the need for changing activation functions by solving for state space. Chen et al. focused on relational verification represented as a Markov Decision Process (MDP) solvable via reinforcement learning [2]. The problem with MDPs is that high-dimensional state spaces are not solvable. In our proposed method, we aim to use the existing policy to find a safe state which does not expand the dimensionality.

7 CONCLUSION

We present an improvement to existing work to overcome unsafe or invalid states in reinforcement learning with the use of backtracking. Our proposed method requires more episodes to solve a scenario when there are no invalid states compared to baselines but performs comparably better when an invalid state is present in the observation space. Further, while the baseline fails to solve for invalid states, our model solves the scenario by opting for a sub-optimal policy for a temporary period and recovering to finish the scenario.

8 FUTURE WORK

Although our model performed well in the presence of an invalid state, we would like to experiment with multiple invalid states to test the robustness and adaptability of the model. One way to do this would be introduce additional invalid states at different stages which would allow us to evaluate the model's capability in such scenarios. One obvious shortcoming of our proposed model is the increased computational intensity. While this depends on the look-ahead factor i.e the number of states to look for in the future, we would like to make our model's performance time faster. One approach to do this would be to shorten the look-ahead factor so as to explore and search for fewer timesteps in the future but we would like to explore some other alternatives such as changing the model architecture as well. To test the model's versatility, we would like to explore other evaluation scenarios where learning a control policy is more nuanced and the environment has bigger observation and action spaces.

9 ONLINE RESOURCES

All our code and implementation can be found on our Github repository at this URL: https://github.com/csci5535-s20/project-rl-verification.

ACKNOWLEDGMENTS

We would like to thank Bor-Yuh Evan Chang and Benno Stein for their mentorship and advice during the course of this project.

REFERENCES

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *CoRR* abs/1606.01540 (2016). arXiv:1606.01540 http://arxiv.org/abs/1606.01540

[2] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational Verification Using Reinforcement Learning. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 141 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360567

[3] Javier García and Fernando Fernández. 2014. Safe Exploration of State and Action Spaces in Reinforcement Learning. *CoRR* abs/1402.0560 (2014). arXiv:1402.0560 http://arxiv.org/abs/1402.0560

[4] Divya Gopinath, Guy Katz, Corina S. Pasareanu, and Clark W. Barrett. 2017. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. *CoRR* abs/1710.00486 (2017). arXiv:1710.00486 http://arxiv.org/abs/1710.00486

[5] Anirudh Goyal, Philemon Brakel, William Fedus, Timothy P. Lillicrap, Sergey Levine, Hugo Larochelle, and Yoshua Bengio. 2018. Recall Traces: Backtracking Models for Efficient Reinforcement Learning. *CoRR* abs/1804.00379 (2018). arXiv:1804.00379 http://arxiv.org/abs/1804.00379

[6] Ruishan Liu and James Zou. 2017. The Effects of Memory Replay in Reinforcement Learning. *CoRR* abs/1710.06574 (2017). arXiv:1710.06574 http://arxiv.org/abs/1710.06574

[7] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. Association for Computing Machinery, New York, NY, USA, 50–56. https://doi.org/10.1145/3005745.3005750

[8] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An Abstract Domain for Certifying Neural Networks. *Proc. ACM Program. Lang.* 3, POPL, Article Article 41 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290354

[9] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. 2019. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC '19)*. Association for Computing Machinery, New York, NY, USA, 147–156. https://doi.org/10.1145/3302504.3311802

[10] R.S. Sutton. 1992. *Reinforcement Learning*. Springer US. https://books.google.com/books?id=GDvW4MNMQ2wC

[11] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. In *Machine Learning*. 279–292.

[12] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An Inductive Synthesis Framework for Verifiable Reinforcement Learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 686–701. https://doi.org/10.1145/3314221.3314638