EVOLVING REINFORCEMENT LEARNING ALGORITHMS

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose a method for meta-learning reinforcement learning algorithms by searching over the space of computational graphs which compute the loss function for a value-based model-free RL agent to optimize. The learned algorithms are domain-agnostic and can generalize to new environments not seen during training. Our method can both learn from scratch and bootstrap off known existing algorithms, like DQN, enabling interpretable modifications which improve performance. Learning from scratch on simple classical control and gridworld tasks, our method rediscovers the temporal-difference (TD) algorithm. Bootstrapped from DQN, we highlight two learned algorithms which obtain good generalization performance over other classical control tasks, gridworld type tasks, and Atari games. The analysis of the learned algorithm behavior shows resemblance to recently proposed RL algorithms that address overestimation in value-based methods.

1 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.

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.

Our learned loss function should generalize across many different environments, instead of being specific to a particular domain. Thus, we design a search language 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. Efficiently searching over the space of useful programs is generally difficult. For the outer loop optimization, we use a genetic programming method, regularized evolution from Real et al. (2019), which 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 design algorithms for reinforcement learning.

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.

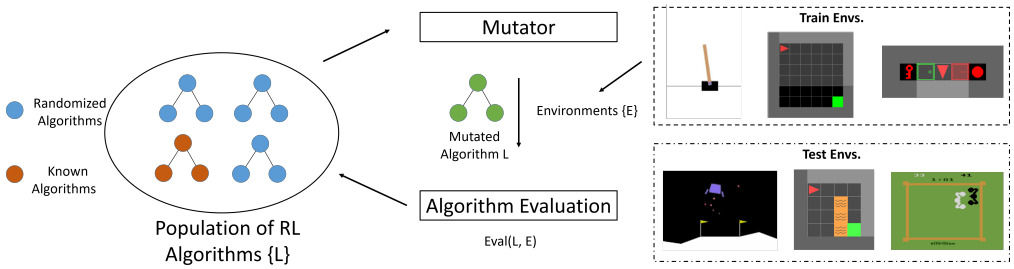


Figure 1: Method overview. We use regularized evolution to evolve a population of RL algorithms. A mutator alters top performing algorithms to produce a new algorithm. The performance of the algorithm is evaluated over a set of training environments and the population is updated. Our method can incorporating existing knowledge by starting the population from known RL algorithms instead of purely from scratch.

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. The learned algorithms are domain agnostic and generalize to new environments. Our search language enables us to embed existing algorithms into the starting graphs which leads to faster learning and interpretable algorithms. We learn two new RL algorithms which outperform existing algorithms in both sample efficiency and final performance on the training and test environments. The environments consist of a suite of discrete action classical control tasks, gridworld style environments, and Atari games. 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.

2 Related Work

Learning to learn is an established idea in in supervised learning, including meta-learning with genetic programming (Schmidhuber, 1987; Holland, 1975; Koza, 1993), learning a neural network update rule (Bengio et al., 1991), and self modifying RNNs (Schmidhuber, 1993). Genetic programming has been used to find new loss functions (Bengio et al., 1994; Trujillo & Olague, 2006). More recently, AutoML (Hutter et al., 2018) aims to automate the machine learning training process. Automated neural network architecture search (Real et al., 2017; 2019; Liu et al., 2017; Stanley & Miikkulainen, 2002; Zoph & Le, 2016; Elsken et al., 2018; Pham et al., 2018) has made large improvements in image classification. Instead of learning the architecture, AutoML-Zero (Real et al., 2020) learns the algorithm from scratch using basic mathematical operations. Our work shares similar ideas, but is applied to the RL setting and assumes additional primitives such as neural network modules. We learn computational graphs with the goal of automating RL algorithm design. In contrast to AutoML-Zero, our learned RL algorithms generalize to new problems, not seen in training.

Automating RL. While RL is used for AutoML (Zoph & Le, 2016; Zoph et al., 2018; Cai et al., 2018; Bello et al., 2017), automating RL itself has been somewhat limited. RL requires different design choices compared to supervised learning, including the formulation of reward and policy update rules. All of which affect learning and performance, and are usually chosen through trial and error. AutoRL addresses the gap by applying the AutoML framework from supervised learning to the MDP setting in RL. For example, evolutionary algorithms are used to mutate the actor network weights (Khadka & Tumer, 2018), learn the rewards for a task (Faust et al., 2019), tune hyperparameters (Tang & Choromanski, 2020; Franke et al., 2020), or search for a neural network architecture (Song et al., 2020; Franke et al., 2020). This paper focuses on task-agnostic RL update rules in the value-based RL setting which are both interpretable and generalizable.

Meta-learning in RL. Recent work has focused on few-shot task adaptation. Finn et al. (2017); Finn & Levine (2018) meta-learns initial parameters which can quickly adapt to new tasks, while RL^2 (Duan et al., 2016) and concurrent work (Wang et al., 2017), formulates RL itself as a learning problem that is learned with an RNN. The meta-learned component of these works is tuned to a particular domain or environment, in the form of NN weights which cannot be used for completely new domains with potentially different sized inputs. Neural Programmer-Interpreters (Reed & De Freitas, 2015; Pierrot et al., 2019) overcome the environment generalization challenge by learning hierarchical neural programs with domain-specific encoders for different environments. Here, the computational graph has a flexible architecture and generalizes across different environments.

Learning RL algorithms or their components, such as a reward bonus or value update function, has been studied previously with meta-gradients (Kirsch et al., 2020; Chebotar et al., 2019; Oh et al., 2020), evolutionary strategies (Houthooft et al., 2018), and RNNs (Duan et al., 2016). Although our work also learns RL algorithms, the update rule is represented as a computation graph which includes both neural network modules and symbolic operators. One key benefit is that the resulting graph can be interpreted analytically and can optionally be initialized from known existing algorithms. Prior work that focuses on learning RL losses, generalizes to different goals and initial conditions *within* a single environment (Houthooft et al., 2018), or learns a domain invariant policy update rule that can generalize to new environments (Kirsch et al., 2020). Another approach searches over the space of curiosity programs using a similar language of DAGs with neural network modules (Alet et al., 2020) and performs the meta-training on a single environment. In contrast, our method is applied to learn general RL update rules and meta-trained over a diverse set of environments.

3 Learning Reinforcement Learning Algorithms

In this section, we first describe the problem setup. An inner loop method $\text{Eval}(L, \mathcal{E})$ evaluates a learned RL algorithm L on a given environment $\mathcal{E}$. Given access to this procedure, the goal for the outer loop optimization is to learn a RL algorithm with high training return over a set of training environments. We then describe the search language which enables the learning of general loss functions and the outer loop method which can efficiently search over this space.

3.1 Problem Setup

We assume that the agent parameterized with policy $\pi_\theta(a_t|s_t)$ outputs actions a_t at 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. 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 $\text{Eval}(L, \mathcal{E})$ 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.

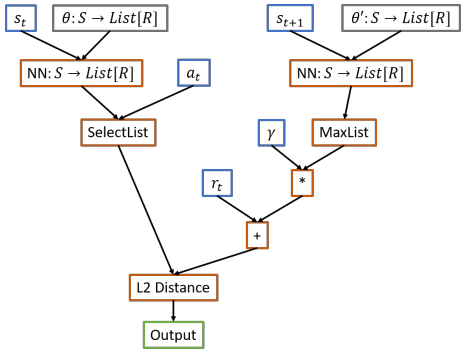


Figure 2: Visualization of a RL algorithm, DQN, as a computational graph which computes the loss $L = (Q(s_t, a_t) - (r_t + \gamma * \max_a Q_{targ}(s_{t+1}, a)))^2$. Input nodes are in blue, parameter nodes in gray, operation nodes in orange, and output in green.

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:

$$L^* = \arg\max_L \left[\sum_{\mathcal{E}} \text{Eval}(L, \mathcal{E}) \right]$$

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

3.2 SEARCH LANGUAGE

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. (2020), 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. We express L as a computational graph or directed acyclic graph (DAG) of nodes with typed inputs and outputs. See Figure 2 for a visualization of DQN expressed in this form. 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. A full list of operation nodes is provided in Appendix A. 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.

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

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

Algorithm 2 Evolving RL Algorithms

Input: Training environments $\{\mathcal{E}\}$, hurdle environment $\mathcal{E}_h$, hurdle threshold α, optional existing algorithm A
Initialize: Population P of RL algorithms $\{L\}$, history H, randomized inputs I. If bootstrapping, initialize P with A.
Score each L in P with $H[L].score \leftarrow \sum_{\mathcal{E}} \text{Eval}(L, \mathcal{E})$
for $c = 0$ **to** C **do**
  Sample tournament $T \sim Uniform(P)$
  Parent algorithm $L \leftarrow$ highest score algorithm in T
  Child algorithm $L' \leftarrow$ Mutate(L)
  $H[L'].hash \leftarrow \text{Hash}(L'(I))$
  if $H[L'].hash$ was new **and** $\text{Eval}(L', \mathcal{E}_h) > \alpha$ **then**
    $H[L'].score \leftarrow \sum_{\mathcal{E}} \text{Eval}(L', \mathcal{E})$
  end if
  Add L' to population P
  Remove oldest L from population
end for
Output: Algorithm L with highest score

3.3 EVOLUTIONARY SEARCH METHOD

Evaluating thousands of programs over a range of complex environments is prohibitively expensive, especially if done serially. We adapt a genetic programming method for the search method, regularized evolution (RE), (Real et al., 2019) which has been shown to work for learning supervised learning algorithms (Real et al., 2020) and can be parallelized across compute nodes. RE 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.

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 Real et al. (2019) with several techniques, detailed below, to make the optimization more efficient. The full training procedure is outlined in Algorithm 2.

Functional equivalence check. 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. This technique is similar to the ones used by Alet et al. (2020) and Real et al. (2020). For regularized evolution, 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.

Early hurdles. 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. 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. We use $\text{Eval}(L, \mathcal{E}_h) < \alpha$ as the threshold for poor performance with α chosen empirically.

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.

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. We learn algorithms from scratch by initializing the population of algorithms randomly. An algorithm is sampled by sampling each operation node sequentially in the DAG. For each node, an operation and valid inputs to that operation are sampled uniformly over all possible options.

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. 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. During regularized evolution, the nodes are not frozen, such that it is possible for the existing sub-graph to be completely replaced if a better solution is found.

4 Learned RL Algorithm Results and Analysis

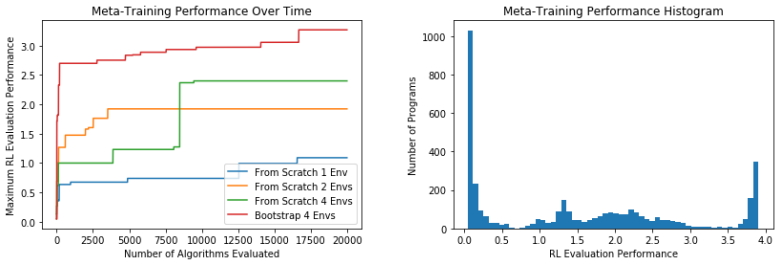
We discuss the training setup and results of our experiments. We highlight two learned algorithms with good generalization performance, DQNClipped and DQNReg, and analyze their structure.

4.1 Training Setup

Meta-Training details: We search over programs with maximum 20 nodes, not including inputs or parameter nodes. A full list of node types is provided in Appendix A. 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. Further meta-training details are in Appendix B.

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. More details are in Appendix C.

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.



(a) Learning curve (b) Performance histogram

Figure 3: Left: Meta-training performance over different number of environments from scratch, and bootstrapping. Plotted as RL evaluation performance (sum of normalized training return across the training environments) over the number of candidate algorithms. More training environments leads to better algorithms. Bootstrapping from DQN speeds up convergence and higher final performance. Right: Meta-training performance histogram for bootstrapped training. Many of the top programs have similar structure (Appendix D.

RL Training details: For training the RL agent, we use the same hyperparameters 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.

4.2 Learning Convergence

Figure 3a 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.

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 (Figure 3a), the learned algorithms are different. The two-environment training setup learns the known TD loss

$$L_{DQN} = (Q(s_t, a_t) - (r_t + \gamma * \max_a Q_{targ}(s_t, a)))^2$$

while the single-environment training setup learns a slight variation $L = (Q(s_t, a_t) - (r_t + \max_a Q_{targ}(s_t, a)))^2$ 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 from scratch versus bootstrapping: In Figure 3a, 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.

4.3 Learned RL Algorithms: DQNClipped and DQNReg

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

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

The first loss function DQNClipped is

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

$L_{\text{DQNClipped}}$ was trained from bootstrapping off DQN using three training environments (LunarLander, MiniGrid-Dynamic-Obstacles-5x5, MiniGrid-LavaGapS5). It outperforms DQN and double-DQN, DDQN, (van Hasselt et al., 2015) on both the training and unseen environments (Figure 4). 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 $Q(s_t, a_t)$.

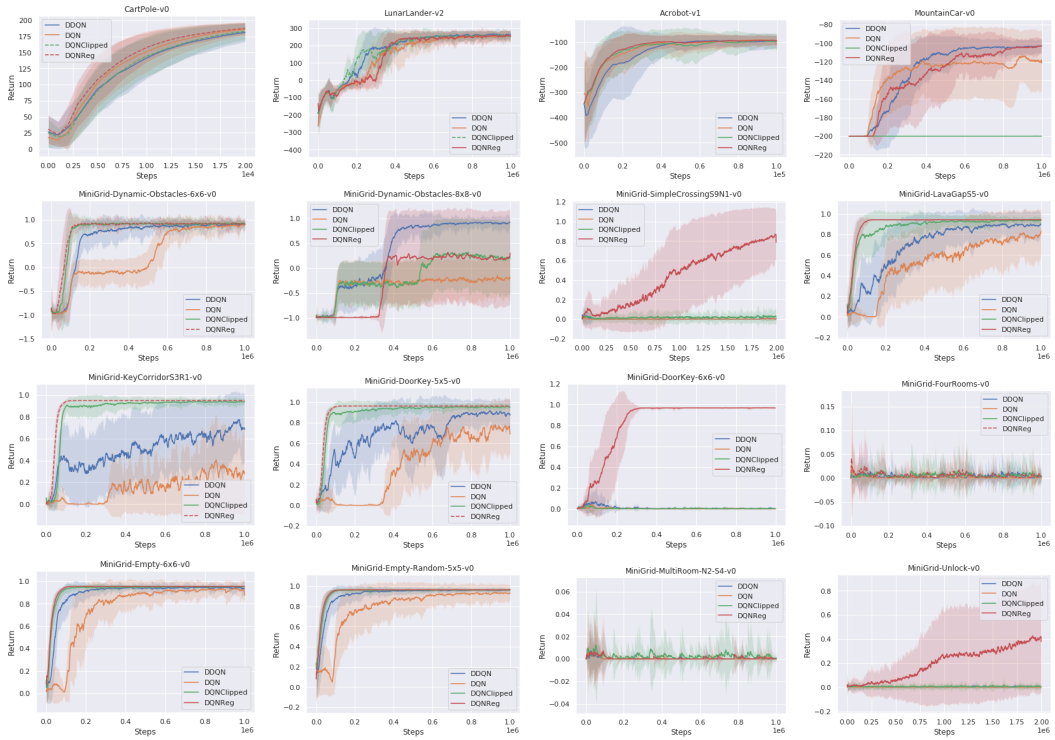


Figure 4: Performance of learned algorithms (DQNClipped and DQNReg) versus baselines (DQN and DDQN) on training and test environments as measured by episode return over 10 training seeds. A dashed line indicates that the algorithm was meta-trained on that environment while a solid line indicates a test environment. DQNReg can match or outperform the baselines on almost all the training and test environments. Shaded regions correspond to 1 standard deviation.

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

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

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).

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**

Table 1: Performance of learned algorithm DQNReg against baselines on several Atari games. Baseline numbers taken from reported papers.

In Table 1, we evaluate DQNReg on a set of Atari games. We use the same architecture as in DQN (Mnih et al., 2013) and use the same no-op evaluation procedure which evaluates a trained policy every 1 million training steps over 200 test episodes. Even though meta-training was on computationally simple, non-image based environments, we find that DQNReg can generalize to image-based environments and outperform baselines. The results for the baselines are taken from their respective papers (Mnih et al., 2013; van Hasselt et al., 2015; Schulman et al., 2017).

These algorithms are related to recently proposed RL algorithms, conservative Q-learning (CQL) (Kumar et al., 2020) and M-DQN (Vieillard et al., 2020). CQL learns a conservative Q-function by augmenting the standard Bellman error objective with a simple Q-value regularizer: $\log \sum_a \exp\left(Q(s_t, a)\right) - Q(s_t, a_t)$ which encourages the agent to stay close to the data distribution while maintaining a maximum entropy policy. DQNReg similarly augments the standard objective with a Q-value regularizer although does so in a different direction by preventing overestimation.

M-DQN modifies DQN by adding the scaled log-policy (using the softmax Q-values) to the immediate reward. Both of these methods can be seen as ways to regularize a value-based policy. This resemblance indicates that our method can find useful structures automatically that are currently being explored manually, and could be used to propose new areas for researchers to explore.

We discover that the best performing algorithms from the experiment which learned DQNReg are consistent, and in the form $L = \delta^2 + k * Q(s_t, a_t)$. This loss could use further analysis and investigation, possibly environment-specific tuning of the parameter k. See Appendix 3 for details.

4.4 Analysis of Learned Algorithms

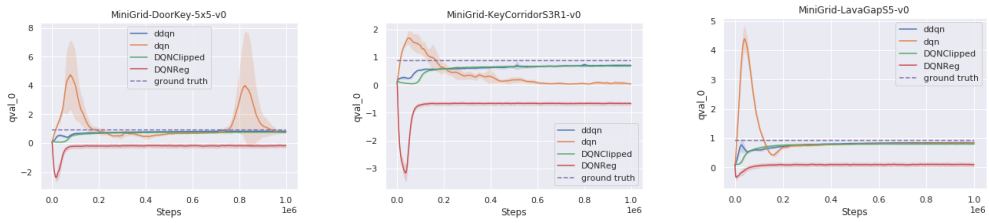


Figure 5: Overestimated value estimates is generally problematic in value-based RL. Our method learns algorithms which regularize the Q-values helping with overestimation. We compare the estimated Q-values for our learned algorithms and baselines with the optimal ground truth Q-values across several environments during training. Estimate is for taking action zero from the initial state of the environment. While DQN overestimates the Q-values, our learned algorithms DQNClipped and DQNReg underestimate the Q-values.

We analyze the learned algorithms to understand their beneficial effect on performance. In Figure 5, 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. 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.

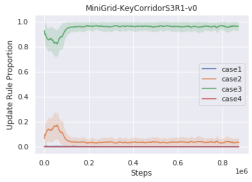


Figure 6: Our learned algorithm, DQNClipped, can be broken down into four update rules where each rule is active under certain conditions. Case 3 corresponds to normal TD learning while case 2 corresponds to minimizing the Q-values. Case 2 is more active in the beginning when value overestimation is a problem and then becomes less active as it is no longer needed.

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 $\max(v_1, v_2) + \max(v_2, v_3)$ 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. In Figure 6, 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 in Figure 6, 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.

5 Conclusion

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. 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.

REFERENCES

Ferran Alet, M. F. Schneider, Tomas Lozano-Perez, and L. Kaelbling. Meta-learning curiosity algorithms. *ArXiv*, abs/2003.05325, 2020.

Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 459–468. JMLR. org, 2017.

S. Bengio, Yoshua Bengio, and J. Cloutier. Use of genetic programming for the search of a new learning rule for neural networks. *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pp. 324–327 vol.1, 1994.

Yoshua Bengio, S. Bengio, and J. Cloutier. Learning a synaptic learning rule. *IJCNN-91-Seattle International Joint Conference on Neural Networks*, ii:969 vol.2–, 1991.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pp. 2787–2794, 2018.

Yevgen Chebotar, Artem Molchanov, Sarah Bechtle, Ludovic Righetti, F. Meier, and Gaurav S. Sukhatme. Meta-learning via learned loss. *ArXiv*, abs/1906.05374, 2019.

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

Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and P. Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *ArXiv*, abs/1611.02779, 2016.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

Aleksandra Faust, Anthony Francis, and Dar Mehta. Evolving rewards to automate reinforcement learning. *arXiv preprint arXiv:1905.07628*, 2019.

Chelsea Finn and S. Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *ArXiv*, abs/1710.11622, 2018.

Chelsea Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *ArXiv*, abs/1703.03400, 2017.

Jörg K. H. Franke, Gregor Köhler, André Biedenkapp, and Frank Hutter. Sample-efficient automated deep reinforcement learning. 2020.

Scott Fujimoto, H. V. Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *ArXiv*, abs/1802.09477, 2018.

Toumas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

Rein Houthooft, Richard Y. Chen, Phillip Isola, Bradly C. Stadie, F. Wolski, Jonathan Ho, and P. Abbeel. Evolved policy gradients. *ArXiv*, abs/1802.04821, 2018.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (eds.). *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31*, pp. 1188–1200. Curran Associates, Inc., 2018.

Louis Kirsch, Sjoerd van Steenkiste, and J. Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. *ArXiv*, abs/1910.04098, 2020.

John Koza. Genetic programming - on the programming of computers by means of natural selection. In *Complex adaptive systems*, 1993.

Aviral Kumar, Aurick Zhou, G. Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. *ArXiv*, abs/2006.04779, 2020.

Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *CoRR*, abs/1712.00559, 2017. URL `http://arxiv.org/abs/1712.00559`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

Junhyuk Oh, Matteo Hessel, Wojciech Marian Czarnecki, Zhongwen Xu, Hado van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. *ArXiv*, abs/2007.08794, 2020.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

Thomas Pierrot, Guillaume Ligner, Scott Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. *NeurIPS*, 2019.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 2902–2911. JMLR.org, 2017.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, volume abs/1802.01548, 2019.

Esteban Real, Chen Liang, David So, and Quoc V. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *ICML*, volume abs/2003.03384, 2020.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Juergen Schmidhuber. Evolutionary principles in self-referential learning. 1987.

Juergen Schmidhuber. A self-referential weight matrix. 1993.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Xingyou Song, Krzysztof Choromanski, Jack Parker-Holder, Yunhao Tang, Wenbo Gao, Aldo Pacchiano, Tamas Sarlos, Deepali Jain, and Yuxiang Yang. Reinforcement learning with chromatic networks for compact architecture search. 2020.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Yunhao Tang and Krzysztof Choromanski. Online hyper-parameter tuning in off-policy learning via evolutionary strategies, 2020.

L. Trujillo and G. Olague. Synthesis of interest point detectors through genetic programming. In *GECCO '06*, 2006.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, 2015.

Nino Vieillard, Olivier Pietquin, and M. Geist. Munchausen reinforcement learning. *ArXiv*, abs/2007.14430, 2020.

Jane X. Wang, Zeb Kurth-Nelson, Hubert Soyer, Joel Z. Leibo, Dhruva Tirumala, Rémi Munos, Charles Blundell, D. Kumaran, and Matt M. Botvinick. Learning to reinforcement learn. *ArXiv*, abs/1611.05763, 2017.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *ICLR*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

A SEARCH LANGUAGE DETAILS

Inputs and outputs to nodes in the computational graph have data types which include state $\mathbb{S}$, action $\mathbb{Z}$, float $\mathbb{R}$, list $List[\mathbb{X}]$, probability $\mathbb{P}$, vector $\mathbb{V}$. The symbol $\mathbb{X}$ indicates it can be of $\mathbb{S}, \mathbb{R}$, or $\mathbb{V}$. We assume that vectors are of fixed length 32 and actions are integers. Operations will broadcast so that for example adding a float variable to a state variable will result in the float being added to each element of the state. This typing allows the learned program to be domain agnostic. The full list of operators is listed below.

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:$\mathbb{S} \to List[\mathbb{R}]$	$\mathbb{S}$	$List[\mathbb{R}]$
(C)NN:$\mathbb{S} \to \mathbb{R}$	$\mathbb{S}$	$\mathbb{R}$
(C)NN:$\mathbb{S} \to \mathbb{V}$	$\mathbb{V}$	$\mathbb{V}$
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)		$\mathbb{R}$

B TRAINING DETAILS

We describe the training details and hyperparameters used. For all environments we use the Adam optimzier with a learning rate of 0.0001.

Common RL training details. All neural networks are MLPs of size (256, 256) with ReLU activations. For optimizing the Q-function parameters we use the Adam optimizer with a learning rate of 0.0001. Target update period is 100. These settings are used for all training and test environments.

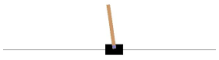
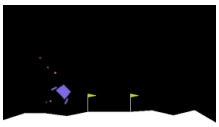
Classical control environments. The value of ϵ is decayed linearly from 1 to 0.05 over 1000 steps. CartPole, Acrobat, and MountainCar are trained for 400 episodes and LunarLander is trained for 1000 episodes.

MiniGrid environments. The value of ϵ is decayed linearly from 1 to 0.05 over 10^5 steps. During meta-training, MiniGrid environments are trained for $5 * 10^5$ steps.

Atari environments. We use the same neural network architecture as in Mnih et al. (2013). Target update period is $1,000$. The value of ϵ is decayed linearly from 1 to 0.1 over 10^6 steps. For evaluation, we use the no-op start condition as in Mnih et al. (2013) where the agent will output the no-op action for x steps where x is a random integer drawn between $[1, 30]$. The evaluation policy uses an ϵ of 0.001 and is evaluated every 10^6 steps for 100 episodes. The best training snapshot is reported.

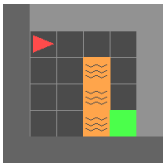
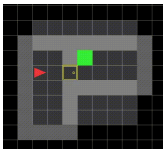
C ENVIRONMENT DETAILS

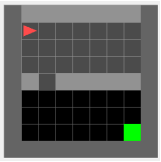
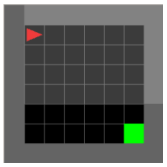
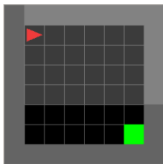
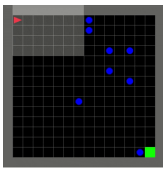
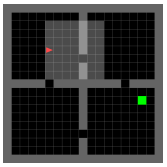
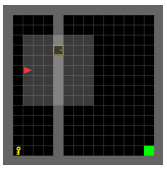
We describe the classical control environments below. CartPole and LunarLander are dense reward while Acrobat and MountainCar are sparse reward.

	Task ID	Description
	CartPole-v0	The agent must balance a pole on top of a cart by applying a force of $+1$ or -1 to the cart. A reward of $+1$ is provided for each timestep the pole remains upright.
	LunarLander-v2	The agent controls a lander by firing one of four thrusters and must land it on the landing pad.
	Acrobat-v1	The goal is to swing a 2-link system upright to a given height by applying 1, 0, or -1 torque on the join between the two links.
	MountainCar-v0	The goal is to drive up the mountain on the right by first driving back and forth to build up momentum.

We describe the MiniGrid environments below. The input to the agent is a fully observed grid which is encoded as an NxNx3 size array where N is the grid size. The 1st channel contains the index of the object type at that location (out of 11 possible objects), the 2nd channel contains the color of the object (out of 6 possible colors), and the 3rd channel contains the orientation of the agent out of 4 cardinal directions. This encoding is then flattened and fed into an MLP. There are 7 possible actions (turn left, turn right, forward, pickup, drop, toggle, done).

Unless stated otherwise, all tasks are sparse reward tasks with a reward of 1 for completing the task. Max steps is set to 100. A size such as 5x5 in the environment name refers to a grid size with width and height of 5 cells.

	Task ID	Description
	KeyCorridorS3R1-v0	The agent has to find a key hidden in one room and then use it to pickup an object behind a locked door in another room. This tests sequential subgoal completion.
	LavaGapS5-v0	The agent has to reach the green goal square without touching the lava which will terminate the episode with zero reward. This tests safety and safe exploration.
	MultiRoom-N2-S4-v0	The agent must open a door to get to the green goal square in the next room.

	SimpleCrossingS9N1-v0	The agent has to reach the green goal square on the other corner of the room and navigate around walls.
	Empty-v0	The agent has to reach the green goal square in an empty room.
	EmptyRandom-v0	The agent has to reach the green goal square in an empty room but is initialized to a random location.
	Dynamic-Obstacles-v0	The agent has to reach the green goal square without colliding with any blue obstacles which move around randomly. If the agent collides with an obstacle it receives a reward of -1 and the episode terminates.
	FourRooms-v0	The agent must navigate in a maze composed of four rooms. Both the agent and goal square are randomly placed in any of the four rooms.
	DoorKey-v0	The agent must pick up a key to unlock a door to enter another room and get to the green goal square.

D GRAPH DISTRIBUTION ANALYSIS

We look at the distribution of top performing graphs and find similarities in their structure. This is summarized in Table 3 where we describe the equations of learned algorithms for differing ranks (if sorted by score). The best performing algorithms from the experiment which learned DQNReg are all variants of adding $Q(s_t, a_t)$ to the standard TD loss in some form, $\delta^2 + k * Q(s_t, a_t)$. We think this kind of loss could use further investigation and that while we did not tune the value of k, this could also be tuned per environment. In Figure 3b, we show the distribution of scores for all non-duplicate programs that have been evaluated.

Raw Equation	Simplified Equation	Score	Rank
$\delta^2 + 0.1 * Q(s_t, a_t) + r_t - (\gamma * Q_{targ} - 0.1 * Q(s_t, a_t))$	$\delta^2 + 0.2 * Q(s_t, a_t)$	3.905	2
$\delta^2 + 0.1 * Q(s_t, a_t) - \gamma + Q_{targ}$	$\delta^2 + 0.1 * Q(s_t, a_t)$	3.904	3
$\delta^2 - (\gamma * Q_{targ} - 0.1 * Q(s_t, a_t))$	$\delta^2 + 0.1 * Q(s_t, a_t)$	3.903	4
$\delta^2 + Q_{targ} + 0.1 * Q(s_t, a_t) - \gamma$	$\delta^2 + 0.1 * Q(s_t, a_t)$	3.902	5
$\delta^2 - (0.1 * Q(s_t, a_t) - Y_t)^2$	$\delta^2 - (0.1 * Q(s_t, a_t) - Y_t)^2$	3.898	6
$\delta^2 + ((r_t + \gamma * Q_{targ} + Q(s_t, a_t)) * (\gamma - \max(\gamma, 0.1 * Q(s_t, a_t)) - \gamma * Q_{targ} - 0.1 * Q(s_t, a_t))$	NA	3.846	11146
$\delta^2 + (\delta^2 + 0.1 * Q(s_t, a_t))^2$	NA	3.65	12146
$\delta^2 + Q(s_t, a_t)$	$\delta^2 + Q(s_t, a_t)$	2.8	12446
δ^2	δ^2	2.28	13246

Table 3: Other programs learned in learning DQNReg which is rank 1 with score 3.907. Rank is if scores are sorted in decreasing order. Score is the sum of normalized RL training performance across four environments. The simplified equations contains only the relevant parts for minimizing the equation output. Q_{targ} refers to $\max_a Q_{targ}(s_{t+1}, a)$.