A Brief Look into How Safe Reinforcement Learning Methods Compare to "Unsafe" Methods

Nathaniel Hamilton
Vanderbilt University
Nashville, Tennessee
nathaniel.p.hamilton@vanderbilt.edu

ABSTRACT

Reinforcement Learning (RL) has become an increasingly popular subject as the success of these algorithms and methods grows. To combat the safety concerns surround the freedom given RL agents while training, there has been an increase in work concerning *Safe* Reinforcement Learning. However, these new and safe methods have been held to less scrutiny than their unsafe peers. In this work we apply scrutiny and formal verification techniques to a new *safe* RL method as well as the original "unsafe" algorithm it is built on.

KEYWORDS

Deep Reinforcement Learning, Safe Reinforcement Learning, Formal Verification, Neural Network Verification

ACM Reference Format:
Nathaniel Hamilton. 2020. A Brief Look into How Safe Reinforcement Learning Methods Compare to "Unsafe" Methods. In *Proceedings of Some Conference (Conference '21).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Deep Reinforcement Learning (RL) has enabled Neural Network controllers to achieve state-of-the-art performance on many high-dimensional control tasks [Haarnoja et al. 2018; Lillicrap et al. 2015; Mania et al. 2018; Schulman et al. 2015, 2017]. However, RL allows agents to learn via trial and error, exploring any behavior during the learning process. In many realistic domains, this level of freedom is unacceptable. Consider the example of an industrial robot arm learning to place objects in a factory. Some behaviors could cause it to damage itself, the plant, or nearby workers. As a result, the realm of Safe Reinforcement Learning (SRL) is extremely important. However, little to no work has been done to formally verify these SRL methods and prove they actually create safe policies.

In this work we reevaluate and compare a recent safe reinforcement learning method that showed exceptionally promising results using compensating and guiding *control barrier functions* (CBF), [Cheng et al. 2019] against the vanilla RL algorithm it was built upon. We modify their provided implementation[1] to test them according to some of the standards written about in [Henderson et al. 2018]. These standards are more rigorous and help provide a better understanding of how well the algorithms and methods actually perform by stripping away some of the practices that have led to results' misrepresentation. We focus only on the DDPG implementations provided as they showed more promising results.

Additionally, we use a state-of-the-art neural network verifying tool to verify whether or not the learned policy networks of each method uphold the safety properties they were trained to uphold. By comparing both the performance metrics and the formal safety of the different methods, we can better understand how training with and without knowledge of the safety constraints impacts the learning process.

2 OVERVIEW/BACKGROUND

The goal of RL is to learn a good strategy for an agent to complete a task from experimental trials that have their levels of success represented as a reward. With the optimal strategy, the agent is capable of actively adapting to the environment to maximize future rewards. This is done by an agent choosing actions in an environment. The mapping of states or observations, s, to an action, a, is referred to as a learned policy, $a = \pi(s)$. The policy is most often optimized to maximize the reward according to the state-action value, also referred to as the Q-value. The mapping of the Q-value to the state-action pairs is the Q-function, $Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$.

2.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a popular and well studied deep reinforcement learning algorithm that was originally written about in [Lillicrap et al. 2015]. Since it's original publication, it has been used as a reference and baseline for the development of more complicated algorithms such as Twin DDPG (TD3) [Fujimoto et al. 2018] and Soft Actor-Critic (SAC) [Haarnoja et al. 2018].

DDPG is an off-policy, actor-critic method that is often referred to as "deep Q-learning for continuous action spaces" [Achiam 2018]. The critic learns the Q-function related to the environment and the actor is updated to maximize the expected reward according to the critic. This is accomplished following the steps in Algorithm 1[2].

2.2 Control Barrier Function Guided Control

In this control architecture, presented in [Cheng et al. 2019], the authors seek to achieve both safe and efficient learning by learning from the deployed controller that enforces safety constraints rather

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference '21, April 15, 2021, City, Earth
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

[1] available at https://github.com/rcheng805/RL-CBF
[2] For more information on DDPG, we recommend this site https://spinningup.openai.com/en/latest/algorithms/ddpg.html

Algorithm 1: DDPG Algorithm

INITIALIZE
critic network $Q(s, a|\theta^Q)$ with random weights θ^Q
actor network $\mu(s|\theta^\mu)$ with random weights θ^μ
target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
replay buffer R

TRAIN
for *episode = 1, 2, ..., M* **do**
  Initialize random process noise N
  $s_1 = initialize\_environment()$
  for t = 1, 2, ..., T **do**
    $a_t = \mu(s_t|\theta^\mu) + N_t$
    $s_{t+1}, r_t = execute(a_t)$
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample B random transitions (s_i, a_i, r_i, s_{i+1}) from R
    $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing
    $L = \frac{1}{B}\sum_i (y_i. - Q(s_i, a_i|\theta^Q))^2$
    Update actor using the sampled policy gradient:
    $\nabla_{\theta^\mu} J \approx$
    $\frac{1}{B}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
    $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end
end

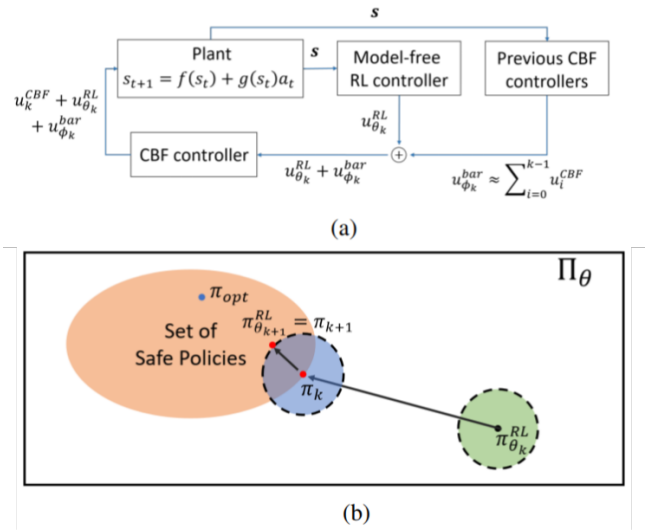

Figure 1: These images from [Cheng et al. 2019] show (a) the control structure of the CBF guiding controller and (b) how the use of CBF controllers guides the policy exploration within the set of safe policies.

than learning only from the RL controller. The authors utilize a *control barrier function* (CBF) control technique to enforce the safety.

The control barrier function controller is a natural tool for enforcing safety throughout the learning process [Cheng et al. 2019]. It utilizes a Lyapunov-like argument to provide a sufficient condition for ensuring forward invariance of the safe set under controlled dynamics. While the approach can be generalized to more barrier functions, this work focuses on affine barrier functions of the form $h = p^T s + q$, $(p \in \mathbb{R}^n, q \in \mathbb{R})$. With this restriction, the safe set has to be comprised of polytopes.

Using an estimate of the system's dynamics from a Gaussian Process estimator, a quadratic problem is solved at each step to compute a minimal control input that maintains safety. By remembering every previously deployed CBF controller, the agent is "guided" into the set of safe policies and forced to explore only within that domain like shown in Figure 1.b.

The architecture and interaction of the CBF controllers is shown in Figure 1.a. This controller combines the outputs of the RL controller and all previous iterations of the CBF controller in the following manner:

$$u_k(s) = u_{\theta_k}^{RL}(s) + \sum_{j=0}^{k-1} u_j^{CBF}(s, u_{\theta_0}^{RL}, ..., u_{\theta_{j-1}}^{RL}) + u_k^{CBF}(s, u_{\theta_k}^{RL} + \sum_{j=0}^{k-1} u_j^{CBF}) \quad (1)$$

Since the process of storing and adding together all previous CBF controllers would be costly and require a large computation time, the authors instead estimate the value by training another neural network.

The authors applied this method to both TRPO and DDPG, but, for the purpose of this paper, we will only focus on the DDPG implementation explained in Algorithm 2 based on their provided implementation.

2.3 Plant Model: Inverted Pendulum

In this work, we focus on OpenAI's *Pendulum* − *v*0[3] environment [Brockman et al. 2016]. It was one of the two environments tested in [Cheng et al. 2019] and is a well studied RL control problem with easily defined safety properties. The goal of the agent is to keep the frictionless pendulum upright and stationary.

The interior plant model changes according to the discrete dynamics

$$\dot{\theta}_{k+1} = \dot{\theta}_k + (\frac{-3g}{2l} * \sin(\theta_k + \pi) + \frac{3u_k}{ml^2}) * \Delta t$$
$$\theta_{k+1} = \theta_k + \dot{\theta}_{k+1}$$

where $g = 10$, $l = 1$, $m = 1$, $\Delta t = 0.05$, and u_k is the control from the neural network in the range $[-2, 2]$. Additionally, within the OpenAI environment, $\dot{\theta}$ is clipped within the range $[-8, 8]$, and θ is aliased within $[-\pi, \pi]$ radians. θ is measured from upright and increases as the pendulum moves clockwise. These values, θ and $\dot{\theta}$ are then used to determine the input values for the neural network

[3]The code implementation of this environment can be found at https://github.com/openai/gym/blob/master/gym/envs/classic_control/pendulum.py

Algorithm 2: DDPG-CBF Guided Control Algorithm

INITIALIZE
critic network $Q(s, a|\theta^Q)$ with random weights θ^Q
actor network $\mu(s|\theta^\mu)$ with random weights θ^μ
target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
replay buffer R
Gaussian Process dynamics model GP

TRAIN
for *episode = 1, 2, ..., M* **do**
  for *k = 1, ..., 5* **do**
    Initialize random process noise N
    $s_1 = initialize\_environment()$
    while s_1 *is unsafe* **do**
      $s_1 = initialize\_environment()$
    end
    for *t = 1, 2, ..., T* **do**
      $u_{RL} = \mu(s_t|\theta^\mu) + N_t$
      $u_{bar} = barrier\_estimator(s)$
      $u_{CBF} = safety\_barrier(GP, s_t, u_{RL} + u_{bar})$
      $a_t = u_{RL} + u_{bar} + u_{CBF}$ $s_{t+1}, r_t = execute(a_t)$
      Store transition in R
      Sample B random transitions from R
      $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
      Update critic by minimizing L
      Update actor using the sampled policy gradient
      Update the target networks
    end
    Update GP using the episode's transitions
  end
  Train *barrier_estimator* using recorded values
end

controller. The input state is

$$s = [\cos(\theta), \sin(\theta), \dot{\theta}]^T$$

The environment also comes with a built-in reward function, which we used to train all of the agents.

$$r(k) = -(\theta_k^2 + 0.1\dot{\theta}_k^2 + 0.001u_k^2)$$

To add a safety aspect to this environment, we decided that the state would only be considered safe if θ was in the range $[-15°, 15°]$. This safety range was encoded into the CBF controller and used during evaluation to determine if a simulation was safe.

2.4 Neural Network Verifier

The Matlab toolbox for *Neural Network Verification* (NNV)[4] [Tran et al. 2020] uses reachability methods to compute exact and over-approximate reachability sets. For the purpose of this work, we utilize the part of their tool focused on *Neural Network Control Systems* (NNCS) similar to the work done in [Tran et al. 2019].

[4]Code available at https://github.com/verivital/nnv

Due to the nonlinearity of the models being evaluated, the over-approximations in the reach set grow exponentially, so we had to define a small initial set about the upright position. This initial set is used throughout the training explained in the next section.

$$s = \begin{bmatrix} [\cos(15°) & , & 1.0] \\ [\sin(-15°) & , & \sin(15°)] \\ [-0.01 & , & 0.01] \end{bmatrix}, \quad (2)$$

3 EXPERIMENT SETUP

In this work, we look at three different ways of training an agent within the *Pendulum* − *v*0 environment and show results for four evaluations. Each evaluation is performed across three random seeds[5]. The first two random seeds were selected at random, the third is the random seed used in the original work [Cheng et al. 2019]. We will compare sample-complexity results, the traditional way of showing how well an algorithm performs in an environment, and the final trained model will be evaluated 100 times in order to get a more general idea of how well we can expect the learned model to perform if it's deployed. Each episode evaluation halts the training and uses the deterministic output of the learned policy for each step. Each evaluation episode starts the agent from within the initial set (Eq. 2) in order to make the evaluations more consistent. The different evaluation traces are explained with more detail below.

3.1 DDPG

The first evaluation, DDPG, shows how the original DDPG implementation performs with no modifications and no changes made to the environment. When the environment is reset, the angle is randomly selected from a uniform distribution from $[-\pi, \pi]$ radians and the velocity from $[-1, 1]$.

3.2 DDPG-C

The second evaluation, DDPG-C, is a constrained version of the DDPG algorithm. Each training episode is started within the initial set (Eq. 2). This training method was included to see the effect of starting the training episode exclusively within the safe region. The following methods make sure to always start training episodes within the safe region, which could have been an underlying reason for their reported success.

3.3 CBF-N and CBF

Both of these evaluations, CBF-N and CBF, are made using the same trained model. The model is trained using the algorithm outlined in Section 2.2. Each training episode is started within the initial set (Eq. 2) and control barrier functions are configured with the safe region trying to keep the pendulum within $[-15°, 15°]$. The difference in the evaluations is CBF-N is evaluated without the learned barrier estimator and safety barrier while CBF is evaluated with both. This comparison is included to observe how the compensator and safety barrier impact the performance.

[5]The work done in [Henderson et al. 2018] shows that an algorithm must be evaluated using multiple random seeds in order to prove a trend since the performance is often linked directly to the random seed.

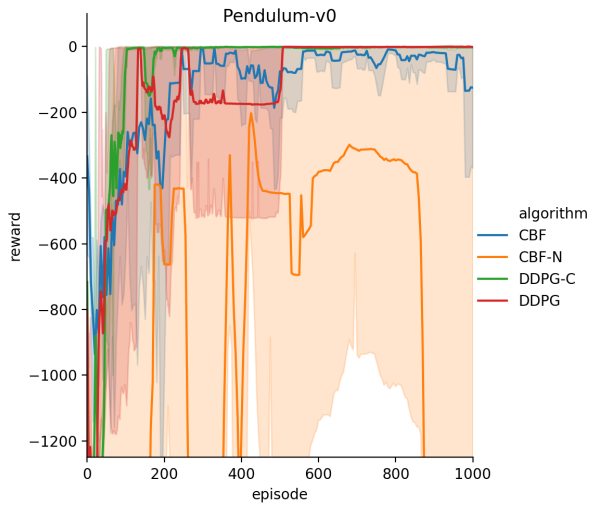

Figure 2: An estimate of the expected reward if the policy network were deployed after training for that many episodes.

3.4 Reachability Evaluation

We used the best performing models of each experiment to compute the reachability analysis. If the best performing model is not verifiably safe, then it is reasonable to believe the others are not as well. As of yet, we cannot evaluate the reachability set of the CBF models. The use of two neural networks to determine the action cannot be replicated in NNV yet.

We converted the final actor model of the selected experiments into a Matlab format used by NNV. These final models were then analyzed alongside the plant model to compute the overapproximate reachability sets after 10 simulated steps using the NNCS **reach** function. Due to some of the nonlinearities in the plant model, the overapproximation quickly explodes after a few steps, so we paired the output with traces of simulated experiments to observe whether or not the reach-sets are reasonable.

4 RESULTS

The following results were collected using the code provided at https://github.com/nphamilton/mlv_2020_project with repeatability instructions provided in the readme.

4.1 Training Results

All of the sample complexity results are shown in Figure 2. This shows the expected reward after each episode of training. The solid line shows the average reward between the three random seeds. The shaded region is the first standard deviation. These results suggest that the CBF algorithm is highly dependent on being deployed with the barrier estimator and safety barrier. Without these, the performance suffers heavily having the worst reported performance with a large variance. The results are discussed more in Section 5.

Table 1: Final Model Evaluated on 100 Runs

Method	Expected Reward	Percent Safe Runs
DDPG_8	-0.487 ± 0.025	100%
DDPG_1964	-0.504 ± 0.019	100%
DDPG_1754	-2.952 ± 0.040	100%
DDPG-C_8	-4.666 ± 0.046	100%
DDPG-C_1964	-0.534 ± 0.011	100%
DDPG-C_1754	-0.647 ± 1.414	100%
CBF-N_8	-1695.5 ± 293.5	0%
CBF-N_1964	-1236.5 ± 18.18	0%
CBF-N_1754	-2.180 ± 0.268	100%
CBF_8	-5.402 ± 0.505	100%
CBF_1964	-87.00 ± 56.19	0%
CBF_1754	-1.417 ± 0.416	100%

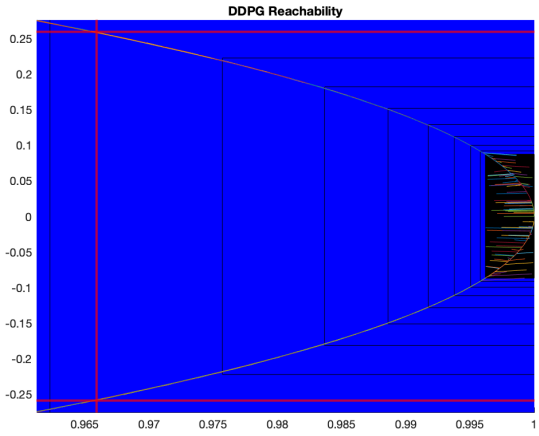

Figure 3: Reachability analysis for the DDPG_8 model represented by boxes representing $\cos(\theta)$ vs $\sin(\theta)$.

4.2 Final Model Evaluation

We evaluated the final learned policy models 100 times to form an estimate of how the systems would respond if deployed in a real system. The results shown in Table 1 suggest that the CBF method does not always produce safe policies, even with the barrier estimator and safety barrier. Both DDPG and DDPG-C produced consistently safe results. The best performing method was DDPG, having the highest expected reward. Additionally, DDPG had the highest average performance across all three random seeds. The worst performing method was CBF-N.

4.3 Reachability Results

We performed the reachability analysis on all three selected models and all of them failed to stay within the safe region for the evaluated 10 steps, i.e. 0.5 seconds. Additionally, the simulated traces also failed to remain within the safe regions, supporting the unsafe finding. However, due to the overapproximations made in the verification process and the inability to put a max speed in the plant

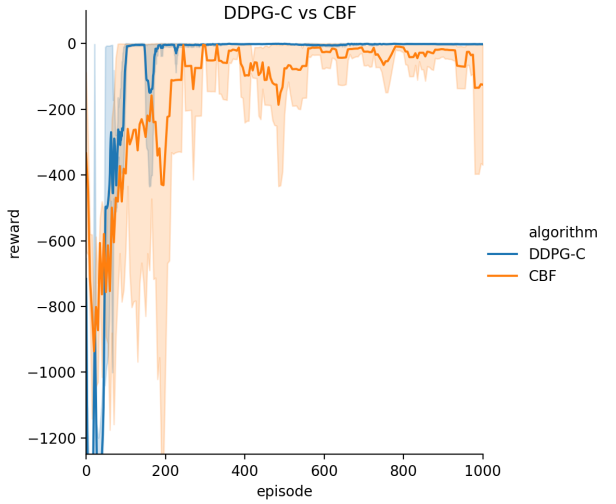

Figure 4: Comparison of learning efficiency between DDPG with initial set constraints and the CBF-guided method.

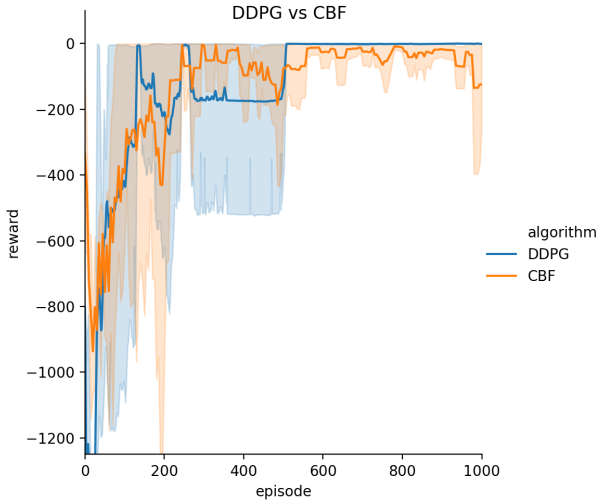

Figure 5: Comparison of learning efficiency between the unsafe and safe methods.

model, we cannot definitively say that these models are unsafe. Upon further inspection, we found that all the results were exactly the same, which might suggest a flaw in our implementation or a limitation in computing the reach set for such a nonlinear system. The reachability result for DDPG is shown in Figure 3 while the other results are included in Appendix B to save space.

In Figure 3, the horizontal axis measures the $\cos(\theta)$ and the vertical axis measures the $\sin(\theta)$. The black rectangle is the initial set and the blue boxes with black borders are the reachable sets. The vertical and horizontal red lines mark the edges of the safe region and the colored lines forming an arch represent 100 simulated traces.

5 DISCUSSION

Our results did not show the same performance benefits of using the CBF-guided methods reported in [Cheng et al. 2019]. Instead, we found that the policy performance was highly reliant on the barrier functions and our experiments suggest the quick convergence was not a result of the CBF-guided method.

In Figure 4, we compare the performance of DDPG modified to start each training episode with the same restrictions used for the CBF-guided method against the CBF-guided performance. Here we see little to no difference in the learning speed as both methods increase performance very quickly and start to converge. It is hard to justify that the addition of CBFs are the cause for the increased learning speed when simply constraining the initial set produces the same improvement.

Furthermore, in Figure 5 we have singled out only the proposed safe method using CBFs and the original RL method to see which is more stable and consistent. The safe method is constantly spiking does not converge to a stable value while the original method converges and maintains a consistent value halfway through training. This suggests that there are some hidden instabilities to using CBFs or the way they're deployed. This is undesirable and could cause strong negative effects if deployed to a safety-critical system. Therefore, we cannot confidently say that the CBF-guided method is safer than using the original, DDPG algorithm.

6 FUTURE WORK AND CONCLUSIONS

This paper presents a single case where the safe method proposed in [Cheng et al. 2019] does not produce safer results than traditional methods, contrary to what was reported in their work. This suggests that other safe reinforcement learning methods may not be tested rigorously enough to make the claim that they are safe.

Future work in this area will involve looking into more cases where safe reinforcement learning algorithms are not safe and possibly create more rigorous benchmark tests and standardized evaluation. These benchmarks and evaluation standards could help shed more light onto which methods are actually safe and can be used in safety critical systems.

7 THINGS I LEARNED

This project has taught me some valuable lessons about goal setting, code standardization, repeatablility, and the trustworthiness of published works.

- Goal setting: I started with a large goal, implementing multiple RL algorithms, training across multiple environments, and comparing all these different methods. Those goals were a bit too much and they did not provide enough "wiggle room" for when things inevitably went wrong. As a result, I panicked and had to reformulate new ideas multiple times to find something that was manageable within a short time frame. I was lucky to find previous work that I was able to rework to fit my original goal, but I need to plan better in the future.
- Code standardization: I started work on a large body of code for this project and used large libraries and tools in order to collect my results. Having to read through all of

the code to figure out what is going on and how to fix the issues I encountered was difficult when the code wasn't well structured. I worked these frustrations into the body of code I developed (https://github.com/nphamilton/rl_library). Each class and method has a rigid structure and uses abstract classes that helps enforce their use. When I encountered issues, this structure really helped me find the bugs and issues.

- Repeatability: I tried to use a lot of other people's work in order to lighten the workload. I focused on papers that were stamped for repeatable results. However, I kept running into issues with dependency errors and confusing structures that made repeating their results next to impossible. I also found a large number of misrepresented results. It has helped solidify my resolve to make all of my future projects easy to repeat and as understandable as I can make them.
- Trustworthiness of published results: As we saw in the results I found in this paper, the claims of the work I used were incorrect. This was a paper I had thought was really good. It was accepted at a good conference and all the methods they described made sense. It should have worked, but we found that it didn't. Therefore, I should be more skeptical in the future.

REFERENCES

Joshua Achiam. 2018. Spinning Up in Deep Reinforcement Learning. (2018).

Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. 2017. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 22–31.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:arXiv:1606.01540

Richard Cheng, Gábor Orosz, Richard M Murray, and Joel W Burdick. 2019. End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3387–3395.

Scott Fujimoto, Herke Van Hoof, and David Meger. 2018. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477* (2018).

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290* (2018).

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

Horia Mania, Aurelia Guy, and Benjamin Recht. 2018. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*. 1800–1809.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International conference on machine learning*. 1889–1897.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv:1707.06347 http://arxiv.org/abs/1707.06347

Hoang-Dung Tran, Feiyang Cai, Manzanas Lopez Diego, Patrick Musau, Taylor T Johnson, and Xenofon Koutsoukos. 2019. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–22.

Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. 2020. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems. In *32nd International Conference on Computer-Aided Verification (CAV)*.

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

Appendices

A HYPERPARAMETERS

DDPG:

- Policy Network: (64, relu, 64, relu, tanh)
- Q Network (64, relu, 64, relu, linear)
- Actor LR: 0.0001
- Critic LR: 0.001
- Noise type: Ornstein-Uhlenbeck Process Noise $\sigma = 0.3$, $\theta = 0.15$
- Soft target update: $\tau = 0.001$
- $\gamma = 0.99$
- Critic L2 reg: 0.01
- buffer size: 10^6
- batch size: $B = 64$
- episode length: $T = 200$
- number of episodes: $M = 1000$

DDPG-CBF:

- Policy Network: (64, relu, 64, relu, tanh)
- Q Network (64, relu, 64, relu, linear)
- Actor LR: 0.0001
- Critic LR: 0.001
- Noise type: Ornstein-Uhlenbeck Process Noise $\sigma = 0.3$, $\theta = 0.15$
- Soft target update: $\tau = 0.001$
- $\gamma = 0.99$
- Critic L2 reg: 0.01
- buffer size: 10^6
- batch size: $B = 64$
- episode length: $T = 200$
- number of episodes: $M = 200$

B OTHER REACHABILITY RESULTS

All of the reachability results took ≈1 hour to compute on a single core.

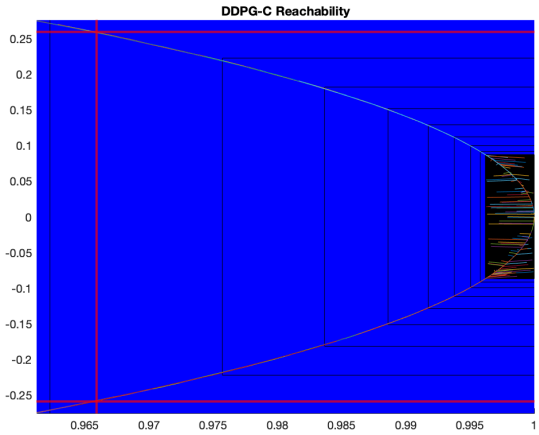

Figure 6: Reachability analysis for the DDPG-C_1964 model represented by boxes representing $\cos(\theta)$ vs $\sin(\theta)$.

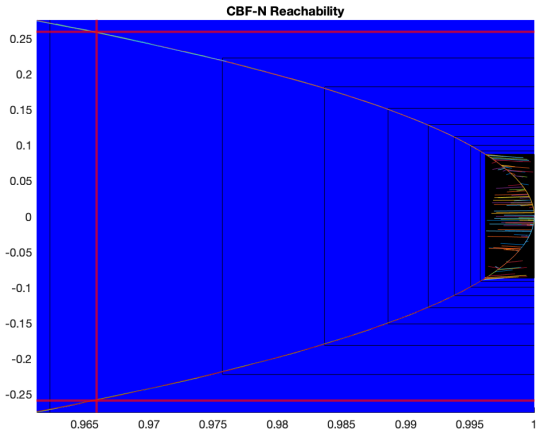

Figure 7: Reachability analysis for the CBF-N_1754 model represented by boxes representing $\cos(\theta)$ vs $\sin(\theta)$.