Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021

옥찬호
utilForever@gmail.com

Introduction

- **Designing new deep reinforcement learning algorithms that can efficiently solve across a wide variety of problems generally requires a tremendous amount of manual effort.**
- Learning to design reinforcement learning algorithms or even small sub-components of algorithms would help ease this burden and could result in better algorithms than researchers could design manually.
- Our work might then shift from designing these algorithms manually into **designing the language and optimization methods for developing these algorithms automatically**.

Introduction

- Reinforcement learning algorithms can be viewed as a procedure that maps an agent's experience to a policy that obtains high cumulative reward over the course of training.
- We formulate the problem of training an agent as one of meta learning: **an outer loop searches over the space of computational graphs or programs that compute the objective function for the agent to minimize** and **an inner loop performs the updates using the learned loss function**.
- The objective of the outer loop is **to maximize the training return of the inner loop algorithm**.

Introduction

- Our learned loss function should generalize across many different environments, instead of being specific to a particular domain.
- Thus, we design a search language based on **genetic programming** (Koza, 1993) that can express general symbolic loss functions which can be applied to any environment.
- Data typing and a generic interface to variables in the MDP allow the learned program to be domain agnostic.
- This language also supports the use of neural network modules as subcomponents of the program, so that more complex neural network architectures can be realized.

Introduction

- Efficiently searching over the space of useful programs is generally difficult.
- For the outer loop optimization, we use **regularized evolution** (Real et al., 2019), a recent variant of classic evolutionary algorithms that employ **tournament selection** (Goldberg & Deb, 1991).
- This approach can scale with the number of compute nodes and has been shown to work for designing algorithms for supervised learning (Real et al., 2020).
- We adapt this method to automatically design algorithms for reinforcement learning.

Introduction

- While learning from scratch is generally less biased, encoding existing human knowledge into the learning process can speed up the optimization and also make the learned algorithm more interpretable.
- Because our search language expresses algorithms as a generalized computation graph, we can embed known RL algorithms in the graphs of the starting population of programs.
- We compare starting from scratch with bootstrapping off existing algorithms and find that while starting from scratch can learn existing algorithms, **starting from existing knowledge leads to new RL algorithms which can outperform the initial programs**.

Introduction

- **We learn two new RL algorithms which outperform existing algorithms in both sample efficiency and final performance on the training and test environments**.
- The learned algorithms are domain agnostic and generalize to new environments.
- Importantly, **the training environments consist of a suite of discrete action classical control tasks and grid-world style environments** while **the test environments include Atari games and are unlike anything seen during training**.

Introduction

- The contribution of this paper is a method for searching over the space of RL algorithms, which we instantiate by developing a formal language that describes a broad class of value-based model-free reinforcement learning methods.
- Our search language enables us to embed existing algorithms into the starting graphs which leads to faster learning and interpretable algorithms.

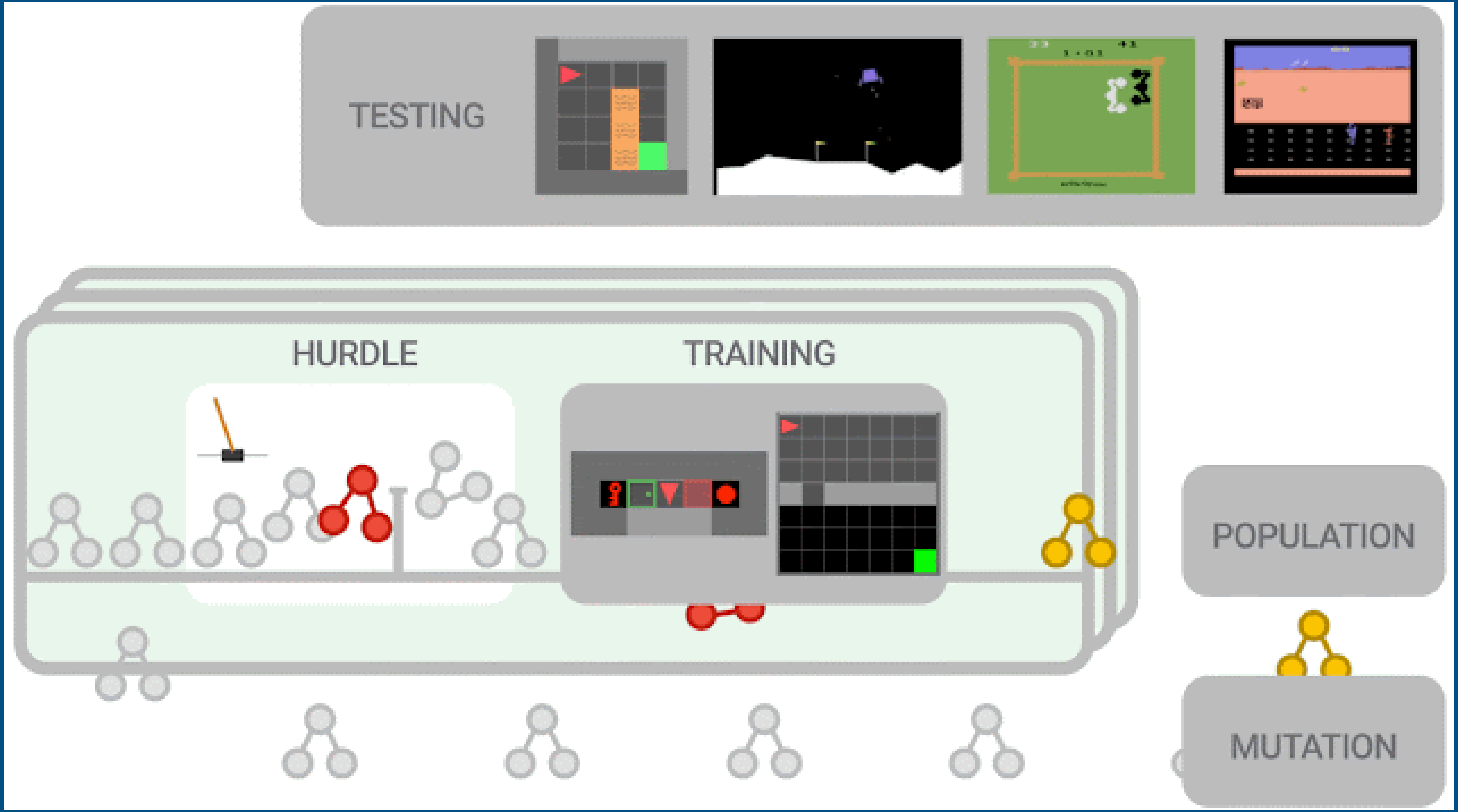
Introduction

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- We highlight two learned algorithms which generalize to completely new environments.
- Our analysis of the meta-learned programs shows that our method automatically discovers algorithms that share structure to recently proposed RL innovations, and empirically attain better performance than deep Q-learning methods.

Method Overview

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021



Problem Setup

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- We assume that the agent parameterized with policy $\pi_\theta(a_t|s_t)$ outputs actions at a_t each time step to an environment $\mathcal{E}$ and receives reward r_t and next state s_{t+1}. Since we are focusing on discrete action value-based RL methods, θ will be the parameters for a Q-value function and the policy is obtained from the Q-value function using an ϵ-greedy strategy.
- The agent saves this stream of transitions (s_t, s_{t+1}, a_t, r_t) to a replay buffer and continually updates the policy by minimizing a loss function $L(s_t, a_t, r_t, s_{t+1}, \theta, \gamma)$ over these transitions with gradient descent.

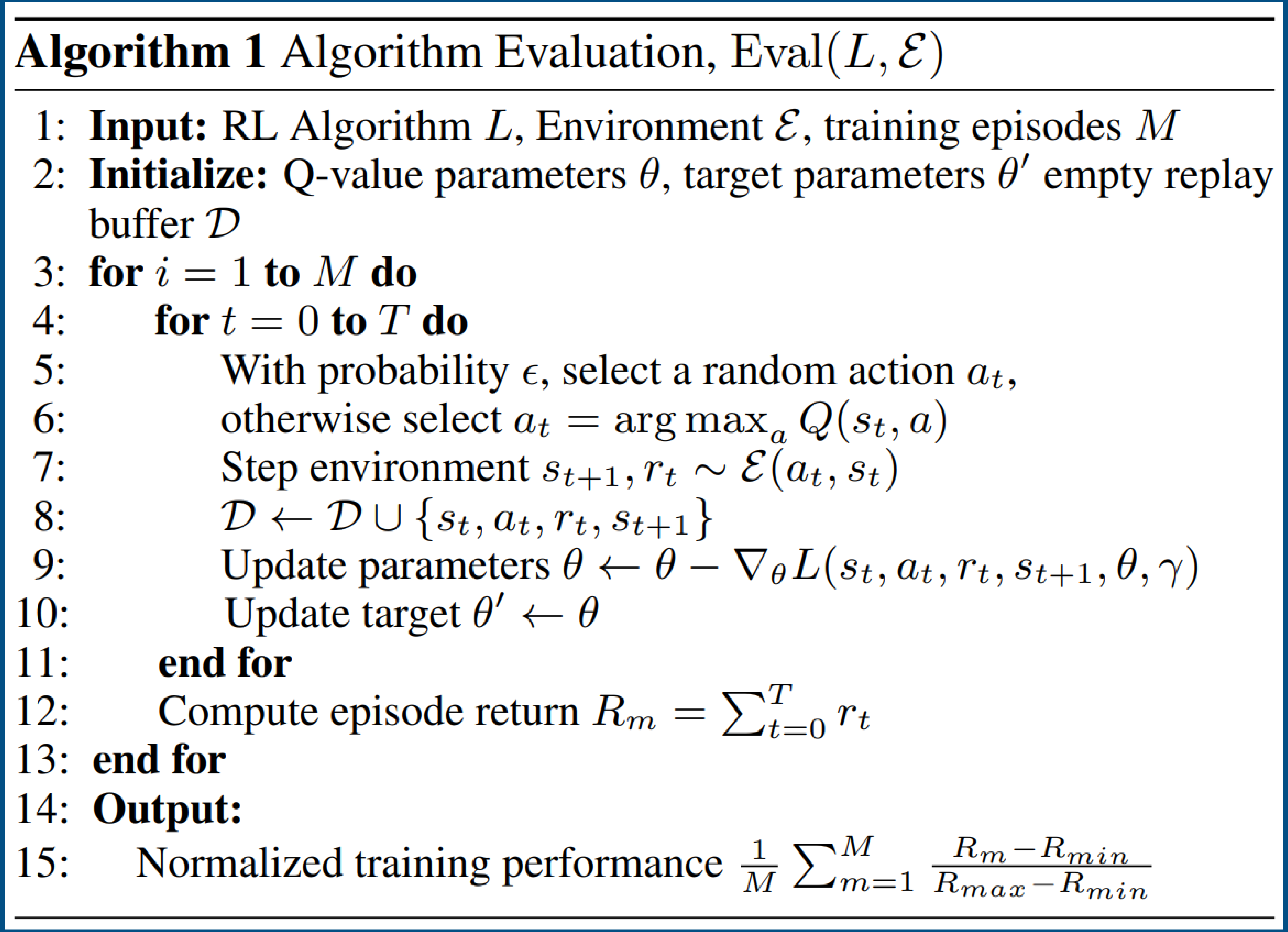
Problem Setup

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- Training will occur for a fixed number of M training episodes where in each episode m, the agent earns episode return $R_m = \sum_{t=0}^{T} r_t$. The performance of an algorithm for a given environment is summarized by **the normalized average training return**, $\frac{1}{M}\sum_{m=1}^{M} \frac{R_i - R_{min}}{R_{max} - R_{min}}$, where R_{min} and R_{max} are the minimum and maximum return for that environment.
- We assume these are known ahead of time. This inner loop evaluation procedure $\mathrm{Eval}(L, \mathcal{E})$ is outlined in Algorithm 1. To score an algorithm, we use the normalized average training return instead of the final behavior policy return because **the former metric will factor in sample efficiency as well**.

Problem Setup

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021



Algorithm 1 Algorithm Evaluation, $\text{Eval}(L, \mathcal{E})$

```
1:  Input: RL Algorithm $L$, Environment $\mathcal{E}$, training episodes $M$
2:  Initialize: Q-value parameters $\theta$, target parameters $\theta'$ empty replay
    buffer $\mathcal{D}$
3:  for $i = 1$ to $M$ do
4:      for $t = 0$ to $T$ do
5:          With probability $\epsilon$, select a random action $a_t$,
6:          otherwise select $a_t = \arg\max_a Q(s_t, a)$
7:          Step environment $s_{t+1}, r_t \sim \mathcal{E}(a_t, s_t)$
8:          $\mathcal{D} \leftarrow \mathcal{D} \cup \{s_t, a_t, r_t, s_{t+1}\}$
9:          Update parameters $\theta \leftarrow \theta - \nabla_\theta L(s_t, a_t, r_t, s_{t+1}, \theta, \gamma)$
10:         Update target $\theta' \leftarrow \theta$
11:     end for
12:     Compute episode return $R_m = \sum_{t=0}^{T} r_t$
13: end for
14: Output:
15:   Normalized training performance $\frac{1}{M} \sum_{m=1}^{M} \frac{R_m - R_{min}}{R_{max} - R_{min}}$
```

Problem Setup

Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021

- The goal of the meta-learner is **to find the optimal loss function** $L(s_t, a_t, r_t, s_{t+1}, \theta, \gamma)$ to optimize π_θ with maximal normalized average training return over the set of training environments.
- The full objective for the meta-learner is:

$$L^* = \underset{L}{\operatorname{argmax}} \left[\sum_{\mathcal{E}} \operatorname{Eval}(L, \mathcal{E}) \right]$$

L is represented as a computational graph which we describe in the next section.

Search Language

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- Our search language for the algorithm L should be expressive enough to represent existing algorithms while enabling the learning of new algorithms which can obtain good generalization performance across a wide range of environments.
- Similar to Alet et al (2020a), we describe the RL algorithm as general programs with a domain specific language, but we target updates to the policy rather than reward bonuses for exploration.
- Algorithms will map transitions (s_t, a_t, s_{t+1}, r_t), policy parameters θ, and discount factor γ into a scalar loss to be optimized with gradient descent.

Search Language

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

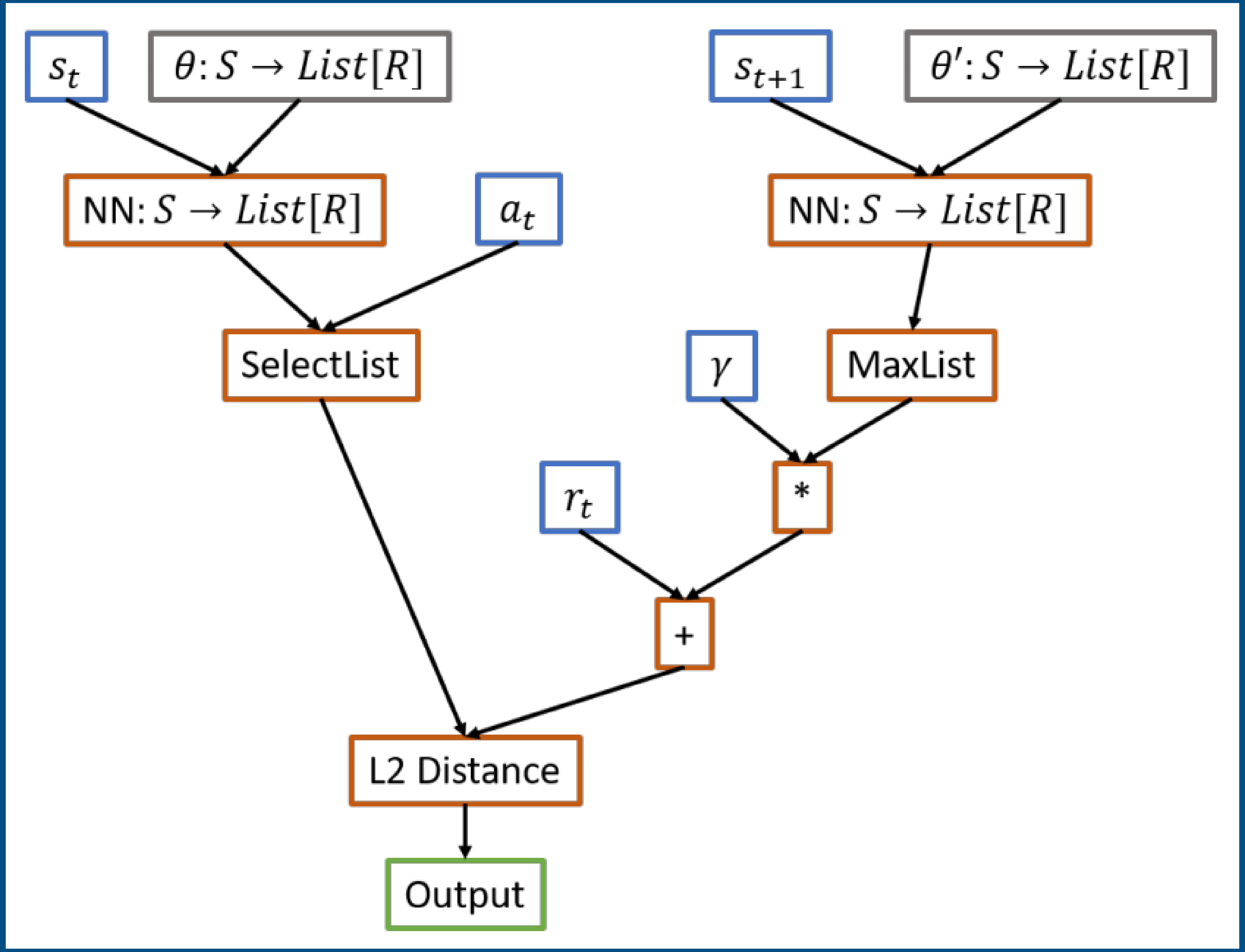
- We express L as **a computational graph or directed acyclic graph (DAG)** of nodes with typed inputs and outputs. Nodes are of several types:
  - **Input nodes** represent inputs to the program, and include elements from transitions (s_t, a_t, s_{t+1}, r_t) and constants, such as the discount factor γ.
  - **Parameter nodes** are neural network weights, which can map between various data types.
    For example, the weights for the Q-value network will map an input node with state data type to a list of real numbers for each action.
  - **Operation nodes** compute outputs given inputs from parent nodes. This includes applying parameter nodes, as well as basic math operators from linear algebra, probability, and statistics. By default, we set the last node in the graph to compute the output of the program which is the scalar loss function to be optimized. Importantly, the inputs and outputs of nodes are typed among (state, action, vector, float, list, probability). This typing allows for programs to be applied to any domain.
    It also restricts the space of programs to ones with valid typing which reduces the search space.

Search Language

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- Visualization of a RL algorithm, DQN

$$L = \left(Q(s_t, a_t) - \left(r_t + \gamma * \max_a Q_{targ}(s_{t+1}, a)\right)\right)^2$$

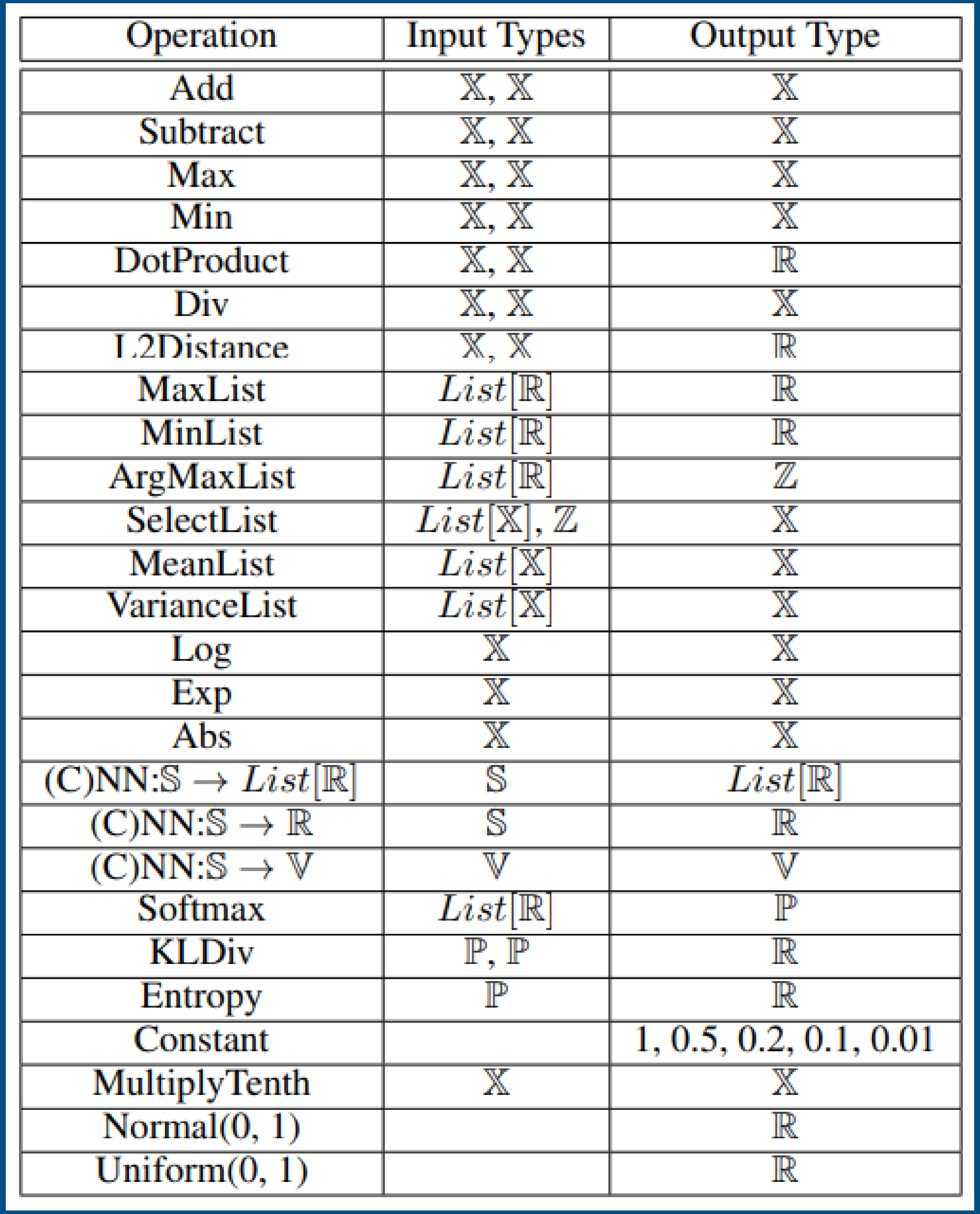


Search Language

Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021

- The full list of operators

  ($\mathbb{S}$: State, $\mathbb{Z}$: Action, $\mathbb{R}$: Float, $List[\mathbb{X}]$: List, $\mathbb{P}$: Probability, $\mathbb{V}$: Vector, $\mathbb{X}$: It can be $\mathbb{S}$, $\mathbb{R}$, or $\mathbb{V}$)



Operation	Input Types	Output Type
Add	$\mathbb{X}, \mathbb{X}$	$\mathbb{X}$
Subtract	$\mathbb{X}, \mathbb{X}$	$\mathbb{X}$
Max	$\mathbb{X}, \mathbb{X}$	$\mathbb{X}$
Min	$\mathbb{X}, \mathbb{X}$	$\mathbb{X}$
DotProduct	$\mathbb{X}, \mathbb{X}$	$\mathbb{R}$
Div	$\mathbb{X}, \mathbb{X}$	$\mathbb{X}$
L2Distance	$\mathbb{X}, \mathbb{X}$	$\mathbb{R}$
MaxList	$List[\mathbb{R}]$	$\mathbb{R}$
MinList	$List[\mathbb{R}]$	$\mathbb{R}$
ArgMaxList	$List[\mathbb{R}]$	$\mathbb{Z}$
SelectList	$List[\mathbb{X}], \mathbb{Z}$	$\mathbb{X}$
MeanList	$List[\mathbb{X}]$	$\mathbb{X}$
VarianceList	$List[\mathbb{X}]$	$\mathbb{X}$
Log	$\mathbb{X}$	$\mathbb{X}$
Exp	$\mathbb{X}$	$\mathbb{X}$
Abs	$\mathbb{X}$	$\mathbb{X}$
(C)NN:$\mathbb{S} \rightarrow List[\mathbb{R}]$	$\mathbb{S}$	$List[\mathbb{R}]$
(C)NN:$\mathbb{S} \rightarrow \mathbb{R}$	$\mathbb{S}$	$\mathbb{R}$
(C)NN:$\mathbb{S} \rightarrow \mathbb{V}$	$\mathbb{V}$	$\mathbb{V}$
Softmax	$List[\mathbb{R}]$	$\mathbb{P}$
KLDiv	$\mathbb{P}, \mathbb{P}$	$\mathbb{R}$
Entropy	$\mathbb{P}$	$\mathbb{R}$
Constant		1, 0.5, 0.2, 0.1, 0.01
MultiplyTenth	$\mathbb{X}$	$\mathbb{X}$
Normal(0, 1)		$\mathbb{R}$
Uniform(0, 1)		$\mathbb{R}$

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- Evaluating thousands of programs over a range of complex environments is prohibitively expensive, especially if done serially.
- We adapt a **genetic programming** (Koza, 1993) for the search method and use **regularized evolution** (Real et al, 2019), a variant of classic evolutionary algorithms that employ **tournament selection** (Goldberg & Deb, 1991).
- Regularized evolution has been shown to work for learning supervised learning algorithms (Real et al., 2020) and can be parallelized across compute nodes.

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- Tournament selection keeps a population of P algorithms and improves the population through cycles. Each cycle picks a tournament of $T < P$ algorithms at random and selects the best algorithm in the tournament as a parent.
- The parent is mutated into a child algorithm which gets added to the population while the oldest algorithm in the population is removed.
- We use a single type of mutation which first chooses which node in the graph to mutate and then replaces it with a random operation node with inputs drawn uniformly from all possible inputs.

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- There exists a combinatorially large number of graph configurations.
- Furthermore, evaluating a single graph, which means training the full inner loop RL algorithm, can take up a large amount of time compared to the supervised learning setting.
- Speeding up the search and avoiding needless computation are needed to make the problem more tractable. **We extend regularized evolution with several techniques to make the optimization more efficient**.

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Functional equivalence** check (Real et al., 2020; Alet et al., 2020b). Before evaluating a program, we check if it is functionally equivalent to any previously evaluated program. This check is done by hashing the concatenated output of the program for 10 values of randomized inputs. If a mutated program is functionally equivalent to an older program, we still add it to the population, but use the saved score of the older program. Since some nodes of the graph do not always contribute to the output, parts of the mutated program may eventually contribute to a functionally different program.

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Early hurdles** (So et al., 2019). We want poor performing programs to terminate early so that we can avoid unneeded computation. We use the CartPole environment as an early hurdle environment $\mathcal{E}_h$ by training a program for a fixed number of episodes. <u>If an algorithm performs poorly, then episodes will terminate in a short number of steps (as the pole falls rapidly) which quickly exhausts the number of training episodes.</u> We use $\mathrm{Eval}(L, \mathcal{E}_h) < \alpha$ as the threshold for poor performance with α chosen empirically.

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Program checks**. We perform basic checks to rule out and skip training invalid programs. The loss function needs to be a scalar value so we check if the program output type is a float ($\text{type}(L) = \mathbb{R}$). Additionally, we check if each program is differentiable with respect to the policy parameters by checking if a path exists in the graph between the output and the policy parameter node.

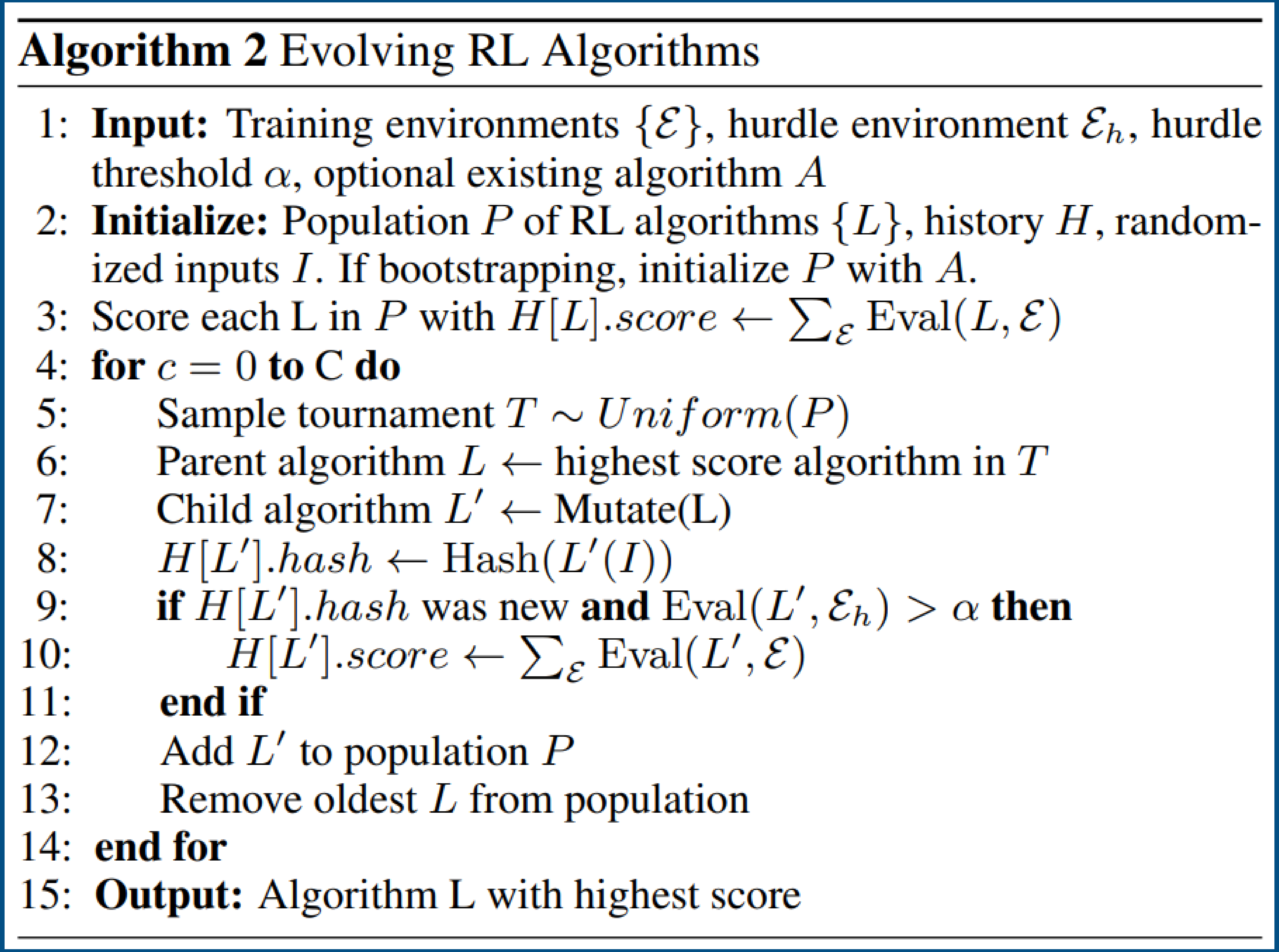
Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Learning from Scratch and Bootstrapping.** Our method enables both learning from scratch and learning from existing knowledge by bootstrapping the initial algorithm population with existing algorithms. <u>We learn algorithms from scratch by initializing the population of algorithms randomly.</u> <u>An algorithm is sampled by sampling each operation node sequentially in the DAG.</u> For each node, an operation and valid inputs to that operation are sampled uniformly over all possible options.

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021



Algorithm 2 Evolving RL Algorithms

```
 1: Input: Training environments $\{\mathcal{E}\}$, hurdle environment $\mathcal{E}_h$, hurdle
    threshold $\alpha$, optional existing algorithm $A$
 2: Initialize: Population $P$ of RL algorithms $\{L\}$, history $H$, random-
    ized inputs $I$. If bootstrapping, initialize $P$ with $A$.
 3: Score each L in $P$ with $H[L].score \leftarrow \sum_{\mathcal{E}} \text{Eval}(L, \mathcal{E})$
 4: for $c = 0$ to C do
 5:     Sample tournament $T \sim Uniform(P)$
 6:     Parent algorithm $L \leftarrow$ highest score algorithm in $T$
 7:     Child algorithm $L' \leftarrow$ Mutate(L)
 8:     $H[L'].hash \leftarrow \text{Hash}(L'(I))$
 9:     if $H[L'].hash$ was new and $\text{Eval}(L', \mathcal{E}_h) > \alpha$ then
10:         $H[L'].score \leftarrow \sum_{\mathcal{E}} \text{Eval}(L', \mathcal{E})$
11:     end if
12:     Add $L'$ to population $P$
13:     Remove oldest $L$ from population
14: end for
15: Output: Algorithm L with highest score
```

Evolutionary Search

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- While learning from scratch might uncover completely new algorithms that differ substantially from the existing methods, this method can take longer to converge to a reasonable algorithm.
- We would like to incorporate the knowledge we do have of good algorithms to bootstrap our search from a better starting point.
- <u>We initialize our graph with the loss function of DQN (Mnih et al., 2013) so that the first 7 nodes represent the standard DQN loss, while the remaining nodes are initialized randomly.</u>
- During regularized evolution, the nodes are not frozen, s.t. it is possible for the existing sub-graph to be completely replaced if a better solution is found.

Training Setup

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Meta-Training details**: We search over programs with maximum 20 nodes, not including inputs or parameter nodes. We use a population size of 300, tournament size of 25, and choose these parameters based on the ones used in (Real et al., 2019). Mutations occur with probability 0.95. Otherwise a new random program is sampled. The search is done over 300 CPUs and run for roughly 72 hours, at which point around 20,000 programs have been evaluated. The search is distributed such that any free CPU is allocated to a proposed individual such that there are no idle CPUs.

Training Setup

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Training environments**: The choice of training environments greatly affects the learned algorithms and their generalization performance. At the same time, **our training environments should be not too computationally expensive to run as we will be evaluating thousands of RL algorithms**. We use a range of 4 classical control tasks (CartPole, Acrobat, MountainCar, LunarLander) and a set of 12 multitask gridworld style environments from MiniGrid (Chevalier-Boisvert et al., 2018). These environments are computationally cheap to run but also chosen **to cover a diverse set of situations**. This includes dense and sparse reward, long time horizon, and tasks requiring solving a sequence of subgoals such as picking up a key and unlocking a door.

Training Setup

- **Training environments**: The training environments always include CartPole as an initial hurdle. If an algorithm succeeds on CartPole (normalized training performance greater than 0.6), it then proceeds to a harder set of training environments. For our experiments, we choose these training environments by sampling a set of 3 environments and leave the rest as test environments. For learning from scratch we also compare the effect of number of training environments on the learned algorithm by comparing training on just CartPole versus training on CartPole and LunarLander.

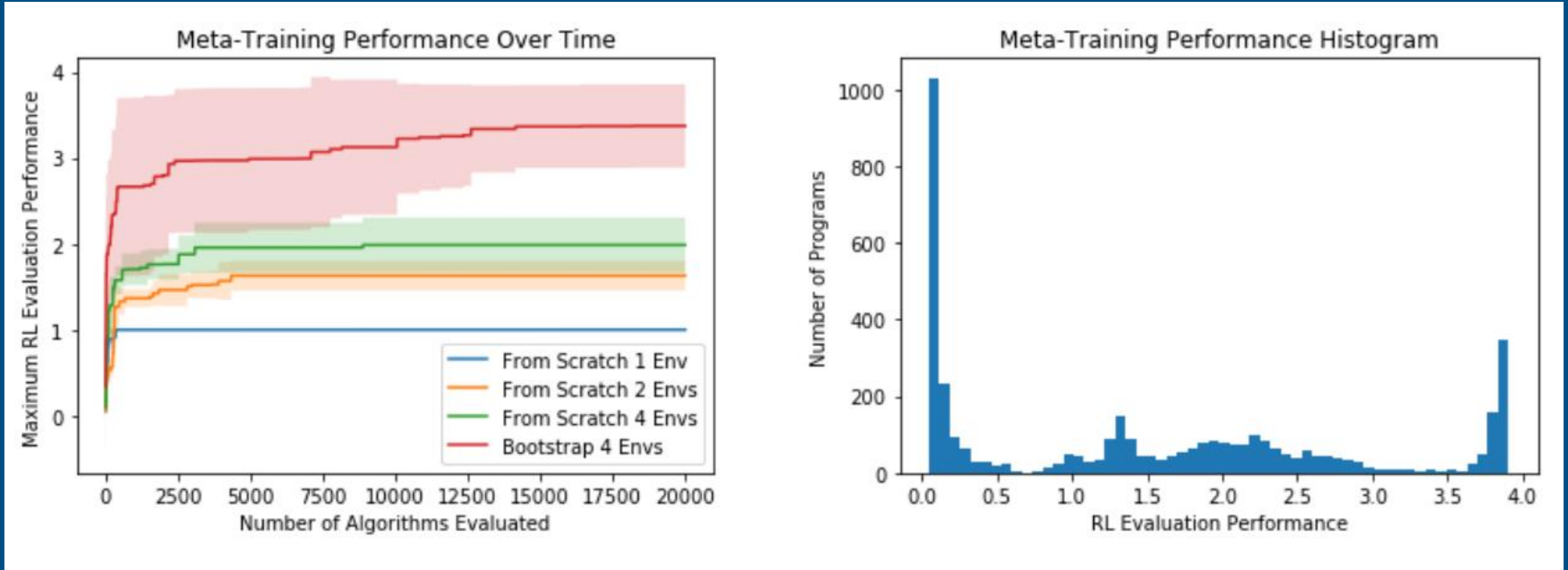
Training Setup

- **RL Training details**: For training the RL agent, we use the same hyper-parameters across all training and test environments except as noted. All neural networks are MLPs of size (256, 256) with ReLU activations. We use the Adam optimizer with a learning rate of 0.0001. ϵ is decayed linearly from 1 to 0.05 over 1e3 steps for the classical control tasks and over 1e5 steps for the MiniGrid tasks.

Learning Convergence

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- It shows convergence over several training configurations. We find that at the end of training roughly 70% of proposed algorithms are functionally equivalent to a previously evaluated program, while early hurdles cut roughly another 40% of proposed non-duplicate programs.



(a) Learning curve (b) Performance histogram

Learning Convergence

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Varying number of training environments**: We compare learning from scratch with a single training environment (CartPole) versus with two training environments (CartPole and LunarLander). While both experiments reach the maximum performance on these environments, the learned algorithms are different.

Learning Convergence

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Varying number of training environments:**
  The two-environment training setup learns the known TD loss

$$L = \left(Q(s_t, a_t) - (r_t + \gamma * \max_a Q_{targ}(s_{t+1}, a))\right)^2$$

while the single-environment training setup learns a slight variation

$$L = \left(Q(s_t, a_t) - (r_t + \max_a Q_{targ}(s_{t+1}, a))\right)^2$$

that does not use the discount, indicating that **the range of difficulty on the training environments is important for learning algorithms which can generalize**.

Learning Convergence

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- **Learning from scratch versus bootstrapping**: We compare training from scratch versus training from bootstrapping on four training environments (CartPole, KeyCorridorS3R1, DynamicObstacle-6x6, DoorKey-5x5). The training performance does not saturate, leaving room for improvement. Bootstrapping from DQN significantly improves both the convergence and performance of the meta-training, resulting in a 40% increase in final training performance.

Learned RL Algorithms

Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021

- We discuss two particularly interesting loss functions that were learned by our method, and that have good generalization performance on the test environments. Let

$$Y_t = r_t + \gamma * \max_a Q_{targ}(s_{t+1}, a), \text{and } \delta = Q(s_t, a_t) - Y_t$$

Learned RL Algorithms

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- The first loss function DQNClipped is

$$L_{\text{DQNClipped}} = \max[Q(s_t, a_t), \delta^2 + Y_t] + \max\left[Q(s_t, a_t) - Y_t, \gamma\left(\max_a Q_{targ}(s_{t+1}, a)\right)^2\right]$$

  - $L_{\text{DQNClipped}}$ was trained from bootstrapping off DQN using three training environments (LunarLander, MiniGrid-Dynamic-Obstacles-5x5, MiniGrid-LavaGapS5).
  - It outperforms DQN and doubleDQN, DDQN, (van Hasselt et al., 2015) on both the training and unseen environments.
  - **The intuition behind this loss function is that, if the Q-values become too large (when $Q(s_t, a_t) > \delta^2 + Y_t$), the loss will act to minimize $Q(s_t, a_t)$ instead of the normal δ^2 loss**. Alternatively, we can view this condition as $\delta = Q(s_t, a_t) - Y_t > \delta^2$. This means when δ is small enough then $Q(s_t, a_t)$ are relatively close and the loss is just to minimize $Q(s_t, a_t)$.

Learned RL Algorithms

Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021

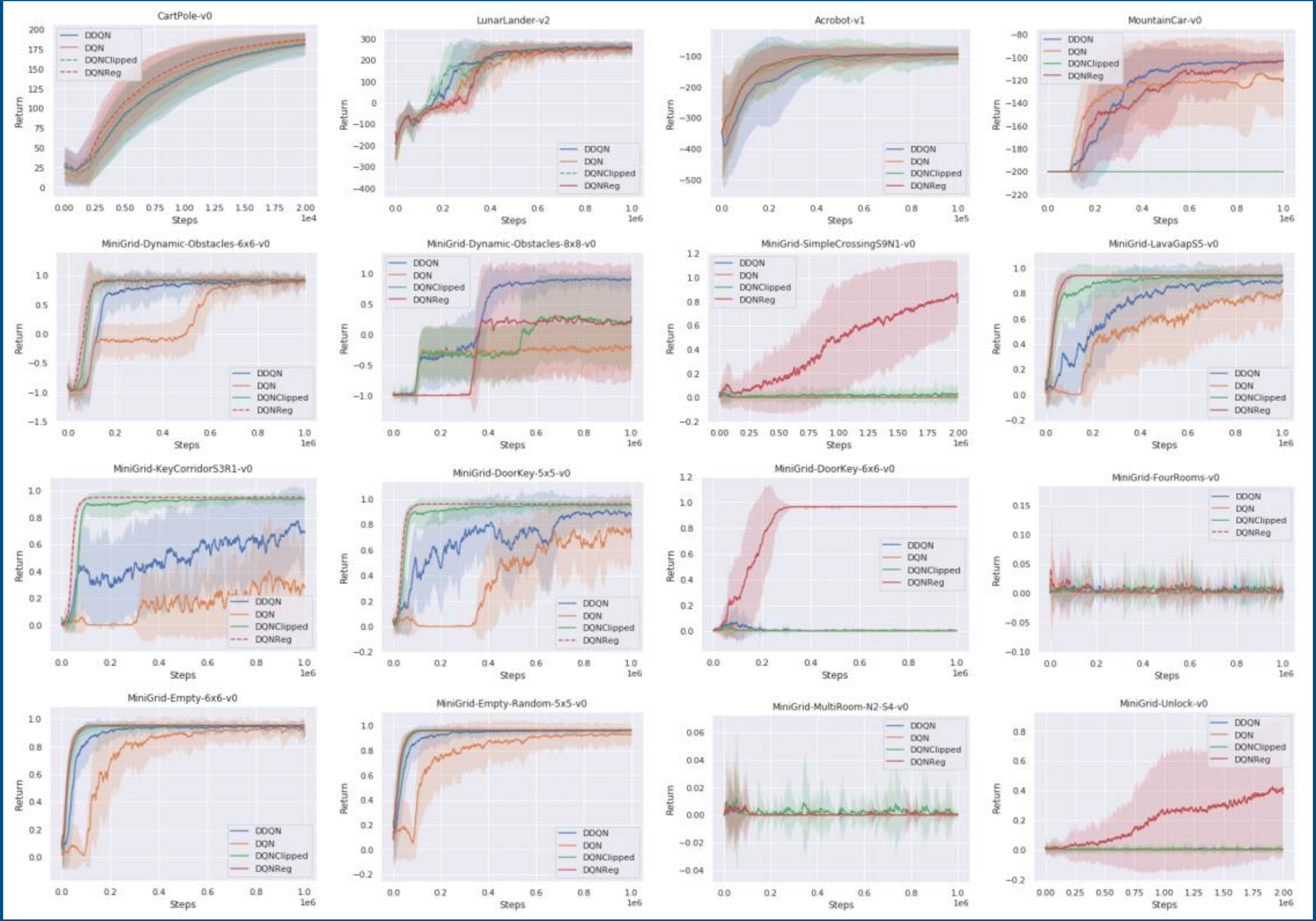
- The second loss function, which we call DQNReg, is given by

$$L_{\mathrm{DQNReg}} = 0.1 * Q(s_t, a_t) + \delta^2$$

  - DQNReg was trained from bootstrapping off DQN using three training environments (KeyCorridorS3R1, Dynamic-Obstacles-6x6, DoorKey-5x5). In comparison to DQNClipped, DQNReg directly regularizes the Q values with a weighted term that is always active.
  - **We note that both of these loss functions modify the original DQN loss function to regularize the Q-values to be lower in value.**
  - While DQNReg is quite simple, it matches or outperforms the baselines on all training and test environments including from classical control and Minigrid. It does particularly well on a few test environments (SimpleCrossingS9N1, DoorKey-6x6, and Unlock) and solves the tasks when other methods fail to attain any reward. It is also much more stable with lower variance between seeds, and more sample efficient on test environments (LavaGapS5, Empty-6x6, Empty-Random-5x5).

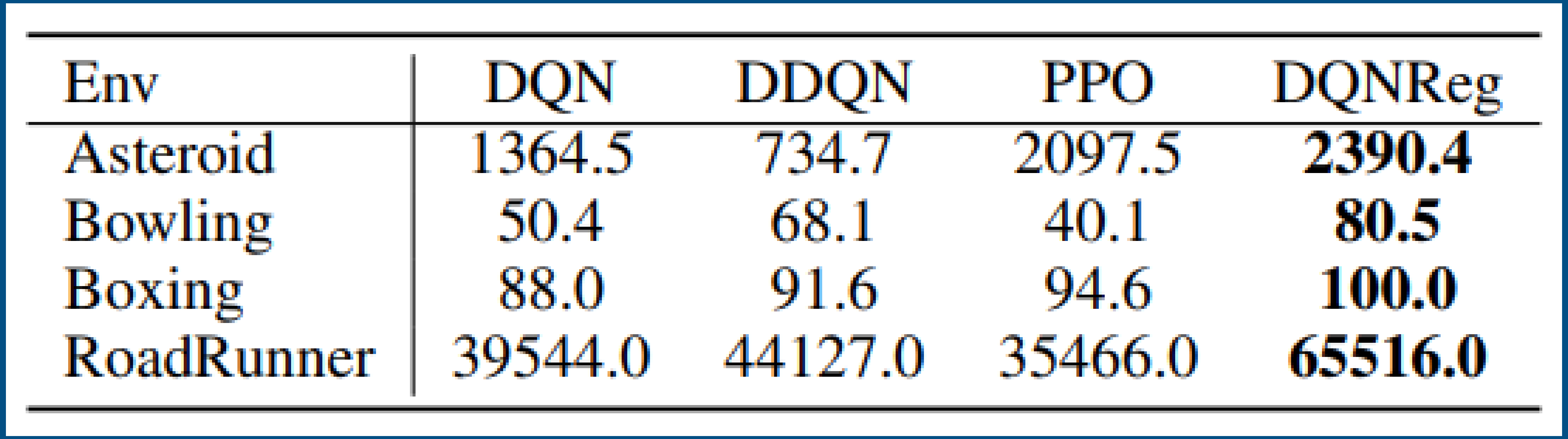
Learned RL Algorithms

Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021



Learned RL Algorithms

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

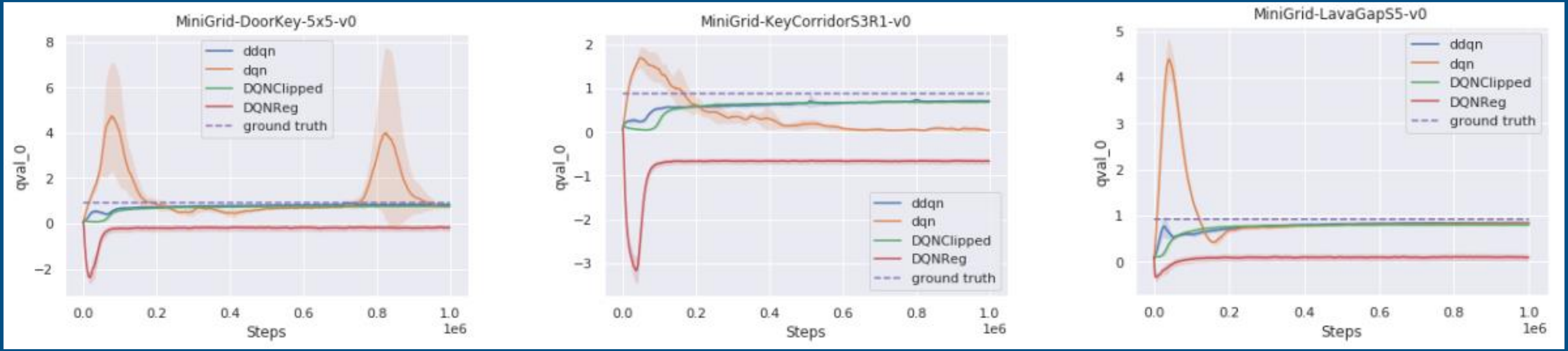


Env	DQN	DDQN	PPO	DQNReg
Asteroid	1364.5	734.7	2097.5	**2390.4**
Bowling	50.4	68.1	40.1	**80.5**
Boxing	88.0	91.6	94.6	**100.0**
RoadRunner	39544.0	44127.0	35466.0	**65516.0**

Analysis of Learned Algos

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- We analyze the learned algorithms to understand their beneficial effect on performance. We compare the estimated Q-values for each algorithm.
- We see that DQN frequently overestimates the Q values while DDQN consistently underestimates the Q values before converging to the ground truth Q value which are computed with a manually designed optimal policy.



Analysis of Learned Algos

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- DQNClipped has similar performance to DDQN, in that it also consistently underestimates the Q values and does so slightly more aggressively than DDQN. DQNReg significantly undershoots the Q values and does not converge to the ground truth.
- Various works (van Hasselt et al., 2015; Haarnoja et al., 2018; Fujimoto et al., 2018) have shown that **overestimated value estimates is problematic and restricting the overestimation improves performance**.

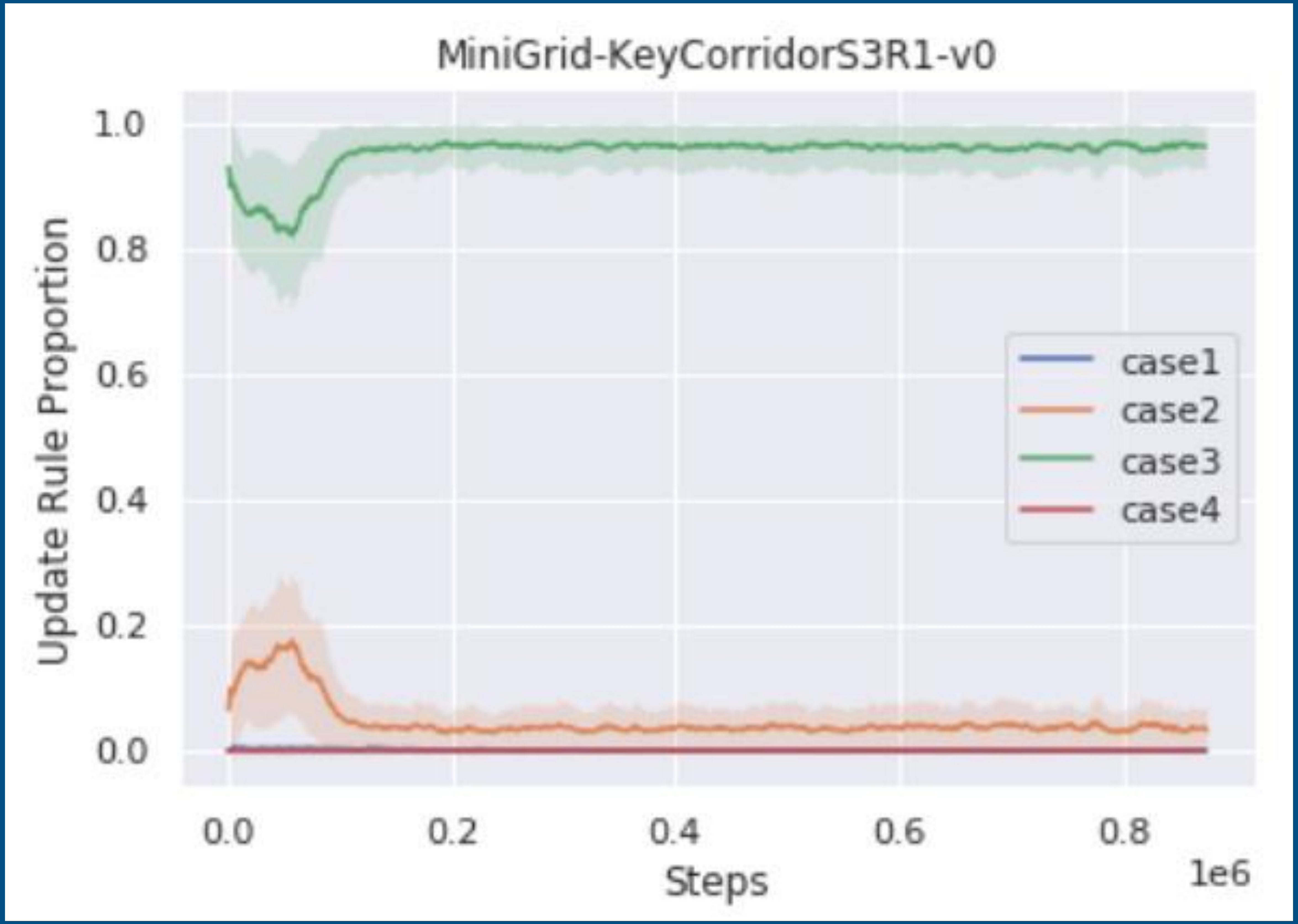
Analysis of Learned Algos

Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021

- The loss function in DQNClipped is composed of the sum of two max operations, and so we can analyze when each update rule is active.
- We interpret DQNClipped as $\max(v_1, v_2) + \max(v_3, v_4)$ with four cases:
  1) $v_1 > v_2$ and $v_3 > v_4$ 2) $v_1 > v_2$ and $v_3 < v_4$
  3) $v_1 < v_2$ and $v_3 < v_4$ 4) $v_1 < v_2$ and $v_3 > v_4$.
  - Case 2 corresponds to minimizing the Q values.
  - Case 3 would correspond to the normal DQN loss of δ^2
    since the parameters of Q_{targ} are not updated during gradient descent.

Analysis of Learned Algos

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021



Analysis of Learned Algos

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- We plot the proportion of when each case is active during training.
- We see that usually case 3 is generally the most active with a small dip in the beginning but then stays around 95%. Meanwhile, case 2, which regularizes the Q-values, has a small increase in the beginning and then decreases later, matching with our analysis, which shows that DQNClipped strongly underestimates the Q-values in the beginning of training.
- This can be seen as a constrained optimization where the amount of Q-value regularization is tuned accordingly. The regularization is stronger in the beginning of training when overestimation is problematic $(Q(s_t, a_t) > \delta^2 + Y_t)$ and gets weaker as δ^2 gets smaller.

Conclusion

Evolving Reinforcement Learning Algorithms,
JD. Co-Reyes et al, 2021

- In this work, we have presented a method for learning reinforcement learning algorithms. We design a general language for representing algorithms which compute the loss function for value-based model-free RL agents to optimize.
- We highlight two learned algorithms which although relatively simple, can obtain good generalization performance over a wide range of environments.
- Our analysis of the learned algorithms sheds insight on their benefit as regularization terms which are similar to recently proposed algorithms.

Conclusion

Evolving Reinforcement Learning Algorithms, JD. Co-Reyes et al, 2021

- Our work is limited to discrete action and value-based RL algorithms that are close to DQN, but could easily be expanded to express more general RL algorithms such as actor-critic or policy gradient methods.
- How actions are sampled from the policy could also be part of the search space.
- The set of environments we use for both training and testing could also be expanded to include a more diverse set of problem types. We leave these problems for future work.

Thank you!