EvoReward: Reinforcement Learning with Evolving Auxiliary Reward

Aravind Balakrishnan
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, N2L 3G1
a4balakr@uwaterloo.ca
CS885: Reinforcement Learning Spring 2018

Abstract

The right reward signal to a reinforcement learning (RL) agent can mean the difference between it converging fast and not converging at all. The external reward from the environment may not always be optimal in nudging the agent in the right direction. Shaping reward functions and reward augmentations have been widely explored, but hand-crafting these are not only hard for human designers but also may not be optimal. This paper provides an additional reward to an RL agent through a reward augmentation function, called EvoReward, which evolves each episode using genetic algorithms. EvoReward takes in the current environment state and agent action to provide a good direction for the agent, even in the absence of external rewards. Experiments with Q-learning and Deep Q-learning agents augmented with EvoReward show that it significantly improves performance in both dense and sparsely rewarded environments and that it has the potential to be practically used.

1 Introduction

A reinforcement learning (RL) agent interacts with the unknown environment and uses these experiences to learn a policy that maximizes a reward signal. This reward signal that is used to nudge the agent in the right direction usually comes from the environment itself, for example, the score obtained from a video game. This works well in many cases, as demonstrated by numerous successes from the RL research community. However, this reward function needs to be manually specified and this may not always be the most optimum signal. A more serious concern is that this fails miserably in environments that have sparse rewards, which requires the agent to learn a long series of actions without encountering any reward signal. This makes it difficult for the agent to attribute which action it took led to the reward it received.

In order to offset these problems, inspiration has been taken from intrinsic motivation literature [1] to provide the agent with additional rewards that are not derived from the external environment. *Intrinsic motivation* is the 'the doing of an activity for its inherent satisfaction rather than for some separable consequence' [1]. For example, *novelty* is where the agent seeks states that it has never seen before and it is a common form of reward augmentation in sparse reward environments, even in high dimensional state spaces [2].

However, these reward augmentations also need to be hand-crafted depending upon the domain; for example, using novelty in an environment that does not require much exploration will make the agent frequently take less optimal unexplored paths leading to slower convergence. Rather than modeling a reward augmentation by hand, automatically searching for one can be beneficial due to a number of factors: 1) A good augmentation can significantly improve the learning rate, 2) It

Submitted to Reinforcement Learning (CS885, 2018 Spring). Do not distribute.

can possibly exploit features of the environment that are not outwardly apparent, 3) It can provide a useful learning signal to the agent in the absence of an external reward.

In this paper, an RL agent is provided with an additional reward from a reward augmentation function. This function is parameterized on the agent's action and the environment in order to exploit features of the environment that may not be obvious to a human designer. A genetic algorithm is used to evolve the model that generates this additional reward and it is shown that it does not lead to too much increase in training time because it is easy parallelize. The RL algorithm with the augmented reward obtained through this process is demonstrated to converge faster than the same base algorithm without the augmentation on both a toy problem as well as a classic OpenAI gym [3] environment. It is shown that the additional reward corresponds to features of the environment. Moreover, experiments also validate that it provides a good direction to the agent in an environment with sparse rewards.

2 Related Work

A large part of reward augmentations seen in literature are based on intrinsic motivation. This is inspired by biology, seeing how humans and other animals have an internal drive to interact with their environment. Bellemare *et al.* [2] popularized novelty as a useful augmentation using a efficient method of computing the novelty of a state in high dimensional state spaces called *pseudocounts*. They achieved state-of-the-art scores in the hard Atari game Montezuma's Revenge, which is infamous for its sparse rewards. Compared to these augmentations, the augmentation in this paper does not need to be well-defined, but rather, adjusts to the environment the agent is in.

Using genetic algorithms in RL has gained some attention recently due to the work by Salimans *et al.* [4] where they show that evolution strategies (ES) is a strong contender to standard RL techniques, even in complex high-dimensional environments. Their implementation is highly scalable as it can be distributed across thousands of workers and it not only performs well in a sparse environment, but also has a few more additional advantages to RL techniques. Although a fixed reward function is used in ES, Houthooft *et al.* builds on this to introduce an approach which evolves a differentiable loss function. This loss is parameterized on convolutions over the agent's experience and the agent tries to minimize this using policy gradient method, thereby obtaining high rewards [5]. Comparatively, the approach taken in this paper does not alter the learning algorithm in any way, but rather choosing to only augment the reward and therefore can be added to any off-the-shelf RL technique.

The idea of searching for reward functions is also not new and have been explored by others [6], but only a few use genetic programming. Niekum *et al.* [7] uses the same idea as this paper, where they evolve the reward function of a Q-Learning agent. In their paper, they use PushGP, a stack-based genetic programming system, to evolve operations and control structures on some features of the environment. But they only tested it in a toy sparse environment problem. In this paper, the additional reward is generated through a linear combination of a set of weights and environment features, which produces a value within some bounds. The weights are evolved through genetic operators which makes this much more scalable because it can easily be replaced by neural networks for more complex environments. In addition, EvoReward was tested in an environment with a large state space as well.

3 Background

3.1 Reinforcement Learning

The standard RL framework is where an agent acts on an environment that provides a feedback reward signal, using which the agent learns its behaviour policy, over a series of time steps. The domain considered here is a Markov Decision Process(MDP), which is defined as the tuple $\langle S, A, R, T, \varepsilon \rangle$, where S is a set of states, A is a set of actions that can be taken, $R(s, a, s')$ is a function representing the reward incurred from transitioning from state s to state s' by taking action $a, T(s, a, s')$ is a function representing the probability of transitioning from s to s' by taking action a, and ε is a set of terminal states that, once reached, prevent any future action.

3.2 Q-learning

The agent's objective is to maximize the cumulative reward it receives. This is usually the discounted cumulative reward, given by $\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$, where r_t is the external reward at time t and γ is the discount factor that specifies to what extent the agent considers future rewards. The agent maintains a Q-function that gives the Q-value of a (s, a) pair, which is the approximation of $Q^*(s, a)$ - the expected discounted cumulative reward to be received if the agent were to take action a in state s and follows the optimal policy thereafter. Q-learning iteratively computes the Q-values using the following update rule at time t, where $\hat{Q}$ is the current approximation of Q^*:

$$\hat{Q}(s_t, a_t) = \hat{Q}(s_t, a_t) + \alpha[R(s_t, a_t, s_{t+1}) + \gamma \max_{a \in A} \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t)]$$

3.3 Genetic Algorithm

Genetic Algorithm (GA) is a metaheuristic optimization algorithm, that uses mechanisms inspired by biological evolution, such as natural selection, mutation and crossover to arrive at solutions to optimization and search problems. The process starts with a population of candidate solutions (called *individuals*) which are randomly initialized. Each individual has some features (its *chromosomes*) that can be altered; which traditionally are strings of 0s and 1s. It is an iterative process, where each iteration is called a *generation*.

At every generation, the fitness score of each individual is evaluated. The *fitness* is a measure of how well the solution does in the optimization problem at hand. The individuals with higher fitness are selected stochastically (*selection*) and the selected individuals are *bred* to produce the population of the next generation through a combination of genetic operators: *crossover* (also called *recombination*) and *mutation*. Each *child* solution in the next generation is produced by breeding two *parent* solutions in the selected pool of the current generation. Crossover combines genetic information of two parents to generate an offspring in order to transfer the best features to the next generation, whereas mutation alters one or more values in the chromosome of the child to maintain genetic diversity and encourage exploration. The algorithm usually ends when an optimum solution is found or when a specific number of generations has passed or when successive iterations no longer produce better results.

4 Methodology

4.1 Algorithm

The basic concept behind EvoReward is to evolve a reward augmentation that helps the RL agent perform better on the problem domain. To initiate the GA algorithm, we need a population, M of n candidate solutions. Here, an individual in the population is an RL agent, $m \in M$ with each having the external reward, r received from the environment augmented at every timestep, t by a function, $EvoReward$. $EvoReward$ takes in the current state, s_t and action, a performed by the agent and returns an additional reward value, r'. It has a set of weights, $w_{m,a}$ $\forall a \in A$, where the $w_{m,a}$ is a vector with size equal to the size of $\hat{s}$ (which is s that has been flattened to a vector). When an action is executed by the agent, a linear combination of $w_{m,a}$ and s produces the additional reward. The chromosome, in this case, is w_m.

$$r' = w_{m,a}.\hat{s}$$
$$AugmentedReward = r + r'$$

At the start of the GA algorithm, a population of agents are created with random weight vectors, $w_m \forall m \in M$. Each generation of agents plays one episode, e in the environment and after all agents complete one episode, the fitness, f of each agent is calculated. The fitness is the average of the episode rewards, $R_{m,e}$ obtained by the agent over the last k episodes. Agents with the best fitness are selected stochastically and their chromosomes (w_m) undergo crossover and mutation (with a probability, p_c and p_m respectively) to produce the chromosomes for the next generation of agents.

This whole process continues until an agent in the population solves the problem or a maximum number of episodes (generations) has been reached.

Algorithm 1: RL algorithm with EvoReward

initialize n agents to give population,M where an individual is $m \in M$
initialize $EvoReward$ function with n random weight vectors, w_m
for *episode, e = 1,...,maxEpisodes or until problem solved* **do**
  for *each agent* $m \in M$ **do**
    $R_{m,e}$ = 0
    for *step t = 1,...,maxSteps or until end of episode* **do**
      get $reward_t$, $state_{t+1}$ using m by taking action, a_t in environment at $state_t$
      add r_t to $R_{m,e}$ to accumulate episode reward for agent
      get r'_t from $EvoReward(m, state_t, a_t)$
      $AugmentedReward_t = r_t + r'_t$
      train agent m with tuple ($state_t$, a_t, $AugmentedReward_t$, $state_{t+1}$)
    end
  end
  calculate fitness, f_m = mean of $R_{m,e}$ over last k episodes
  crossover and mutate w_m with probability, p_c and p_m in $EvoReward$ using $f_m \forall M$
end

The evolution process in $EvoReward$ takes place as follows:

- Selection: Select n agents from M with replacement according to probability distribution given by $f_m \ \forall m \in M$ to give selected population, M_s.
- Crossover: With probability p_c, perform crossover for each agent $m \in M_s$ to produce a child, c for each. If no crossover is performed, the weight vectors in c remain the same as m. To perform crossover, choose another agent, $m_a \in M_s$ randomly and copy values from random locations in its weight vector, w_{m_a} to corresponding location in w_m.
- Mutate: For each child, c created in above step, mutate weight vectors w_c by randomizing each value with a probability, p_m.
- Set the newly created children as the population M for the next episode/generation.

4.2 Implementation Details

Any RL technique can be augmented with EvoReward but for the purposes of this paper, Q-learning and Deep Q-Networks (DQN) were chosen depending upon whether the environment is simple or complex. The Q-learning algorithm uses a table maintaining Q-values for each (state, action) pair as the Q-function. This is suitable for simple environments with small state spaces. A discount factor of 0.95 and exploration probability, ϵ of 0.3 is used. The learning rate, α is set to the inverse of the state visit count.

The Deep Q-Network is used for more complex environments. The Q-function is modelled using a neural network with two hidden layers of 16 nodes each. The activation function used is Rectified Linear Units (ReLU) and training is performed using the Adam optimizer. A discount factor of 0.99 and a learning rate of 0.001 is used for training. The exploration probability, ϵ is decayed exponentially with the number of timesteps. An experience replay memory buffer of size 10000 is used to train with a batch size of 32. No target network is used for training.

The GA algorithm for evolving rewards has a crossover probability of 0.4 and mutation probability of 0.01. The bounds for the additional reward is chosen depending upon bounds of the external reward from the environment. The agents in the population can run simultaneously using multiprocessing since only the fitness needs to be shared between agents, and that too only at the end of each episode. Therefore, the agents are executed in batches of 8 (or the number of available CPU cores) with each running only one episode. After finishing an episode, it shares its fitness to a high-level controller and waits for all other agents to finish before starting the next iteration. In fact, EvoRe-

ward can be parallelized efficiently even across networks, since only one value needs to be shared by each agent.

5 Experimental Evaluation

Evaluation is done by comparing the performance of the Q-Learning or DQN algorithm with and without the EvoReward augmentation. Experiments on three different environments are performed to evaluate different aspects of the performance:

- A toy maze problem
- A large state space dense-reward OpenAI gym [3] environment, CartPole-v0
- A sparse reward grid world environment, MiniGrid[8]

5.1 Toy Maze Problem

The environment is a simple maze MDP, a grid world, of size 4x4. The states in the grid are numbered in row-major order starting from 0 to 15. The agent always starts at 0 and the goal state is 15. Each step taken gives the agent a reward of -1. The goal state gives a reward of 100. 9 is a bad state the gives a negative reward of -70. There is an extra state, 16, which is an absorbing state that the agent transitions to after visiting the goal state. Thus, there are 17 states in total (including the end state) and 4 actions (up, down, left, right). The bounds for the additional reward were set to $[-10, 10]$.

Since the state space is small, a normal Q-learning algorithm is used. For EvoReward, I use weight vectors of size 4x17(action space x state space). The states are one-hot encoded before passing to EvoReward function to allow for the linear combination of weights and states (which produces the additional reward). I ran an experiment comparing normal Q-learning and Q-learning with EvoReward of varying population sizes for 200 episodes and 100 steps per episode over 100 trials. This serves to show the effect of the number of agents in a population to the performance. Figure 1 shows the averaged cumulative scores per episode for each model. For the models with EvoReward, I plot the highest episode score achieved among all the agents in the population.

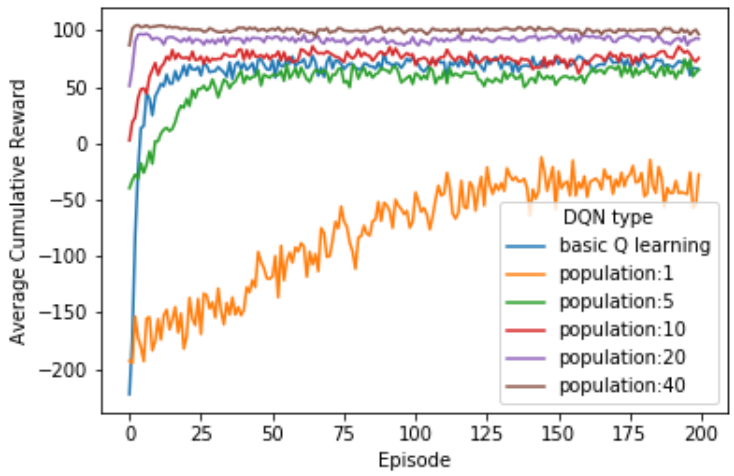

Figure 1: Performance of EvoReward of varying populations on toy maze problem over 100 trials

It can be seen that basic Q-learning performs similar (but slightly worse) to an EvoReward model with a population of 10. The population of 1, as expected, perform very poorly because the weights do not evolve. The population of 5 converges slower but reaches the same final score as the normal Q-learning algorithm. EvoReward with population 10 and above perform very well, with even the starting score in the first episode being better than the best score of basic Q-learning. This

is because, out of the large population of random reward weights, at least one would have had a good combination of starting weights (over 100 trials), that lead it to the goal state faster. It should be noted that since the EvoReward models were run using multiprocessing over 8 cores, the runtime till population 10 is almost the same as normal Q-learning. In addition, from the reward augmentation weight vector, it can be seen that the additional rewards provided to the agent give a solid direction. For example, actions that lead the agent into the bad state produce negative rewards, thereby discouraging the agent from taking that action.

5.2 CartPole-v0

CartPole-v0 is a classic control environment available in the OpenAI gym [3] package that is used for evaluating RL algorithms. The goal is to prevent pole attached to a cart from falling by increasing or reducing the cart's velocity. The state space consists of 4 observations, each being a floating point number between some bounds, which makes this a large state space problem. The action space is 2 - to push the cart to the left, or to the right. The agent receives a reward of 1 for every timestep, making this a dense reward environment. The episode ends when the pole becomes unbalanced or after 200 timesteps, which makes 199 the maximum possible episode reward that can be obtained. For this paper, the environment is considered solved when the average reward over the last 20 episodes is greater than 195.0. Therefore, the performance is not about the maximum score but about the number of episodes taken to solve it.

I ran an experiment comparing DQN with and without EvoReward (with a population of 20) over 10 trials. This population was chosen from the previous experiment as it gave the best trade-off between performance and runtime. The bounds for additional reward from EvoReward were set to $[-10, 10]$. As can be seen from Figure 2, the EvoReward model solves it much faster than normal DQN with an average of 118 episodes (Min:55, Max 366) and a standard deviation of 85. Whereas, Normal DQN took an average 156 episodes (Min: 83, Max: 336) and a standard deviation of 92. EvoReward (parallelized over 8 cores) took on average approximately 2.5 times longer than normal DQN for completing one trial. But since EvoReward can be parallelized easily, the time taken would not be an issue for scalability.

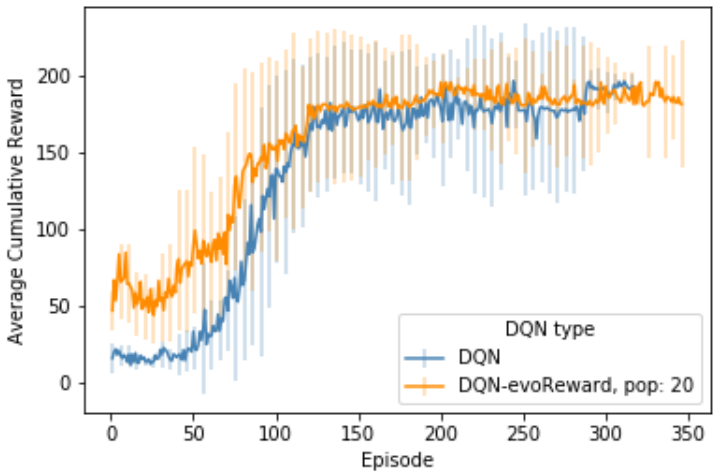

Figure 2: Performance of EvoReward of population size 20 on CartPole-v0 over 10 trials

5.3 MiniGrid

MiniGrid[8] has many lightweight environments with sparse rewards, ranging from a simple test maze to complex puzzle solving mazes. One of the provided environment, called MiniGrid-Empty-6x6-v0, is similar to the first toy problem. It is a 6x6 grid world with the agent starting at top left corner and the goal state at the bottom-right corner. The agent receives a reward only when it reaches the goal state. This reward is further penalized for the number of steps taken to reach the goal. The

action space of the agent is 7, but only the first 4 are related to movement (the rest of the actions are useful only in other environments) and this adds another layer of sparsity. Unlike the first problem, this agent has a field of view and can observe a 7x7 grid in front of it as can be seen in Figure 3. This means that as the agent turns, the observation rotates. The maximum reward possible in this problem is 0.935. Therefore, the bounds for additional rewards from EvoReward were set to $[-1, 1]$.

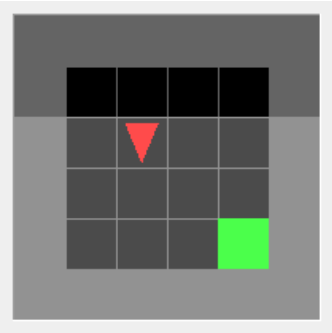

Figure 3: MiniGrid-Empty-6x6-v0

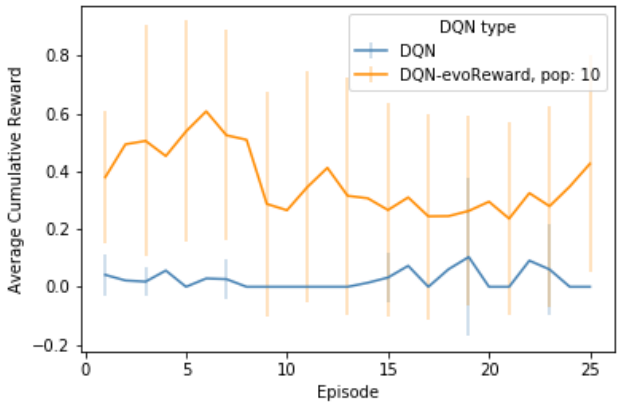

Figure 4: Performance of EvoReward of population 10 over 8 trials on MiniGrid-Empty-6x6-v0

I ran the same experiment as before with the DQN, but this time, with a population of 10 for EvoReward (as the environment was comparatively simple), for 25 episodes over 8 trials. Figure 3 shows the results of this experiment and it can be seen that EvoReward performs better, albeit not as good as the previous two results. But the average reward for a normal DQN is very close to 0. On observing the cumulative reward graph of an individual trial of normal DQN, spikes in reward appear occasionally which corresponds to when the agent somehow reaches the goal. But the DQN never learns anything from this and continues to obtain 0 on the subsequent episodes. I ran the experiment on the normal DQN for 100 episodes just to see if it will converge eventually, but the result was the same.

The EvoReward however, once it obtains a reward, can be seen to improve its score from thereafter most of the time. It reached close to the maximum score multiple times during the trials but it does not converge and starts regressing after a while. This can be attributed to the fact that EvoReward uses a linear combination of weights and environment features to generate the additional rewards. The 7x7 observation grid of the agent is flattened to achieve this and a lot of spatial information is lost in this process. This leads to EvoReward not being able to represent the environment correctly and thus, failing to supply a consistently useful direction to the agent. This is aggravated by the fact the observation matrix rotates as the agent rotates, so a linear combination is bound to fail. This can possibly be alleviated by using a Convoluted Neural Network (CNN) to generate the additional reward (where the weights of the CNN are evolved using GA).

6 Conclusion

From the experiments, it can be clearly seen that the RL algorithms augmented with evolving rewards converge faster and also achieves higher scores than the base algorithm. They also perform well in environments with sparse rewards, but not as much as expected. This can possibly be due to the design of the reward generation model not meeting the complexity of the environment, but more experiments need to be conducted to shed more light on this.

Due to multiprocessing, the EvoReward model did not take too much time to train compared to the base algorithm. If the number of cores available are more than or equal to the population of the agents, EvoReward can be made to be almost as fast as the base algorithm. Altogether, this is a promising direction for research especially since Evolutionary Strategies for RL [4] showed that this technique is highly scalable. The best advantage, however, is the fact that EvoReward can be

coupled with any off-the-shelf RL algorithm and it would valuable to see how EvoReward would fare in combination with ES.

However, a few problems are immediately noticeable with EvoReward and require further inquiry. The additional rewards produced depend on the accuracy of the environment representation by EvoReward. But in order to improve the representation, more complex models with more parameters are required and this may not be scalable and needs to be investigated. Another problem is that the evolution is mostly random and has no direction other than the fitness. Therefore, the variance between different trials can be huge. Network parallelization is another direction for future work which can be very useful in increasing the population size beyond the cores available locally and this could dramatically improve the performance.

References

[1] P. Y. Oudeyer, F. Kaplan, and V. V. Hafner, "Intrinsic motivation systems for autonomous mental development," *IEEE Transactions on Evolutionary Computation*, vol. 11, pp. 265–286, April 2007.

[2] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," *CoRR*, vol. abs/1606.01868, 2016.

[3] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba, "Multi-goal reinforcement learning: Challenging robotics environments and request for research," 2018.

[4] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," *ArXiv e-prints*, Mar. 2017.

[5] R. Houthooft, R. Y. Chen, P. Isola, B. C. Stadie, F. Wolski, J. Ho, and P. Abbeel, "Evolved policy gradients," *CoRR*, vol. abs/1802.04821, 2018.

[6] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg, "Intrinsically motivated reinforcement learning: An evolutionary perspective," *IEEE Transactions on Autonomous Mental Development*, vol. 2, pp. 70–82, June 2010.

[7] S. Niekum, A. G. Barto, and L. Spector, "Genetic programming for reward function search," *IEEE Transactions on Autonomous Mental Development*, vol. 2, pp. 83–90, June 2010.

[8] L. W. Maxime Chevalier-Boisvert, "Minimalistic gridworld environment for openai gym." https://github.com/maximecb/gym-minigrid, 2018.