Policy Gradient Algorithms on Continuous Control Tasks - OpenAI Gym Car Racing Case Study

Yifan Bai
yifan.bai@mail.mcgill.ca

Bhavya Patwa
bhavya.patwa@mail.mcgill.ca

April 30, 2020

Abstract

Policy gradient methods are fundamental to recent breakthroughs in using deep neural networks for control. Yet things such as sensitivity of hyperparameters choices may hinder performances. This project seeks to explore a few popular algorithms on a continuous control task, trying to understand pros and cons and their potential in applications. ***GitHub Repository:*** `code` ***Presentation Video:*** `video`

1 Introduction

The combination of Deep Learning and Reinforcement Learning have been successful at solving complicated riddles and extracting nonlinear features from high dimensional media such as images. This enabled us to combine the best of both worlds. On the one hand, Deep Learning enable accurate computations of large amount of data and flexibility to adapt; on the other hand, Reinforcement Learning enabled us to construct models that represent real-world continuous tasks and allow incorporating temporal aspects. Named as 'Deep Reinforcement Learning'(DRL), this burgeoning field gathered our interest and is the main motivation for this project.

Policy gradient methods are a type of reinforcement learning techniques that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward). They do not suffer from many of the problems that have been marring traditional approaches such as the lack of guarantees of a value function, the intractability problem resulting from uncertain state information and the complexity arising from continuous states actions. For this project, we analyzed 3 Policy Gradient algorithms, namely *PPO (Proximal Policy Optimization)* Schulman, Wolski, Dhariwal, Radford, and Klimov (2017), *DDPG (Deep Deterministic Policy Gradient)* and *TD3 (Twin Delayed DDPG)* Fujimoto, van Hoof, and Meger (2018) on a continuous control task – OpenAI Gym's ***CarRacing-v0***. We also explored different hyperparameters trying to improve performance.

1.1 Model Description

CarRacing-v0 Brockman et al. (2016) is a game engine developed by OpenAI Gym, part of continuous control tasks to lean from pixel images, in Box2D simulator environments. It also allows on/off discrete controls. We modeled using Markov Decision Process. The Observation Space is a set of $96 \times 96 \times 3$ vectors that represents the window of images being fed to the model (width-height-channel). The Action Space is a set of 3D vectors of *[steering angle, throttle, brake]* representing inputs made by the agent. Computing 2D curvature-based accelerations, gyroscopes will determine if ABS is activated to stabilize driving.This means aggressive driving, such as braking under turning or carrying too much speed mid-corner could result in loss of control. Reward is -0.1 for every frame and $+1000/N$ for every track tile visited, with N being total number of tiles in one lap. Episode either finishes when all tiles are completed, or terminates with a die state if the car goes off track too far, with a reward of -100.

1.2 Algorithms

Every Reinforcement Learning algorithm must follow some policy in order to decide which actions to perform at each state. Still, the learning procedure of the algorithm does not have to take into account that policy while learning. Algorithms which concern about the policy which yielded past state-action decisions are referred to as on-policy algorithms, while those ignoring it are known as off-policy. As an on-policy algorithm, PPO Berseth (2018) is $\epsilon - greedy$ and uses Gaussian exploration. It also uses Stochastic Gradient Descent during optimization, which makes it parallelizable. However, it does not account for how exploration is performed, and loss can change siginificantly between updates. As off-policy algorithms, DDPG Berseth (2018) uses Ornstein-Uhlenbeck process to explore and Q-function to learn actions. However, its performance hinges on Q-function's quality, which tends to overestimate q-values; TD3 *Twin Delayed DDPG* (2018) is an improved version of DDPG which incorporates clipped double learning, delayed policy updates and target policy smoothing, with the downside being more computationally expensive.

Core to DRL is the usage of neural networks, manifested in policy gradient problems via Actor-Critic. The "Critic" estimates the value functions (Action - Q value or State - V). The "Actor" updates the policy distribution in the direction suggested by the Critic. With DRL, they are parameterized with neural networks whose architectures are shown in Tables 1 2 in the Appendix. Detailed implementations of this method will be illustrated in the following section.

2 Methodology and Results Discussions

To compare the performances, we standardized the process that attempt to balance between meaningful understanding and appropriate computation times, given the limitation of Google Colab. Each experiment was ran with 1000 episodes and 5 random seeds were averaged for reproducibility purposes.

2.1 PPO

PPO attempts to balance ease of implementation, sample complexity and tuning by tring to compute an update at each step to minimize cost function while controlling deviation from previous policy. Specifically, a novel implementation by OpenAI implements a Trust Region update that is compatible with SGD, and simplifies things by removing KL penalty and need to make adaptive updates John Schulman (2017):

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)A_t, clip(r_t(0, 1-\epsilon, 1+\epsilon)A_t)]$$

Information of our best model is provided in Table 3 in Appendix. From our results, we can see that the algorithm could indeed achieve learning with certain settings. In terms of reward and ending an episode, customized with additional reward of 100 if the episode ends and an additional -0.05 if the state space image has high amount of green, i.e. the car runs off the track and is unlikely to rejoin under a reasonable number of frames. Episode ends if reward remains -0.1 for 100 consecutive actions. Using best model, we could see significant improvements very quickly. However, there are significant noise throughout due to the inter-update variations in policies.

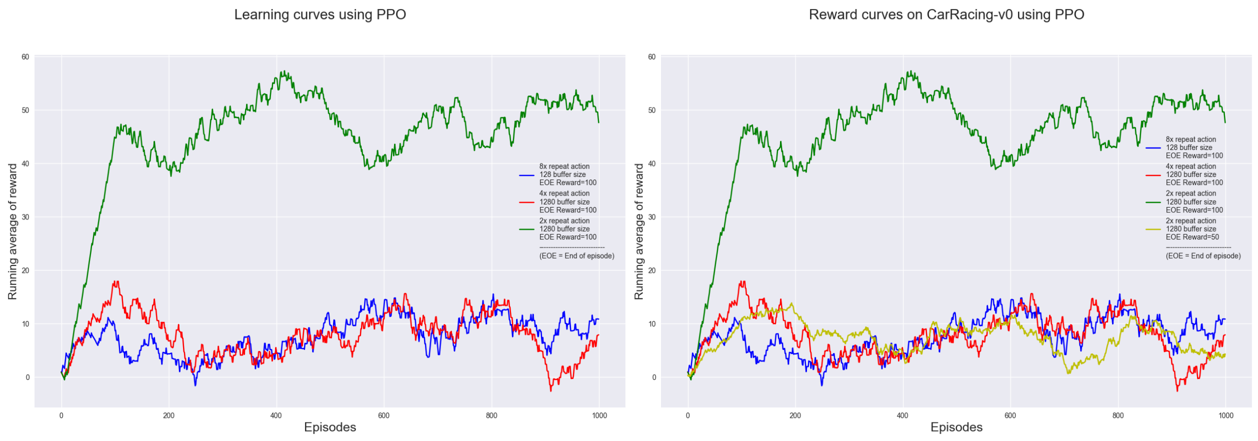

Figure 1: Running Reward Curve, *Left*: DDPG *Right*: TD3

Referring to right part of Fig 1, we can see that changing end of episode (EOE) reward plays huge role in reinforcing the action of the model during training. For example, by simply increasing from 50 to 100 and using repeat actions in 2 frames, we see a significant increase reward (green line). In addition, decreasing the number of times an action is repeated help us reach a reward with a running average of 50, as shown in the left part, where green curve shows using 2 repeated actions, blue using 8 and red using 4. This enables the agent to avoid carrying out 'mistakes' for too long and could save itself in situations such as a slow corner followed by a long straight where it needs to orchestrate braking, steering and thorttling. Repeating the same action could cause the agent to miss the corner and go off track. Last but not least, taking a high buffer size (1280 vs 128, i.e. 10 times batch size) also helps model train on a large variety of data.

2.2 DDPG and TD3

DDPG Yoon (2019) uses four neural networks: a Q network θ^Q, a deterministic policy network θ^μ, a target Q network $\theta^{Q'}$, and a target policy network $\theta^{\mu'}$. Actor directly maps states to actions instead of outputting the probability distribution across a discrete action space. The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning.

During learning, it uses a replay buffer to sample experience to update neural network parameters. During each trajectory roll-out, we save all the experience tuples (state, action, reward, next state) and store them in a finite sized replay buffer. Then, we sample random mini-batches of experience from the replay buffer when we update the value and policy networks. In optimization asks, we want the data to be independently distributed. This fails to be the case when we optimize a sequential decision process in an on-policy way, because the data then would not be independent of each other. When we store them in a replay buffer and take random batches for training, we overcome this issue. During Actor-Critic network updates, the next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the updated Q value and the original Q value:

$$Loss = \frac{1}{N}\sum_i (y_i - Q(S_i, a_i|\theta^Q))^2$$

For the policy function, our objective is to maximize the expected return. To calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter. Since we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch.

$$\Delta_{\theta^\mu} J(\theta) \approx \frac{1}{N}\sum_i [\Delta_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \Delta_{\theta^\mu}(\mu^{s|\theta^\mu})|_{s=s_i}]$$

To update target network, we make a copy of parameters and have them slowly track those of learned networks.

$$\theta^{Q^{'}} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q^{'}}$$

$$\theta^{\mu^{'}} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu^{'}}$$

Where τ is usually 0.005 by default. For discrete action spaces, exploration is done via probabilistically selecting a random action. For continuous action spaces, exploration is done via adding noise to the action itself. DDPG uses Ornstein-Uhlenbeck Process, a version of such process Finch (2004).

TD3 is an improved version of DDPG with 3 major differences Bryne (2019). Firstly, it uses 2 separate critic networks and uses the smallest value of the two when forming targets. This provides an underestimation prevents bias from propagting through the algorithm, increasing stability. Secondly, it uses deplayed updates of the actor network, only updating it every 2 time steps instead of after each time step, resulting in more stable and efficient training. This will prevent us from sampling bad policies where avoidable, reducing errors and variance in value estimates. Thirdly, TD3 smoothes target policy via noise regularization to prevent overfitting from deterministic policies which produces high variance in target values. Clipped noise is added to the selected action when calculating the targets.

The implementation of these algorithms referred to a paper [ref]. In addition, we also attempted adding reward criteria. To prevent being stuck off-track and on grass, the agent is given a reward of -0.02 for each frame of 4-wheel off-track. Episode terminates when there is no significant gain in reward (-0.1 between frames). However, due to the fact that we cannot obtain a good enough Q function to represent the action space properly and incorporate complex changes, neither algorithm produced satisfatory results. Due to limitation of computation resources and to extend the comparisons for PPO, we only investigated the number of frames with repeated actions to see if off-policy algorithms could update action space well. From local machine rendering, the car showed tendency of entering an action state and unable to correct it. Note that a maximum of 1000 episodes were ran if possible, however, Google Colab forced shutdown of runtime due to out-of-memory issues.

As Fig 2 shows, DDPG had its best results when 4 frames of repeated actions, and worst with 8. This is in contrary to PPO, and could be explained by the fact that the ability to learn is bounded by quality of Q-function, which may overestimate Q-values. In this case, it is not an advantage to have as less repeated actions as possible. TD3 showed the same trend, and it was unable to complete the experiment due to hardware limits, which confirms our guess of it being compuationally expensive. It is also notable that at worst settings, both algorithm converge at a negative reward of around -17. The detailed reason needs to be investigated more thoroughly

Parameters of the best models used is available in Table 3 in the Appendix. For DDPG, the running average of rewards stabilizes at around 0 and for TD3, -7.6, even though the machine shut off at Episode 755. Both algorithms showed high noise and the rewards per-episode vary significantly, suggesting that we need to investigate further into our models as to why it failed to learn properly. For all cases, resources permitted, we should train with lots more episodes and investigate thoroughly into hyperparameters to find better comibnations.

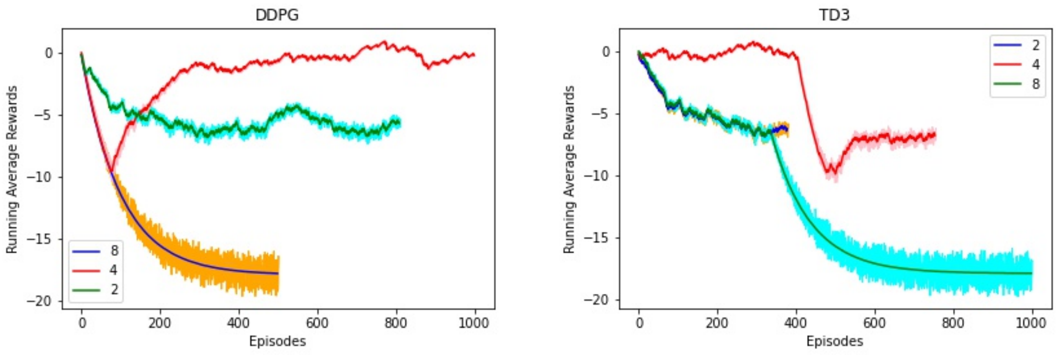

Figure 2: Reward Curve, *Left*: DDPG *Right*: TD3. Solid line represents and the filled represent standard deviation

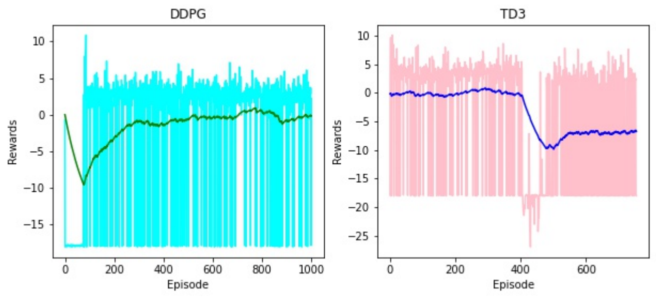

Figure 3: Best Models, Per-episode(Light Line) and Running Average Rewards(Dark Line), *Left*: DDPG *Right*: TD3.

3 Conclusions and Recommendations

To conclude, this project benchmarked 3 popular DRL policy gradient algorithms that uses Actor-Critic on a continuous control task. The on-policy PPO was the best performing in terms of rewards. However, it also showed the most significant noise as it falls short on exploration. Off policy algorithms, in its most baseline versions, showed limitations on its reliance of Q-function. While for simpler tasks this may be less problematic, for a complex problem like this, its tendency of over-estimating the Q-value contributed to poor performance. TD3, its sibling, suffered similar problems and did not show the expected improvements, and its computation cost is higher. The Twin Critic aspect did not help, hypothetically also due to the limited performance of Q-functions applied to this problem. However, all three algorithms share one trait: increasing number of repeated actions frames deteriorate performance.

There are a few suggestions and recommendations. Firstly, resource permitted, we should run the model with many more episodes and conduct a thorough hyperparameter search, such as learning rate, as well as trying different neural network techniques, including normalization and regulation techniques to improve function approximation. State-of-the-art models published by renowned institutes are mostly trained with over 100 000 episodes. Secondly, we should try engineering the action space, by one way incorporating inspirations from real-world human driving techniques, such as maximizing the total G-force allowed on the car imposed by inputs to prevent over-driving, or the other discretizing the action space via pre-defining strategy sets. For example, via image inputs, we could try to compute the most plausible actions, such as heavy braking after a long straight, and refer back to the strategy dictionary to enforce actions, such as no steering/throttle while braking. Another potential benefit of doing so is saving computing times. We could also set a maximum on rewards for one episode to restrict the car's max speed to prevent losing control, which would be particularly important in early training phases to prevent the car from getting stuck in one failure. Last but not least, we could try explore adding different reward conditions. In this project, we implemented penalties for going off track. We could extend on this to improve training.

4 Statement of Contribution

Both team members contributed equally. Each team member worked independently on code (Bhavya for PPO, Yifan for DDPG/TD3), through which code reviews were done consistently. Yifan wrote the report and Bhavya made the slides, followed by proof-reading from both. Due to technical limitations, presentation video was commentated by Yifan only.

References

Berseth, G. (2018). *Demystifying the many deep reinforcement learning algorithms.* Retrieved from `https://www.cs.ubc.ca/~gberseth/blog/demystifying-the-many-deep-reinforcement-learning-algorithms.html`

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *Openai gym.*

Bryne, D. (2019). *Td3: Learning to run with ai.* Retrieved from `https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93`

Finch, S. (2004). *Ornstein-uhlenbeck process.*

Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. *CoRR, abs/1802.09477*. Retrieved from `http://arxiv.org/abs/1802.09477`

John Schulman, F. W. P. D. A. R., Oleg Klimov. (2017). *Proximal policy optimization.* Retrieved from `https://openai.com/blog/openai-baselines-ppo/#ppo`

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR, abs/1707.06347*. Retrieved from `http://arxiv.org/abs/1707.06347`

Twin delayed ddpg. (2018). Retrieved from `https://spinningup.openai.com/en/latest/algorithms/td3.html`

Yoon, C. (2019). *Deep deterministic policy gradients explained.* Retrieved from `https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b`

5 Appendix

5.1 Model and Results Summary

Layer	Output Size
Input	[96, 96, 4]
Conv2d(args.img$_stack, 8, kernel_size = 4, stride = 2)$	
ReLU()	
Conv2d(8, 16, kernel$_size = 3, stride = 2)$	(8, 47, 4)
ReLU()	
Conv2d(16, 32, kernel$_size = 3, stride = 2)$	(16, 23, 23)
ReLU()	
Conv2d(32, 64, kernel$_size = 3, stride = 2)$	(32, 11, 11)
ReLU()	
Conv2d(64, 128, kernel$_size = 3, stride = 1)$	(64, 5, 5)
ReLU()	
Conv2d(128, 256, kernel$_size = 3, stride = 1)$	(128, 3, 3)]
ReLU()	
Flatten()	[256, 1, 1]
Value	**Action**
Linear(256, 100)	Linear(256, 100)
Relu()	Relu()
Linear(100, 1)	Alpha/Beta, Linear(100, 3)
	Relu()

Table 1: PPO Actor-Critic Network Architecture

Layer	Output Size
Input	[96, 96]
Conv2d(img$_stack, 16, 5, 2, padding = 2)$	[16, 48, 48]
ReLU()	
BatchNorm2d(16)	
Conv2d(16, 32, 5, 2, padding=2)	[32, 24, 24]
ReLU()	
BatchNorm2d(32)	
Conv2d(32, 64, 5, 2, padding=2)	[64, 12, 12]
ReLU()	
BatchNorm2d(64)	
Conv2d(64, 128, 5, 2, padding=2)	[128, 3, 3]
ReLU()	
BatchNorm2d(128)	
Conv2d(128, 256, 5, 2, padding=2)	[256, 3, 3]
ReLU()	
BatchNorm2d(256)	
Conv2d(256, 512, 5, 2, padding=2)	[512, 1, 1]
Flatten()	
Output	
Linear(512, 30)	
Relu()	
Linear(30, 3)	
Tanh()	

Table 2: DDPG/TD3 Actor-Critic Network Architecture

Algorithm	**Learning Rate**	**Repeated Actions**	**Batch Size**	**Buffer Size**
PPO	0.001	2	128	1280
DDPG/TD3	0.001	4	100	-

Table 3: Best Hyperparameters

5.2 Reproducibility Checklist

This list is made in compliance with provided for this course, tailored specifically for experimental projects. Computation was done on Google Colab GPUs.

The Machine Learning Reproducibility Checklist (v2.0, Apr.7 2020)

For all **models** and **algorithms** presented, check if you include:

- [x] A clear description of the mathematical setting, algorithm, and/or model.
- [x] A clear explanation of any assumptions.
- [x] An analysis of the complexity (time, space, sample size) of any algorithm.

For any **theoretical claim**, check if you include:

- [ ] A clear statement of the claim.
- [ ] A complete proof of the claim.

For all **datasets** used, check if you include:

- [ ] The relevant statistics, such as number of examples.
- [ ] The details of train / validation / test splits.
- [ ] An explanation of any data that were excluded, and all pre-processing step.
- [ ] A link to a downloadable version of the dataset or simulation environment.
- [ ] For new data collected, a complete description of the data collection process, such as instructions to annotators and methods for quality control.

For all shared **code** related to this work, check if you include:

- [x] Specification of dependencies.
- [x] Training code.
- [x] Evaluation code.
- [ ] Pre-trained model(s).
- [x] README file includes table of results accompanied by precise command to run to produce those results.

For all reported **experimental results**, check if you include:

- [x] The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results.
- [x] The exact number of training and evaluation runs.
- [x] A clear definition of the specific measure or statistics used to report results.
- [x] A description of results with central tendency (e.g. mean) & variation (e.g. error bars).
- [ ] The average runtime for each result, or estimated energy cost.
- [x] A description of the computing infrastructure used.

Reproduced from: *www.cs.mcgill.ca/~jpineau/ReproducibilityChecklist-v2.0.pdf*