Policy Gradient RL Algorithms as Directed Acyclic Graphs

Juan Jose Garau Luis
Massachusetts Institute of Technology
Department of Aeronautics and Astronautics
garau@mit.edu

Abstract

Meta Reinforcement Learning (RL) methods focus on automating the design of RL algorithms that generalize to a wide range of environments. The framework introduced in [1] addresses the problem by representing different RL algorithms as Directed Acyclic Graphs (DAGs), and using an evolutionary meta learner to modify these graphs and find good agent update rules. While the search language used to generate graphs in the paper serves to represent numerous already-existing RL algorithms (e.g., DQN, DDQN), it has limitations when it comes to representing Policy Gradient algorithms. In this work we try to close this gap by extending the original search language and proposing graphs for five different Policy Gradient algorithms: VPG, PPO, DDPG, TD3, and SAC.

1 Introduction

Novel Reinforcement Learning (RL) algorithms are recurrently being proposed to address a wide range of performance gaps. Each of these algorithms involves a human-driven design process that does not scale well. In order to reduce the design cost, the research community is focusing on meta RL algorithms, which automate the process of developing new policy update rules that generalize to multiple unseen environments.

Broadly, meta RL methods consider a double-loop algorithm in which an agent attempts to learn good policies for arbitrary environments in an inner loop and a meta learner attempts to produce good update rules in an outer loop. The inner agent uses a loss function provided by the meta learner, whereas the meta learner uses the agent's return to guide its search.

Among all the meta RL methods that are currently being studied, in this work we focus on the framework presented in [1], in which authors introduce a search language with 26 operators used by the meta learner to form different update rules by connecting multiple operators and constructing a computation graph in the form of a Directed Acyclic Graph (DAG). The meta learner explores the space of graphs by means of evolutionary strategies and keeps track of a population with the best loss functions. The goal of the meta learner is to find the best RL algorithm given by loss function L^*, defined as

$$L^* = \arg\max_{L} \left[\sum_{\varepsilon} \text{Eval}(L, \varepsilon) \right] \tag{1}$$

where each ε belongs to a set of different training environments, all of them used to evaluate the agent's performance, defined in turn as

$$\text{Eval}(L, \varepsilon) = \frac{1}{M_\varepsilon} \sum_{m=1}^{M_\varepsilon} \frac{R_m - R_{min}}{R_{max} - R_{min}} \tag{2}$$

where R_m corresponds to the return during episode m, M_ε is the total number of episodes using environment ε, and R_{min} and R_{max} are the minimum and maximum possible returns for the environment, respectively. This method can start a search from scratch or use existing algorithms as warm-starts.

The authors test their approach using a set of training environments consisting of 4 classical control tasks from OpenAI Gym [2] and 12 multitask gridworld styile environments from MiniGrid [3]. They use 4 of these environments to meta train and 12 to meta test, and the DQN loss function [4] as an input algorithm. They find two improved versions of the DQN loss function; one of them, defined as DQNReg, outperforms both DQN and DDQN [5] in 15 out of the 16 environments. DQNReg is then tested on 4 Atari environments [6], where it is also shown to outperform PPO [7].

One of the salient conclusions of the paper is that representing RL algorithms as DAGs is a good way to obtain interpretable solutions. The authors reinforce this idea by including a comprehensive analysis of two of the novel update rules found by their framework based on their graph structure. While it is a promising research direction, the search language proposed in the original paper is not complete enough to build on other types of RL algorithms, such as Policy Gradient algorithms [8]. In this work we try to close this research gap by proposing extensions to their framework and representing five different Policy Gradient algorithms as DAGs: VPG, PPO, DDPG, TD3, and SAC.

2 Related work

Representing algorithms as computational graphs is also proposed in other studies. In [9] the same concept is used to find appropriate image classifiers. This idea is also present in [10], where a meta learner produces curiosity graphs that drive the agent's exploration by shaping the environment's reward. The agent then uses the modified reward with a fixed update rule. Conversely, the evolutionary strategy in [1] allows flexible update rules and is meta trained on a diverse set of environments.

As oposed to changing the form of RL algorithms, other studies consider a fixed-form update rule which takes in parameters provided by the meta learner. It is the case of [11], where authors propose an evolutionary strategy to learn good policy update parameters $\hat{\pi}$ that guide the policy optimization. They train the meta RL framework on several MuJoCo tasks [12] and test on similar versions of those tasks. The same train and test procedure is followed by the authors in [13], although a meta-gradient descent strategy is adopted instead of evolutionary methods. This approach is also followed by the authors in [14], which propose MetaGenRL to attain higher generalization capabilities. Those are proven in test environments based on unseen MuJoCo tasks.

In addition to learning appropriate policy update rules $\hat{\pi}$, the method presented in [15], defined as Learned Policy Gradient, also learns a prediction update rule $\hat{y}$, i.e., the semantics of the agent's prediction. In some experiments, the discovered semantics converge towards a notion of value function. The authors claim LPG is able to attain better generalizability this way, as proven on the performance on unseen Atari test environments after meta training on toy environments.

3 Methods

The focus of this work is to extend the applicability of the framework presented in [1] by adapting the search language to Policy Gradient algorithms [8]. First, we present a high-level description of the method from the original paper, summarized in Figure 1. Then, we introduce our extensions to the search language.

3.1 General Description

The input to the algorithm consists of a set of training environments, one additional "hurdle environment", and a search language that conforms the nodes of the computational graphs. The algorithm starts by creating a population of N [1] RL algorithms or loss functions, which can be constructed from scratch using the search language or from an input already-existing RL algorithm. The starting population is scored in each of the training environments.

[1] This symbol is defined by us, it is not in the original paper

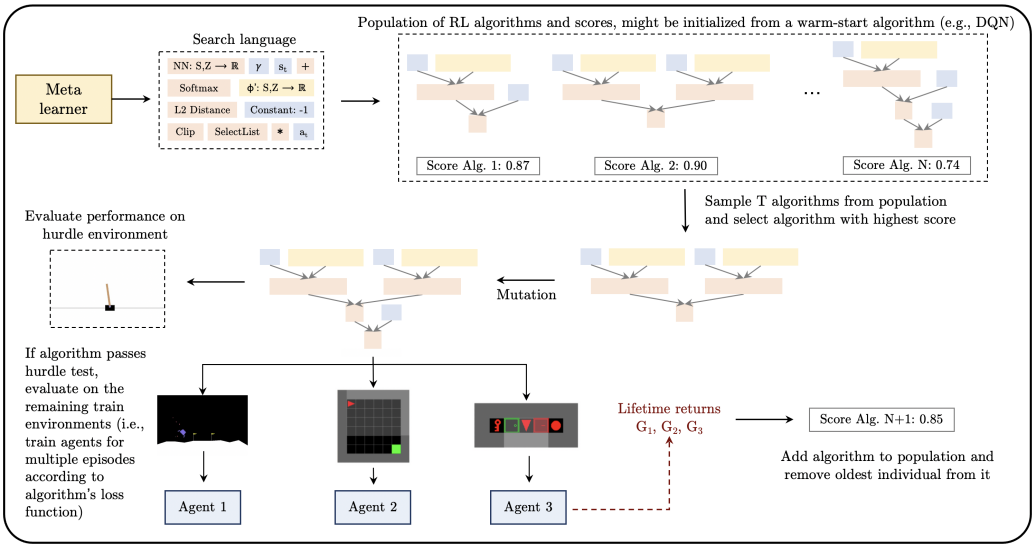

Figure 1: Evolutionary Reinforcement Learning method [1] overview. The meta learning framework constructs a population of RL algorithms or loss functions from a predefined search language, randomly creating each one or using an already-existing algorithm as a warm-start. For a fixed number of iterations, the best RL algorithm from a random set of T algorithms undergoes mutation and a child algorithm is generated. This new algorithm is evaluated, first on a hurdle environment, and then on a set of training environments. The meta learner collects the returns from each environment and computes the score of the new algorithm. The oldest algorithm from the population is then removed.

For a defined number of C iterations, the meta learner first samples a set of T algorithms from the population and selects the one with the highest score as a parent algorithm. Then, a child algorithm is created by mutating the parent algorithm. The hurdle environment is used to assess a minimum performance threshold α, which determines whether the child algorithm is considered for the population or not. If it passes the threshold, the child algorithm is evaluated on each of the training environments by independent agents, and an aggregated score is generated. The new algorithm is added to the population, which is kept to a constant size of N individuals by removing the oldest one.

3.2 Search Language Extensions

The framework expresses RL algorithms as DAGs of nodes with typed inputs and outputs. The input nodes consist of the elements in the tuple (s_t, a_t, r_t, s_{t+1}), the network parameters θ, the target network parameters θ', the discount factor γ, and other numerical constants (e.g., 1, 0.1). Other types of nodes include parameter nodes, which encode neural network operations, and operation nodes, which perform scalar and list operation over the inputs (e.g., *Add*, *MaxList*). A full list of the available nodes can be found in the appendix of the original paper [1]. For reference, Figure 2 shows the DDQN algorithm [5] encoded as a computational graph using the *original* set of nodes from the paper.

The output of the computational graph, given by the node *Output*, is then used as the loss function L_θ the agent must minimize, updating its parameters by gradient descent $\theta \longleftarrow \theta - \alpha_{lr} \nabla_\theta L_\theta$. We follow this gradient descent convention throughout the rest of this work.

We now introduce each of the extensions to the original language, divided into new input nodes, new operation nodes, and new constant nodes. We follow the data type notation convention proposed by the authors, which includes state $\mathbb{S}$, action $\mathbb{Z}$, float $\mathbb{R}$, and list $List[\mathbb{X}]$, where $\mathbb{X}$ indicates it can be of $\mathbb{S}$ or $\mathbb{R}^2$.

[2]Notation extracted from the original paper [1]

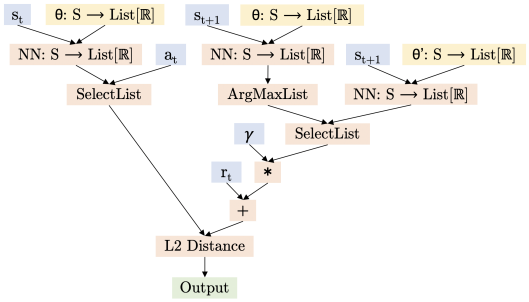

Figure 2: Computational graph representing DDQN loss function $L_\theta = \left(Q_\theta\left(s_t, a_t\right) - \left(r_t + \gamma Q_{\theta'}\left(s_{t+1}, \arg\max_a Q_\theta\left(s_{t+1}, a\right)\right)\right)\right)^2$.

3.2.1 Input nodes

In the paper it is assumed each independent agent initializes Q-value parameters θ, target network parameters θ', and an empty replay buffer $\mathcal{D}$, where the agent stores all the transition tuples encountered in order to do off-policy learning. To adapt the framework to Policy Gradient algorithms, we propose the following extensions:

- We register the episode-termination signal d_t in the replay buffer $\mathcal{D}$.
- We add the parameter λ [8] to the language.
- We allow the tuple $(s_t, a_t, r_t, d_t, s_{t+1})$ to represent a sequence of consecutive timesteps, instead of a single timestep. According to the original paper, operations broadcast in the cases in which inputs have a different number of dimensions.
- We use additional network parameters ϕ (e.g., for the value network when needed) and their target counterparts ϕ'.
- In addition to the network parameter nodes from the original paper, which allow mappings from states $\mathbb{S}$ to floats $\mathbb{R}$ and lists $List[\mathbb{R}]$, we also allow to map both a state $\mathbb{S}$ and an action $\mathbb{Z}$ to a float $\mathbb{R}$, i.e., $\theta : \mathbb{S}, \mathbb{Z} \longrightarrow \mathbb{R}$. Likewise, we introduce the parameter node $NN : \mathbb{S}, \mathbb{Z} \longrightarrow \mathbb{R}$ to carry out operations of such kind. We do this in order to address *continuous* state and action spaces, as required by certain algorithms, such as DDPG [16].

3.2.2 Operation nodes

Apart from the operation nodes proposed in the original paper, we introduce the additional nodes:

- *SumAndDiscount*: It takes in a vector v of length l_v and a numerical value b, and outputs a vector v' of length l_v where each element is computed as $v'_i = \sum_{k=i}^{l_v} v_k \cdot b^{k-i}$. This is necessary to compute certain elements of different Policy Gradient methods, such as computing the rewards-to-go (in that case, $v = r$ and $b = 1$).
- *Clip*: It clips a scalar or vector component-wise given minimum and maximum float values.
- *Squashing*: Given a state s_t from data type $\mathbb{S}$ and network parameters $\theta : \mathbb{S} \longrightarrow \mathbb{R}$, it computes actions $\tilde{a}$ following a squashed Gaussian policy, i.e., $\tilde{a} = \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t) \odot \xi)$, with $\xi \sim \mathcal{N}(0, I)$. This node is specifically introduced to represent the Soft Actor-Critic algorithm [17].
- *Prob*: In cases in which the action space is continuous, and therefore the node *Softmax* can not be used, this node computes the probability of taking a certain action in a given state with given network parameters θ. It assumes the probabilities can be computed, such as in the case of Gaussian or categorical policies.

3.2.3 Constant nodes

Finally, we encode useful constant values as new constant nodes for the language. These are general values, such as -1, or algorithm-specific constants, such as $1+\epsilon$ and $1-\epsilon$, for the example case of PPO. We introduce all algorithm-specific constant nodes in their related sections.

4 Example Graphs

Based on the language extensions introduced in the previous section, we propose computational graphs in the form of DAGs for the following Policy Gradient algorithms: Vanilla Policy Gradient (VPG), Proximal Policy Optimization (PPO) [7], Deep Deterministic Policy Gradient (DDPG) [16], Twin Delayed DDPG (TD3) [18], and Soft Actor-Critic (SAC) [17]. We base our graph constructions on the descriptions and implementations provided in [19]. For all algorithms, we leave the target parameters update considerations (i.e., $\theta' \longleftarrow \theta$, $\phi' \longleftarrow \phi$) for the framework user to decide.

4.1 Vanilla Policy Gradient (VPG)

The work in [1] only considers off-policy algorithms that sample batches of tuples (s_t, a_t, r_t, s_{t+1}). In this work we also allow s_t to represent a sequence of consecutive states. We apply this idea to construct a computational graph for the VPG algorithm that can be used as an input to the meta learning framework. We need to consider two different loss functions: one for the policy π_θ and another for the value function V_ϕ.

First, Figure 3 shows the computational graph for the policy loss, in which we assume the use of GAE-Lambda advantage estimation method [8] to compute the discounted advantage for each timestep. In this example, we introduce the novel node *SumAndDiscount* and a constant node with the value -1, in order to follow the loss minimization convention from the original paper.

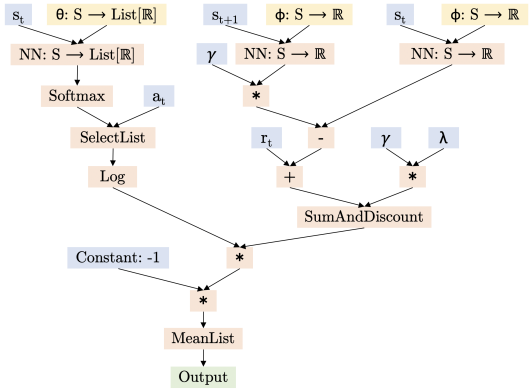

Figure 3: Computational graph representing Vanilla Policy Gradient (VPG) policy π_θ loss $L_\theta = -\frac{1}{T}\sum_{t=0}^{T} \log \pi_\theta\left(a_t|s_t\right) \hat{A}_t$.

Then, Figure 4 shows the computational graph for the value function loss, which computes the MSE between the value function and the rewards-to-go.

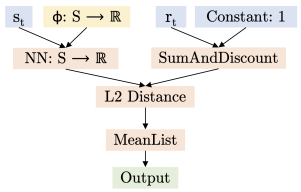

Figure 4: Computational graph representing Vanilla Policy Gradient (VPG) value function V_ϕ loss $L_\phi = \frac{1}{T}\sum_{t=0}^{T}\left(V_\phi(s_t) - \hat{R}_t\right)^2$.

4.2 Proximal Policy Optimization (PPO)

In the case of the PPO algorithm we consider the policy parameters θ as well as an early version θ_k to compute the policy change ratio. Figure 5 shows the computational graph for this algorithm, which includes the novel node *Clip*, as well as the constant nodes $1+\epsilon$ and $1-\epsilon$, input to the clipping function. We use the same advantage estimation method used in the VPG case, and also consider the same value function update method, already presented in Figure 4.

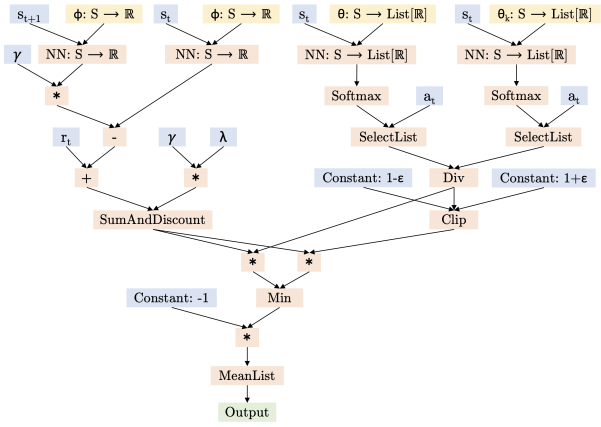

Figure 5: Computational graph representing Proximal Policy Optimization (PPO) policy π_θ loss $L_\theta = -\frac{1}{T}\sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}\hat{A}_t, \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)$.

4.3 Deep Deterministic Policy Gradient (DDPG)

Next, we examine the DDPG algorithm, which in the original paper [16] was proposed to address continuous action spaces. As explained in the previous section, we adapt the framework to satisfy continuity requirements by introducing the parameter node $NN : \mathbb{S}, \mathbb{Z} \longrightarrow \mathbb{R}$, which accounts for a Q-function taking a state and a continuous action as inputs. DDPG algorithm considers both a policy μ_θ and a Q-function Q_ϕ, as well as their target counterparts, $\mu_{\theta'}$ and $Q_{\phi'}$, respectively. Figure 6 shows the computational graph to update the Q-function parameters ϕ, which makes use of the episode-termination signal d_t, also introduced in this work.

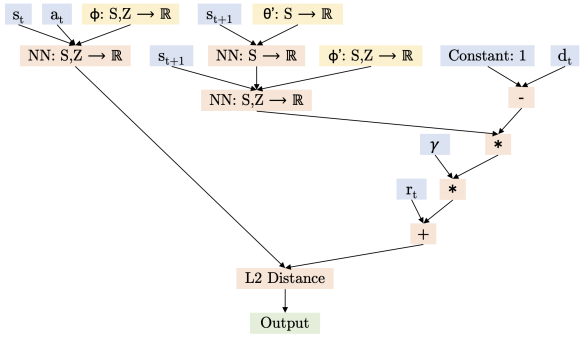

Figure 6: Computational graph representing Deep Deterministic Policy Gradient (DDPG) Q-function Q_ϕ loss $L_\phi = \left(Q_\phi\left(s_t, a_t\right) - \left(r_t + \gamma\left(1-d_t\right)Q_{\phi'}\left(s_{t+1}, \mu_{\theta'}\left(s_{t+1}\right)\right)\right)\right)^2$.

Then, Figure 7 shows the computational graph to update policy parameters θ.

4.4 Twin Delayed DDPG (TD3)

The TD3 loss functions can be constructed with all input, parameter, and operation nodes presented so far. However, new constant nodes are needed for these functions. Figure 8 shows the computational

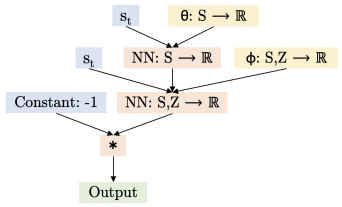

Figure 7: Computational graph representing Deep Deterministic Policy Gradient (DDPG) policy μ_θ loss $L_\theta = -Q_\phi\left(s_t, \mu_\theta\left(s_t\right)\right)$.

graph to update the Q-function parameters ϕ_i. In the TD3 algorithm, clipped noise is added to the actions as a regularizer. In the graph, this is achieved by using input nodes with constant values $-c$ and c, as well as a constant ϵ sampled from the normal distribution $\mathcal{N}(0, \sigma)$. To make sure actions are kept within bounds, additional constant nodes with action bounds a_{low} and a_{high} are used.

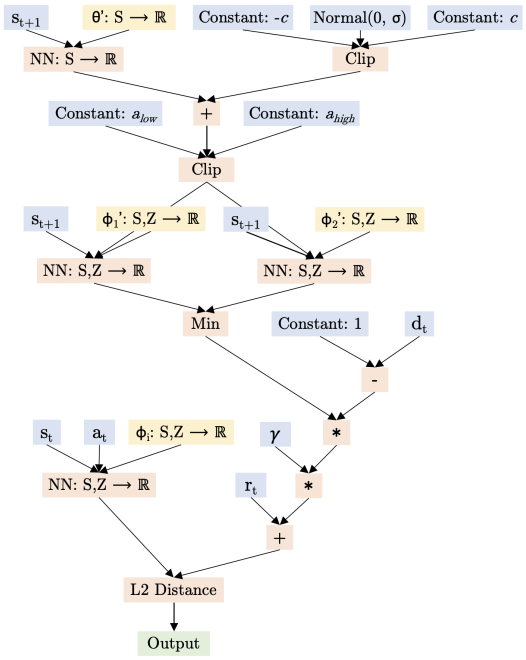

Figure 8: Computational graph representing Twin Delayed DDPG (TD3) Q-function Q_{ϕ_i} loss, for $i = 1, 2$, given by $L_{\phi_i} = \left(Q_{\phi_i}\left(s_t, a_t\right) - \left(r_t + \gamma\left(1 - d_t\right) \min_{i=1,2} Q_{\phi_i'}\left(s_{t+1}, a\left(s_{t+1}\right)\right)\right)\right)^2$, where $a(s_{t+1}) = \text{clip}(\mu_{\theta'}(s_{t+1}) + \text{clip}(\epsilon, -c, c), a_{\text{low}}, a_{\text{high}})$, with $\epsilon \sim \mathcal{N}(0, \sigma)$.

Note this graph corresponds to generic Q-function parameters ϕ_i; TD3 algorithm considers two different Q-functions, Q_{ϕ_1} and Q_{ϕ_2}, the framework user can decide whether to use the same graph to update both or use two different DAGs. In contrast, only Q-function Q_{ϕ_1} is used to update policy parameters θ, as shown in Figure 9.

4.5 Soft Actor Critic (SAC)

Finally, we address SAC loss functions. Following the implementation in [19], we assume the use of environments with continuous action spaces. Figure 10 shows the computational graph corresponding to the generic Q-function Q_{ϕ_i} – SAC uses two different Q-functions following the same rationale as TD3. In this case we use the two last operation nodes introduced in this work, *Squashing* and *Prob*. We follow the convention from the original paper of considering single-output nodes. In case the framework user wants to consider multiple-output nodes, the *Squashing* and *Prob* nodes could be

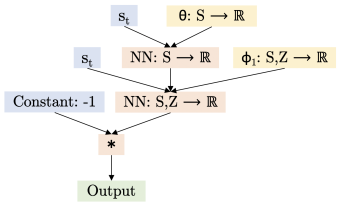

Figure 9: Computational graph representing Twin Delayed DDPG (TD3) policy π_θ loss $L_\theta = -Q_{\phi_1}(s_t, \mu_\theta(s_t))$.

integrated in a single node. To construct this graph, we also use an additional constant node with hyperparameter α [17].

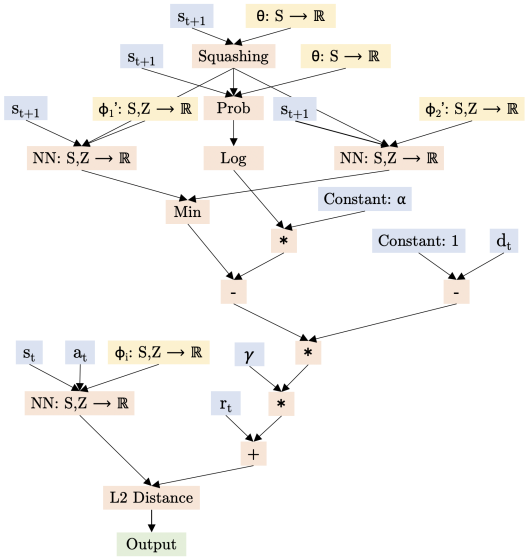

Figure 10: Computational graph representing Soft Actor-Critic (SAC) Q-function Q_ϕ loss $L_\phi = \left(Q_{\phi_i}(s_t, a_t) - \left(r_t + \gamma(1-d_t)\min_{i=1,2} Q_{\phi_i'}(s_{t+1}, \tilde{a}) - \alpha \log_{\pi_\theta}(\tilde{a}|s_{t+1})\right)\right)^2$, where $\tilde{a} = \tanh(\mu_\theta(s_{t+1}) + \sigma_\theta(s_{t+1}) \odot \xi)$, with $\xi \sim \mathcal{N}(0, I)$.

Figure 11 shows the computational graph to update policy parameters θ, which makes use of the same new nodes.

5 Conclusions

In this work we have presented an extension of the search language introduced in [1], which serves as a set of building blocks to generate multiple RL algorithms in the form of Directed Acyclic Graphs. Specifically, we have focused on adapting the language to Policy Gradient algorithms and, to that end, proposed four additional operation nodes as well as five modifications to the input structure of the graphs. By means of these extensions, we have shown computational graphs for five different Policy Gradient algorithms: VPG, PPO, DDPG, TD3, and SAC.

Overall, representing RL algorithms as graphs offers high interpretability to the framework users, especially when RL algorithms are novel. As part of the future work, these graphs should be tested as warm starts to the original evolutionary framework and the resulting child graphs validated in different unseen environments. In addition, other RL algorithms not addressed in this work could be represented in the same format.

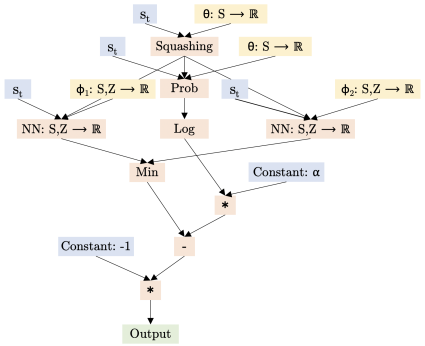

Figure 11: Computational graph representing Soft Actor-Critic (SAC) policy π_θ loss $L_\theta = -\left(\min_{i=1,2} Q_{\phi_i}(s_t, \tilde{a}) - \alpha \log_{\pi_\theta}(\tilde{a}|s_t)\right)$, where $\tilde{a} = \tanh(\mu_\theta(s_t) + \sigma_\theta(s_t) \odot \xi)$, with $\xi \sim \mathcal{N}(0, I)$.

6 Github Repository

A Github repository with all of the figures shown in this work can be found at https://github.com/jjgarau/DAGPolicyGradient.

References

[1] John D. Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Sergey Levine, Quoc V. Le, Honglak Lee, and Aleksandra Faust. Evolving reinforcement learning algorithms, 2021.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[5] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[9] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search, 2019.

[10] Ferran Alet, Martin F. Schneider, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Meta-learning curiosity algorithms, 2020.

[11] Rein Houthooft, Richard Y. Chen, Phillip Isola, Bradly C. Stadie, Filip Wolski, Jonathan Ho, and Pieter Abbeel. Evolved policy gradients, 2018.

[12] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[13] Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta-learning via learned loss, 2020.

[14] Louis Kirsch, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives, 2020.

[15] Junhyuk Oh, Matteo Hessel, Wojciech M. Czarnecki, Zhongwen Xu, Hado van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms, 2020.

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

[17] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[19] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.