Learning Curriculum Policies for Reinforcement Learning*

Sanmit Narvekar
Department of Computer Science
University of Texas at Austin
sanmit@cs.utexas.edu

Peter Stone
Department of Computer Science
University of Texas at Austin
pstone@cs.utexas.edu

Abstract

Curriculum learning in reinforcement learning is a training methodology that seeks to speed up learning of a difficult target task, by first training on a series of simpler tasks and transferring the knowledge acquired to the target task. Automatically choosing a sequence of such tasks (i.e. a *curriculum*) is an open problem that has been the subject of much recent work in this area. In this paper, we build upon a recent method for curriculum design, which formulates the curriculum sequencing problem as a Markov Decision Process. We extend this model to handle multiple transfer learning algorithms, and show for the first time that a *curriculum policy* over this MDP can be learned from experience. We explore various representations that make this possible, and evaluate our approach by learning curriculum policies for multiple agents in two different domains. The results show that our method produces curricula that can train agents to perform on a target task as fast or faster than existing methods.

1 Introduction

Over the past two decades, transfer learning [5, 12] is one of several lines of research that have sought to increase the efficiency of training reinforcement learning agents. In transfer learning, agents train on simple *source* tasks, and transfer knowledge acquired to improve learning on a more difficult *target* task. Typically, this has been a one-shot process, where information is transferred from one or more sources directly to the target task. However, as the problems we task reinforcement learning agents with become ever more complex, it may be beneficial (and even necessary) to gradually acquire skills over multiple tasks *in sequence*, where each subsequent task builds upon knowledge gained in a previous task. This insight is the basis for *curriculum learning* [1, 6].

The goal of curriculum learning is to design a sequence of source tasks (i.e. a curriculum) for an agent to train on, such that after training on that sequence, learning speed or performance on a *target* task is improved. Automatically designing a curriculum is an open problem that has only recently begun to be examined [4, 3, 8, 2, 7, 10]. One recent approach [7] proposed formulating the selection of tasks using a (meta-level) curriculum Markov Decision Process (MDP). A policy over this MDP, called a curriculum policy, maps from the current knowledge of an RL agent to the task it should learn next. However they did not demonstrate whether the curriculum policy could actually be learned. Instead, they proposed an algorithm to approximate a single execution of the curriculum policy, corresponding to an individual curriculum.

Until now, it was not known if curriculum policies could be learned: that is, whether it is possible to find a representation that is both compact enough and generalizable enough to facilitate learning. Our main contribution is to demonstrate that curriculum policies can indeed be learned, and we explore various representations that make this possible. In addition, we generalize the curriculum MDP model

*A full-length, complete version of this paper can be found at: https://arxiv.org/abs/1812.00285

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

proposed by Narvekar et al. [7] to handle different kinds of transfer learning algorithms. Finally, we empirically show that the curricula produced by our method are at least as good as, or better than those produced by two existing curriculum methods on two different domains. We also demonstrate that curriculum policies can be learned for agents with different state and action spaces, agents that use different transfer learning algorithms, and different representations for the curriculum MDP.

2 Learning Curriculum Policies

Our work extends the model proposed by Narvekar et al. [7], which formulates curriculum generation as an interaction between two separate Markov Decision Processes: one is for a *learning agent* that is trying to solve a specific target task MDP M_t, as is the standard case in reinforcement learning. The second is a *curriculum agent*, which interacts in a second, higher level *curriculum MDP* (CMDP), and whose goal is to sequence tasks M for the learning agent. A curriculum MDP was defined to be an MDP where: (1) the state space $\mathcal{S}^C$ consists of all policies the learning agent can represent; (2) the action space $\mathcal{A}^C$ is the set of tasks to train on; (3) the transition function p^C reflects the change in the learning agent's policy as a result of learning a task; and (4) the reward r^C is the cost in time to learn the task. The starting state corresponds to a random learning agent policy, and terminal states are learning agent policies that can achieve a predefined performance level on a target task.

One limitation of the previous definition is that it assumes the underlying transfer learning mechanism is value function or policy transfer. Intuitively, the state space of a CMDP should represent different states of knowledge. The goal of the agent is to reach a state of knowledge that allows solving the target task in the least amount of time. As a case study, in this paper we consider learning agents that use two different transfer algorithms: value function transfer and potential-based reward shaping. In value function transfer, the value function learned in a source task is used to initialize the value function in a subsequent task. In potential-based reward shaping, the value functions of a set of source tasks are used as potential functions to create a shaping reward for a next task (see Svetlik et al. [10] for details). Thus, for an agent that uses reward shaping, the CMDP state is represented as a set of potential functions, and the goal is to find a state whose potentials allow learning the target task as fast as possible.

2.1 Representing CMDP State Space

In the standard reinforcement learning setting, the agent perceives its state as a set of state variables. These are typically used to extract basis features $\boldsymbol{\phi}(\mathbf{s})$, which transform the state variables into a space more suitable for learning and use in function approximation. Given these features and a functional form, the goal is to learn weights $\boldsymbol{\theta}$ for the value function or policy. We introduce an analagous process for curriculum design agents acting in CMDPs. We will ground the discussion assuming both the learning and curriculum agents use a value-function-transfer-based approach. However, the idea is easily applied to a reward-shaping setting by noting that the reward can also be expressed as a product of state features and weights $r(s,a) = \boldsymbol{\phi}(s,a) \cdot \boldsymbol{\theta}$.

The first question is how to represent the raw state variables of a CMDP state. The representation chosen must be able to represent *any* policy the underlying learning agent can represent (or equivalently, any shaping reward). Assuming the learning agent derives its policy from an action-value function $Q_{\boldsymbol{\theta}}(s,a)$, the form of the function (such as network architecture, etc.) determines the class of policies that can be represented. This functional form $Q_{\boldsymbol{\theta}}(s,a)$ and how learning agent features $\boldsymbol{\phi}$ are extracted are fixed. Thus, it is specific values of the weight vector $\boldsymbol{\theta}$ that actually instantiates a policy in this class. Therefore, it follows that we can represent the state variables for a particular CMDP state s^C using the instantiated vector of learning agent weights: $s^C = \boldsymbol{\theta}$. Different instantiations of $\boldsymbol{\theta}$ correspond to different CMDP states. Typically, these weights $\boldsymbol{\theta}$ will take on continuous values. Therefore, in order to learn a CMDP action-value function $Q^C_{\boldsymbol{\theta}^C}(s^C, a^C)$, it will be necessary to do some kind of function approximation.

First we consider one way of extracting CMDP state features and performing function approximation, when the domain has a finite state space. Assume again the learning agent learns an action-value function $Q_{\boldsymbol{\theta}}(s,a)$, for each state-action pair in the task. We can represent Q as a linear function of "one-hot" features $\boldsymbol{\phi}(s,a)$ and their associated weights $\boldsymbol{\theta}$ as $Q_{\boldsymbol{\theta}}(s,a) = \boldsymbol{\theta} \cdot \boldsymbol{\phi}(s,a)$. In other words, all the action-values are stored in $\boldsymbol{\theta}$, and $\boldsymbol{\phi}(s,a)$ is a one-hot vector used to select the activated action-value from $\boldsymbol{\theta}$. One approach for designing $\boldsymbol{\phi}^C$ is to utilize tile coding [9] over subsets of action-values in $\boldsymbol{\theta}$. Specifically, the idea is to create a separate tiling for each primitive state s in

the domain. Each such tiling will be defined over the action-values in $\boldsymbol{\theta}$ associated with state s. Thus, this creates $|\mathcal{S}|$ tiling groups, where each group is defined over $|\mathcal{A}|$ CMDP state variables (i.e. action-values). To create the feature space, multiple overlapping tilings are laid over each group.

The representation problem is harder in the continuous case, since each parameter θ_i is not local to a state, and we cannot use a state-by-state approach to create a basis feature space. In principle, any continuous feature extraction and function approximation scheme can form the basis of $\boldsymbol{\phi}^C$ (tile coding, neural nets, etc.), and would need to be tailored to the domain.

3 Experiments

We evaluated learning curriculum policies for agents on a grid world domain used in previous work [7] (See Appendix A for more details) as well as a Ms. Pac-Man domain (see Appendix B). We will show the results as CMDP learning curves. The x-axis on these learning curves are over *CMDP episodes*. Each CMDP episode represents an execution of the current curriculum policy for the agent. Thus, multiple tasks are selected over the course of a single episode, with each task taking a varying number of steps/episodes, which contributes to the cost on the y-axis. Tasks are selected until the desired performance can be achieved in the target task, at which point the CMDP episode is terminated. In short, the curves show how long it would take to achieve a certain performance threshold on the target task following a curriculum, where the curriculum is represented by the CMDP policy, which is being learned over time.

We compare curriculum policies learned for each agent to two static curricula. The first is the baseline *no curriculum* policy. In this case, on each episode, the agent learns tabula rasa directly on the target task. The flat line plotted represents the average time needed to directly learn the target task. Note that the line is flat because the curriculum is fixed and does not change over time. The second is a curriculum produced by following an existing curriculum algorithm ([7] for the gridworld, [10] for Ms. Pac-Man, to compare with past work). We also compare to a naive learning-based approach, which represents CMDP states using a list of all tasks learned by the learning agent. For example, the start state is the empty list. Upon learning a task M_1, the CMDP agent transitions to a new state $[M_1]$. In order to deal with the combinatorial explosion of the size of the state space, we limit the number of tasks that can be used as sources in the curriculum to a constant (between 1 and 3 in our experiments), and force the selection of the target task after.

3.1 Gridworld Experiments

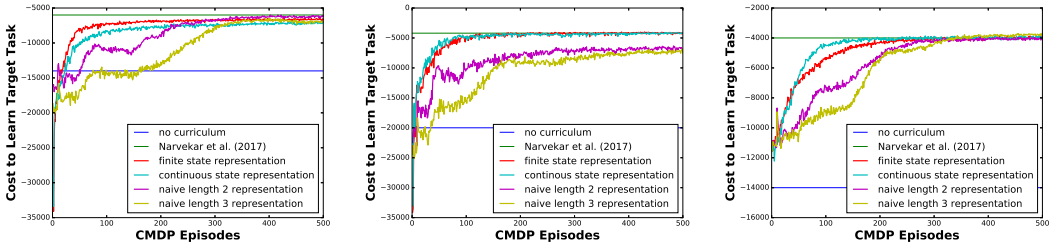
In this experiment, we examine learning curriculum policies for 3 learning agents that have different state and action spaces [7], but use the same transfer learning algorithm (value function transfer), in a simple grid world domain. In particular, we examine and compare two different types of representations for the CMDP state. The first CMDP representation is based on the finite state space representation discussed earlier. The learning agents use Sarsa(λ) with an *egocentric* feature space. Thus, the parameters $\boldsymbol{\theta}$ learned are not action-values for each state. However, since the underlying domain has a fixed number of states, we can move the learning agent to each of the states in the target task and compute action values for each grid cell. Let this new parameter of weights be $\boldsymbol{\theta'}$. We can now utilize the procedure described in Section 2 to create a CMDP feature space $\boldsymbol{\phi}^C(\boldsymbol{\theta'})$. The second CMDP representation was created directly from $\boldsymbol{\theta}$ without using an intermediary state-based action-value representation. We did this by creating a separate tile group directly for each θ_i.

A total of 9 different tasks were created to form the action space $\mathcal{A}^C$ of the CMDP agent. Each of the learning agents was trained until it could receive a return of 700 on the target task. The CMDP learning curves for each agent are shown in Figures 1(a) - 1(c). The results show that each agent successfully learned curriculum policies using both CMDP representations that were comparable in performance to the curricula generated by previous work [7].

3.2 Ms. Pac-Man Experiments

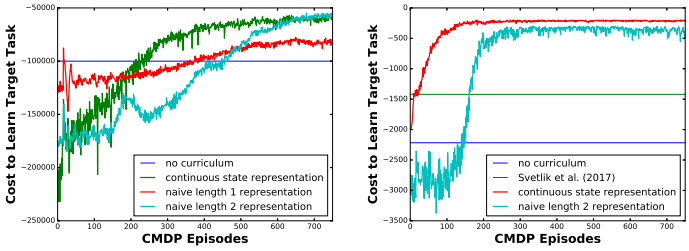
We also evaluated learning CMDP policies in the game of Ms. Pac-Man. In this experiment, we explore learning a curriculum policy for a Ms. Pac-Man agent, when the agent uses 2 different types of transfer learning methods: value function transfer and reward shaping.

In the value function case, the raw CMDP state variables s^C are the weights $\boldsymbol{\theta}$ of the Ms. Pac-Man agent's linear function approximator. To create the CMDP space $\boldsymbol{\phi}^C$, we normalize $\boldsymbol{\theta}$ and use tile



(a) (b) (c)

Figure 1: CMDP learning curves for the (a) basic agent, (b) action-dependent agent, and (c) rope agent using different curriculum design approaches and CMDP state space representations. All curves are averaged over 500 runs.



(a) (b)

Figure 2: CMDP learning curves on the Ms. Pac-Man target task, using (a) value function transfer and (b) transfer with reward shaping. All curves are averaged over 500 runs. Cost is measured in game steps for (a), and episodes for (b).

coding, creating a separate tiling over each θ_i. In the reward shaping setting, each source task in the curriculum is associated with a potential function (derived from the value function). As multiple tasks are learned, the potentials are added together, and used to create a shaping reward (as done in [10]). Thus, the raw CMDP state variables are the summed weights of the potential functions. As in the value function case, we use tile coding to create a separate tiling over each potential weight feature to create the CMDP basis space.

We used the same 15 tasks used in the code release of Svetlik et al. [10] to form the action space $\mathcal{A}^C$. The set of terminal states S_f^C were all states where the learning agent could achieve a return of at least 2000 on the target task. Figure 2(a) shows CMDP learning curves for Ms. Pac-Man using value function transfer and Figure 2(b) shows the curves using reward shaping. The results again clearly show that curriculum policies can be learned, and that such policies are more useful than training directly on the target task. In addition, we compared the reward shaping approach with that of Svetlik et al.[10], and found that a much better curriculum is possible in this more complex domain.[2]

4 Conclusion

In this paper, we showed that a more general representation of a curriculum than previous work, a *curriculum policy*, can be learned. The key challenge of learning a curriculum policy is creating a CMDP state representation that allows efficient learning. We extended the original curriculum MDP definition to handle multiple types of transfer learning algorithms, and described how to construct CMDP representations for both discrete and continuous domains to faciliate such learning. Finally, we demonstrated that curriculum policies can be learned on a gridworld and pacman domain. The results show that our approach is successful at creating curricula that can train agents to perform on a target task as fast or faster than existing methods. Furthermore, our approach is robust to multiple learning agent types, multiple transfer learning algorithms, and different CMDP representations.

[2]Our results are based on a reproduction of their experiments using their publicly released code. Interestingly, we get slightly better results for their method than they report in their paper.

References

[1] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 41–48. ACM, 2009.

[2] Felipe Leno Da Silva and Anna Helena Reali Costa. Object-oriented curriculum generation for reinforcement learning. In *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2018.

[3] Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1515–1528, Stockholmsmässan, Stockholm Sweden, July 2018.

[4] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. In *Proceedings of the 1st Annual Conference on Robot Learning*, 2017.

[5] A. Lazaric. Transfer in reinforcement learning: a framework and a survey. In M. Wiering and M. van Otterlo, editors, *Reinforcement Learning: State of the Art*. Springer, 2011.

[6] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)*, May 2016.

[7] Sanmit Narvekar, Jivko Sinapov, and Peter Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, August 2017.

[8] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing-solving sparse reward tasks from scratch. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[9] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[10] Maxwell Svetlik, Matteo Leonetti, Jivko Sinapov, Rishi Shah, Nick Walker, and Peter Stone. Automatic curriculum graph generation for reinforcement learning agents. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, February 2017.

[11] Matthew E. Taylor, Nicholas Carboni, Anestis Fachantidis, Ioannis Vlahavas, and Lisa Torrey. Reinforcement learning agents providing advice in complex video games. *Connection Science*, 26(1):45–63, 2014.

[12] Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

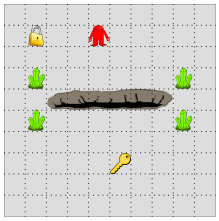


Figure 3: Grid world target task.

A Gridworld Domain Description

The world consists of a room with 4 different types of objects. *Keys* are objects the agent can pickup by executing a *pickup* action. These are used to unlock *locks*, which can depend on one or more keys, by executing an *unlock* action. *Pits* are obstacles that terminate the episode upon contact. Finally, *beacons* are landmarks placed on the 4 corners of a pit.

The goal of the agent is to traverse the world and unlock all the locks. To do so, at each step, the agent can move in one of the 4 cardinal directions, execute a pickup action, or an unlock action. Successfully picking up keys give a reward of +500, unlocking a lock rewards +1000, falling into a pit ends the episode with a reward of -200, and a constant step penalty of -10 is applied for all other actions.

A.1 Learning Agent Descriptions

We created 3 different learning agents that have varied sensing and action capabilities, based on the agents presented in [7]. Creating these specific agents allows us to show that our approach works regardless of the state and action representation used by the learning agents, and also allows us to compare with the results of [7]. We refer the reader there for full details of the agents, but recap the main elements and differences here for completeness.

The first agent, the *basic agent*, has 16 sensors, grouped into 4 on each side. The first sensor in each quadruple measures the Euclidean distance to the closest key from that side, the second the distance to the closest lock, the third the distance to the closest beacon, and the fourth detects whether there is a pit adjacent to the agent in that direction. An additional sensor indicates whether all keys in the room have been picked up, referred to as the *noKey* sensor. The agent used Sarsa(λ) with ϵ-greedy action selection, value function transfer for transfer learning, and CMAC tile coding with linear function approximation, where tile widths were set to 1.

For the basic agent, we created two tilings: one over the 13 percepts from the key, beacon, pit, and noKey sensors, and another over the 13 percepts from the lock, beacon, pit, and noKey sensors. These tilings formed $\phi(s)$ for the basic agent. The exploration rate ϵ was set to 0.1, eligibility trace parameter λ to 0.9, and learning rate α to 0.1.

The second, *action-dependent agent*, has the same sensors as the basic agent, but they are tiled differently, leading to a different $\phi(s)$: one tiling is over the lock, pit, and noKey features; a second is over the key, pit, and noKey features; and a third is over the beacon and pit features. In addition, unlike the basic agent, the state representation is action-dependent. That is, when considering the *move right* action, the agent's feature vector uses values only from the right side sensors. The weights in the tilings are shared, so that the same set of weights is used for the state in each of the directions.

Finally, the *rope agent* is like the basic agent, except that it has 4 additional actions, which are to use a rope in one of the four directions. Doing so opens a path across a pit if one is present, and incurs the step cost of -10.

B Ms. Pac-Man Domain Description

Our implementation was based on code released by [11] and augmented by [10]. The goal of the Ms. Pac-Man agent is to traverse a maze and accumulate points by eating edible objects such as food pellets, while avoiding ghosts. At each time step, Ms. Pac-Man can move along one of the 4 cardinal

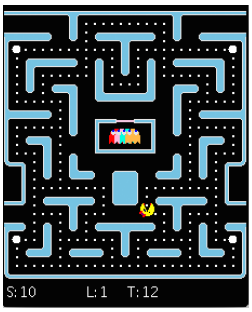


Figure 4: Ms. Pac-Man target task

directions (though not every action is available in every state). The agent receives a reward of 10 for eating a pill and 50 for a power pill. Eating a power pill temporarily makes the ghosts edible. Eating the first ghost gives a reward of 200; each subsequent ghost eaten multiplies this reward by a factor of 2. The game ends when all food pellets and power pills have been eaten, a ghost eats Ms. Pac-Man, or when a time limit of 2000 game steps have occurred.

While the domain is technically discrete, it has a combinatorially large state space. There are over a thousand positions in the target task maze, and the state consists of the locations of Ms. Pac-Man, each food pellet, power pill, each of the 4 ghosts, the last move of each ghost, and whether each ghost is edible. Thus, it is essential for the Ms. Pac-Man learning agent to use function approximation.

B.1 Learning Agent Description

We created a Ms. Pac-Man learning agent using the low-asymptote feature set described in [11, 10]. The state space of the agent is represented by a set of action-dependent egocentric features, that count the fraction of pills, power pills, ghosts, and edible ghosts there are in each direction up to different "depths." The depth is represented in terms of junctions, i.e. locations in the maze where there are more than 2 possible actions. For example, the ghost feature for depth 1 would return the fraction of ghosts there are along one direction until the first junction. In our experiments, we calculated features up to depth 6. The features were used to learn a simple linear value function approximator.

The agent was trained using Sarsa(λ), with $\epsilon = 0.05$, $\alpha = 0.001$, $\gamma = 0.999$, and $\lambda = 0.9$. Value function transfer was used for transfer learning. See the code by [10] for implementation details. Following the experimental setup of [10], a task is considered learned when at least 35% of the maximum reward possible for that task can be achieved.