The Dissertation Committee for Sanmit Santosh Narvekar
certifies that this is the approved version of the following dissertation:

Curriculum Learning in Reinforcement Learning

Committee:

Peter Stone, Supervisor

Scott Niekum

Raymond Mooney

Emma Brunskill

Curriculum Learning in Reinforcement Learning

by

Sanmit Santosh Narvekar

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2021

To my parents, Santosh and Smita, and my brother Neel,
for always believing in me

Acknowledgments

This thesis would not have been possible without the support of many people. First, I would like to thank my advisor Peter Stone for giving me the opportunity, support, and guidance to pursue research in this area. Despite having so many students and other commitments, Peter is always able to find time to meet and discuss ideas and potential next steps. He is supportive, encouraging, and the strongest advocate for all of his students, and I could not be more grateful to have him as my advisor. I also want to thank my other committee members Scott Niekum, Raymond Mooney, and Emma Brunskill for their valuable advice and feedback on this thesis.

My time at UT has led to a lot of personal and professional growth. This is in large part due to my labmates in the Learning Agents Research Group (LARG) who have always been there to bounce ideas, share a meal, or play soccer. I was also fortunate to be able to participate in the Standard Platform League competition as a part of UT Austin Villa, which was the source of many sleepless nights but also great joy when things finally worked on the robots. And of course, both of these were part of the broader Computer Science Department at UT, which fostered friendships across labs. Among the people I would like to thank are Michael Albert, Stefano Albrecht, Shani Alkoby, Samuel Barrett, Sid Desai, Ishan Durugkar, Katie Genter, Keya Ghonasgi, Harsh Goyal, Josiah Hanna, Justin Hart, Matthew Hausknecht, Eddy Hudson, Elad Liebman, Yuqian Jiang, Haresh Karnan, Josh Kelle, Piyush Khandelwal, Brad Knox, Matteo Leonetti, Bo Liu, Shih-Yun Lo, Patrick Macalpine, William Macke, Bharath Massetty, Jake Menashe, Reuth Mirsky, Aishwarya Padmakumar, Bei Peng, Ashay Rane, Guni Sharon, Jivko Sinapov, Matthew Taylor, Jesse Thomason, Faraz Torabi, Daniel Urieli, Garrett Warnell, Nick Wilson, Xuesu Xiao, Zifan Xu, Harel Yedidsion, Ruohan

Zhang, and Shiqi Zhang.

I want to give a special shout out to my "theoretical bros" Elad and Ishan. Your friendship over the years, those late nights and early mornings scrambling to put together the vision systems for Robocup, and our random conversations made grad school much more enjoyable. That coauthored publication did not end up coming together so far, but who knows, maybe this could be the year! Thank you – I couldn't imagine grad school without you guys.

When I first moved to Austin, I never thought it would feel like home the way California did. I would like to thank Hien Nguyen-Phuoc, Sherry Reynolds, Felipa Mendez, and Arianna Mahan for filling my time outside the lab with memories and friendships I never expected. Whatever the future holds, I will always remember your kindness, generosity, and passion. Meeting you all was truly a gift.

I would like to thank my undergraduate advisor Valentino Crespi at Cal State LA for strongly influencing my decision to pursue graduate school in computer science, and for his time and mentorship in cultivating my interest in machine learning. I would also like to thank my friends from my undergraduate years – Michael Levitin, Charissa Kim, and Catrina Chitjian – who I've been fortunate to keep in sporadic contact with as we navigated life from middle school to full grown adults.

And finally, I would like to thank my parents Santosh and Smita, and my brother Neel, for believing in me – even more than I do in myself. You all have been part of this rollercoaster of a journey from the beginning, and I would not be who I am today without your love and support.

Sanmit Santosh Narvekar

The University of Texas at Austin

May 2021

Curriculum Learning in Reinforcement Learning

Sanmit Santosh Narvekar, Ph.D.
The University of Texas at Austin, 2021

Supervisor: Peter Stone

In recent years, reinforcement learning (RL) has been increasingly successful at solving complex tasks. Despite these successes, one of the fundamental challenges is that many RL methods require large amounts of experience, and thus can be slow to train in practice. Transfer learning is a recent area of research that has been shown to speed up learning on a complex task by transferring knowledge from one or more easier source tasks. Most existing transfer learning methods treat this transfer of knowledge as a one-step process, where knowledge from all the sources are directly transferred to the target. However, for complex tasks, it may be more beneficial (and even necessary) to gradually acquire skills over multiple tasks *in sequence*, where each subsequent task requires and builds upon knowledge gained in a previous task. This idea is pervasive throughout human learning, where people learn complex skills gradually by training via a *curriculum*.

The goal of this thesis is to explore whether autonomous reinforcement learning agents can also benefit by training via a curriculum, and whether such curricula can be designed fully autonomously. In order to answer these questions, this thesis first formalizes the concept of a curriculum, and the methodology of *curriculum learning* in reinforcement learning.

Curriculum learning consists of 3 main elements: 1) task generation, which creates a suitable set of source tasks; 2) sequencing, which focuses on how to order these tasks into a curriculum; and 3) transfer learning, which considers how to transfer knowledge between tasks in the curriculum. This thesis introduces several methods to both create suitable source tasks and automatically sequence them into a curriculum. We show that these methods produce curricula that are tailored to the individual sensing and action capabilities of different agents, and show how the curricula learned can be adapted for new, but related target tasks. Together, these methods form the components of an autonomous *curriculum design agent*, that can suggest a training curriculum customized to both the unique abilities of each agent and the task in question. We expect this research on the curriculum learning approach will increase the applicability and scalability of RL methods by providing a faster way of training reinforcement learning agents, compared to learning tabula rasa.

Contents

List of Tables

List of Figures

1. Introduction

In recent years, autonomous reinforcement learning (RL) agents have successfully solved increasingly complex problems. For example, they have been used to play Atari games with human-level performance [77], have bested the world's top professional Go player [110], and have also "solved" variants of poker [14]. However, training agents and systems like these typically require collecting massive amounts of training experience, which may not always be possible, especially in time-limited scenarios. Transfer learning [66, 127] is one recent area of research that seeks to speed up learning on a target problem by *transferring* knowledge from one or more related source problems. Most existing transfer learning techniques have treated this procedure as a one-shot process, where an agent trains on one or more source problems, and directly transfers the knowledge gained to the target problem. However, they have stopped short of asking whether those source problems themselves could benefit from training on even simpler subproblems. This breakdown could be necessary when solving a source problem depends on having learned other behaviors as prerequisites.

One source of inspiration for this idea can be found in human learning. Humans learn to solve complex problems by incrementally acquiring and building the necessary skills via a *curriculum*. Such curricula are present throughout early human development, formal education, and life-long learning all the way to adulthood. Whether learning to play a sport, or learning to become an expert in mathematics, the training process is organized and structured so as to present new concepts and tasks in a sequence that leverages what has previously been learned. In a variety of human learning domains, the quality of the curricula has been shown to be crucial in achieving success. Curricula are also present in animal training, where it is commonly referred to as shaping [89, 113].

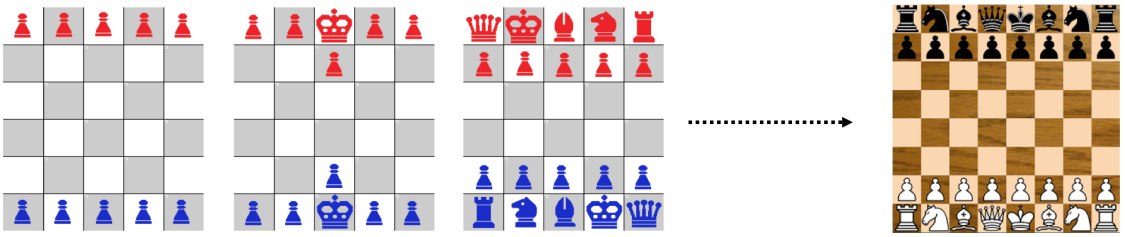


Figure 1.1: Different subgames in Quick Chess

As a motivating example, consider the game of Quick Chess[1] (Figure 1.1). Quick Chess is a game designed to introduce human players to the full game of chess, by using a sequence of progressively more difficult "subgames." For example, the first subgame is a 5x5 board with only pawns, where the player learns how pawns move and about promotions. The second subgame is a small board with pawns and a king, which introduces a new objective: keeping the king alive. In each successive subgame, new elements are introduced (such as new pieces, a larger board, or different configurations) that require learning new skills and building upon knowledge learned in previous games. The final game is the full game of chess.

This thesis explores the extent to which similar ideas can be used to improve the learning ability of autonomous reinforcement learning agents. Specifically, the goal of this thesis research is to answer the following question:

Can reinforcement learning agents benefit from learning via a curriculum, and how can an autonomous curriculum design agent automatically create a curriculum tailored to both the abilities of individual learning agents and the task in question?

[1] http://www.intplay.com/uploadedFiles/Game_Rules/P20051-QuickChess-Rules.pdf

1.1 Contributions

This thesis answers this question by making the following contributions to the field of curriculum learning in reinforcement learning:

1. **Problem Definition**

   This thesis formalizes the concept of a curriculum, and the method of curriculum learning in the context of reinforcement learning. It describes curriculum learning and each of its components, and also defines metrics to quantify the utility of a curriculum. These ideas are discussed in Chapter 3.

2. **Methods for Creating Source Tasks**

   In order to create a curriculum, a curriculum designer must first be able to create a space of source tasks that would form components of a curriculum. This thesis presents a series of methods to create source tasks relevant for an agent by using knowledge of the domain, and also by observing the agent's learning progress on the target task. These methods are discussed in Chapter 4.

3. **Method to Evaluate Task Transferability**

   In order to determine how to sequence tasks into a curriculum, we need to know how useful learning one task will be to bootstrap learning of another. This thesis presents an approach that uses parameterized representations of tasks to learn a task-transferability model, which can be used to predict the utility of a source task for a target task, in Chapter 5.

4. **Methods for Sequencing Tasks into a Curriculum**

   The core question in curriculum learning is how to best sequence a set of potential source tasks into a curriculum. This thesis proposes algorithms to automatically create a curriculum in Chapters 6 and 7.

In real world domains, different robots/agents may be called upon to perform a task (for example, due to availability), where each has its own sensing and acting capabilities. These differences suggest each agent would benefit from an individualized curriculum. Therefore, the sequencing methods this thesis presents will also produce curricula tailored to the individual abilities of each agent.

5. **Methods for Adapting a Curriculum Created for one Task to a Different Task**

   Many existing curriculum learning methods for RL agents generate curricula independently for each agent and target task. This process can be very expensive, especially if there are multiple target tasks or agents that require curricula. Therefore, this thesis proposes algorithms to reuse this knowledge, so that a curriculum generated for one task can be adapted for another. These ideas are presented in Chapter 8.

6. **A Taxonomy of Curriculum Learning Approaches for RL**

   Over the past few years, several different methods for curriculum learning in reinforcement learning have been devised. Each of these methods makes different assumptions about the way tasks in the curriculum are generated, sequenced, and evaluated. These design choices affect the types of settings each method can be applied in. This thesis presents a taxonomy of these methods in Chapter 9, highlighting common themes of approaches thus far, which is designed to inform future research in this area.

7. **Empirical Validation**

   This thesis evaluates the above contributions in both a simple domain that allows prototyping and ablation analysis of the different elements involved in generating a curriculum, as well as a more complex domain that will test the ability of the methods to scale up. These experiments appear in Chapters 4 to 8, for Contributions 2 to 5 as described above.

Curriculum learning is a method to improve the efficiency of RL agents. Taken together, the contributions of this thesis make advances in each of the core components of curriculum learning, providing methods to both generate and sequence tasks into a curriculum. This thesis also formalizes the curriculum learning method and situates it with respect to related reinforcement learning work, providing a common basis to discuss and advance ideas in this area.

1.2 Thesis Overview

The rest of this thesis is organized as follows. Although the chapters of this thesis are written to be read in order, doing so is not required. Figure 1.2 specifies the dependencies between chapters.

- **Chapter 2 - Background.** This chapter provides the necessary background to understand the rest of this thesis. Here, I review the basics of reinforcement learning in Markov Decision Processes with function approximation. I also review transfer learning methods, and how they are evaluated.

- **Chapter 3 - The Curriculum Learning Method.** In this chapter, I formalize the concept of a curriculum and the methodology of curriculum learning. I also discuss how curricula can be evaluated by extending the metrics devised for single stage transfer learning. This chapter addresses Contribution 1 of this thesis.

- **Chapter 4 - Task Generation.** In this chapter, I introduce a set of methods to semi-automatically create relevant source tasks for a given target task. These methods use a parameterized model of the domain, and observations of the agent interacting in the target task to create useful source tasks. This chapter addresses Contribution 2 of this thesis.

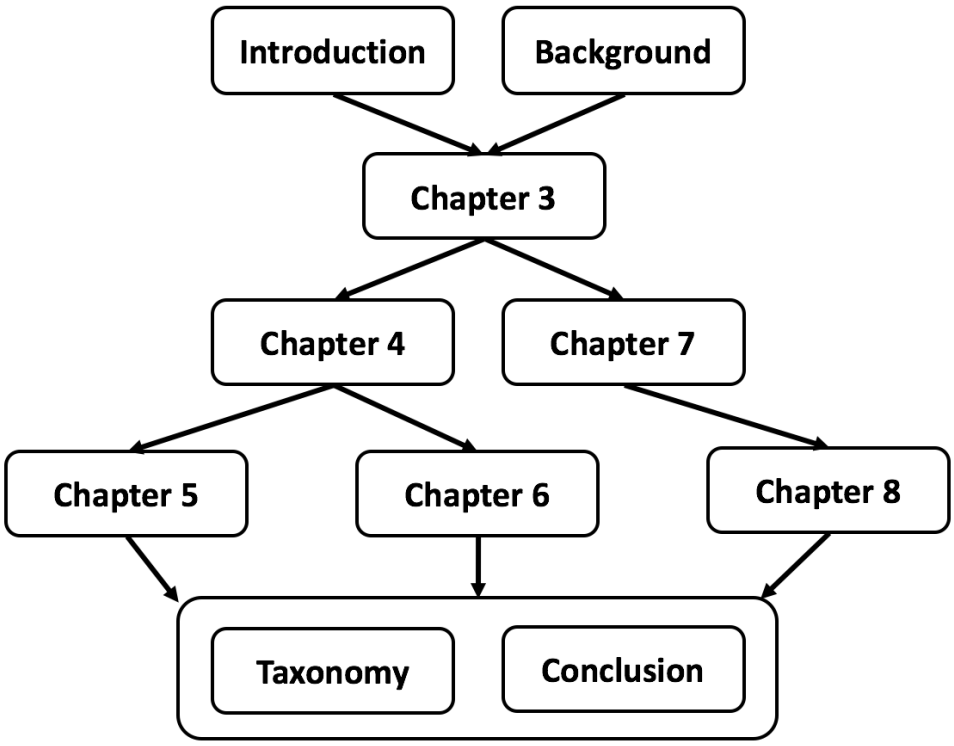


Figure 1.2: A visual illustration of how the chapters in this thesis depend on each other. Arrows denote that one chapter should be read before another.

- **Chapter 5 - Measuring Inter-task Transferability.** In this chapter, I present a method to model the inter-task transferability of a source task for a target task; i.e., how much benefit (evaluated by the jumpstart metric) one can expect when transferring from a specified source task to a specified target task. This chapter addresses Contribution 3 of this thesis.

- **Chapter 6 - Heuristic-based Approaches for Sequencing.** In this chapter, I present a heuristic method to automatically sequence tasks into a curriculum. The method uses samples of the agent's behavior on the target task to select the next source task in the curriculum. It relies on the task generation methods from Chapter 4, and produces individualized curricula for different agents. This chapter addresses Contribution 4 of this thesis.

- **Chapter 7 - Learning-based Approaches for Sequencing.** In this chapter, I present a learning-based method to automatically sequence tasks into a curriculum. It formulates curriculum generation as an interaction between a student and teacher MDP, and learns a curriculum policy, which is a mapping from the student's knowledge to what task it should learn next. Like in the method from Chapter 6, the resulting curriculum is tailored to each agent. This chapter addresses Contribution 4 of this thesis.

- **Chapter 8 - Generalizing Curricula.** In this chapter, I discuss an approach to generalize the model described in Chapter 7 over different target tasks. This process allows the method to produce curricula for novel, unseen target tasks. This chapter addresses Contribution 5 of this thesis.

- **Chapter 9 - Taxonomy of CL Methods and Related Work.** In this chapter, I present a taxonomy for curriculum learning methods, and provide a detailed survey addressing each of the elements of curriculum learning. I also describe how curriculum

learning relates to other ideas in reinforcement learning, and how curriculum learning has been used in supervised learning and human education. This chapter addresses Contribution 6 of this thesis.

- **Chapter 10 - Conclusion and Future Work.** In this chapter, I conclude by giving a recap of the work presented in previous chapters. I also describe some ongoing work connecting the methods devised in this thesis back to human learning, and present ideas for future work.

2. Background

Curriculum learning in reinforcement learning builds upon two key fields of study. In this chapter, I provide the necessary background information on these fields and introduce the notation that will be used throughout this thesis. The first field is reinforcement learning, which frames decision making problems as a Markov Decision Process (MDP) and is concerned with how to act to maximize an environmental reward signal. The second is transfer learning, which studies how an agent can reuse knowledge acquired in one task to improve performance on another task, and forms the basis for curriculum learning.

Throughout this thesis, I will use the following conventions for notation (adapted from Sutton and Barto [121]). Any exceptions to these conventions will be noted by footnotes in the thesis:

Sets will be denoted using capital caligraphic letters (e.g., $\mathcal{S}$).

Random variables will be denoted with capital letters (e.g., R_t is the reward observed at time t)

Elements of sets, instantiations of random variables, functions, and constants will be denoted by lowercase letters (e.g., $s \in \mathcal{S}$ for an element of the state space $\mathcal{S}$).

Vectors will be indicated in bold (e.g., $\boldsymbol{\theta}$)

2.1 Reinforcement Learning

Reinforcement learning is a branch of machine learning that considers how an agent should act in an environment. Unlike other branches of machine learning such as supervised learning,

the agent is given a delayed (and possibly sparse) numeric reward signal, rather than explicit labels on each step indicating whether the action taken was correct or not. The goal of the agent is to learn through interaction which actions to take in order to maximize the cumulative rewards from the signal over time.

2.1.1 Markov Decision Processes

We can formalize the interaction of an agent with its environment (i.e. a *task*) as a Markov Decision Process (MDP). In this work, we restrict our attention to *episodic* MDPs:

Definition 2.1. An episodic MDP M is a 6-tuple $(\mathcal{S}, \mathcal{A}, p, r, \Delta s_0, \mathcal{S}_f)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $p(s'|s,a)$ is a transition function that gives the probability of transitioning to state s' after taking action a in state s, and $r(s,a,s')$ is a reward function that gives a scalar reward for taking action a in state s and transitioning to state s'. In addition, we shall use Δs_0 to denote the initial state distribution, and $\mathcal{S}_f$ to denote the set of terminal states. In some situations, instead of referring to an initial state distribution, we will instead refer to a set of initial states $\mathcal{S}_0$. In this case, it is implied that the initial state distribution is uniform over the set of initial states.

We consider time in discrete time steps. At each time step t, the agent observes its state and chooses an action according to its *policy* $\pi(a|s)$. The goal of the agent is to learn an *optimal policy* π^*, which maximizes the expected *return* G_t (the cumulative sum of rewards R) until the episode ends at a terminal state on timestep T:[2]

$$G_t = \sum_{i=1}^{T-t} R_{t+i}$$

There are two main classes of methods to learn π^*: value function approaches and policy search approaches. In *value function approaches*, a value $v_\pi(s)$ is first learned for each

[2]We use a capital T here to match conventional notation even though it is a constant

state s, representing the expected return achievable from s by following policy π. Through policy evaluation and policy improvement, this value function is used to derive a policy better than π, until convergence towards an optimal policy. Using a value function in this process requires a model of the reward and transition functions of the environment. If the model is not known, one option is to learn an action-value function instead, $q_\pi(s, a)$, which gives the expected return for taking action a in state s and following π after:

$$q_\pi(s, a) = \sum_{s'} p(s'|s, a)[r(s, a, s') + q_\pi(s', a')] \text{ , where } a' \sim \pi(\cdot|s')$$

The action-value function can be iteratively improved towards the optimal action-value function q_* with on-policy methods such as SARSA [121]. The optimal action-value function can also be learned directly with off-policy methods such as Q-learning [140]. An optimal policy can then be obtained by choosing action $\text{argmax}_a q_*(s, a)$ in each state. If the state space is large or continuous, the (action-)value function can be approximated as a function of state *features* $\boldsymbol{\phi}(s)$ and a weight vector $\boldsymbol{\theta}$. We discuss several common representations for function approximation using such vectors in the next section.

In contrast, *policy search* methods directly search for or learn a parameterized policy $\pi(a|s, \boldsymbol{\theta})$, without using an intermediary value function. Typically, the parameter $\boldsymbol{\theta}$ is modified using search or optimization techniques to maximize some performance measure $J(\boldsymbol{\theta})$[3]. For example, in the episodic case $J(\boldsymbol{\theta})$ could correspond to the value of the policy parameterized by $\boldsymbol{\theta}$ from the starting state s_0: $v_{\pi_\theta}(s_0)$. One example of a policy search method is Proximal Policy Optimization (PPO) [108].

2.1.2 Function Approximation

There are many different types of functions we can use to represent a value function or policy. I will ground the discussion below by considering function approximation for estimating the

[3]We use capital J to match convention even though it is a function

value function. However, the function can also be used to approximate an action-value function or policy as well.

In value function learning without function approximation, a separate entry in a lookup table is maintained for each state s, that represents the current estimate of the value of that state $\hat{v}(s)$. With function approximation, we instead *approximate* the value of state s using an approximate value function $\hat{v}(s, \boldsymbol{\theta})$ that is parameterized by a weight vector $\boldsymbol{\theta}$. There are many types of functions we can use to approximate $\hat{v}$. For example, one common approach is a linear function approximator [121]. In this case, the value is represented as the inner product of the weight vector $\boldsymbol{\theta}$ and a feature vector $\boldsymbol{\phi}(s)$ derived from the state s:

$$\hat{v}(s, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s) = \sum_i \theta_i \phi_i(s) \tag{2.1}$$

where $\phi_i(s)$ is the number corresponding to feature i in state s, and θ_i is the corresponding weight for that feature. There are many options for extracting and representing features $\boldsymbol{\phi}(s)$ for a state: common choices include radial basis functions, polynomial basis functions, and tile coding [121].

As we use tile coding for encoding features in some experiments in later chapters, I briefly describe it here. Tile coding is a way of representing a continuous feature space by overlaying several overlapping tilings over subsets of state variables (see Figure 2.1). The value of the state variable is used to determine which *tile* is activated in each tiling, and each activated tile contributes a weighted value to the output for a given state. Increasing the number of tilings or their size allows the encoding to generalize better, while decreasing it allows better representation of finer details.

Another option for function approximation is to use a nonlinear function approximator, such as a neural network. Recently, deep neural networks have become a popular choice, as they are able to learn complex nonlinear policies and can be trained end-to-end from raw state input. Reinforcement learning algorithms using deep nets for function approximation

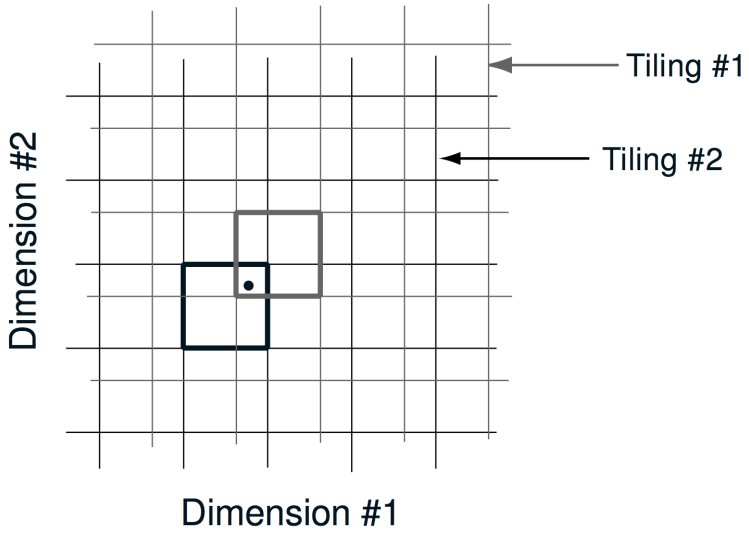


Figure 2.1: Tile coding over 2 dimensions of state variables. Image from Taylor and Stone [126].

have been successful at achieving human or better than human level performance in Atari games [77] and the ancient Chinese game of Go [110].

The right function approximator to use very much depends on the domain. The methodology we develop throughout this thesis is designed to be independent of the type of function approximator used; we indicate in the experimental section of each chapter which type was used.

2.2 Transfer Learning

Let the task the agent must learn be denoted as the *target task*. In the standard reinforcement learning setting, an agent usually starts with a random policy, and directly attempts to learn an optimal policy for the target task. When the target task is difficult, for example due to adversarial agents, poor state representation, or sparse reward signals, learning can be very slow.

Transfer learning is one class of methods and area of research that seeks to speed up

training of RL agents. The idea behind transfer learning is that instead of learning on the *target task* tabula rasa, the agent can first train on one or more *source task* MDPs, and *transfer* the knowledge acquired to aid in solving the target. This knowledge can take the form of samples [67, 68], options [114], policies [29], models [25], or value functions [126].

Some of these methods assume that the source and target MDPs either share state and action spaces, or that a *task mapping* [128] (see Figure 2.2) is available to map states and actions in the target task to known states and actions in the source. Such mappings can be specified by hand, or learned automatically [4, 129]. Other methods assume the transition or reward functions do not change between tasks. The best method to use varies by domain, and depends on the relationship between source and target tasks. While most methods assume that knowledge is transferred from one source task to one target task, some methods have also been proposed to transfer knowledge from several source tasks directly to a single target [123]. See Taylor and Stone [127] or Lazaric [66] for a survey of transfer learning techniques.

Curriculum learning utilizes transfer learning as a component to transfer information acquired in tasks of a curriculum, as we discuss in Chapter 3. The methods we propose are designed to be general enough to work with any type of transfer learning method used. However, in the next section I describe two specific transfer learning methods that we used as part of the methods devised in this thesis.

2.2.1 Methods

In this section, I provide background on two types of transfer learning methods used in the methods of this thesis: value/policy transfer and transferring a shaping reward.

Value Function and Policy Transfer

The first type of transfer learning method we use in this thesis is value function transfer [128]. In value function transfer, the parameters of an action-value function $q_{source}(s, a)$ learned in a source task are used to initialize the action-value function in the target task $q_{target}(s, a)$. This process biases exploration and action selection in the target task based on experience acquired in the source task. Closely related to this idea is policy transfer, where instead of transferring the weights of the action-value function, we instead transfer the weights of the parameterized policy. In the context of deep learning and neural networks, this idea is commonly referred to as finetuning.

These types of transfer typically assume that the source and target MDPs either share state and action spaces (such that a policy derived from the source task can be directly applied to the target task), or that a task mapping [128] is available to transform states and actions in the target task to states and actions in the source. In this work, the representation either does not change between tasks, or we use egocentric feature spaces that scale in size and pad the extra dimensions. For example, a more difficult task may have additional objects or opponents; weights for features for these extra objects are initialized to 0 (this is commonly referred to as padding). Thus the mapping is available (as part of the transfer learning method, which we treat as a black box) if necessary to facilitate transfer.

Reward Shaping

Reward shaping is a method where the reward function in the target task MDP is augmented by adding an additional shaping reward f, that is derived from the source tasks. When the target task is difficult due to sparse rewards, adding a shaping reward can provide more dense feedback, and allow an RL algorithm to learn faster. The new reward function in the

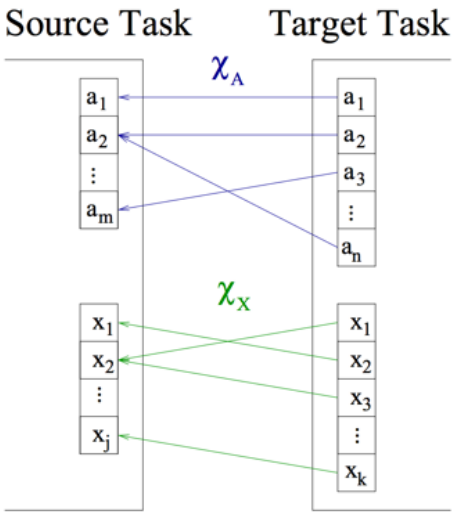


Figure 2.2: An inter-task mapping from states and actions in the target task to states and actions in a source. Image from Taylor and Stone [127].

target task MDP thus becomes:

$$r'(s, a, s') = r(s, a, s') + f(s, a, s') \tag{2.2}$$

In order to prevent this process from changing the optimal policy, we use potential-based advice [144], which restricts the form of f to be a difference of potential functions:

$$f(s, a, s') = \Phi(s', \pi(s')) - \Phi(s, a) \tag{2.3}$$

where Φ is a potential function. Choosing shaping rewards of this form is both necessary and sufficient to guarantee that adding f to the reward does not change the optimal policy [85]. In order to use shaping rewards for transfer, we follow the work of Svetlik et al. [123], where the value function learned in a source task is used as the potential function: $\Phi(s, a) = q_{source}(s, a)$. When multiple source tasks are present, as will be the case in curriculum learning, the potential function is composed as the sum of value functions from

the set of sources $\mathcal{D}$:

$$\Phi(s, a) = \sum_{i \in \mathcal{D}} q_i(s, a) \tag{2.4}$$

2.2.2 Evaluation Metrics

There are several metrics to quantify the benefit of transferring from a source task to a target task [127]. Typically, they compare the learning trajectory on the target task for an agent after transfer, with an agent that learns directly on the target task from scratch (see Figure 2.3a). One metric is *time to threshold*, which computes how much faster an agent can learn a policy that achieves expected return $G_0 \geq \delta$ on the target task if it transfers knowledge, as opposed to learning the target from scratch, where δ is some desired performance threshold. Time can be measured in terms of CPU time, wall clock time, episodes, or number of actions taken. Another metric is *asymptotic performance*, which compares the final performance after convergence in the target task of learners when using transfer versus no transfer. The *jumpstart* metric instead measures the difference between the initial performance after transfer and the initial performance without transfer. Finally, the *total reward* ratio compares the total reward accumulated by the agent during training up to a fixed stopping point, using transfer versus not using transfer. For more details on these metrics, see Taylor and Stone [127].

An important evaluation question is whether to include time spent *learning in source tasks* into the cost of using transfer. The transfer curve in Figure 2.3a shows performance on the target task, and starts at time 0, even though time has already been spent learning one or more source tasks. Thus, it does not reflect time spent training in source tasks before transferring to the target task. This situation is known in transfer learning as the *weak transfer* setting, where time spent training in source tasks is treated as a sunk cost. On the other hand, in the *strong transfer* setting, the learning curves must account for time spent in all source tasks. One way to account for this cost is to offset the curves to reflect time

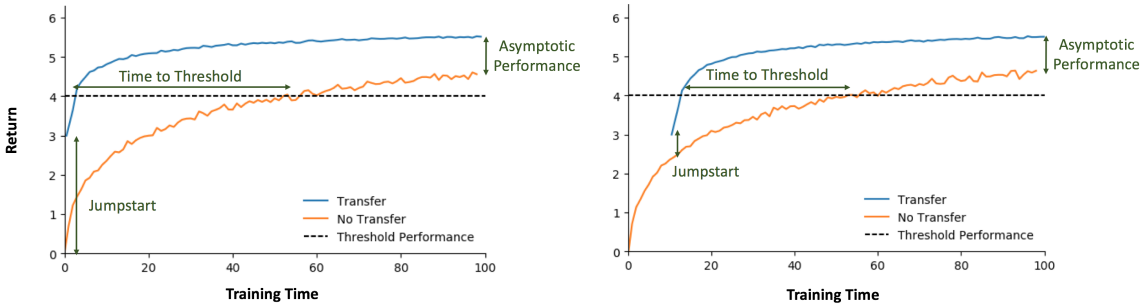


Figure 2.3: Performance metrics for transfer learning using (a) weak transfer and (b) strong transfer with offset curves.

spent in source tasks, as shown in Figure 2.3b. Another option is to freeze the policy while learning on source tasks, and plot that policy's performance on the target task.

2.3 Summary

In this chapter, I introduced the notation that will be used throughout this thesis, as well as provided background on reinforcement learning and transfer learning. These two ideas form the basis for this thesis, as it considers how to use curriculum learning in reinforcement learning domains, leveraging existing ideas from transfer learning. In Chapter 3, I will specifically formalize how these ideas fit into the curriculum learning framework. Together, these two chapters will form the foundation for the rest of the ideas explored in this thesis.

3. The Curriculum Learning Method

The idea of using a curriculum to train artificial agents can be traced back at least as far as Elman [24] in 1993. Over the years, curricula have been used to train agents on complex reinforcement learning tasks in areas such as robotics [6, 71] and games [147]. These curricula were often manually defined using expert knowledge of the problem domain.

Very recently, several research groups have begun examining how such curricula can be designed automatically. The recent spark of interest has been a result of two phenomena: 1) the problems we hope to solve have become increasingly complex and 2) prerequisite fields for curriculum learning, such as transfer learning, have sufficiently matured. However, what exactly constitutes a curriculum and what precisely qualifies an approach as being an example of curriculum learning is not clearly and consistently defined in the literature.

A curriculum has been represented in many ways: for example, the most common way is as an ordering of tasks. At a more fundamental level, a curriculum can also be represented as an ordering of individual experience samples. In addition, a curriculum does not necessarily have to be a simple linear sequence. One task can build upon knowledge gained from multiple source tasks, just as courses in human education can build off of multiple prerequisites.

In this chapter, I formalize the concept of a *curriculum*, with a definition that is broad enough to encompass many of the ideas and methods present in the literature. I also formalize the methodology of *curriculum learning*, which focuses on how to create a curriculum, and describe how to evaluate the costs and benefits of training reinforcement

This chapter is based on work that was published in the Journal of Machine Learning Research [84]. It was done in collaboration with Bei Peng, Jivko Sinapov, Matteo Leonetti, Matthew E. Taylor, and Peter Stone. My collaborators assisted in surveying some of the papers and in writing the article.

learning agents using this methodology. This chapter addresses Contribution 1 from Chapter 1 of this thesis.

3.1 Curricula

A curriculum is a general concept that encompasses both schedules for organizing past experiences, and schedules for acquiring experience by training on tasks. As such, we first propose a fully general definition of curriculum, and then follow it with refinements that apply to special cases common in the literature.

We assume a *task* is modeled as a Markov Decision Process, and define a curriculum as follows:

Definition 3.1 (Curriculum)**.** Let $\mathcal{T}$ be a set of tasks, where $M_i = (\mathcal{S}_i, \mathcal{A}_i, p_i, r_i, \Delta s_{i_0}, \mathcal{S}_{i_f})$ is a task in $\mathcal{T}$. Let $\mathcal{D}^{\mathcal{T}}$ be the set of all possible transition samples from tasks in $\mathcal{T}$: $\mathcal{D}^{\mathcal{T}} = \{(s, a, r, s') \mid \exists\, M_i \in \mathcal{T} \text{ s.t. } s \in \mathcal{S}_i, a \in \mathcal{A}_i, s' \sim p_i(\cdot|s, a), r \leftarrow r_i(s, a, s')\}$. A *curriculum* $C = (\mathcal{V}, \mathcal{E}, g, \mathcal{T})$ is a directed acyclic graph, where $\mathcal{V}$ is the set of vertices, $\mathcal{E} \subseteq \{(x, y) \mid (x, y) \in \mathcal{V} \times \mathcal{V} \wedge x \neq y\}$ is the set of directed edges, and $g : \mathcal{V} \to \mathcal{P}(\mathcal{D}^{\mathcal{T}})$ is a function that associates vertices to subsets of samples in $\mathcal{D}^{\mathcal{T}}$, where $\mathcal{P}(\mathcal{D}^{\mathcal{T}})$ is the power set of $\mathcal{D}^{\mathcal{T}}$. A directed edge $\langle v_j, v_k \rangle$ in C indicates that samples associated with $v_j \in \mathcal{V}$ should be trained on before samples associated with $v_k \in \mathcal{V}$. All paths terminate on a single sink node $v_t \in \mathcal{V}$.[4]

A curriculum can be created online, where edges are added dynamically based on the learning progress of the agent on the samples at a given vertex. It can also be designed completely offline, where the graph is generated before training, and edges are selected based on properties of the samples associated with different vertices.

Creating a curriculum graph at the sample level can be computationally difficult for large tasks, or large sets of tasks. Therefore, in practice, a simplified representation for a

[4]In theory, a curriculum could have multiple sink nodes corresponding to different target tasks. For the purpose of exposition, I assume a separate curriculum is created and used for each task.

curriculum is often used. There are 3 common dimensions along which this simplification can happen. The first is the single-task curriculum, where all samples used in the curriculum come from a single task:

Definition 3.2 (Single-task Curriculum)**.** A *single-task curriculum* is a curriculum C where the cardinality of the set of tasks considered for extracting samples $|\mathcal{T}| = 1$, and consists of only the target task M_t.

A single-task curriculum essentially considers how best to organize and train on experience acquired from a single task. This type of curriculum is common in experience replay methods [106].

A second common simplification is to learn a curriculum at the task level, where each vertex in the graph is associated with samples from a single task. At the task level, a curriculum can be defined as a directed acyclic graph of *intermediate* tasks:

Definition 3.3 (Task-level Curriculum)**.** For each task $M_i \in \mathcal{T}$, let $\mathcal{D}_i^{\mathcal{T}}$ be the set of all samples associated with task M_i: $\mathcal{D}_i^{\mathcal{T}} = \{(s, a, r, s') \mid s \in \mathcal{S}_i, a \in \mathcal{A}_i, s' \sim p_i(\cdot|s, a), r \leftarrow r_i(s, a, s')\}$. A *task-level curriculum* is a curriculum $C = (\mathcal{V}, \mathcal{E}, g, \mathcal{T})$ where each vertex is associated with samples from a single task in $\mathcal{T}$. Thus, the mapping function g is defined as $g : \mathcal{V} \to \{\mathcal{D}_i^{\mathcal{T}} \mid M_i \in \mathcal{T}\}$.

In reinforcement learning, the entire set of samples from a task (or multiple tasks) is usually not available ahead of time. Instead, the samples experienced in a task depend on the agent's behavior policy, which can be influenced by previous tasks learned. Therefore, while generating a task-level curriculum, the main challenge is how to order tasks such that the behavior policy learned is useful for acquiring good samples in future tasks. In other words, selecting and training on a task M induces a mapping function g, and determines the set of samples $\mathcal{D}_i^{\mathcal{T}}$ that will be available at the next vertex based on the agent's behavior policy as a result of learning M. The same task is allowed to appear at more than one vertex,

similar to how in Definition 3.1 the same set of samples can be associated with more than one vertex. Therefore, tasks can be revisited when the agent's behavior policy has changed. Several works have considered learning task-level curricula over a graph of tasks [71, 123]. An example can be seen in Figure 3.1b.

Finally, another simplification of the curriculum is the linear *sequence*. This is the simplest and most common structure for a curriculum in existing work:

Definition 3.4 (Sequence Curriculum)**.** A *sequence curriculum* is a curriculum C where the indegree and outdegree of each vertex v in the graph C is at most 1, and there is exactly one source node and one sink node.

These simplifications can be combined to simplify a curriculum along multiple dimensions. For example, the sequence simplification and task-level simplification can be combined to produce a task-level sequence curriculum. This type of curriculum is the one we primarily consider in this thesis, and can be represented as an ordered list of tasks $[M_1, M_2, ...M_n]$. An example can be seen in Figure 3.1a [83].

A final important question when designing curricula is determining the stopping criteria: that is, how to decide *when* to stop training on samples or tasks associated with a vertex, and move on to the next vertex. In practice, typically training is stopped when performance on the task or set of samples has converged. Training to convergence is not always necessary, so another option is to train on each vertex for a fixed number of episodes or epochs. Since more than one vertex can be associated with the same samples/tasks, this experience can be revisited later on in the curriculum.

3.2 Curriculum Learning

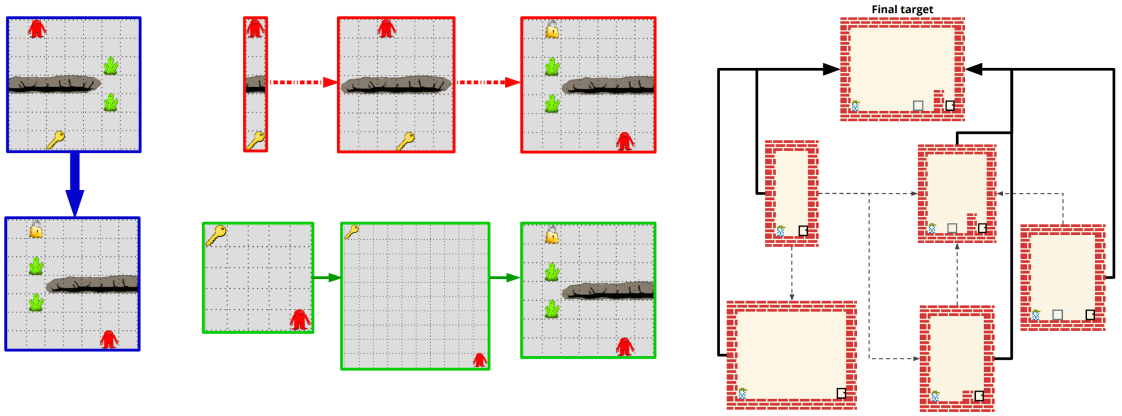
Curriculum learning is a methodology to *optimize* the order in which experience is accumulated by the agent, so as to increase performance or training speed on a set of final tasks.

Through generalization, knowledge acquired quickly in simple tasks can be leveraged to reduce the exploration of more complex tasks. In the most general case, where the agent can acquire experience from multiple intermediate tasks that differ from the final MDP, there are 3 key elements to this method:

- **Task Generation.** The quality of a curriculum is dependent on the quality of tasks available to choose from. Task generation is the process of creating a good set of intermediate tasks from which to obtain experience samples. In a task-level curriculum, these tasks form the nodes of the curriculum graph. This set of intermediate tasks may either be pre-specified, or dynamically generated during the curriculum construction by observing the agent. This thesis will present methods to generate tasks both offline before curriculum construction, as well as methods that produce tasks based on a dynamic analysis of the agent's progress on a task (Chapter 4).

- **Sequencing.** Sequencing examines how to create a partial ordering over the set of experience samples $\mathcal{D}$: that is, how to generate the edges of the curriculum graph. Most existing work has used manually defined curricula, where a human selects the ordering of samples or tasks. However, recently automated methods for curriculum sequencing have begun to be explored. This thesis will present several methods for sequencing, each of which make different assumptions about the tasks and transfer methodology used. These ideas will form the core of this thesis (Chapters 5 to 8).

- **Transfer Learning.** When creating a curriculum using multiple tasks, the intermediate tasks may differ in state/action space, reward function, or transition function from the final task. Therefore, transfer learning is needed to extract and pass on reusable knowledge acquired in one task to the next. Typically, work in transfer learning has examined how to transfer knowledge from one or more source tasks directly to the target task. Curriculum learning extends the transfer learning scenario to consider



(a) (b)

Figure 3.1: Examples of structures of curricula from previous work. (a) Linear sequences in a gridworld domain [83], where the goal of the agent is to pick up a key and use it to unlock a lock. Based on the agent's sensing and action capabilities, the curriculum sequentially teaches skills such as navigating to keys and locks while avoiding pits. (b) Directed acyclic graphs in block dude [123], where to goal is to build a staircase of blocks that allow the agent to reach the exit door. In a graph form, a curriculum allows different skills to be learned in parallel and then combined.

training sessions in which the agent must repeatedly transfer knowledge from one task to another, up to a set of final tasks.

3.3 Evaluating Curricula

Curricula can be evaluated using the same metrics as for transfer learning (cf. Chapter 2), by comparing performance on the target task after following the complete curriculum, versus performance following no curriculum (i.e., learning from scratch). If there are multiple final tasks, the metrics can easily be extended: for example, by comparing the average asymptotic performance over a set of tasks, or the average time to reach a threshold performance level over a set of tasks.

Similarly, it is possible to distinguish between weak and strong transfer. However, in curriculum learning, there is the additional expense required to *build* the curriculum itself,

in addition to training on intermediate tasks in the curriculum, which can also be factored in when evaluating the cost of the curriculum. As in the transfer learning case, cost can be measured in terms of wall clock time, or data/sample complexity.

Most existing applications of curricula in reinforcement learning have used curricula created by humans. In these cases, it can be difficult to assess how much time, effort, and prior knowledge was used to design the curriculum. Automated approaches to generate a curriculum also typically require some prior knowledge or experience in potential intermediate tasks, in order to guide the sequencing of tasks. Due to these difficulties, these approaches have usually treated curriculum generation as a sunk cost, focusing on evaluating the performance of the curriculum itself, and comparing it versus other curricula, including those designed by people.

The best set of evaluation criteria to use ultimately depends on the specific problem and settings being considered. For example, how expensive is it to collect data on the final task compared to intermediate tasks? If intermediate tasks are relatively inexpensive, we can treat time spent in them as sunk costs. Is it more critical to improve initial performance, final performance, or reaching a desired performance threshold? If designing the curriculum will require human interaction, how will this time be factored into the cost of using a curriculum? Many of these questions depend on whether we wish to evaluate the utility of a specific curriculum (compared to another curriculum), or whether we wish to evaluate the utility of using a curriculum design approach versus training without one.

In this thesis, I will be evaluating methods using the asymptotic performance (Chapter 4), jumpstart (Chapter 5), and time to threshold metrics (Chapter 6). In these chapters, strong transfer will be shown by accounting for time spent training in source tasks of the curriculum, but will not account for time spent *generating* the curriculum. In Chapters 7 and 8, I will then present methods that produce strong transfer while accounting for time spent both generating the curriculum and time spent in source tasks of the curriculum, using

the asymptotic performance and time to threshold metrics.

3.4 Summary

In this chapter, I formalized the concept of a curriculum as a directed acyclic graph over sets of experience samples. This definition is able to represent many types of curricula present in the literature, including linear sequences, curricula over sets of tasks, and their combinations. After that, I presented the methodology of curriculum learning, which consists of 3 parts – task generation, sequencing, and transfer learning – and discussed how these approaches can be evaluated. In the next few chapters of the thesis, I will discuss methods we have designed to address the task generation and sequencing components of curriculum learning. These methods will utilize existing methods for transfer learning.

4. Task Generation

In this chapter, I consider the problem of how to generate a space of source tasks for use in a curriculum. This problem is a crucial part of any curriculum sequencing approach that builds on experience through multiple tasks, since the size and quality of the tasks available to choose from will affect the speed and quality of the resulting curriculum. In existing curriculum learning work, source tasks have often been manually hand-crafted by domain experts independently and tailored for each problem. To the best of my knowledge, very limited work (see Chapter 9) has been dedicated to formally studying domain-independent methods to address this subproblem.

In this chapter, I present an approach for semi-automatically creating an appropriate space of source tasks that can be used in both an online and offline fashion. Our approach relies on 2 key ideas. 1) Tasks should be created using some knowledge of the domain. In this work, we assume this knowledge comes from a parameterized model of the domain. 2) Tasks should be tailored to the abilities and progress of the agent over the course of its learning cycle. We do this tailoring by collecting and analyzing experience trajectories from the agent as it interacts with different tasks.

The approach consists of a series of methods that create tasks using these two ideas by altering different aspects of the target task MDP, such as the state space, action space, initial and terminal state distributions, reward function, and/or transition function. This chapter addresses Contribution 2 from Chapter 1 of this thesis.

This chapter is based on work that was published in the proceedings of the Autonomous Agents and Multi-agent Systems (AAMAS) conference [82]. It was done in collaboration with Jivko Sinapov, Matteo Leonetti, and Peter Stone. My collaborators assisted in formalizing some of the methods presented.

4.1 A Space of Tasks

Before I define methods to generate source tasks for a particular target task, I first define the domain $\mathcal{D}$, which forms the space of all possible tasks that could be created from the target task:

Definition 4.1. A domain $\mathcal{D}$ is a set of MDPs that can be expressed by varying a set of *degrees of freedom*, and applying a set of *restrictions*.

The *degrees of freedom* $\boldsymbol{f}$ of a domain are a vector of features $[f_1, f_2, \ldots f_n]$ that parameterize the domain. For example, in the Quick Chess domain (see Chapter 1), possible degrees of freedom could be the size of the board, the number of each type of piece, or whether special rules such as castling or en passant are allowed. Each $f_i \in \boldsymbol{f}$ has a range of values $\text{Rng}(f_i)$ that represents the possible values that feature can take. Furthermore, we assume there is an ordering defined over each $\text{Rng}(f_i)$ that corresponds to task complexity. Collectively, the ordering over these degrees of freedom encodes our domain knowledge of the task.

An instantiation of $\boldsymbol{f}$ in $\mathcal{D}$ results in a specific task (an MDP). We assume we have a generator τ that can create tasks given a domain and degree of freedom vector:

$$\tau : \mathcal{D} \times \boldsymbol{f} \mapsto M$$

By *restrictions*, we mean the set of tasks that can be formed by eliminating certain actions or states, modifying the transition or reward function, or changing the starting or terminal distributions of MDPs generated by τ.

Informally, $\mathcal{D}$ captures the universe of possible source tasks for use within the curriculum and could be potentially infinite in size. The goal is to create a subset of tasks in $\mathcal{D}$ that might be suitable for learning a given target task, using knowledge of the domain (specified by the ordering over the degrees of freedom), and tailored to the performance and

abilities of the learning agent (by observing the agent learning on a task).

Formally, given a target task MDP M_t and trajectory samples X consisting of tuples (s, a, s', r) from following some policy π_t on M_t, the goal is to create suitable source tasks $M_s \in \mathcal{D}$ that will lead to a policy in M_t that is better than π_t. Specifically, we want functions c of the following form:

$$c : M_t \times X \mapsto M_s$$

In the next section, I describe several methods that can serve as c to create suitable source tasks for a target task.

4.2 Methods

Intuitively, there are many different ways in which a task could be a useful source for transfer to M_t: it could have a smaller or more abstract state space; it could have some actions removed; it could focus on a useful subgoal; or it could drill a common mistake. Some of these source tasks could be generated by simply manipulating the degrees of freedom $\boldsymbol{f}$, and indeed we consider that case first. However, in the rest of the section, I define additional *domain-independent* instantiations for c that modify particular aspects of the target task MDP based on the agent's experience in the target task.

4.2.1 Task Simplification

The first method we propose, TaskSimplification (Algorithm 1), simplifies a task using knowledge of the domain's parameterization. Here, Simplify is a function that changes one of the degrees of freedom $f_i \in \boldsymbol{f}$ to a new $f_i' \in \text{Rng}(f_i)$, in order to make the task smaller or easier. In many domains, there is a natural interpretation for Simplify. For example, in Quick Chess, we could reduce the value of parameters such as the size of the board or the number of specific pieces. In multiagent settings, we can add cooperative agents or remove

adversarial ones.

Algorithm 1 Task Simplification

1: **procedure** TaskSimplification($M, X, \mathcal{D}, \boldsymbol{f}, \tau$)
2: $\quad \boldsymbol{f'} = \text{Simplify}(\boldsymbol{f})$
3: $\quad M' \leftarrow \tau(\mathcal{D}, \boldsymbol{f'})$
4: $\quad$ **return** M'

TaskSimplification transforms the state, action, transition, and reward functions of an MDP simultaneously, in a domain-specific way.

4.2.2 Promising Initializations

The second method is designed for tasks that have a sparse reward signal. In many RL problems, positive outcomes can be rare, especially at the onset of learning. An agent may have to reach the goal randomly or through some exploration scheme many times before the policy stabilizes. PromisingInitializations creates a task that initializes an agent near states that were found to have high reward.

Algorithm 2 Promising Initializations

1: **procedure** PromisingInitializations(M, X, d, δ, ρ)
2: $\quad Y \leftarrow \{(s, a, s', r) \in X : r \geq \rho^{th}$ percentile of all rewards in $X\}$
3: $\quad M' \leftarrow M$
4: $\quad \mathcal{S}'_0 \leftarrow \{\}$
5: $\quad$ **for** $(s, a, s', r) \in Y$ **do**
6: $\quad\quad \mathcal{S}'_0 \leftarrow \mathcal{S}'_0 \cup \text{FindNearbyStates}(s, X, d, \delta)$
7: $\quad M'.\mathcal{S}_0 \leftarrow \mathcal{S}'_0$
8: $\quad$ **return** M'

Here, the parameter $\rho \in [0, 100]$ is a percentile that defines the fraction of rewards an agent has seen in its experience trajectory X that it should consider to be positive outcomes. FindNearbyStates is a domain-dependent function that returns a set/distribution of states that are close to a given state, using either a distance metric $d : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}$ or a pseudo-distance based on steps away in a trajectory. The exact form depends on the

representation used for the MDP.

If the state space is factored, we can perturb the state vector by some amount δ such that the distance from the original state to the perturbed state (measured by d) is less than δ. In our Quick Chess example, if the state space consists of the positions of all pieces on the board, we can use a distance metric that measures the least number of "moves" needed to transform one board configuration to another. FindNearbyStates would return all configurations that are δ steps away. If the state space is not factored (for example, in a tabular representation), then we can use the trajectory samples X to find states that are at most δ steps away from a high reward state, and explore these further.

As I discuss in the related work (Chapter 9), ideas based on PromisingInitializations have subsequently been used to perform sequencing, such as the Reverse Curriculum Generation approach by Florensa et al. [32].

4.2.3 Mistake-Driven Subtasks

The next set of methods we introduce here create subtasks to help an agent avoid and correct its *mistakes*. In principle, a mistake is any action or sequence of actions (e.g., an option [122]) taken in a state that deviates from the optimal policy.

In practice, the agent does not know the optimal policy while learning, so we propose 3 alternative characteristics to automatically identify mistakes. The first is any action that leads to unsuccessful termination of an episode, such as not reaching a goal state. Second is any action that results in no change in state. Finally, a mistake could be any action that incurs a large negative reward. In the following methods, I use IsMistake to denote whether a mistake was detected, using these criteria.

Action Simplification

The first mistake-driven subtask generation method I propose, ActionSimplification, prunes the action set to create a subtask where mistakes are less likely. Action set pruning is especially useful in settings where actions have preconditions for success. For example, a robot must grasp an object before manipulating it. An autonomous car must be standing still before opening the doors. However, it is also useful when the agent has more abilities/behaviors than are necessary to complete the target task or subtask. Algorithm 3 shows a simple example of ActionSimplification that removes actions which commonly cause mistakes. The parameter $\alpha \in \mathbb{Z}$ is a threshold on the number of times an action should lead to a mistake before it is pruned. In practice, it may be useful to set these thresholds so that only one action is eliminated at a time, or only eliminated in certain states.

Algorithm 3 Action Simplification

procedure ActionSimplification(M, X, α)
  $M' \leftarrow M$
  $count(a) = 0, \forall a \in \mathcal{A}$
  $Y \leftarrow \{(s, a, s', r) \in X : \text{IsMistake}(s, a, s', r)\}$
  for $(s, a, s', r) \in Y$ **do**
    $count(a) += 1$
  $\mathcal{A}' = \{a \in \mathcal{A} : count(a) > \alpha\}$
  $M'.\mathcal{A} = M'.\mathcal{A} \setminus \mathcal{A}'$
  return M'

Mistake Learning

In contrast, the second method, MistakeLearning (Algorithm 4), directly tries to correct mistakes by rewinding the game back some number of steps, and having the agent learn a revised policy from there. Intuitively, focusing training on areas of the state space where the agent made a "mistake," gives access to this experience much faster, allowing the agent to also learn to correct itself much faster.

The question of how far back in the trajectory to rewind is an interesting challenge

Algorithm 4 Mistake Learning

```
1: procedure MistakeLearning(M, X, ε)
2:     M' ← M
3:     S'_0 ← {}
4:     Y ← {(s, a, s', r) ∈ X : IsMistake(s, a, s', r)}
5:     for (s, a, s', r) ∈ Y do
6:         S'_0 ← S'_0 ∪ Rewind(X, s, ε)
7:     M'.S_0 ← S'_0
8:     return M'
```

in and of itself. For now, Rewind is a simple method that looks back ϵ steps from S in trajectory X, and returns the found state. However, in principle it could be more complex, based on the type of mistake made or the situation where it was made. In our example of Quick Chess, we could rewind the game to determine what should have been done differently to avoid a checkmate.

4.2.4 Option-based Subgoals

The next method creates subtasks for learning subgoals. The options literature [122] identifies many approaches to finding subgoals. Many take a state-based approach, where the learner tries to find states that may have strategic value to reach. For example, McGovern and Barto [75], identify subgoals as states that occur frequently in successful trajectories. Menache et al. [76] try to find "bottleneck" states. Simsek and Barto [111] seek to create subgoals for "novel" states, since they facilitate exploration of regions of the state space that the agent normally does not reach. Finally, graph-based approaches such as Mannor et al. [73] identify states by clustering over a state-transition map.

OptionSubGoals (Algorithm 5) is designed to take any option discovery method (FindOption) to create a subtask. Specifically, it creates a task to learn an option given the option's termination set $\mathcal{S}_f$ and a pseudo-reward function r for completion. Since an option typically only involves a subset of the task's complete state space, this subtask allows quick learning of how to reach important states. For example, in Quick Chess, capturing the

queen would be an example of a useful subgoal.

Algorithm 5 Option Sub-goals

```
1: procedure OptionSubGoals(M, X, V, φ)
2:     M' ← M
3:     (S_f, r) ← FindOption(M, X, V, φ)
4:     M'.S_f = S_f
5:     M'.r = r
6:     return M'
```

Since my work takes place in the context of transfer learning, I introduce one additional option discovery method, FindHighValueStates (Algorithm 6), that uses high value states learned in a previous task as a subgoal. Specifically, it checks whether any of the learned values $V(s)$ for states encountered in our trajectory X exceed a threshold ϕ.

Algorithm 6 Find High Value States

```
1: procedure FindHighValueStates(M, X, V, φ)
2:     S_f ← {}
3:     r ← M.r
4:     for (s, a, s', r) ∈ X do
5:         if V(s) > φ then
6:             S_f ← S_f ∪ s
7:             r(s, a, s') = V(s)
8:     return (S_f, r)
```

Instead of using trajectory samples X, we can also extract high value states directly from the value function. For example, with a tabular representation, we can simply lookup states of high value. With function approximation, an optimization routine would be used to solve for high value states.

4.2.5 Task-based Subgoals

An alternative to creating subgoals within an MDP is to create them directly at the task level. Specifically, we set the termination set $\mathcal{S}_f$ of the input MDP to be the initiation set $\mathcal{S}_0$ of some other subtask, as shown in Algorithm 7:

Algorithm 7 Link Subtask

```
1: procedure LinkSubTask(M, M_s, V)
2:     M' ← M
3:     for s' ∈ M_s.S_0, s ∈ M.S, a ∈ M.A do
4:         M'.r(s, a, s') = V(s')
5:     M'.S_f ← M_s.S_0
6:     return M'
```

For example, we can create a subtask that terminates where PromisingInitializations starts as follows:

$$M_1 = \textsc{PromisingInitializations}(M_t, X, C, \delta, \rho)$$

$$M_s = \textsc{LinkSubTask}(M_t, M_1, M_1.V)$$

Applied to Quick Chess, this would create a task to reach configurations that are likely to lead to checkmate. The reward for reaching this terminal set is the value of the state in the subsequent task. This idea is similar to skill chaining [65], except that instead of learning options linking target regions to initiation sets, we link directly on tasks.

4.2.6 Composite Subtasks

Lastly, each of the previous subroutines c takes as input an MDP M_t and trajectory samples X, and returns a modified task MDP M_s. By passing the samples and resulting M_s as input to another function c, we can chain together arbitrary many subroutines to compose new source tasks.

Mathematically, let b and c be any two functions above. Assume we are given a target task MDP M_t and trajectory samples X from it. Then, the composite task $(b \circ c) = b(c(M_t, X), X)$, where for ease of exposition, we've left out the task specific threshold parameters.

Most of the domain-independent functions described previously make specific modifi-

cations to a particular part of the target task MDP. In contrast, TaskSimplification can potentially make changes to the state and action space, as well as the transition and reward functions all at once. Thus, in practice, tasks should be composed using TaskSimplification first, followed by the others.

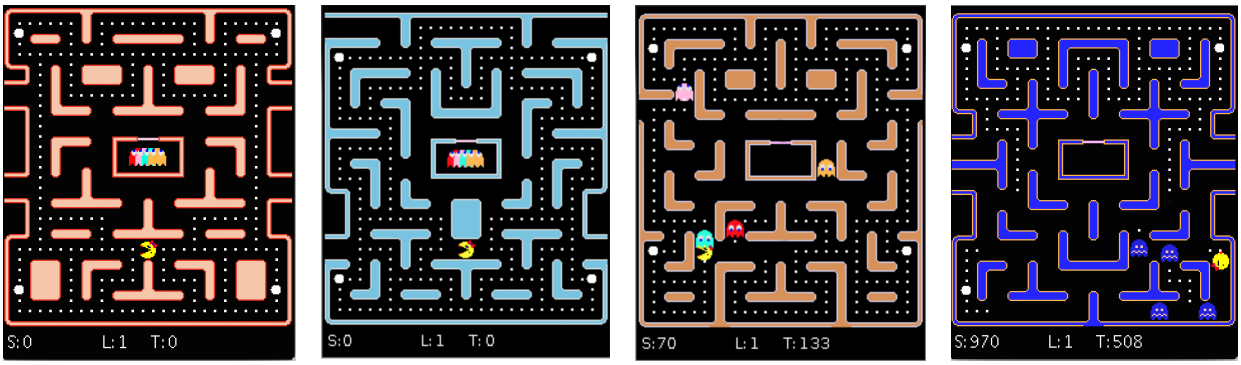
4.3 Ms. Pac-Man Experiments

We evaluated the ability of these heuristic methods to create useful source tasks for curricula in two challenging multi-agent domains – Ms. Pac-Man and Half Field Offense – to show the domain-independent applicability of these methods. In this section, we first demonstrate the effectiveness of domain-dependent and domain-independent subtasks in a simple one-stage curriculum (i.e. classic transfer learning paradigm) applied to Ms. Pac-Man.

Ms. Pac-Man (see Figure 4.1) is a game in which the agent's goal is to traverse a maze and earn points by eating edible items such as pills, while avoiding ghosts. The game typically starts with a large number of pills, four power pills located near each corner, and four ghosts that are initially placed in a lair that is inaccessible to Ms. Pac-Man. Shortly after the game starts, the ghosts leave their lair and may either chase Ms. Pac-Man or move about randomly. If a ghost catches Ms. Pac-Man, the game is over (we did not model the number of lives that are typically available to a human player). Whenever the agent eats one of the four power pills, the ghosts themselves become edible by Ms. Pac-Man for a short amount of time and their speed is reduced. If a ghost is eaten during that time, Ms. Pac-Man earns points and the ghost is sent back to the lair for a fixed amount of time, after which it starts to operate as normal. The agent's action space consists of four actions – up, down, left, and right – though not every action is available in every state. Ms. Pac-Man eats pills, power pills, and ghosts (when edible) whenever she gets within a small distance threshold of the object. Table 4.1 lists the rewards Ms. Pac-Man can get for different events in the game. The game ends when all the pills are gone, Ms. Pac-Man is eaten by a ghost, or 2000 time

Table 4.1: The Reward Structure of the Ms. Pac-Man Domain

Event	Reward (points)
Ms. Pac-Man eats a pill	10
Ms. Pac-Man eats a power pill	50
Ms. Pac-Man eats a ghost	200
Ms. Pac-Man eats an additional ghost while they are still edible	Apply a multiplier of 2 to the usual reward for each additional ghost that is eaten
Ms. Pac-Man is eaten by a ghost	Game Over



(a) (b) (c) (d)

Figure 4.1: Examples of tasks in Ms. Pac-Man. (a) Maze 1 (b) Maze 2 (c) Maze 3 (d) Maze 4

steps pass.[5]

We used the Ms. Pac-Man implementation described in Taylor et al. [130]. The raw state space of the game is high dimensional and also specific to each maze: there are over 1200 unique positions in the maze, and a complete state consists of the locations of Ms. Pac-Man, the ghosts, the pills, the power pills, each ghost's previous move, and whether or not each ghost was edible. These features make it unsuitable for learning. Therefore, in practice the state space in the Ms. Pac-Man game is typically represented by a set of local features that are ego-centric with respect to Ms. Pac-Man's position on the board (see [16, 99, 124] for

[5]A game play video (not associated with our work) can be found at http://youtu.be/c4n_6NFYvLY

a representative sample of approaches). In this work, we used 7 heavily-engineered features from Taylor et al. [130]. These features were chosen because they were already defined in existing work, and were more suitable for transfer than using the raw state representation of the game. In particular, these features calculate properties such as the safety of junctions, and scores for the amount of pills and ghosts that could potentially be eaten along a certain direction (see Taylor et al. [130] for full details on these features). Learning was done using Q-learning [121], and transfer via value function transfer. The action-value function was represented by a simple linear function approximator over those 7 features. This domain is also used for experiments in Chapters 5 and 7.

4.3.1 Maze Simplification Task

The first experiment is an application of the TaskSimplification method. The domain of Ms. Pac-Man comes with four different maze levels, some of which are easier for the agent to learn than the others. Thus, intuitively, one way to apply the TaskSimplification method is to train an agent on an easier maze and transfer the learned policy to a harder one. The results of such an application are shown in Figure 4.2. Here, the target task was maze level four (Figure 4.1d). The TaskSimplification principle was used to generate a source task by changing the maze level from four to one (Figure 4.1a). The transfer curve shows the effects of learning for 5 episodes on the source task and then learning for an additional 20 episodes on the target task. The baseline curve in contrast shows the result of learning for 25 episodes directly on the target task. Both curves are averaged over 20 runs. The results clearly show that applying TaskSimplification results in jumpstart and substantial improvement in the expected reward over the first 25 episodes.

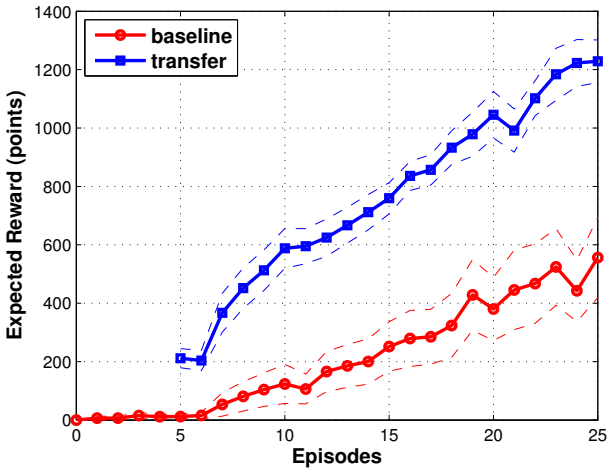


Figure 4.2: Results of TaskSimplification applied to the Ms. Pac-Man domain. Dashed lines indicate standard error.

4.3.2 Avoiding Ghosts Task

Next, we illustrate the use of an agent-specific source task, MistakeLearning, in the Ms. Pac-Man domain. We consider a mistake to be the event where Ms. Pac-Man is eaten by a ghost, which is a terminal non-goal state. Whenever a mistake occurrs, we spawn the following task:

$$M_{mistake} = \text{MistakeLearning}(M_t, X_t, \epsilon)$$

This call creates a subtask that rewinds $\epsilon = 50$ game steps from the moment the episode was terminated. The agent subsequently trains for 5 episodes in the generated subtask, after which training in the target task is resumed. The result of this test is shown in Figure 4.3. For this experiment, we measured the agent's performance as a function of the number of game steps, since episodes spent on learning in the generated subtasks were much shorter. Results are averaged over 20 trials. The plot shows that the application of MistakeLearning results in much faster learning when compared to the baseline approach of restarting each episode from the initial configuration upon episode termination.

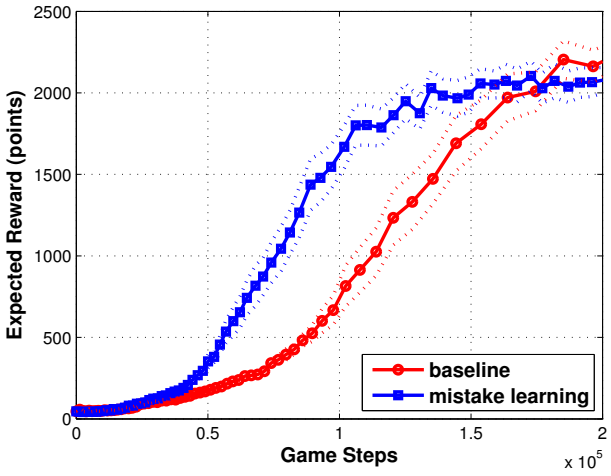


Figure 4.3: Results of MistakeLearning applied to the Ms. Pac-Man domain. See Section 4.3.2 for details. Dashed lines indicate standard error.

4.4 Half Field Offense (HFO) Experiments

In this section, we evaluate the ability of the methods to create tasks in a different domain, and also empirically evaluate their usefulness as tasks in a manually sequenced curriculum. The results show that the order of tasks in the curriculum has a significant impact on final performance.

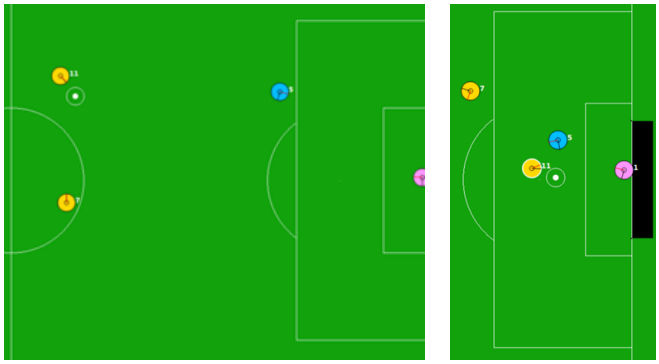
Half field offense [59] is a subtask of Robocup simulated soccer in which a team of m offensive players tries to score a goal against n defensive players while playing on one half of a soccer field. The domain poses many challenges, including a large, continuous state and action space, coordination between multiple agents, and multi-agent credit assignment. Each of these difficulties makes learning hard, especially early on when goal scoring episodes can be rare.

Each HFO episode starts with the ball and offensive team placed randomly near the half field line. Likewise, the defensive team is randomly initialized near the goal box. A sample starting configuration can be seen in Figure 4.4a. The goal of the offensive team is to move the ball up the field while maintaining possession, and take shots to score on goal.

Event	Reward
Goal	1.0
Ball out of bounds	-0.1
Ball with offense	0
Ball captured by defense	-0.2
Ball captured by goalie	-0.1
Episode times out	-0.1

Table 4.2: Reward structure in HFO

An episode ends when either (1) a goal is scored, (2) the ball goes out of bounds, (3) the defense captures the ball, or (4) the episode times out. The reward structure of the domain is shown in Table 1.



(a) (b)

Figure 4.4: Examples of tasks in Half Field Offense. (a) HFO initial configuration and 2v2 dribble task (b) 2v2 shoot task. Offensive players are colored yellow, defensive players are blue, and the goalie is pink. The ball is shown by the white circle.

As done by Kalyanakrishnan et al. [59], we focus on learning behaviors for the player with the ball. The player with the ball has to choose one of the following actions:

- Pass k: A direct pass to the teammate that is k-th closest to the ball, where $k = 2, 3, \ldots, m$.
- Dribble: A small kick in the cone formed between the player and the goalposts, that maximizes its distance to the closest defender also in the cone.

- Shoot j: A full power kick towards one of j evenly spaced points on the goal line.

Offensive players without the ball follow one of several fixed formations to provide support. The agent's state space consists of distances and angles to points of interest, such as other players, the goal posts, the ball, etc. These are listed in Table 4.3. We used CMAC tile coding for function approximation, Sarsa for the learning algorithm [121], and value function transfer to transfer knowledge (see Chapter 2 for a review).

Feature	Description
dist-to-goalie	Distance from O_1 to the goalie
dist-to-defender-in-cone	Distance from O_1 to the closest defender in the dribble cone
dist-to-teammate$_i$	Distance from O_1 to each teammate O_i, for $i = 2, 3, \ldots m$
dist-teammate$_i$*-to-closest-defender*	For each O_i, the distance to its closest defender, $i = 2, 3, \ldots m$
dist-teammate$_i$*-pass-intercept*	For each O_i, the shortest distance between a defender and the line between O_1 and O_i, $i = 2, 3, \ldots m$
min-ang-teammate$_i$*-defender*	For each O_i, the smallest angle between O_i, O_1, and a defender, $i = 2, 3, \ldots m$
dist-to-shot-target$_i$	Distance from O_1 to location i on the goal line, $i = 1, 2, \ldots j$
dist-goalie-to-shot-target$_i$	Distance from goalie to location i on the goal line, $i = 1, 2, \ldots j$
dist-shot$_i$*-intercept*	Shortest distance between a defender and the line between O_1 and location i on the goal line, $i = 1, 2, \ldots j$
ang-goalie-shot-target$_i$	Angle between goalie, O_1, and location i on the goal line, $i = 1, 2, \ldots j$
ang-defender-shot-target$_i$	Smallest angle between a defender, O_1, and location i on the goal line, $i = 1, 2, \ldots j$

Table 4.3: Feature space for the player with the ball in HFO. We index offensive players by their distance to the ball. Thus, the player with the ball is O_1 and its teammates are $O_2, O_3, \ldots O_m$.

Parameter	Range
Number Offense Players	$\{0, 1, \ldots 4\}$
Number Defense Players	$\{0, 1, \ldots 5\}$
Defense Behavior	{Agent-2D, Helios, WrightEagle}
Formation Type	{ Flat, Box, Trapezoid}
Field Width	20 – 68
Field Length	20 – 52.5
Max ball speed	0 – 5
Max player speed	0 – 1
Wind Noise	0 – 1

Table 4.4: Half Field Offense degrees of freedom

4.4.1 Space of Tasks

Half field offense has a number of degrees of freedom that allow creating many different types of tasks. We list some of the relevant degrees of freedom in Table 4.4. In addition to these, various aspects of the field (such as the size of the goals, the goal box, etc.), the players (such as visibility, stamina, etc.), and the world physics can also be changed.

These degrees of freedom allow us to quickly create many domain-specific source tasks, using the TaskSimplification rule. For example, we can add more teammates or reduce the number of defenders to give the offense more options. We can change the defensive team behavior to train against opponents of varying difficulty. We could also change various aspects of the world size and physics to make scoring and movement easier.

However, we can also create agent-specific source tasks by observing the behavior of the agent on the target task. For example, after observing generally unsuccessful trajectories on the target task, we could use MistakeLearning to recreate situations where the agent lost the ball or failed to score, in order to learn how to avoid or resolve them. Another option would be to build upon successful trajectories using PromisingInitializations, which would create tasks that initialize the offense at different positions near the goal, allowing them to drill on how to shoot.

4.4.2 Manual Sequencing Process

In this work, we stopped short of *automatically sequencing* tasks into a curriculum. The overall process we proposed was an incremental development of subtasks culminating in a full curriculum: an agent first tries learning M_t, but gets stuck at suboptimal policy π_t. Experience tuples X are generated from π_t, and used to generate a space of possible source tasks tailored for *this* agent at *this* particular point in its learning process. Here, we assumed a separate process (specifically, a human) was available to select a suitable source task M_s from this space. The whole procedure then repeats, with M_s possibly becoming the new M_t, until a curriculum emerges. In later chapters of this thesis, we will discuss ways this sequencing procedure can be automated (Chapters 6 to 8).

In the next sections, we illustrate the formal specification of tasks and creation of curricula that we found to be useful for 2 scenarios in half field offense.

4.4.3 2v2 HFO Curriculum

We first consider the target task of 2v2 half field offense, where 2 attackers must score against 1 defender and 1 goalie. We used agents from the released binaries of the Helios team to form the defensive team [1]. Helios and WrightEagle consistently place among the top teams in the annual Robocup 2D Simulation League tournament, making even this small version of half field offense a challenging task.

Let M_{2v2} denote the target task's MDP, and X_{2v2} be a set of (presumably generally unsuccessful) samples collected from M_{2v2}. We can generate this task $M_{2v2} = \tau(\mathcal{D}, \boldsymbol{f}_{2v2})$, using the following instantiations for the degree of freedom vector (the order of parameters is the same as in Table 4.4):

$$\boldsymbol{f}_{2v2} \quad = \quad [2, 2, \text{Helios}, \text{flat}, 68, 52.5, 2.7, 1, 0.3]$$

The following are specific subtasks that could be created using the methods from Section 4.2.

Shoot Task

One useful skill to learn is where a goal can be scored from. After having obtained some experience in the target task with at least a few goals, it is very likely that similar scenarios are also possible to score from. We can gradually expand this set of states that lead to a high reward termination using PROMISINGINITIALIZATIONS, where we use a Euclidean distance metric d over the agent's relative distances and angles to other players, to measure state proximity:

$$M_{shoot} = \textsc{PromisingInitializations}(M_{2v2}, X_{2v2}, d, \delta, \rho)$$

A sample scenario can be seen in Figure 4.4b. Essentially, this task creates different configurations of players near the goal, and drills shooting. In our experiments, we set $\delta = 3$ and $\rho = 0.10$.

Dribble Task

Initially while exploring, the agent takes many shots on goal from far away, which are unlikely to score. A skill the agent needs is the ability to move the ball up the field, maintaining possession away from defenders, until the agent reaches a state that it can score from. This can be accomplished by chaining ACTIONSIMPLIFICATION with M_{shoot} using LINKSUBTASK:

$$\begin{aligned} M_1 &= \textsc{LinkSubTask}(M_{2v2}, M_{shoot}, V_{shoot}) \\ M_{dribble} &= \textsc{ActionSimplification}(M_1, X_{2v2}, \alpha) \end{aligned}$$

LINKSUBTASK creates a subtask M_1 where the goal is to reach situations that the agent is likely to score from, as learned in M_{shoot}. ACTIONSIMPLIFICATION prevents the

agent from taking shots on goal from far away, since these actions usually lead to defense captures, and adds this restriction to M_1. An example of the initial configuration for the dribble task is shown in Figure 4.4a. In our experiments, we set $\alpha = 100$.

2v2 Curriculum Results

Figure 4.5 shows the performance on the target task of 2v2 HFO for learners following various curricula composed of the 2 tasks above. For each curriculum, we trained on sub tasks until convergence. Offsets in the curves represent time spent training in source tasks. Labels indicate the curricula used; baseline is learning on the target task without transfer.

The teams of agents were evaluated on their goal scoring ability: the fraction of times they are able to score a goal. Since each episode results in binary goal or no goal scored result, we used a sliding window of 200 episodes around each point to determine the average goal-scoring rate at each time step. All results are averaged over 25 trials. From Figure 4.5, it is clear to see that using a sequence of tasks from the space created significantly improves the final aymptotic performance, even when accounting for time spent training in source tasks.

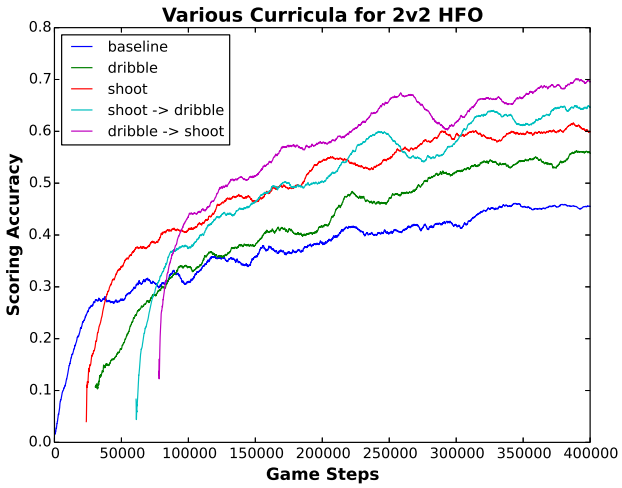


Figure 4.5: Goal scoring accuracy on 2v2 HFO for agents following different curricula. Standard error (not shown to avoid clutter) ranged from 0.015 to 0.027 over the last 200 episodes for all curves.

4.4.4 Extension to 2v3 HFO

In this section, we describe results extending the problem to the harder task of 2v3 half field offense, where there are now 2 defenders and a goalie. 2v3 is fundamentally harder than 2v2, since the additional defender means both attackers can now be marked. We can generate this target task $M_{2v3} = \tau(\mathcal{D}, \boldsymbol{f}_{2v3})$ using the following degree of freedom vector:

$$\boldsymbol{f}_{2v3} = [2, 3, \text{Helios}, \text{flat}, 68, 52.5, 2.7, 1, 0.3]$$

This time, we can use TaskSimplification to simplify the degree of freedom vector to recreate the 2v2 task from the last section, allowing us to use it as a source for 2v3:

$$M_{2v2} = \textsc{TaskSimplification}(M_{2v3}, X_{2v3}, \mathcal{D}, \boldsymbol{f}_{2v3}, \tau)$$

Doing this simplification also allows us to utilize the dribble and shoot tasks, since they are derived from M_{2v2}. Thus, we now consider 3 possible source tasks for a curriculum: $M_{dribble}$, M_{shoot}, and M_{2v2}. Results of various curricula composed of these source tasks can be seen in Figure 4.6. Again, using a multistage sequence of tasks provides better asymptotic performance than a curriculum composed of a subset of its source tasks. Interestingly, we also find that the most effective curriculum in 2v2 HFO is a subset of the best curriculum in 2v3 HFO when considering this space of tasks. This observation suggests that an automated procedure to create curricula could be designed recursively.

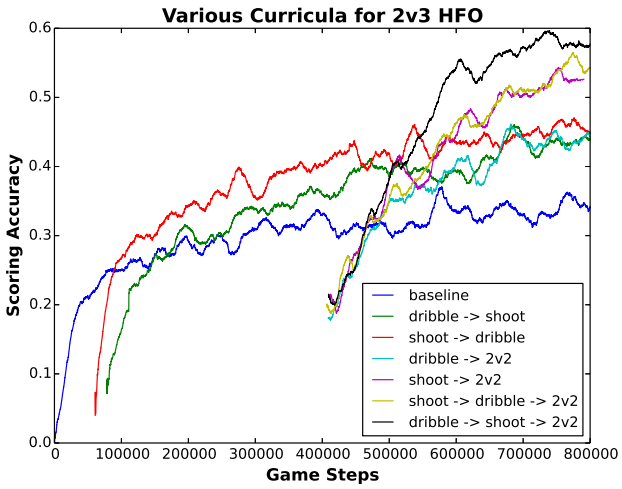


Figure 4.6: Goal scoring accuracy on 2v3 HFO for agents following different curricula. Standard error (not shown to avoid clutter) ranged from 0.010 to 0.039 over the last 200 episodes for all curves.

4.5 Summary

Task generation is the problem of creating tasks that could serve as suitable components of a curriculum for a target task. This subproblem is an important component of the overall methodology of curriculum learning (see Chapter 3), because it defines the space of curricula that can be considered.

In this chapter, I defined a space of possible tasks, and presented several functions that could create suitable source tasks for a given target task. The methods can be categorized into two types: the first allows for task creation using domain knowledge. The others are largely domain-independent, and rely directly on trajectory samples in the target task to create *agent-specific* tasks tailored for an agent at a particular point in its learning process. We claim that the functions outlined are broadly and generally useful. However, they are not the only possible methods; nor would every method apply to every domain. Nonetheless, the experiments showed that these methods could be applied to 2 complex RL domains. Furthermore, I showed that they could be used to successfully create curricula that provide a higher asymptotic performance than training without one, and that the order of tasks in such a curriculum is important.

In this chapter, the curriculum was sequenced manually. In subsequent chapters of this thesis, the methods described in this chapter will be used to form the space of tasks that will be used in a curriculum, and I will describe automated approaches towards sequencing (Chapters 6 to 8).

5. Measuring Inter-task Transferability

In this chapter, I consider the problem of source task selection: how to choose a good source task for transfer to a specified target task. Source task selection is a simplified version of the sequencing subproblem of curriculum learning; specifically, it evaluates how good a single source task is for transfer to a single target task, and does not explicitly consider the effect or selection of a chain of multiple source tasks (i.e., a curriculum). It also assumes the set of sources is given beforehand.

While many source task selection methods use samples or a model of the target task to aid in task selection [3, 68, 86], in this work we take a different approach, and try to predict task transferability using only a task descriptor for each task. This task descriptor is an instantiated vector of the degrees of freedom from Chapter 4. We use these task descriptors to train a regression model that predicts the benefit of transfer between source-target task pairs, and produces a transferability matrix that gives the expected benefit between any pair of tasks. We quantify the benefit of transfer using the jumpstart metric from Chapter 2, though other metrics could also be used, and evaluate our approach in a large-scale experiment involving 192 different variations of the game of Ms. Pac-Man.

As a first step towards multi-stage sequencing, we also evaluate using the transferability matrix to chain source tasks together into a curriculum. We do this evaluation by recursively finding an appropriate source task for the target task, and then finding an appropriate source task for the previous source task. While this process is a naive "one-step" approach that ignores possible overlaps and interactions between sources, it simplifies the

The ideas in this chapter are based on work that was published in the proceedings of the Autonomous Agents and Multi-agent Systems (AAMAS) conference [112]. It was done in collaboration with Jivko Sinapov, Matteo Leonetti, and Peter Stone. My collaborators helped with formulating the idea and experiments, as well as writing the paper.

otherwise combinatorial search for a curriculum. This chapter addresses Contribution 3 from Chapter 1 of this thesis.

5.1 Modeling Task Transferability

In this section, I introduce our proposed framework for modeling inter-task transferability. First, in Section 5.1.1 I introduce notation specific to this setting, and formulate the problem. Next, in Section 5.1.2 I discuss the main idea of how we will predict the benefit of transfer from one task to another. Finally, in Section 5.1.3 I discuss how this approach will be evaluated.

5.1.1 Notation and Problem Formulation

Let $\mathcal{M}$ be the set of possible tasks. Let $\mathcal{M}_{source} \subset \mathcal{M}$ be a set of tasks for which the agent has learned a policy and let $\mathcal{M}_{target} \subset \mathcal{M}$ be another set of tasks that represents the set of target tasks to be learned by the agent. For each task $M_i \in \mathcal{M}$, let $\boldsymbol{f}_i \in \mathbb{R}^n$ be a feature descriptor for the task that is known to the agent (as in Chapter 4).

Given a target task $M_j \in \mathcal{M}_{target}$, the goal of the agent is to select a task $M_i \in \mathcal{M}_{source}$ such that M_i serves as an effective source for learning M_j. Thus, given a task pair, M_i and M_j, let $B(M_i, M_j) \in \mathbb{R}$ denote the benefit of transferring the policy learned in N_i to the task M_j, where $B(M_i, M_j) > 0$ indicates positive transfer, while $B(M_i, M_j) < 0$ indicates negative transfer. In this work, the transfer benefit is estimated using the jump-start measure defined in Chapter 2, though in principle, other measures can be appropriate as well.

We assume that for each pair of source tasks (M_i, M_j) such that $M_i, M_j \in \mathcal{M}_{source}$, the agent has a reliable estimate for $B(M_i, M_j)$. Next, we describe how the agent can use these estimates to predict the expected transfer benefit between tasks in $\mathcal{M}_{source}$ and tasks in $\mathcal{M}_{target}$.

5.1.2 Predicting the Benefit of Transfer

Here, the task of the agent is to learn a function which, given two arbitrary tasks M_i and M_j from $\mathcal{M}$, can predict whether M_i is a good source task for M_j. More specifically, the function should produce the estimate $\hat{B}(M_i, M_j)$, i.e., the expected benefit of transferring from M_i to M_j. Since $B(M_i, M_j) \in \mathbb{R}$, a natural solution for modeling the transferability between two tasks is to train a regression model.

Let $\boldsymbol{f}_i = [f_{i,1}, f_{i,2}, \ldots, f_{i,n}]$ and $\boldsymbol{f}_j = [f_{j,1}, f_{j,2}, \ldots, f_{j,n}]$ be the features for a pair of tasks (M_i, M_j). To train a regression model on task pairs, a third feature vector is computed, Z^{ij}, such that it captures some aspects of how the two feature vectors $\boldsymbol{f}_i$ and $\boldsymbol{f}_j$ are related. The feature vector Z^{ij} was computed such that each element z_k^{ij} is defined by:

$$z_k^{ij} = \frac{f_{i,k} - f_{j,k}}{max(f_{i,k}, \epsilon)}$$

where ϵ is a very small number to avoid divisions by 0. In other words, the vector represents the change along the n-dimensional features space relative to the feature values of the first task in the pair.[6] The function that computes how two tasks are related was designed to be sensitive to the order of the tasks in the pair since preliminary experiments suggested that task transferability is not always symmetric.

Given this representation and a dataset $\{Z^{ij}\}_{M_i, M_j \in \mathcal{M}_{source}}$, a regression model Y is trained such that:

$$Y(Z^{ij}) \approx B(M_i, M_j)$$

Once trained on pairs of tasks from $\mathcal{M}_{source}$, the regression model is subsequently used to select source tasks for the tasks in $\mathcal{M}_{target}$. Given a target task M_j, the task $M_i \in \mathcal{M}_{source}$ that maximizes $Y(Z^{ij})$ is selected as the source task. Next, we describe the performance measures that were used to evaluate the framework proposed here.

[6] Other representations for the vector Z^{ij} were explored as well, including raw difference (i.e., $f_{i,k} - f_{j,k}$) as well as ratio (i.e., $f_{i,k}/f_{j,k}$). Representations that captured the absolute or squared distance between $\boldsymbol{f}_i$ and $\boldsymbol{f}_j$ did not perform as well as they were not sensitive to the order of the tasks in the pair.

5.1.3 Evaluation

For each target $M_j \in \mathcal{M}_{target}$, the best possible source task is defined by:

$$M^* = \arg\max_{M_i \in \mathcal{M}_{source}} B(M_i, M_j)$$

Let M_i be the source task selected by the model. To compare the model's choice for a source task to the optimal source task, we define the *loss* as:

$$loss(M_i) = B(M^*, M_j) - B(M_i, M_j)$$

We also evaluated the ranking of source tasks induced by the regression model. For a given target task M_j, let $O_j = [M_{\{1\}}, M_{\{2\}}, \ldots, M_{\{P\}}]$ be the ranked list of source task according to the learned regression model, i.e., $\hat{B}(M_{\{k\}}, M_j) \geq \hat{B}(M_{\{k+1\}}, M_j)$. For each position k in the ranking, let $rel_k = B(M_{\{k\}}, M_j)$ be a measure of the relevance of the result at that position. A common measure to evaluate the quality of a ranking is the Discounted Cumulative Gain (DCG) [57]:

$$DCG_p(O_j) = rel_1 + \sum_{k=2}^{p} \frac{rel_k}{log_2(k)}$$

where $p \leq P$. The normalized DCG (NDCG) is computed by $\frac{DCG(O_j)}{DCG(O_j^{best})}$ where O_j^{best} is the true (i.e., best possible) ranking of source tasks. A normalized DCG of 1.0 would indicate a perfect ranking.

For a baseline comparison, we consider the naive approach of selecting the most similar task according to the feature vectors used to describe the tasks. In other words, given target task M_j, the naive method would select the source task M_i that minimizes the squared distance between $\boldsymbol{f}_i$ and $\boldsymbol{f}_j$. The baseline approach does not perform any learning but nevertheless, we hypothesize that it will perform better than randomly selecting a source task.

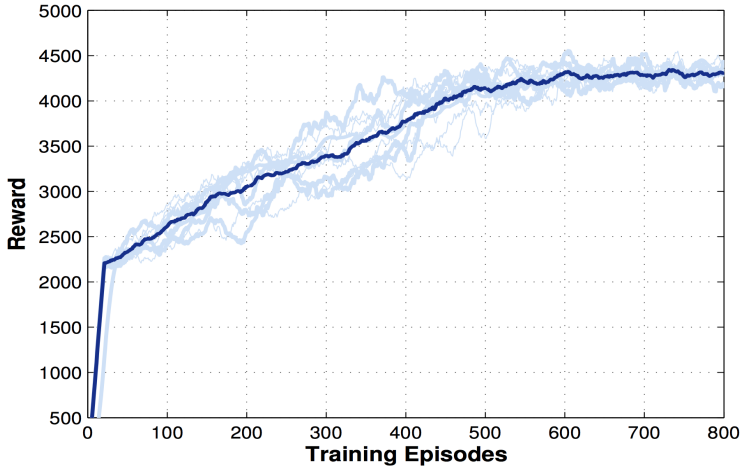


Figure 5.1: An example baseline test for one of the 192 tasks. The dark line indicates the reward averaged after 10 different runs (shown as the lighter lines), each starting with a different random seed. In this example, the policy converged after about 700 episodes.

5.2 Experimental Domain and Methodology

To evaluate the proposed framework, we conducted a large-scale experiment in the Ms. Pac-Man domain (described previously in Section 4.3). The agent was trained using Sarsa [121].

We generated 192 variations of the Ms. Pac-Man task by varying several parameters that dictate the dynamics of the game:

- **Maze**: each game was played on one of four different mazes, shown in Figure 4.1.
- **Number of ghosts**: the number of ghosts present in the game was varied from 1 to 4.
- **Ghost slowdown**: when Ms. Pac-Man eats a power pill, the ghosts become edible and their movement speed is reduced. The *ghost-slowdown* parameter specified the amount of speed reduction and varied from 1 to 4, in increments by 1. When the Ghost slowdown is set to n, then the ghosts remain stationary every n^{th} game step when they are edible. Thus, a lower value makes the ghosts move slower.

- **Ghost type**: the ghosts behaved according to one of three different modes: *Standard*, *Random*, and *Chaser*

The three different ghost behaviors are as follows: (1) *Standard ghosts* chase Ms. Pac-Man 80% of the time and move randomly the other 20%. When Ms. Pac-Man eats a power pill, the ghosts start moving away from the agent and eventually revert to their original behavior once they are no longer edible; (2) *Random ghosts* choose a random direction when reaching a junction 100% of the time. This makes it easier for Ms. Pac-Man to avoid them, but harder for Ms. Pac-Man to catch consecutive ghosts after eating a power pill. (3) *Chaser ghosts* have the same behavior as the Standard ghosts when inedible. However, after Ms. Pac-Man eats a power pill, they continue moving towards Ms. Pac-Man instead of fleeing. This makes it easy for Ms. Pac-Man to learn to eat ghosts (sometimes also too easy, since Ms. Pac-Man can learn to just stay in place and let the ghosts come to it, which does not transfer well to the normal setting).

Varying the four parameters resulted in $4 \times 4 \times 4 \times 3 = 192$ versions of the game. These 192 tasks constituted the full set of tasks $\mathcal{M}$. To compute transferability for all pairs of tasks, the agent first learned to play each task from scratch for $2,500$ episodes (the number of total episodes was chosen such that the agent's policy converged on each of the 192 tasks). Each episode consisted of playing a full game of Ms. Pac-Man. After each episode, the policy was frozen and the agent played an additional 10 games to compute a reliable estimate for the expected reward at each point during training. This procedure was repeated 10 times for each task in order to account for the stochastic nature of the domain. Thus, the agent played a total of $192 \times 2,500 \times (1 + 10) \times 10 = 50,800,000$ games to compute the baseline performance reward curves. Figure 5.1 shows an example baseline test for one of the 192 tasks. The bold line indicates the average reward curve from the 10 different runs.

Once the baseline curves were computed, the benefit of transfer was estimated for all task pairs. To do so, for each of the $36,672$ pairs of tasks (M_i, M_j) in $\mathcal{M}$, the agent

learned on task M_j for 30 episodes starting with the policy learned on task M_i (i.e., the agent transferred the policy from source task M_i to target task M_j). This process was repeated 10 times for each pair, such that in each run, a different one of the 10 policies computed during the baseline run was used as a starting point. Thus the agent played $36,672 \times 30 \times (1+10) \times 10 = 120,101,760$ games. The average reward with transfer and the average baseline reward over the first 30 episodes were then used to compute the jump start measure (see Chapter 2). Specifically, let $\boldsymbol{G}^{baseline} \in \mathbb{R}^K$ be the return curve after learning the target task for K episodes such that $G_k^{baseline} \in \mathbb{R}$ is the expected return after learning for k episodes. Similarly, let $\boldsymbol{G}^{transfer} \in \mathbb{R}^K$ be the return curve for learning the target task after transferring a policy from the source task. The jump start metric can then be defined by:

$$jumpstart(w) = \frac{\sum_{k=1}^{w}(G_k^{transfer} - G_k^{baseline})}{w} \tag{5.1}$$

The parameter w determines the size of the temporal window which is used to compute the jump start after the onset of training on the target task. We chose to use the jump start measure because of the large-scale nature of our experiment, and because computing it requires a relatively small number of training episodes on the target task. We computed the jump start measure for $w = 1$, 3, 5, 10, 15, and 30.

All told, to compute both the baseline reward curves as well as the transfer reward curves, the agent had to play over 170 million games. This type of an experiment would be next to impossible on a single computer and therefore, we used our department's Condor Cluster system [37]. A learning episode typically took about $0.5 - 0.75$ seconds, though this duration could vary depending on the cluster machine being used. Based on logged data, the experiment took over 2,300 hours of compute time spread over 192 individual machines.

The framework for learning task transferability proposed in Section 5.1.2 requires that the agent has access to a task descriptor vector that describes each task. Table 5.1

shows the task features that were used in our experiments. All of the features, except for *ghost-type*, are numeric. The *ghost-type* feature was originally nominal and therefore was converted into 3 different binary features, one for each type of ghost behavior. Thus, $\boldsymbol{f}_i \in \mathbb{R}^{17}$. The features that were used to describe the tasks corresponded to the parameters used to generate the tasks, as well as to graph-based features induced by the maze in each task. The features were not specifically selected or tuned to maximize performance. The graph-based features included domain-specific attributes (e.g., the distance between Ms. Pac-Man's starting position and the Ghosts' lair) as well as general graph-based features such as *eccentricity* as well as a histogram of the nodes' degrees (the last three features in the Table 5.1).

In our experiments, we explored two different implementations for the regression model Y described in section 5.1.2: 1) Linear Regression, and 2) M5 Model trees [91]. Linear Regression was selected due to its simplicity, while the M5 Model tree was selected as it is able to handle non-linear problems. Both implementations can be found in the WEKA machine learning library [48]. The WEKA implementation uses a modified version of the original tree induction algorithm, called M5P [139] which added pruning as a part of the training stage.

5.3 Experimental Results

We evaluated our task transferability approach to evaluate its utility in selecting good source tasks, and the ability of those tasks to be chained into a multi-stage curriculum. To do this evaluation, in Section 5.3.1, we first compute a ground truth transferability matrix for all pairs of tasks, that gives the true jumpstart for each pair of tasks. Then in Section 5.3.2, we partition the set of tasks in this matrix to train a regression model as described in Section 5.1.2. Next, in Section 5.3.3, we evaluate how well the learned regression model performs compared to the ground truth jumpstart metrics. Finally, in Section 5.3.4, we

Table 5.1: The task features that were known to the agent

Feature	Description
number-of-ghosts	The number of ghosts in the game (1 to 4)
ghost-slowdown	The amount of speed reduction that the ghost undergoes when Ms. Pac-Man eats a power pill. The values ranged from 1 to 4.
ghost-type	The behavior of the ghosts. There are three possible values: *Random*, *Standard*, and *Chaser*.
num-nodes	The number of nodes in the maze graph
num-pills	The number of regular pills in the maze
distance-to-ghost	The distance between Ms. Pac-Man and the ghosts at the start of the game
distance-power	The average distance between power pills
distance-lair	The average distance between the ghost lair and the power pills
junctions-between-junctions	The average number of junctions that lie on the shortest path between any pair of junctions
eccentricity	The average eccentricity of nodes in the graph. The eccentricity for a node u is defined as $e(u) = max\{d(u,v) : v \in V\}$ where d is the shortest-path function for a pair of nodes and V is the total set of nodes in the graph.
eccentricity-junction	The average eccentricity of junctions (i.e., nodes with more than 2 neighbors). The eccentricity for a junction node u is defined as $e(u) = max\{d(u,v) : v \in J\}$ where $J \subset V$ is the set of nodes that are junctions.
graph-diameter	The diameter of the graph is defined as $diam(G) = max\{e(u)|u \in V\}$.
num-nodes-d2	Number of nodes with 2 neighbors
num-nodes-d3	Number of nodes with 3 neighbors
num-nodes-d4	Number of nodes with 4 neighbors

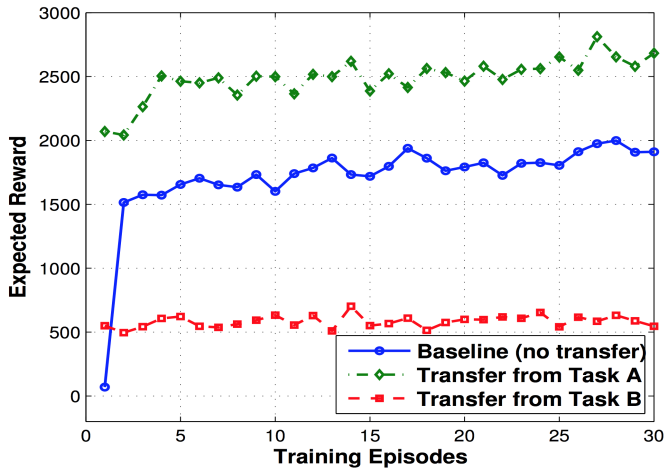


Figure 5.2: An example transfer result for a given target task and two potential source tasks. Task A is clearly the better source task, resulting in a large positive transfer.

explore whether the task selection method derived from the transferability matrix can be used to chain multiple tasks into a task-level sequence curriculum.

5.3.1 The Transferability Matrix

Figure 5.2 shows an example transfer result for a target task and two different source tasks. In this case, transferring the policy from one of the source task to the target task results in positive transfer, while the other source task induces negative transfer. Figure 5.3 shows the whole transferability matrix computed for the set of 192 tasks considered in our experiments. In this example, each entry contains the expected benefit of transfer according to the *jumpstart*(30) measure for each pair of tasks (in other words, the jump start was computed over the first 30 training episodes on the target task). White values indicate high jump start while black values indicate low (possibly negative) jump start.

The order of the columns and rows of the matrix is not random but rather, the entries are sorted first according to the *maze*, then *ghost-type*, then *ghost-slowdown*, and then finally, *number-of-ghosts*. The last 1/4 set of columns in the matrix appear brighter than the rest because those tasks were much more likely to benefit from transfer. These tasks corresponded

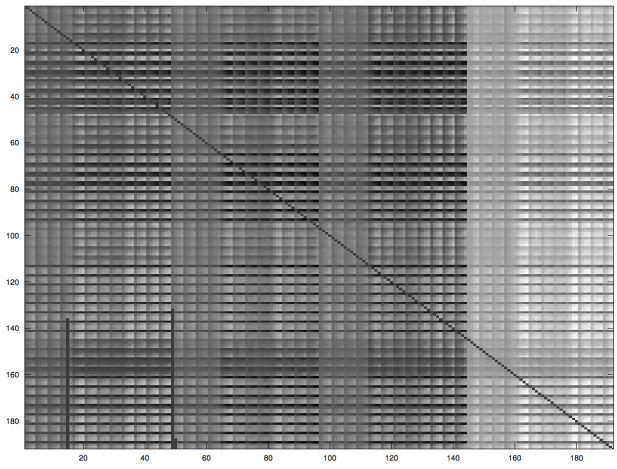


Figure 5.3: An example transferability matrix computed for each pair of the 192 tasks considered in our experiments. In this matrix, the entry at i, j amounts to the resulting $jumpstart(30)$ measure after transferring the policy learned on task M_i to task M_j. Light values indicate high jump start while black values indicate low (possibly negative) jump start.

to tasks with the fourth maze, which proved to be much more difficult for the agent than the other three mazes. The grid-like pattern shows that transfer is not random and hence, we hypothesized that the parameters that define the tasks may be useful in predicting the benefit of transfer across tasks.

Figure 5.4 shows a histogram of the jump start measures for two randomly chosen target tasks (i.e., a histogram over the values in a given column of the transferability matrix). Even though the shapes of the histograms are similar, one of the target tasks is much more likely to benefit from transfer. For the first target task (top histogram), virtually all source tasks result in positive transfer. For the second target task, however, there are a large number of source tasks that induce negative transfer, which further motivates the need for effective source task selection.

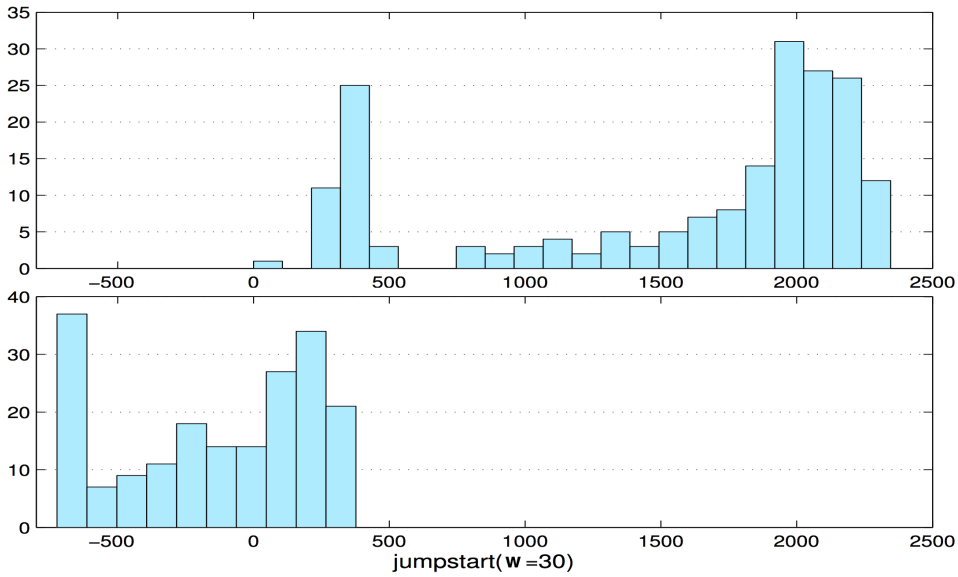


Figure 5.4: Example histograms of the jump start measures for two randomly chosen target tasks (i.e., a histogram over the values in a given column of the transferability matrix). For the first target task (top histogram), virtually all source task result in positive transfer, while for the second, there are a large number of source tasks that induce negative transfer.

5.3.2 Regression Model Performance

The performance of the regression model used to estimate transferability was evaluated using 10-fold cross validation at the task level. In other words, during each run, the tasks were split into 10 sets such that 9 of these formed the set $\mathcal{M}_{source}$ while the remaining fold was considered as the set of target tasks $\mathcal{M}_{target}$. The regression model was trained on all pairs of tasks (M_i, M_j) such that $M_i, M_j \in \mathcal{M}_{source}$ and then tested on all pairs of tasks induced by the cross product of $\mathcal{M}_{source} \times \mathcal{M}_{target}$.

Table 5.2 shows the performance of the two regression algorithms that were used to predict the $jumpstart(w)$ measure for different values of w, the size of the temporal window used to computed the jump start. The results are reported in terms of the Correlation Coefficient (CC) between the actual and the predicted values. These results show that the difficulty of modeling task transferability depends on the measures used to estimate the benefit of transfer. For example, modeling the jump start after just 1 training episode on

Table 5.2: Regression Model Performance measured by Correlation Coefficient

Transferability Measure	Linear Regression	M5P Model Tree
$jumpstart(w = 1)$	0.54	0.74
$jumpstart(w = 3)$	0.64	0.85
$jumpstart(w = 5)$	0.65	0.87
$jumpstart(w = 10)$	0.66	0.87
$jumpstart(w = 15)$	0.65	0.86
$jumpstart(w = 30)$	0.61	0.83

the target task is more difficult than modeling the jump start after 10 episodes on the target task. Overall, the CCs are high enough that we expect the ranking induced by the regression models to be useful for source task selection.

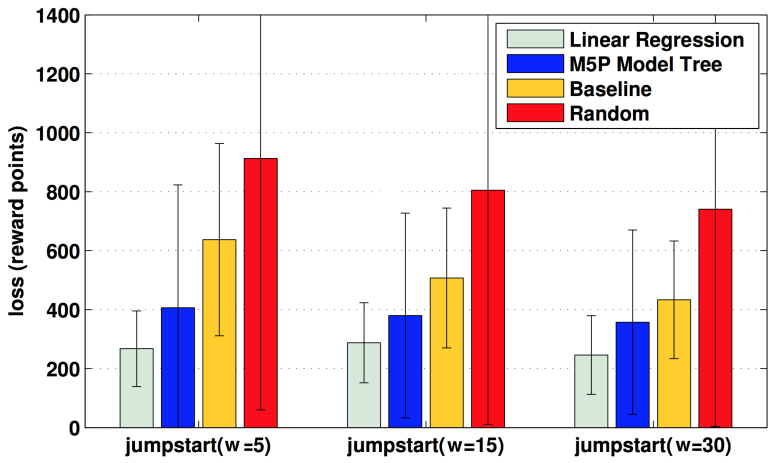


Figure 5.5: Source Task Selection *loss* for three transferability measures. The two regression models were compared with the baseline source task selection model and with random source task selection.

5.3.3 Source Task Ranking and Selection

Next, the framework for source task selection proposed in this paper was evaluated in terms of the expected *loss*, i.e., if the agent selects the source task that maximizes the expected transferability according to the regression model, how much worse does it do compared to selecting the optimal source task that it has already learned. Figure 5.5 shows the result of this test for two different regression algorithms, as well as the baseline approach. In addition, as a sanity check we computed the loss when randomly selecting a source task.

As we expected, the baseline approach which selects a source task based on task similarity in the task feature space performs better than randomly selecting a source task. Furthermore, the proposed method for learning task transferability substantially outperforms the baseline approach. While the Linear Regression (LR) model performed worse in terms of Correlation Coefficient when compared to the M5P Tree (M5P), the top source task selected when using LR tended to be a better source task than the one selected by M5P. An important question is whether performance would suffer as the set of tasks used to train the regression model becomes smaller. To obtain an answer, the number of tasks used to train the model was varied from 2 to 30 and we found that the expected loss converges after about 20 tasks (i.e., 400 pairs) are available for learning the regression.

The quality of the rankings were further evaluated using the Normalized Discounted Cumulative Gain measure. The results of this test are shown in Figure 5.6. Overall, LR performed the best. These results conclusively show that inter-task transferability can be learned even without samples or models of the target task. In particular, when faced with a new target task, a single good source task can be selected for transfer. These results naturally raise the question of whether it is possible to chain together multiple such source tasks sequentially to do even better. We examine that question next.

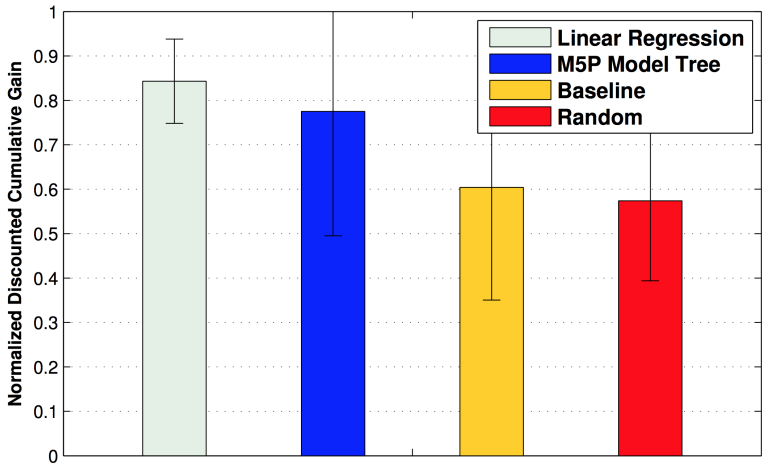


Figure 5.6: Evaluation of source task ranking using the learned regression model and the baseline case-based reasoning approach. The ranking was evaluated using the Normalized Discounted Cumulative Gain (DCG_p) and the $jumpstart(w = 5)$ measure (the results were similar for the remaining values of w used in this study). The value for p, the number of elements to be considered in the ranking (starting at position 1) was set to 20.

5.3.4 Multi-stage Transfer

In this section, we take our first steps towards automatic sequencing of a task-level sequence curriculum, by exploring whether we can chain together a sequence of tasks $M_1 \rightarrow M_2 \rightarrow \ldots \rightarrow M_{target}$, such that learning M_1 makes it "easier" to learn M_2, which makes it "easier" to learn M_3, and so on. For simplicity, consider two stage transfer: we are looking for source tasks M_1 and M_2 such that transferring from $M_1 \rightarrow M_2 \rightarrow M_{target}$ gives better performance than training directly on M_{target} or any of the one-stage transfers $M_1 \rightarrow M_{target}$ and $M_2 \rightarrow M_{target}$.

Candidates for the tasks M_1 and M_2 can be determined recursively using the transferability matrix. We simply look at the column corresponding to the target task, and select the row (i.e. source task) that provides the best transfer. The selected task then becomes the column for the next recursive stage.

A key question is how to decide how many episodes to spend on each source task. We used a heuristic approach based on the intuition that an agent should train on a source task

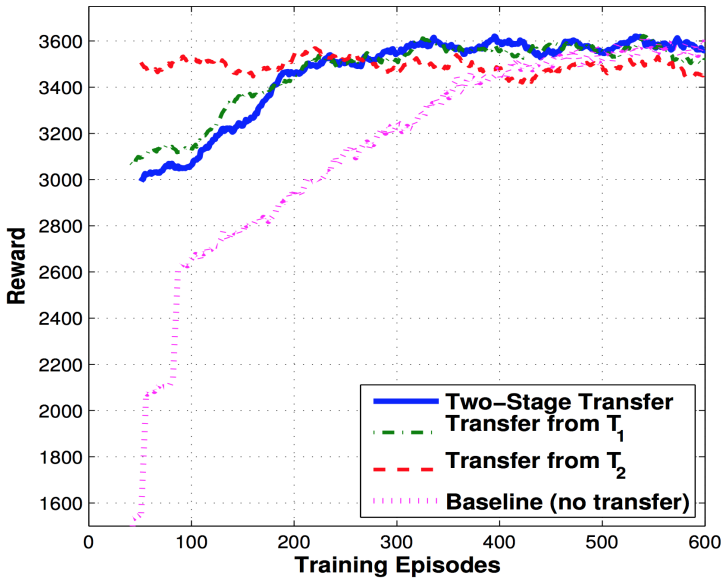


Figure 5.7: Performance on the target task using one and two-stage transfer. Note that the transfer curves are offset to reflect time spent training in their source tasks. In this example, all methods of transfer result in jump start but there is no benefit of two-stage transfer relative to single-stage transfer

until additional training does not improve performance on the target. Specifically, we define the target performance to be the total reward accumulated by the agent on the target task, for a fixed number of episodes (i.e. the area under the learning curve). Let A_{base} be the total reward accumulated by training directly on the target task without using transfer, and let $A^x_{transfer}$ be the total reward accumulated on the target task after training on the source task for x episodes, and using value function transfer. We used an incremental approach where the agent trained on the source task for 10 episodes, and used this to compute $A^x_{transfer}$. If the difference $(A^x_{transfer} - A_{baseline})$ was positive and increased, the agent trained on the source for 10 more episodes. This process was repeated until the difference no longer increased, at which point training on the source task was halted.

We hand-selected several of the more challenging tasks to serve as M_{target}. The results for one such target task are shown in Figure 5.7. All methods of transfer resulted in a jump start, but there was no benefit to using two stage transfer over single stage transfer. The

results were similar for the other target tasks and overall, we were not able to find a two-stage transfer that was significantly better than its one-stage counterpart. We suspect that this result is partly due to the fact that the mazes are very similar, especially from the view of agent using the highly engineered feature set. Therefore, a single stage transfer already initializes the policy in some area of the search space, and adding more stages does not noticeably refine or change this area. For a multi-stage curriculum to be useful, the source tasks ideally need to teach orthogonal skills, and/or allow the agent to accumulate rather than overwrite previously learned skills.

5.4 Summary

In this chapter, I introduced a framework for source task selection in settings where neither samples from the target task, nor a model of the task, are available to the learning agent. Instead, the agent used task descriptors (i.e., a low-dimensional feature vector describing the degrees of freedom of the task) to learn the expected benefit of transfer (i.e., transferability) between source tasks and target tasks. The framework was evaluated using a large-scale experiment in which the agent learned to play 192 variations of the Ms. Pac-Man game. Our results show that an agent can indeed learn to predict the transferability for an arbitrary pair of source-target tasks, provided training pairs for which the benefit (or detriment) of transfer is known. The learned transferability model was then used to effectively select relevant source tasks that improve the agent's learning performance on a given target task.

Identifying which source task is useful for a particular next target task is an important part of any sequencing method for curriculum learning. We further evaluated whether the approach could be used to select a multi-stage curriculum consisting of at least two source tasks, by using the transferability matrix to chain tasks together. However, we found that at least in this domain, there was no added benefit to using a second source task in the curriculum. This result could in part be due to the types of source tasks that were available

in this experiment, combined with a transfer learning method that was not able to accumulate information from these tasks.

In the next chapters (Chapters 6 to 8), we focus on sequencing methods that explicitly optimize for multi-stage curricula. In addition, we opt to take a different approach from the backward chaining method proposed here. The main reasons are that 1) the approach presented in this chapter is expensive to compute, especially when there are many source tasks; 2) the whole process needs to be redone if the agent's representation or learning algorithm changes; and 3) the approach does not account for the transfer learning method used. In this thesis, we assume the TL algorithm used is part of the agent and not within our control to change. The goal is to create a curriculum without changing any aspects of the agent itself.

6. Heuristic-based Approaches for Sequencing

Task sequencing is the subproblem most commonly associated with curriculum learning. The goal of task sequencing is to order a set of source tasks into a curriculum. In most existing reinforcement learning work that uses curricula, this sequencing has been done manually by domain experts separately for each problem. One reason for the prevalence of this approach is that finding a good sequence is hard – even when considering a simple task-based linear sequence, the number of possible curricula grows combinatorially with the number of tasks available. In some situations, the set of sources available may dynamically change to better reflect the types of experience the agent needs to acquire, rather than being statically fixed beforehand. In this case, the space of possible curricula is even larger.

Part of the goal of this thesis is to evaluate the extent to which this process can be automated. In this chapter, I start by presenting a heuristic-based approach that automatically sequences tasks into a curriculum. The core idea is to use trajectory samples of the agent's experience on the target task to guide the selection of appropriate source tasks. This approach utilizes the task generation methods of Chapter 4 to dynamically suggest candidate source tasks on the fly. Using dynamic sources and the agent's experience samples on the target task allows the approach to learn an individualized curriculum that is tailored to both the capabilities and learning progress of the agent. This chapter addresseses Contribution 4 from Chapter 1 of this thesis.

The ideas in this chapter are based on work that was published in the proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) [83]. It was done in collaboration with Jivko Sinapov and Peter Stone. My collaborators assisted in formalizing some of the ideas and editing the paper.

6.1 Method Intuition and Overview

For any curriculum sequencing method, we first need to consider the metric we are trying to optimize and the type of curriculum we are trying to design. In this chapter, we consider minimizing the time to threshold metric, which tries to find a curriculum that uses the least amount of time/experience to achieve a performance threshold δ on the target task. We are also looking to create a task-level sequence curriculum (see Chapter 3), which represents a curriculum as an ordered list of tasks $[M_1, M_2, \ldots M_n]$.

The intuition for the approach we propose is as follows. Assume the learning agent starts with some initial policy π_0. Our goal is to learn a policy π_f through training on a sequence of tasks that allows the agent to achieve a return of δ on the target task M_t as quickly as possible. Although we do not know what π_f looks like before training, we can identify what parts of the state space $\mathcal{S}$ are relevant to an optimal policy for the target task by *sampling* state trajectories in the target task. We can then use those samples to guide the selection of a source task. While initially these samples would be random and concentrated near the starting state, as the agent updates its policy through learning on source tasks, they will shift towards regions of the state space that are on the boundary between what the agent is already able to do, and what it still needs further practice on.

Thus, the main idea behind our proposed algorithm is to incrementally build up the policy using states and experiences the agent is currently facing. The algorithm samples from the target task to figure out what the learning agent needs to learn about. It then creates a set of sources tailored for those experiences using the heuristic functions defined in Chapter 4. Tasks that are too difficult for the agent to solve are recursively broken down by reapplying the task generation methods on the created source task.

In order to decide which task to select for the curriculum, we need to trade off how useful that task is to learn versus how difficult the task is to learn. A useful task is one that helps the agent make progress in the target task, which we can evaluate using the sample

trajectories collected on the target task. We make one important assumption, which is that the source tasks created by the methods do not result in negative transfer. We discuss the impact of this assumption in Section 6.2. With this assumption, we can then select the task which as a result of learning changes the policy the most according to the target samples. Learning the task updates the learning agent's policy. This update in turn leads to a different set of samples from the target task. This process repeats until the agent is able to solve the target directly. Note that the goal is not to learn a policy π for every state in the state space $\mathcal{S}$ of a target task M_t, as some of these states may not be encountered by the agent, and hence are irrelevant for executing the optimal policy π^*.

At the same time, a task could be very useful to train on, but the time needed to learn on it may not be worth it, especially if that same information can be acquired through easier tasks. The difficulty of a task can be quantified by the amount of time needed to solve the task. We will define what it means to solve a task in the next section (Section 6.2). This cost depends on many different factors, such as the policy the learning agent starts with, the learning algorithm being used, and also aspects of the task itself such as the size of its state and action spaces.

The only way to determine the true cost is to solve the task, which is unbounded and unknown ahead of time. Therefore, we introduce the idea of a budget or learning capacity β for the agent, which limits the amount of time an agent will spend trying to learn a task before it decides the task is too difficult. This process encourages learning easier tasks first, and tackling harder tasks once the learning agent has accumulated knowledge from these easy tasks. Finally, to prevent unnecessary tasks from being used in the curriculum, we require tasks to affect the policy and be relevant to the target task by at least fraction ϵ (described in detail in the next section).

Algorithm 8 GENERATECURRICULUM($M_t, \pi, \beta, \delta, \epsilon$)

```
1: C ← ∅
2: while true do
3:     size = |C|
4:     (π′, C) ← RECURSETASKSELECT(M_t, π, β, ϵ, C)
5:     if π′ = null then
6:         Increase β
7:         POP(C, |C| − size)
8:         continue
9:     π ← π′
10:    if EVALUATE(M_t, π′) ≥ δ then
11:        break
12: return (π′, C)
```

6.2 Algorithm Details

We now formalize the intuition given into pseudocode. The main call is to Algorithm 8, GENERATECURRICULUM, which takes as input the target task M_t that we want to generate a curriculum for, the learning agent's initial policy π (typically a uniform random policy), the learning budget β, the return threshold δ desired on the target task, and the minimum policy change and relevance parameter ϵ. It returns a policy π' that can achieve a return of δ on M_t, and the curriculum $\mathcal{C}$.

Each iteration of the loop in Algorithm 8 attempts to add a task to the curriculum by calling RECURSETASKSELECT (Algorithm 8 Line 4). If a task is found and added to the curriculum, the updated policy π' is evaluated on the target task (Algorithm 8 Line 10). The loop terminates if a return greater than δ is received. If no tasks are found, the budget β is increased, any tasks that were added in this phase are cleared, and the search is repeated.

Algorithm 9, RECURSETASKSELECT, is the core method that adds tasks to the curriculum and updates the policy π. It starts by calling LEARN, which attempts to *solve* the given task M starting with initial policy π, using at most β time steps. LEARN returns a boolean *solved* indicating whether the task was solved or not, a set of state-action-reward samples X for each trajectory experienced, and the updated policy π' as a result of learning

Algorithm 9 RecurseTaskSelect($M, \pi, \beta, \epsilon, \mathcal{C}$)

(solved, X, π') = Learn(M, π, β)
if solved **then**
    Enqueue($\mathcal{C}, M$)
    return ($\pi', \mathcal{C}$)
$\mathcal{M}_s \leftarrow$ CreateSourceTasks(M, X)
$\mathcal{P} \leftarrow \emptyset$
$\mathcal{U} \leftarrow \emptyset$
for $M_s \in \mathcal{M}_s$ **do**
    (solved$_{M_s}$, X_{M_s}, π_{M_s}) = Learn(M_s, π, β)
    if solved$_{M_s}$ **then**
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{(\pi_{M_s}, M_s)\}$
    else
        $\mathcal{U} \leftarrow \mathcal{U} \cup \{(M_s, X_{M_s})\}$
if $|\mathcal{P}| > 0$ **then**
    ($\pi_{\text{best}}, M_{\text{best}}$, score) = GetBestPolicy($\mathcal{P}, \pi, X$)
    if score $> \epsilon$ **then**
        Enqueue($\mathcal{C}, M_{\text{best}}$)
        return ($\pi_{\text{best}}, \mathcal{C}$)
SortBySampleRelevance($\mathcal{U}, X, \epsilon$)
for $(M_s, X_{M_s}) \in \mathcal{U}$ **do**
    ($\pi'_s, \mathcal{C}$) $\leftarrow$ RecurseTaskSelect($M_s, \pi, \beta, \epsilon, \mathcal{C}$)
    if $\pi'_s \neq$ null **then**
        return RecurseTaskSelect($M, \pi'_s, \beta, \epsilon, \mathcal{C}$)
return (null, $\mathcal{C}$)

M.

We propose two methods for determining whether a task has been solved. The first is *policy convergence*, which checks whether the policy has converged (i.e. not changed) for the states the agent has visited over the past few episodes. In addition, the episodes must terminate in a goal state. This requirement is in order to prevent an agent that has learned to quickly fail a task from being considered as successfully solving a task. The second is based on the *maximum return* possible in a task, where an agent that receives the maximum return possible on a task can be said to have solved it. The first method assumes the agent can detect a successful completion of a task, while the second assumes the max return (which is task specific) is known.

If the task M can be solved, it is added to the curriculum and the updated policy is returned (Algorithm 9 Line 4). Note that we only update the policy if the task can be solved. Otherwise, the learned policy may not be correct. If the task M cannot be solved, RecurseTaskSelect recursively tries to find and solve a simpler source task.

CreateSourceTasks(M, X) creates a set of source tasks $\mathcal{M}_s$ tailored to the agent's experiences X on M, using the heuristic functions defined previously in Chapter 4. We partition this set into two groups over lines 8 - 13 based on whether the source can be solved or not. $\mathcal{P}$ contains source tasks that could be solved and their corresponding updated learning agent policies. $\mathcal{U}$ contains tasks that could not be solved, and experience trajectories from the learning agent's attempts on those tasks.

If solvable tasks exist in $\mathcal{P}$, the curriculum design agent needs to select a task to add. We propose a heuristic GetBestPolicy (see Algorithm 10) that selects the policy-task pair $(\pi_{M_s}, M_s) \in \mathcal{P}$ that results in the greatest change in policy when evaluated on samples X from the target task. This heuristic implicitly assumes that the source tasks created are relevant to the target task and do not result in negative transfer.[7] Formally, for each state s encountered in the state sequence from samples X, we compare the action selected by $\pi(s)$, the policy before learning M_s, to $\pi_{M_s}(s)$, the policy after learning M_s, and count the number of states for which the action changed. This number is normalized by the number of states in the sequence X to produce a score. Note that states where the learning agent spends more time in M occur more often in X, and hence bias the score towards policies that update these states. The policy-task pair with the highest score is returned from GetBestPolicy, and if the score meets a minimum threshold ϵ, the task is added to the curriculum (Algorithm 9 Line 17). The threshold ϵ is used to prevent tasks that do not significantly impact the policy from entering the curriculum.

[7] In this setting, this is not an unreasonable assumption because the source tasks are explicitly created to be relevant to the target task, using the methods defined in Chapter 4. However, in the sequencing methods described in the next 2 chapters of this thesis (Chapters 7 and 8), we do not make this assumption.

Algorithm 10 GetBestPolicy($\mathcal{P}, \pi, X$)

$\boldsymbol{c} \leftarrow$ zeroes($|\mathcal{P}|$)
for $(s, a, r) \in X$ **do**
    $a_\pi \leftarrow \pi(s)$
    for $(\pi_{M_s}, M_s) \in \mathcal{P}$ **do**
        $a_{\pi_{M_s}} \leftarrow \pi_{M_s}(a)$
        if $a_\pi \neq a_{\pi_{M_s}}$ **then**
            $\boldsymbol{c}[M_s] = \boldsymbol{c}[M_s] + 1$
$(\pi_{\text{best}}, M_{\text{best}}) \leftarrow \mathcal{P}[\arg\max(\boldsymbol{c})]$
score $= \boldsymbol{c}[M_{\text{best}}]/|X|$
return $(\pi_{\text{best}}, M_{\text{best}}, \text{score})$

Algorithm 11 SortBySampleRelevance($\mathcal{U}, X, \epsilon$)

$\mathcal{U}' \leftarrow \emptyset$
for $(M_s, X_{M_s}) \in \mathcal{U}$ **do**
    $S_t \leftarrow \{s : (s, a, r) \in X\}$
    $S_s \leftarrow \{s : (s, a, r) \in X_{M_s}\}$
    score $= |S_t \cap S_s|/|S_t|$
    if score $> \epsilon$ **then**
        $\mathcal{U}' = \mathcal{U}' \cup (M_s, X_{M_s}, \text{score})$
sort($\mathcal{U}'$) ▷ Sort by score
$\mathcal{U} \leftarrow \{(M_s, X_{M_s}) : (M_s, X_{M_s}, \text{score}) \in \mathcal{U}'\}$

If no solvable source task is selected, the algorithm instead finds the most relevant unsolvable source tasks (Algorithm 11), and attempts to break them down further. We calculate the relevance of a source task by computing the overlap between samples from a source X_{M_s} and the samples of the target X. Specifically, for each task-sample pair $(M_s, X_{M_s}) \in \mathcal{U}$, we compute the fraction of states s in the target samples X that are also present in the source X_{M_s}. If function approximation is used, a distance metric such as that by Ferns et al. [30] can be used to do this. The task-sample pairs in $\mathcal{U}$ are sorted by their relevance, dropping any that have relevance less than ϵ, and are recursively broken down by calling RecurseTaskSelect, which tries to find a sub-source task for the current source task. If no tasks can be solved, the recursion ends.

Assuming the target task is solvable, Algorithm 8 is guaranteed to terminate once β

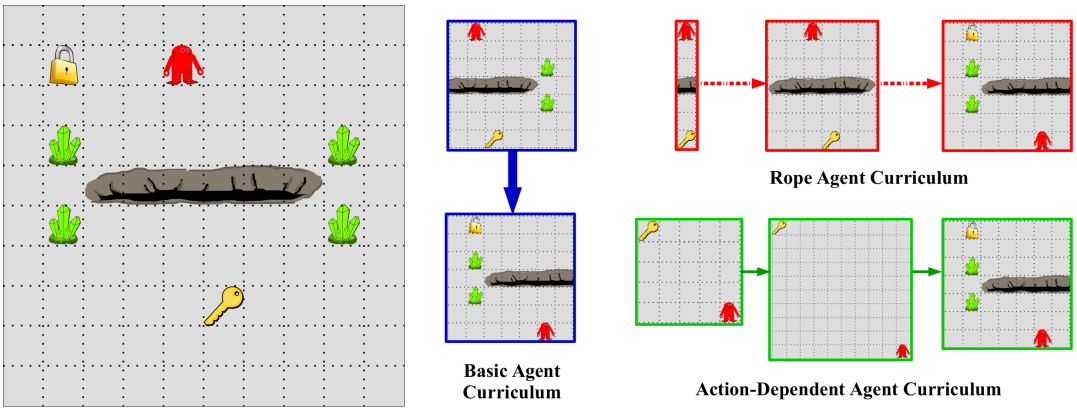


Figure 6.1: (a) Grid world target task (b) Sample curricula generated for each of the agents. Each one ends in the target task.

increases enough to solve the target task directly. In the worst case, no source tasks are useful. If there are m total source tasks, and it takes n iterations of increasing β to learn the target task, then the whole process makes at most $O(mn)$ recursive calls.

6.3 Experiments

We evaluate our curriculum generation algorithm on a grid world domain, inspired by the lights world domain used by Konidaris and Barto [65]. The world consists of a room, which can contain 4 types of objects. *Keys* are items the agent can pick up by moving to them and executing a pickup action. These are used to unlock *locks*. Each lock in a room is dependent on a set of keys. If the agent is holding the right keys, then moving to a lock and executing an unlock action opens the lock. *Pits* are obstacles placed throughout the domain. If the agent moves into a pit, the episode is terminated. Finally, *beacons* are landmarks that are placed on the corners of pits. A sample domain is pictured in Figure 6.1a.

The goal of the learning agent is to traverse the world and unlock all the locks. At each time step, the learning agent can move in one of the four cardinal directions, execute a pickup action, or an unlock action. Moving into a wall causes no motion. Sucessfully picking

up a key gives a reward of +500, and sucessfully unlocking a lock gives a reward of +1000. Falling into a pit terminates the episode with a reward of -200. All other actions receive a constant step penalty of -10.

This domain features a number of skills that must be learned in order to complete a task. For example, navigation, picking up keys, and unlocking locks are all different skills an agent must learn, and each could be learned in a separate task as part of a curriculum. The complexity of the task can also be increased or decreased by adding or removing new objectives and obstacles.

6.3.1 Learning Agent Descriptions

We created multiple reinforcement learning agents that have different representation and action abilities, and used the algorithm proposed to generate curricula for them. We can create agents that vary along the representation dimension by using features that increase or decrease bias. We can change action capabilities by adding obstacles such as pits, and giving an agent a "rope" action that allows it to cross pits.

Using multiple agents allows us to verify that the algorithm works regardless of the implementation of the RL agent used. It also allows us to potentially answer another question: whether different agents can benefit from tailored curricula, just as humans often benefit from individualized curricula, and whether the method proposed facilitates that.

To evaluate these ideas, we created 3 different agents. 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 measures 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, which we refer to as the *noKeys* sensor. For example, the perception vector for the agent in Figure 6.1a is [7.07, 5.10, 6, 6.32, 3.16, 3.16,

4, 2, 4.24, 3.16, 3.61, 2.83, 0, 0, 0, 0, 0], where the first 4 elements are key features for the north, south, east, and west side sensor, followed by the 4 for locks, 4 for beacons, 4 for pits, and the noKey.

The agent used Sarsa(λ) with ϵ-greedy action selection for the learning algorithm $\mathcal{L}$, value function transfer for transfer learning algorithm $\mathcal{T}$, and CMAC tile coding for function approximation (see Chapter 2 for a review). For all our agents, the tile widths were 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. Tiling in this way allows the agent to generalize knowledge about keys and locks learned in source tasks separately. 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: one tile 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. For example, the feature vector for the agent in Figure 6.1a considering the move right action is [6, 4, 3.61, 0, 0], where the values correspond to the key, lock, beacon, pit, and noKey features. The weights in the tilings are shared, so that the same set of weights is used for the state in each of the directions. Sharing weights like this increases the agent's level of generalization.

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. Depending on the task, this action capability can result in a different optimal policy, and thus could benefit from a customized curriculum.

6.3.2 Curriculum Generation and Results

We used the algorithm presented in Section 6.2 to automatically generate curricula for each of the 3 agents. The target task M_t was a 10x10 grid world with 1 lock and 1 key separated by a 6 tile pit, as shown in Figure 6.1a. This task requires agents to learn at least 3 different behaviors: picking up keys, navigating around pits, and unlocking locks.

Each agent was initialized with a uniform random policy, and given an initial learning budget β of 500, which was increased by 500 in each iteration of the loop in Algorithm 8. In order to add a source task, we specified it had to affect the policy by at least $\epsilon = 0.1$. Curriculum generation was terminated when a return $\delta = 700$ was reached.

Tasks were identified as solved using the policy convergence method described in Section 6.2. We applied the TaskSimplification and OptionSubGoals heuristics defined in Chapter 4 to create source tasks. These created source tasks that varied elements such as the size of the domain, the number of pits, or changed the goal of the task to be picking up certain keys. A total of 15 unique tasks were considered for use by the curriculum algorithm, and 9 were used to compose curricula for the different agents.

We evaluated the performance of each agent on the target task using no curriculum, a curriculum tailored for that specific agent, curricula tailored for each of the other two agents, and a random curriculum consisting of 3 randomly selected tasks. Figures 6.2 - 6.4 show the results for the basic agent, action-dependent agent, and rope agent, respectively. The results clearly show that training via the curriculum customized for an agent provides the best benefit. Using a different agent's curriculum was usually suboptimal, and in some cases even hurt performance. Using a random curriculum generally led to performance quite similar to learning from scratch, only delayed. The random tasks added training time without improving learning speed (results are shown for the rope agent. For the other agents, the shape of the random curve was similar to the "no curriculum" curve, but the randomly selected tasks led to horizontal offsets greater than the scale of the graph axes). Examples

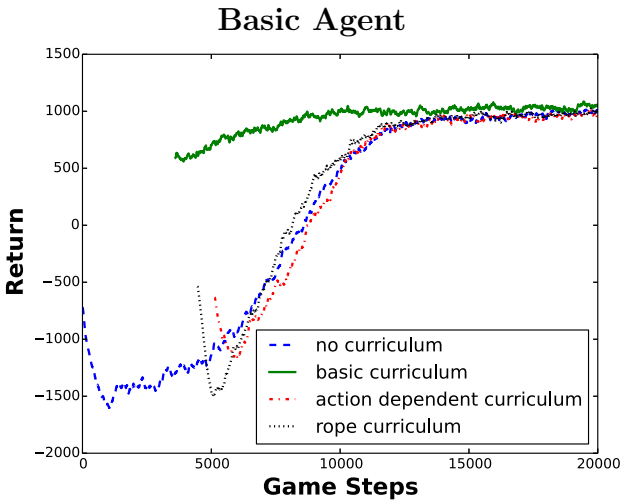


Figure 6.2: Performance on the target task by the basic agent after training using various curricula. Each curve was averaged over 500 runs, and is offset to reflect time spent training in source tasks. The basic curriculum is statistically significantly better than the other curricula until game step 12292, using a 2-tail t-test with $p < 0.05$.

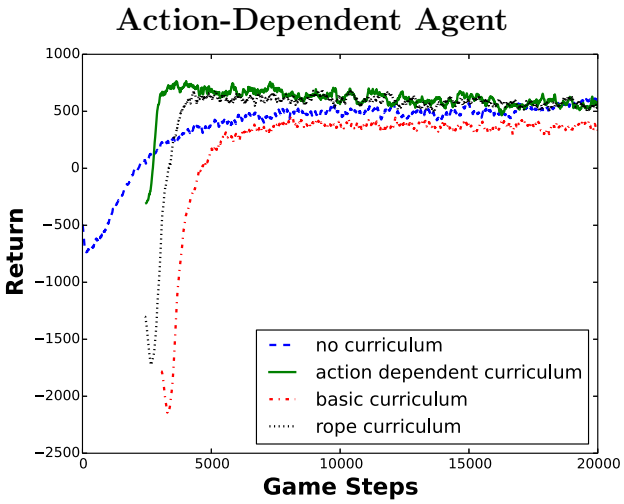


Figure 6.3: Performance on the target task by the action-dependent agent after training using various curricula. Each curve was averaged over 500 runs, and is offset to reflect time spent training in source tasks. The action dependent curriculum is statistically significantly better than the other curricula between game steps 2809 and 4258, using a 2-tail t-test with $p < 0.05$.

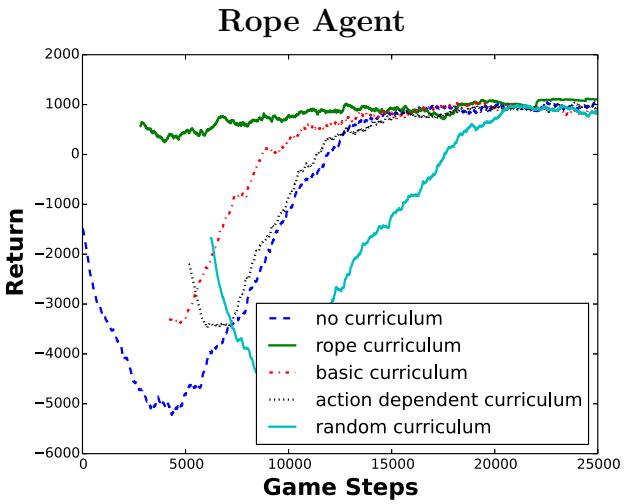


Figure 6.4: Performance on the target task by the rope agent after training using various curricula. Each curve was averaged over 500 runs, and is offset to reflect time spent training in source tasks. The rope curriculum is statistically significantly better than the other curricula until game step 12510, using a 2-tail t-test with $p < 0.05$.

of produced curricula for each agent are shown in Figure 6.1b.

6.4 Summary

In this chapter, I introduced our first approach for automatically sequencing tasks into a curriculum. The main idea is to repeatedly use samples of an agent's experience on the target task to determine which source tasks to train on next. In particular, we use a heuristic which measures how much change learning each source task will have on these target task samples to decide which task to select. The task that elicits the most change is selected. After training on this source task, the agent's policy is updated, and a new set of samples is acquired on the target task based on the agent's updated policy. This process is repeated until a curriculum is formed.

We evaluated our approach in a grid world domain on 3 different types of RL agents that varied in sensing and action capabilities. We were able to show that different agents do benefit from individualized curricula, and that the proposed method is able to learn them.

An individualized curriculum is better than both a baseline of not using a curriculum, and a curriculum designed for the other agents.

The approach designed in this chapter uses a heuristic for identifying which source task would be most useful to learn next. In the next chapter, we instead pose the curriculum generation problem as an MDP, and directly use learning to find a curriculum.

7. Learning-based Approaches for Sequencing

In Chapter 6, I presented the first automated sequencing method of this thesis. This method performs a cost versus benefit computation to determine which task to add next in the curriculum. Part of this computation uses a heuristic to determine the benefit of learning a task – specifically, that a larger change in the policy on a set of target task samples after learning a task implies that it is more beneficial. However, this heuristic relies on one key assumption – that the source tasks generated as possible components of the curriculum do not induce negative transfer. While in some settings, this assumption may hold, we would also like methods that are robust to settings where this assumption does not hold.

In order to deal with this limitation, we need to have additional information about the source and target task MDPs, or acquire additional experience in the target task MDP after transfer to evaluate the direction of transfer. Therefore, in this chapter, I present an alternative approach that uses experience trajectories and *learning* to perform sequencing. This method poses curriculum generation as an interaction between two MDPs. One MDP is a standard MDP for a learning agent (i.e., a student) interacting with a task. The second MDP is a higher level MDP for the curriculum agent (i.e., teacher), which learns to select tasks for the student to train on. A policy over the curriculum agent's MDP (referred to as a curriculum policy) is a mapping from the student agent's current state of knowledge and abilities, to the task it should learn next, to optimize one of the curriculum learning metrics from Chapter 2. A curriculum policy can be learned using any standard RL method. However, the key challenge of this approach is to represent the curriculum agent MDP's

This chapter is based on work that was published in the proceedings of the Autonomous Agents and Multi-agent Systems (AAMAS) conference [80]. It was done in collaboration with Peter Stone, who aided in formalizing the ideas and editing the paper.

state space in a way that facilitates efficient learning. I discuss this and other challenges in this chapter, and show that this approach produces curricula that are at least as good or better than previous methods. This chapter addresses Contribution 4 from Chapter 1 of this thesis.

7.1 Curriculum Generation as an MDP

As in Chapter 6, I first define the type of curriculum our approach will produce, as well as the metric it will optimize. In this chapter, I will again consider task-level sequence curricula which represent curricula as an ordered list of tasks $[M_1, M_2, \ldots M_n]$. In contrast to the previous chapter, the approach I will discuss assumes the set of source tasks is prespecified beforehand, and can be used to optimize for multiple different transfer metrics. I will ground the discussion and perform experimental evaluation using the time to threshold metric. However, I will also discuss how it can be applied for asymptotic performance or jumpstart.

The core idea of our approach is to formulate curriculum generation as an interaction between two agents acting in two different MDPs. One is 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*, and whose goal is to sequence tasks M for the learning agent.

The overall process in a CMDP unfolds as follows. The learning agent starts with some initial state of knowledge – which for now we will assume can be encapsulated by its policy – π_0, which is represented as the initial state s_0 of the CMDP. The curriculum agent then selects an action a_0, where each action corresponds to a different task that can be learned by the learning agent using some learning algorithm. Learning a task transforms the learning agent's state of knowledge to a new policy π_1, represented in the CMDP as the next state s_1, by means of a transfer learning algorithm. It also incurs a cost, which is the amount of time needed by the learning agent to learn the task (when optimizing for time to

threshold). This process repeats until the curriculum agent transitions to a terminal state, which is a state where the policy of the learning agent can achieve a return $G_0 \geq \delta$ on the target task.

We now define this process formally as an MDP. To distinguish the curriculum MDP from task MDPs, we will use the superscript C to refer to elements of the curriculum MDP.

Definition 7.1. A *curriculum MDP (CMDP)* M^C is a 6-tuple $(\mathcal{S}^C, \mathcal{A}^C, p^C, r^C, \Delta s_0^C, \mathcal{S}_f^C)$, where:

State Space The set of states $\mathcal{S}^C$ consist of the set of all policies π the learning agent can represent, in a form that is executable on the target task. For example, the initial state s_0^C could be the uniform random policy. In the time to threshold setting, the terminal states $\mathcal{S}_f^C$ are states whose policies achieve a return of at least some desired performance threshold δ on the target task. In the asymptotic performance or jumpstart setting, the MDP terminates after a prespecified amount of time or number of episodes.

Action Space The set of actions $\mathcal{A}^C$, are the set of tasks a learning agent can train on.

Transition Function The transition function $p^C(s^C, a^C, s'^C)$ gives the probability that s'^C is the learning agent's policy after training on a^C and starting with policy s^C.

Reward Function The reward function $r^C(s^C, a^C, s'^C)$ varies depending on the transfer learning metric being optimized. In the time to threshold setting, $r^C(s^C, a^C, s'^C)$ is the negative of the time (measured e.g., in experience samples or wall clock time) needed to learn task a^C starting from policy s^C. In the asymptotic performance setting, it is 0 unless s'^C is a terminal state, in which case it is the final performance on the target task. Likewise, in the jumpstart setting, $r^C(s^C, a^C, s'^C)$ is 0 except on terminal states, where it is the value of the jumpstart performance on the target task.

A policy π^C on a CMDP specifies which task to train on given a learning agent policy s^C. Executing π^C for a particular learning agent produces a task-level sequence curriculum.

Learning a full policy over a CMDP can be very difficult, due to stochasticity in the learning algorithm (which leads to stochasticity in the CMDP transition function), a very large and continuous state space, and the high cost of taking a CMDP action. In this rest of this chapter, we explore the challenges involved in learning π^{C^*}.

Before doing so, I would like to briefly comment on representing CMDP states and its relation to the CMDP transition function and transfer learning algorithm being used by the learning agent. We used the learning agent's policy as one example of how to represent the CMDP state. However, this representation assumes the underlying transfer learning mechanism is value function or policy transfer. Intuitively, the state space of a CMDP represents different states of knowledge. A transition between states reflects the change in knowledge from training on a task and *transferring/incorporating* the information acquired. In value function transfer, the knowledge learned from a task is represented by the value function of the agent itself. Similarly, in policy transfer, the knowledge learned from a task is encapsulated by the policy. However, for other transfer learning techniques, such as transfer via reward shaping (see Chapter 2), knowledge can be represented in other forms; for instance, as a potential-based shaping reward.

Thus, the CMDP state space and transition function are directly related to the transfer learning algorithm being used. The goal of the agent is to reach a state of knowledge that allows solving the target task in the least amount of time. Therefore, for an agent that uses reward shaping, the CMDP state can be represented as a set of potential functions, derived from the value functions of source tasks already learned. The goal is to find a CMDP state whose sum of potential functions creates a shaping reward that allows learning the target task as fast as possible. In the next section, we describe in detail how to represent CMDP states.

7.2 Representing CMDP State Space

We now detail how to represent the CMDP state to facilitate learning of curriculum policies. Recall that in the standard reinforcement learning setting, the agent perceives its state as a set of raw 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 for use in function approximation. Given these features and a functional form (such as a linear representation), the goal is to learn weights $\boldsymbol{\theta}$ for the value function or policy (e.g., in the linear case: $v_{\boldsymbol{\theta}}(s) = \boldsymbol{\theta} \cdot \boldsymbol{\phi}(s)$). We introduce an analagous process for curriculum design agents acting in CMDPs. We will ground the discussion assuming the learning agent uses value function transfer. However, the idea is easily applied to the reward shaping setting by noting that the potential-based reward, like the value function, can be expressed as a function of state features and weights.

The first question is how to represent the raw state variables s^C of the CMDP state space. The representation chosen must be able to represent *any* policy the underlying learning agent can represent. Assuming the learning agent derives its policy from an action-value function $q_{\boldsymbol{\theta}}(s, a)$, the form of the function – in particular, the way values are calculated from $\boldsymbol{\phi}(s, a)$ and $\boldsymbol{\theta}$ (for example, the architecture of a neural network) – determines the class of policies that can be represented. The functional form of $q_{\boldsymbol{\theta}}(s, a)$ 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. It follows that we can represent the state variables for a particular CMDP state s^C using the instantiated vector of learning agent weights $\boldsymbol{\theta}$.

$$s^C = \boldsymbol{\theta} \tag{7.1}$$

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. While it is possible to directly use the raw $\boldsymbol{\theta}$ as features for function approximation in the CMDP, learning may be more efficient in an alternative basis space. Thus, it may be beneficial to extract *CMDP basis features* $\boldsymbol{\phi}^C(s^C, a^C)$, mirroring what is done in the standard MDP setting. For example, with linear value function approximation, $q^C_{\boldsymbol{\theta}^C}(s^C, a^C) = \boldsymbol{\theta}^C \cdot \boldsymbol{\phi}^C(s^C, a^C)$. The goal then is to learn the weights $\boldsymbol{\theta}^C$ for the CMDP's value function. Any standard RL algorithm can be used to do this.

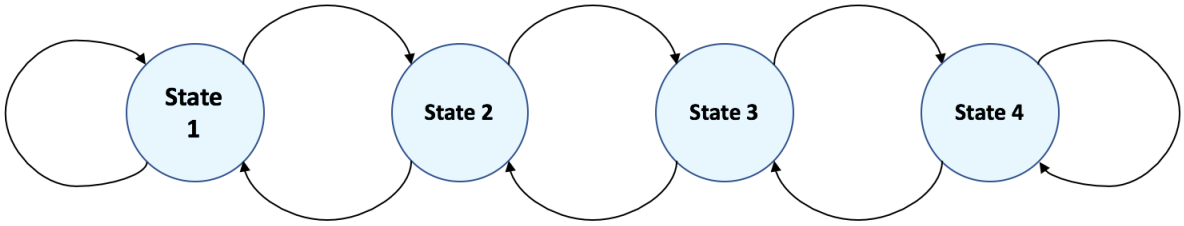
The questions that remain are: (1) how to convert raw CMDP state variables to CMDP basis features, i.e., the form of $\boldsymbol{\phi}^C(s^C, a^C)$; and (2) what kind of functional form to use to represent the function approximation. The best way to resolve these issues will vary by domain. However, the key idea will be to choose representations that allow similar CMDP states to be close in feature space, whereas those that are different to be farther away. A simple example illustrating this idea for a 4 state MDP can be seen in Figure 7.1. In the next 2 subsections, I provide specific examples and guidelines for representations and function approximations that can apply across a broad class of domains.

7.2.1 Discrete State Representations

First we propose one specific way of extracting CMDP state features and performing function approximation, that can be applied when the parameters $\boldsymbol{\theta}$ are tied to specific states, as is common in tabular reinforcement learning.

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 $\boldsymbol{\theta}$:

$$q_{\boldsymbol{\theta}}(s, a) = \boldsymbol{\theta} \cdot \boldsymbol{\phi}(s, a) \tag{7.2}$$



CMDP State 1			
	Left	Right	Policy
State 1	0.3	0.7	→
State 2	0.1	0.9	→
State 3	0.4	0.6	→
State 4	0.0	1.0	→

CMDP State 2			
	Left	Right	Policy
State 1	0.2	0.8	→
State 2	0.2	0.8	→
State 3	0.2	0.8	→
State 4	0.3	0.7	→

CMDP State 3			
	Left	Right	Policy
State 1	0.7	0.3	←
State 2	0.9	0.1	←
State 3	0.6	0.4	←
State 4	0.0	1.0	→

Figure 7.1: A simple 4 state task MDP, and 3 examples of CMDP states over this task. Each CMDP state corresponds to a different policy over the task MDP. Values under the "Left" and "Right" columns are weights (such as q-values or probabilities) for taking those actions in a primitive state in the task MDP, and correspond to $\boldsymbol{\theta}$ from Equation 7.1. CMDP states 1 and 2 have similar policies. Therefore, we want them to be close in the featurized CMDP state space. In contrast, CMDP state 3 has a more different policy, and should be farther away in CMDP state space.

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 $\boldsymbol{\theta}$. Our approach for designing $\boldsymbol{\phi}^C$ is to utilize tile coding 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 process 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. An example of this process for the 4 state MDP from Figure 7.1 is shown in Figure 7.2.

Since action-values can take a large range of values, we suggest normalizing the action-values within each tiling. Thus, each tiling is over the relative preferences of the different actions in a state. The entire CMDP basis state is the concatenation of all of these tiled features. The effect of this approach is that when computing the value of a CMDP state s^C,

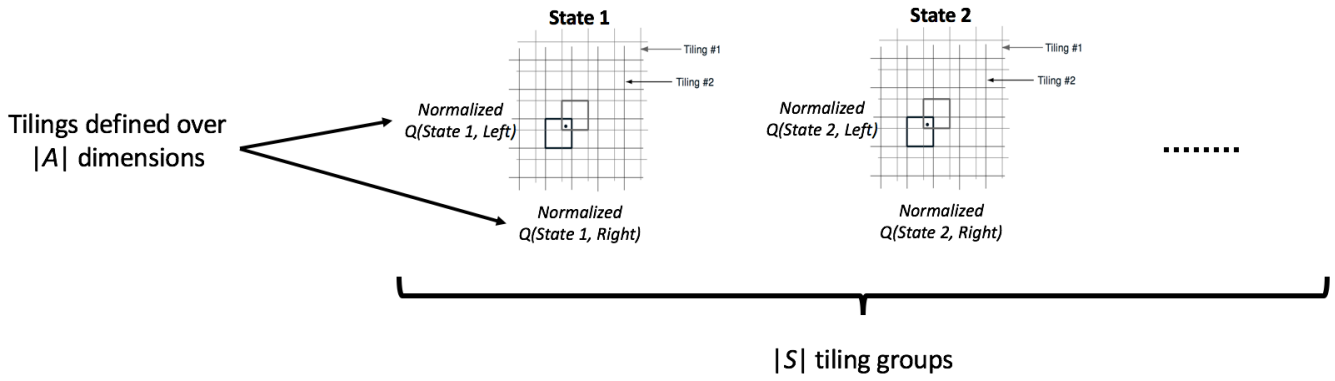


Figure 7.2: An example of how tile coding can be used to create CMDP features for the 4 state MDP from Figure 7.1. In this case, $|\mathcal{S}| = 4$ for the 4 primitive states, and $|\mathcal{A}| = 2$ for the left and right primitive actions. We treat the raw state variables $\boldsymbol{\theta}$ in Figure 7.1 as q-values and normalize them before applying the tilings.

the policy for each primitive state contributes equally towards the total value. Two CMDP states will be "closer" in representation space the more $\boldsymbol{\phi}^C$ activates the same tiles – which will happen if they have similar action preferences for primitive states in their task state spaces.

7.2.2 Continuous State Representations

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.). We offer 2 guidelines that we found useful in defining successful $\boldsymbol{\phi}^C$ representations in our experiments.

The first is that the precise form of $\boldsymbol{\phi}^C$ should be informed by the domain and the structure of the learning agent's function approximation. The discrete case discussed previously is a special case of this setting. In the discrete case, aggregating action-values in a state-by-state basis could be thought of as exploiting the structure and what we know about the parameter vector $\boldsymbol{\theta}$: namely, that it consists of action values that share states.

Depending on the function approximation used by the learning agent, it may be possible to draw similar insights to design $\boldsymbol{\phi}^C$.

The second guideline for creating $\boldsymbol{\phi}^C$ is to capture the *relative* effect of each θ_i on different action preferences. In the discrete case, this process was done by normalizing the action values within each state to create preferences. However, since in general parameters may not be local to a state, the normalization needs to be done directly on the parameter values. In other words, we need to think about how each parameter θ_i affects the policy as a whole over all states, and how each parameter θ_i relates to another. If the parameters $\boldsymbol{\theta}$ are not related, one option would be to create a separate tiling over each parameter, and normalize over all the parameter values. We will demonstrate a specific example of creating $\boldsymbol{\phi}^C$ for the continuous case in Section 7.3.

7.3 Experimental Setup

We evaluated learning curriculum policies for agents on a grid world domain (see Section 6.3) as well as the Ms. Pac-Man domain (see Section 4.3) introduced in previous chapters. These domains were selected because they allow us to compare to previous methods; test our approach using different agent representations, different transfer learning algorithms, and different CMDP representations; and test its scalability to a more complex setting.

I 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 CMDP 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 learn the target task directly. 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 (from Narvekar et al. [83] for the gridworld and from Svetlik et al. [123] 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 $[M_1]$. If the CMDP agent subsequently selects task M_t, the resulting state is $[M_1, M_t]$. Note that this representation is a cruder approximation of the underlying process, as learning 2 different tasks that impart the same knowledge will lead to 2 different states under this representation. In order to deal with the combinatorial explosion of the size of the state space with this naive representation, 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.

Hyperparameters for the learning agents were chosen using previously reported results in the respective domains. Hyperparameters for the CMDP agents were set as described in Sections 7.4.1 and 7.5.2. They were not extensively optimized.

7.4 Gridworld Experiments

In our first set of experiments, our goal is to evaluate the ability of our method to learn curriculum policies for 3 learning agents that have different state and action spaces, but use the same transfer learning algorithm (value function transfer), in a grid world domain (see Figure 6.1 and Section 6.3). This domain was chosen because it allows us to compare the

curriculum generated against our previous curriculum sequencing approach from Chapter 6. In addition, we also evaluate the effect of two different types of representations for the CMDP state. The first CMDP representation is based on the finite state space representation discussed earlier, while the second CMDP representation is created directly from $\boldsymbol{\theta}$ without using an intermediary state-based action-value representation. A description of the domain and learning agents can be found in Section 6.3. In the next subsection, we describe the representations for the CMDP state space and their effects on learning curriculum policies.

7.4.1 CMDP Description

We defined our curriculum MDP as follows:

State space. The start state s_0^C was derived from an untrained, uniformly initialized learning agent. The set of terminal states $\mathcal{S}_f^C$ were all states where the learning agent's policy allowed it to achieve a return of at least 700 on the target task. This performance threshold was the maximum that all the agents could achieve after training to convergence on the target task. Representations used for the CMDP state space are described in the next section.

Action space. Source tasks were created using the TaskSimplification and OptionSubGoals heuristics (see Chapter 4). These heuristics create source tasks by simplifying the domain, for example by reducing the size of the grid or the number of keys, locks, and pits, and by changing the goal of the task to be picking up keys. A total of 10 different tasks were created, and with the target task, these formed the action space $\mathcal{A}^C$ of the CMDP agent. The properties of these source tasks are summarized in Table 7.1.

Transition function. The (unknown) transition function is stochastic, describing how learning a task changes a learning agent's policy.

Reward function. The environment returns a reward $r^C(s^C, a^C, s'^C)$ as the negative of the time needed to learn task a^C from state s^C. A task is considered learned once the

Task Num	Grid Size	Num Keys	Num Locks	Pit Present	Rope Required
1	5x5	1	0	No	No
2	10x10	1	0	No	No
3	5x5	0	1	No	No
4	10x10	0	1	No	No
5	7x1	1	0	Yes	Yes
6	7x6	1	0	Yes	Yes
7	7x1	0	1	Yes	Yes
8	7x6	0	1	Yes	Yes
9	7x7	1	0	Yes	No
10	7x7	0	1	Yes	No
Target	10x10	1	1	Yes	No

Table 7.1: Properties of tasks in the gridworld experiments. "Rope required" indicates tasks where a pit blocks direct paths from the agent to the goal, necessitating a rope action. When a lock is not present, the episode terminates when all keys are picked up.

policy ceases to change for 10 episodes. Time is measured using game steps.

Learning on the CMDP was done using Sarsa(λ) with $\epsilon = 0.001$, $\lambda = 0.9$, and $\alpha = 0.1$.

7.4.2 CMDP State Space Representations

One of the main challenges addressed in this research is identifying a representation for the CMDP state space that is both generalizable and compact enough to enable efficient learning of a curriculum for a range of agents. To this end, we instantiated and evaluated two forms for $\boldsymbol{\phi}^C$.

Recall that the learning agents use tile coding with linear function approximation. Here, $\boldsymbol{\phi}$ is a feature vector that indicates which tiles have been activated for state s and action a, and $\boldsymbol{\theta}$ are the corresponding weights in each tile. These weights $\boldsymbol{\theta}$ form the raw CMDP state variables s^C. We discuss two different ways to construct $\boldsymbol{\phi}^C(\boldsymbol{\theta})$, which will convert the raw state variables into a CMDP basis feature space suitable for learning.

Finite State Representation

The learning agents use Sarsa(λ) with an *egocentric* feature space, which consists of relative distances to objects of interest from the current position of the agent. Thus, the parameters $\boldsymbol{\theta}$ learned are not actually action-values for each grid world cell, but are weights for these

egocentric features. However, since the underlying domain has a fixed number of grid cells, we can simulate the finite state representation case by "moving" the learning agent to each of the grid cells in the target task and computing action values. Let this new parameter of weights be $\boldsymbol{\theta'}$. We can now utilize the procedure described in Section 7.2 to create a CMDP feature space $\boldsymbol{\phi}^C(\boldsymbol{\theta'})$.

Continuous State Representation

The above representation is only well-defined in environments with a discrete underlying state space. We therefore also explore a CMDP representation that can apply in continuous domains by creating $\boldsymbol{\phi}^C$ directly from $\boldsymbol{\theta}$ without using an intermediary state-based action-value representation. Recall that the CMDP state variables $s^C = \boldsymbol{\theta}$ are the weights associated with all the tiles. Each of these tiles is part of a tiling group. For example, the basic and rope agents had 2 tiling groups over different subsets of its sensor percepts, while the action-dependent agent had 3 tiling groups. All tiles in a tiling group are related to each other. Thus there is an inherent structure to the parameters in the tiles.

However, forming a $\boldsymbol{\phi}^C$ tiling group over the weights of all the tiles associated with a $\boldsymbol{\phi}$ tiling would not generalize well, because it would require nearly identical action-preferences in every state to activate common tiles. Therefore, we created a separate tile group for each θ_i. Since the weights $\boldsymbol{\theta}$ within each learning agent's tilings $\boldsymbol{\phi}$ are still correlated, we normalized the weights associated within each $\boldsymbol{\phi}$ tile group.

7.4.3 Results and Discussion

We learned curriculum policies for all 3 learning agents using both the finite and continuous state representations for the CMDP state space. The target task M_t is shown in Figure 6.1. The corresponding CMDP learning curves are shown in Figures 7.3 - 7.5. The results show that each agent successfully learned curriculum policies using both CMDP representations

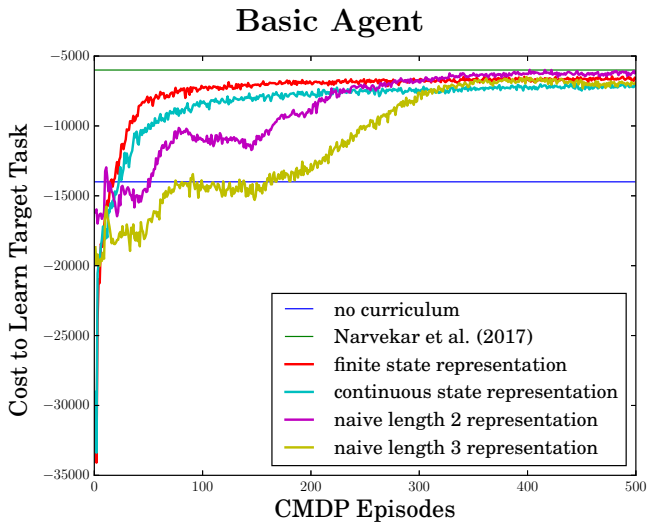


Figure 7.3: CMDP learning curves for the basic agent using different curriculum design approaches and CMDP state space representations. The y-axis represents the cost (i.e., negative of the time needed) to reach a performance of 700 on the target task, following the curriculum policy at episode X. All curves are averaged over 500 runs. Each curriculum method was statistically significantly better than no curriculum using a 2 tail t-test with $p < 0.05$.

that were better than learning without a curriculum, and comparable to the curricula generated by previous work [83]. However, unlike this previous work, our approach does not require additional prior information about source tasks (such as task descriptors). In addition, the results show that our approach is robust to different predefined agent and CMDP representations.

7.5 Ms. Pac-Man Experiments

In the previous section, we demonstrated that CMDPs can be learned for agents with different actions and/or state representations. Another relevant way in which agents can differ is the algorithm by which they transfer knowledge from a source to a target task. Thus, in this section, we evaluate the robustness of our approach to different underlying transfer learning methods, while simultaneously evaluating the scalability to a significantly more complex

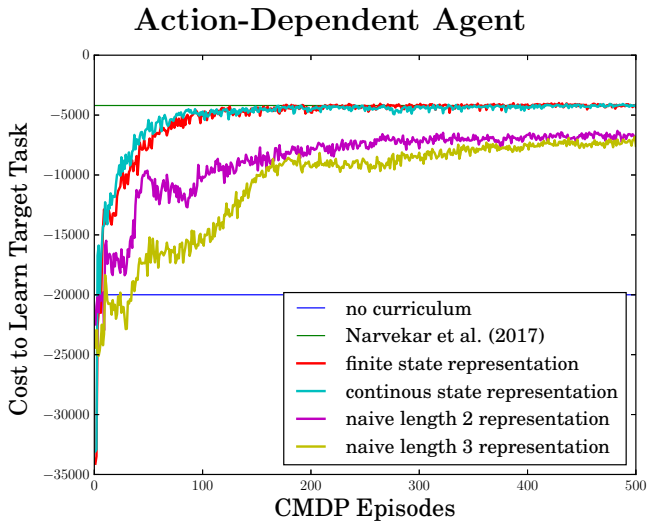


Figure 7.4: CMDP learning curves for the action-dependent agent using different curriculum design approaches and CMDP state space representations. The y-axis represents the cost (i.e., negative of the time needed) to reach a performance of 700 on the target task, following the curriculum policy at episode X. All curves are averaged over 500 runs. Each curriculum method was statistically significantly better than no curriculum using a 2 tail t-test with $p < 0.05$.

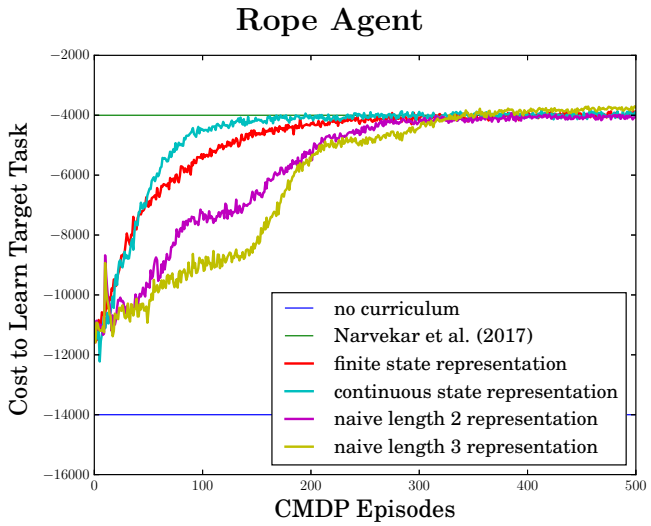


Figure 7.5: CMDP learning curves for the rope agent using different curriculum design approaches and CMDP state space representations. The y-axis represents the cost (i.e., negative of the time needed) to reach a performance of 700 on the target task, following the curriculum policy at episode X. All curves are averaged over 500 runs. Each curriculum method was statistically significantly better than no curriculum using a 2 tail t-test with $p < 0.05$.

Ms. Pac-Man domain. In particular, we examine the case when the learning agent stays the same, but uses 2 different types of transfer learning methods: value function transfer and reward shaping. The change in transfer algorithm affects both the CMDP state space representation, and the CMDP transition function, which we will describe in the following subsections. A description of the domain can be found in Section 4.3.

7.5.1 Learning Agent Description

We created a Ms. Pac-Man learning agent using the low-asymptote feature set described in Svetlik et al. [123], Taylor et al. [130]. The state space of the agent is represented by a set of 24 action-dependent egocentric features. These are divided into 4 sets of features for pills, power pills, ghosts, and edible ghosts, that count the fraction of each object type in a direction up to 6 different "depths." The depth refers to junctions, i.e., locations in the maze where there are more than 2 possible actions.[8] For example, the ghost feature for depth 1 would return the fraction of ghosts there are along one direction until the first junction. The pill feature for depth 2 would return the fraction of pills present up to two junctions away, and so on. These features were used to learn a linear value function approximator.

The agent was trained using Sarsa(λ), with $\epsilon = 0.05$, $\alpha = 0.001$, $\gamma = 0.999$, and $\lambda = 0.9$. See the code by Svetlik et al. [123] for implementation details.

7.5.2 CMDP Description

We defined our curriculum MDP as follows:

State space. As before, the start state s_0^C was an untrained, randomly initialized learning agent. The set of terminal states $\mathcal{S}_f^C$ were all states where the learning agent could achieve a return of at least 2000 on the target task.

[8] Two possible actions in a state means the agent is in a corridor, whereas a "T" and "+" junction has 3 and 4 actions respectively.

Task Num	Num Junctions	Num Ghosts	Num Pills	Num Power Pills
1	2	0	53	1
2	2	1	65	2
3	40	2	234	4
4	36	4	240	4
5	8	0	179	4
6	8	2	179	4
7	8	4	179	4
8	13	2	209	4
9	13	4	209	4
10	13	0	209	4
11	24	0	231	4
12	24	2	231	4
13	24	4	231	4
14	24	4	231	4
Target	36	4	240	4

Table 7.2: Properties of source tasks in the Ms. Pac-Man experiments. "Num Junctions" indicates how many maze positions had 3 or more direction actions possible. Note that some tasks have similar properties; however, the layout of the maps in these tasks differed. See the code release from Svetlik et al. [123] for more details.

Action space. We used the same 15 tasks used in the code release of Svetlik et al. [123] to form the action space $\mathcal{A}^C$. These tasks were formed by varying the type of maze, as well as the number of pills, ghosts, and power pills. Their properties are summarized in Table 7.2.

Transition function. As before, the (unknown) transition function is stochastic, describing how Ms. Pac-Man's value function or set of shaping potentials changes as a result of learning a task.

Reward function. We measure the cost of learning a task in terms of the number of game steps needed. Following the experimental setup of Svetlik et al. [123], a task is considered learned when at least 35% of the maximum reward possible for that task can be achieved. The maximum reward for a task is calculated analytically by summing the points accrued for eating all the pills, and all the edible ghosts for each power pill.

Learning on the CMDP was done using Sarsa(λ) with $\epsilon = 0.001$, $\lambda = 0.9$, and $\alpha = 0.05$.

7.5.3 CMDP State Space Representations

We consider 2 different CMDP state space represenations that result from the use of 2 different transfer learning algorithms. In the value function transfer 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 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 Svetlik et al. [123]). 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.

7.5.4 Results and Discussion

Figure 7.6 shows CMDP learning curves for Ms. Pac-Man using value function transfer and Figure 7.7 shows the curves using transfer with reward shaping. The results again clearly demonstrate that curriculum policies can be learned, and that such policies are more useful than training directly on the target task. They also show that the approach is adjustable to different types of transfer learning algorithms. In addition, we compared the reward shaping approach with that of Svetlik et al. [123], who also use reward shaping for transfer in their curriculum algorithm, and found that a much better curriculum is possible in this more complex domain.[9]

Finally, we also study the effect of the hyperparameter that controls when to finish training on a source task. For the previous two experiments in Ms. Pac-Man, training on a source was stopped after 35% of the max possible return in the task was achieved, to

[9]Our results are based on a reproduction of their experiments using their publicly released code. Interestingly, we also get slightly better results for their method than they report in their paper. We measure cost in episodes for this experiment only to facilitate comparison to their work.

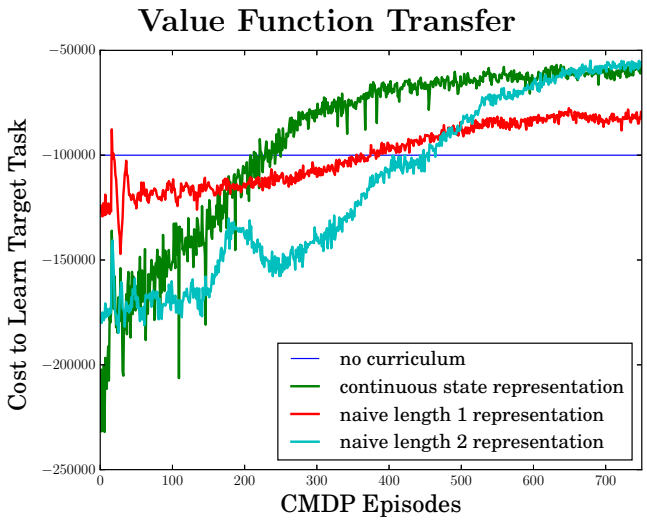


Figure 7.6: CMDP learning curves on the Ms. Pac-Man target task, using value function transfer. All curves are averaged over 500 runs and cost is measured in game steps. Each curriculum method was statistically significantly better than no curriculum at convergence. These were tested using a 2-tail t-test with $p < 0.05$.

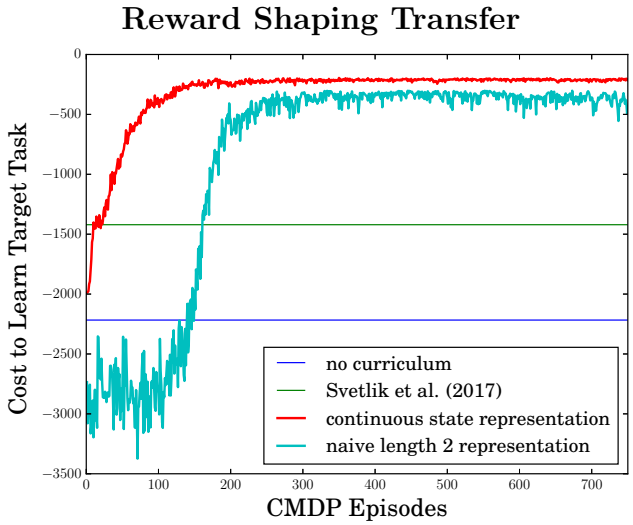


Figure 7.7: CMDP learning curves on the Ms. Pac-Man target task, using transfer with reward shaping. All curves are averaged over 500 runs, and cost is measured in episodes. Each curriculum method was statistically significantly better than no curriculum at convergence. In addition, the CMDP-based approaches were statistically better than Svetlik et al. [123]. These were tested using a 2-tail t-test with $p < 0.05$.

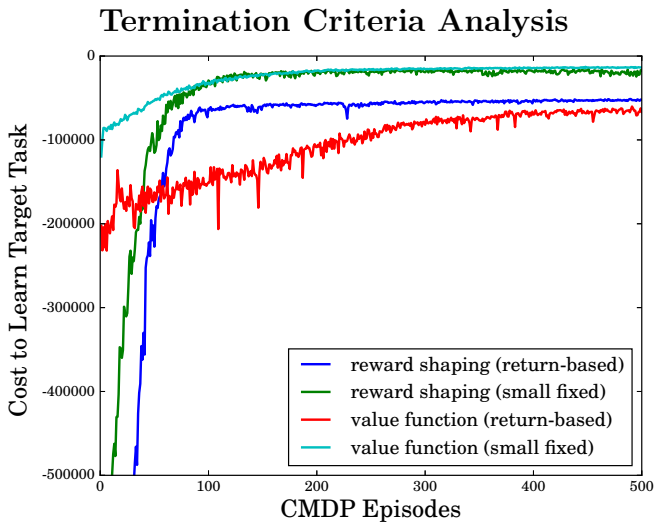


Figure 7.8: A CMDP learning curve comparison between the continuous representations for value function and reward shaping transfer, using different criteria to determine when to stop training on source tasks. All curves are averaged over 500 runs and cost is measured in game steps. The "small fixed approaches were statistically better than their corresponding "return-based methods at convergence. These were tested using a 2-tail t-test with $p < 0.05$.

replicate the experimental conditions of Svetlik et al. [123]. Since their approach precomputes a curriculum and does not model the state of the learning agent's progress, this termination condition must be carefully chosen to ensure something can be learned in each source task. In contrast, with our approach, we can train on source tasks for an arbitrarily small amount of time, as the curriculum policy can learn to reselect a task if additional experience in that task is required.

In Figure 7.8, we reproduce the continuous state representation CMDP learning curves using value function transfer from Figure 7.6 and reward shaping from Figure 7.7. These are denoted in the figure by "(return-based)", and train on sources until 35% of the max return is achieved. We compare them against an approach that is identical to "(return-based)" approaches, but that trains for 5 episodes on a task at a time. These CMDP learning curves are denoted with "(small fixed)." The results show that agents do not need to train for a long time or to convergence on source tasks, and that our approach can adapt to this

hyperparameter setting.

7.6 Summary

In Chapter 6, I introduced our first method for automatically sequencing tasks into a curriculum, that used a heuristic to guide the selection of tasks. In this chapter, I introduced a second method for automatically sequencing tasks into a curriculum that instead relies on learning. The approach formulates curriculum generation as an interaction between two MDPs: one for the learning agent (i.e., a student) which learns tasks, and one for the curriculum agent (i.e., a teacher) that sequences tasks for the learning agent. A policy over the curriculum agent's MDP (i.e., a CMDP) is a mapping from the state of knowledge of an agent to the task it should learn next, to optimize some curriculum learning metric (such as time to threshold or asymptotic performance from Chapter 2). A key challenge in learning curriculum policies is representing the CMDP state. I discussed how this can be done for both discrete and continuous domains.

We evaluated this method on both a grid world domain, and a large, discrete Ms. Pac-Man domain. The results show that the approach is robust to multiple learning agent types, multiple transfer learning algorithms, and different CMDP representations. Learned curricula were also as good as or better than previous CL approaches on the same domain. We also showed an additional benefit of this method, which is that it can implicitly learn how long to spend on each task in the curriculum, as opposed to learning each task to convergence.

A key limitation of this approach is that learning a full curriculum policy can take significantly more experience data than simply learning the target task from scratch. In the next chapter (Chapter 8), I show how this cost can be amortized by learning a curriculum policy that can adapt to multiple different target tasks.

8. Generalizing Curricula

Over the years, many techniques have been designed to both manually and automatically design curricula for RL agents. In Chapters 6 and 7, I presented two such algorithms for automatic sequencing. However, these methods and – to the best of my knowledge – all existing methods have one key limitation: the curriculum must be regenerated from scratch for each new agent or task encountered. In many cases, this generation process can be very expensive. However, there is structure that can be exploited between tasks and agents, such that knowledge gained developing a curriculum for one task should be able to be reused to speed up creating a curriculum for a new task. Just as curricula designed for humans are used to teach many different students, and can easily be adapted to teach people how to solve similar tasks, we would like curricula designed for artificial autonomous agents to have similar versatility.

This chapter thus considers the problem of *curriculum generalization*: how can knowledge gained about designing a curriculum for one task be generalized to speed up learning of a curriculum for a similar, but novel, unseen task? In other words, how can we transfer or adapt a curriculum learned for one task to a new target task?

This chapter builds on the representation of a curriculum as a policy as described in Chapter 7, which maps from the state of knowledge of an RL agent to the task it should learn next. The primary contribution is to show that by combining curriculum policies with universal value functions, where the task is encoded as the goal, we can learn a curriculum policy that can generalize to produce curricula for new unseen tasks. This combination allows

This chapter is based on work that was presented at the 4th Lifelong Learning Workshop at the International Conference on Machine Learning [81]. It was done in collaboration with Peter Stone, who aided in formalizing the ideas and editing the paper.

us to essentially perform "zero-shot" curriculum learning, where a curriculum is generated for a novel target task based on experience generating curricula for similar tasks. This chapter addresses Contribution 5 from Chapter 1 of this thesis.

8.1 Curriculum Generalization

Our main idea is to learn a universal value function over a curriculum MDP so that we can generalize over CMDP states and goals.

In standard reinforcement learning, the value function $v_\pi(s)$ estimates the return of a policy from a given state s. In deep reinforcement learning, the value function is typically represented by a deep neural network, and exploits the structure in the state space to learn values for observed states and to generalize to unseen states. In goal-oriented tasks, where the environment transition dynamics stay the same but the goal state may differ, much of the structure of a value function can also be shared across goals. Thus, the idea behind Universal Value Functions (UVFA) [105] is to create a value function $v_\pi(s, g)$ that generalize over both states s and goals g by creating an embedding over state features and goal features:

$$v_\pi(s, g) = \mathbb{E}^\pi \left[\sum_{t=0}^{\infty} r_g(s_t, a, s_{t+1}) \middle| s_0 = s \right] \tag{8.1}$$

A universal value function can also be learned over a curriculum MDP. The key questions are how to represent CMDP states, goals, and the architecture for the UVFA.

8.1.1 CMDP States and Goals

Recall from Chapter 7 that a CMDP state parametrically represents the agent's knowledge. One way to represent the agent's state of knowledge is by its policy π_θ. In particular, the class of policies the agent can represent is determined by the structure of the function approximator used, and the instantiation of weights $\boldsymbol{\theta}$ determines the exact policy in this

class. Thus, when access to the internal representation of the agent is available, we can represent the agent's raw state of knowledge in the CMDP state $\mathcal{S}^C$ using the vector of weights of the student agent's value function or policy $\boldsymbol{\theta}$. We can then use tile coding with linear function approximation as in Chapter 7 or some other feature extractor and approximator (in this chapter we will use neural networks) to perform learning.

A goal in the universal value framework is represented as a single state: $g \in \mathcal{S}$. In the CMDP setting, we instead propose to represent goals g^C as target tasks that the agent could be trained on, with one goal for each target task. A key question is how to represent tasks. In this work, we restrict our attention to goal-based navigational tasks, which are defined by a starting position and an ending position. This assumption allows us to easily create a parameterized representation of the task by using the concatenated vector of coordinates corresponding to the starting and ending states. As an example, consider the gridworld environment in Figure 8.1a, using a coordinate system where the origin $(0, 0)$ is at the bottom left tile. The agent's (red triangle) starting position is $(1, 10)$ and the end position (green circle) is $(9, 3)$. Thus, we would represent this task parametrically as $[1, 10, 9, 3]$. An important direction for future work is to extend these ideas to non-goal-based tasks, such as those described by language or vision commands [18].

8.1.2 Architecture

Given a representation for both the CMDP state and goal, we use a two-stream neural network architecture as used by Schaul et al. [105] to learn a universal value function over the CMDP. A two-stream architecture assumes the problem can be factorized into two components. In our case, one component is $\phi : \mathcal{S}^C \mapsto \mathbb{R}^n$, which creates an embedding for CMDP states. The second is $\psi : \mathcal{G}^C \mapsto \mathbb{R}^n$, which creates an embedding for CMDP goals. The two streams are combined using an output function $h : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}^m$. In our case, the mappings ϕ and ψ are represented by multi-layer perceptrons, and the output function is

the Hadamard product. See Figure 8.3 for the architecture we use in the experiments.

A policy extracted from this value function is then able to suggest a task to the student based both on what the student knows, and the task it needs to learn. Given enough experience on a set of "training" target tasks, our experiments will show that learning such a universal value function allows the curriculum policy to generalize and produce curricula for unseen "test" target tasks.

8.2 Gridworld Navigation Domain

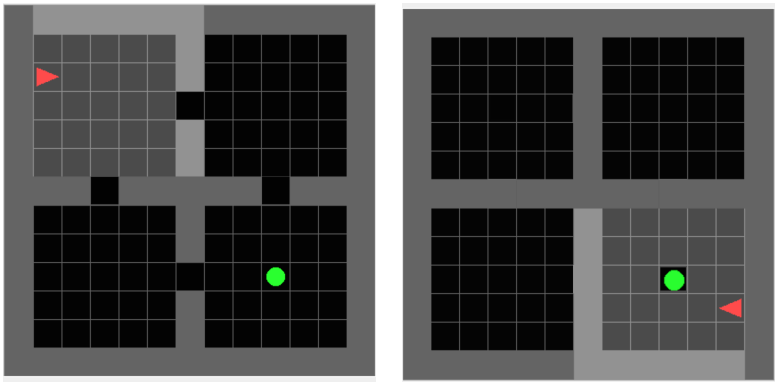
We evaluated curriculum transfer on navigation tasks in a static gridworld environment. Our goal was to train a CMDP teacher agent to learn a curriculum policy on a subset of tasks, and show that it can produce curricula for a student on novel unseen target tasks.

The gridworld environment considers goal-oriented navigation tasks in a standard 4-room grid world. The environment consists of 4 connected rooms, where each room is 5x5 in size and connected at one cell to each adjacent room. A navigation task is defined by a pair of (x, y) coordinates for the starting position and goal position. There are 100 possible starting and ending positions. We ignore tasks that start and end at the same position, thus there are 9900 possible different target tasks. See Figure 8.1 for examples of tasks.

Our student agent has a tabular representation for the state space, and learns using Sarsa(λ) with exploration $\epsilon = 0.1$, learning rate $\alpha = 0.1$, discount factor $\gamma = 1.0$, and eligibility trace decay rate $\lambda = 0.7$. We use value function transfer to transfer information between tasks in the curriculum.

8.3 Teacher (CMDP) Agent Description

In this section, I describe the representation, architecture, and learning parameters for the teacher CMDP agent.



(a) (b)

Figure 8.1: Examples of tasks in the gridworld environment. The red arrow is the agent, and the green circle is the goal. (a) An example of a target task. (b) An example of a dynamic source task for the target task in (a).

State and Goal Space

The CMDP needs to learn to generalize over both the agent's knowledge and task space. Conceptually, the agent's current policy – its function for selecting actions in each state, which we assume to be known to the teacher – is its state of knowledge. As done in Chapter 7, we represent the agent's knowledge using the vector of weights associated with the student's q-function table. In this chapter, we limit ourselves to goal-oriented navigational tasks. Thus, tasks can be represented using the pair of (x, y) coordinates associated with the starting and goal states.

Action Space

We create nine different source tasks. Eight of these tasks are static tasks that don't change based on the target task. These tasks initialize the agent in one of the 4 rooms, and terminate when the agent moves into one of the adjacent two rooms. There is one such task for each room and adjacent room pair. In addition, all corridors between rooms are blocked except for the one required to complete the task. The 8 source tasks can be seen in Figure 8.2.

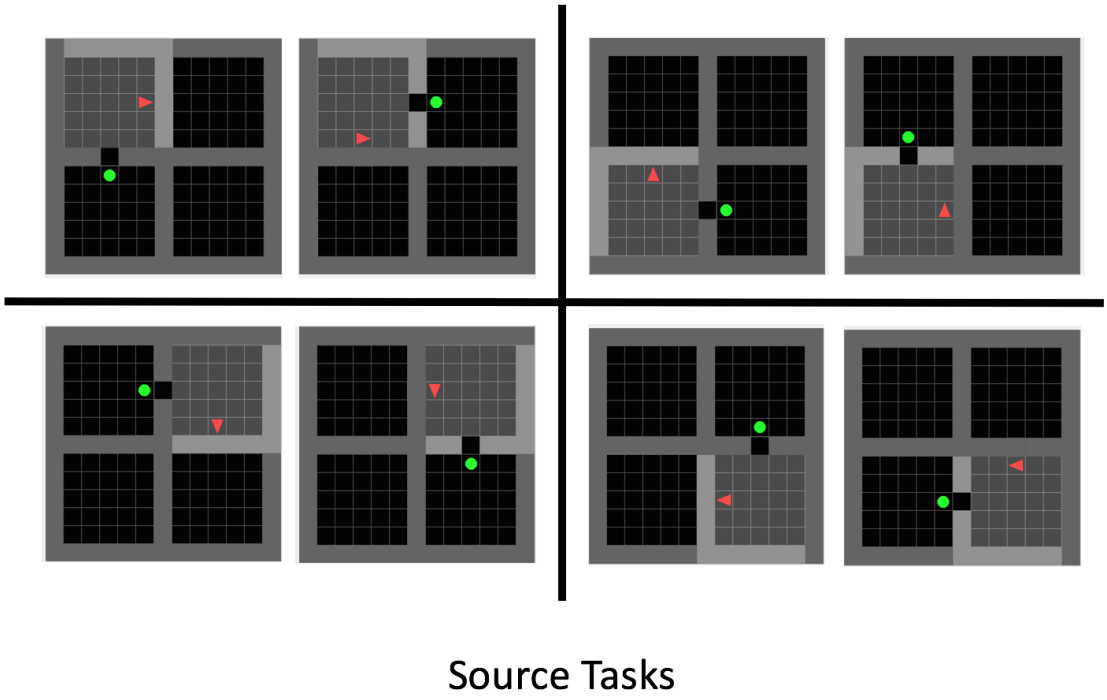


Figure 8.2: The 8 static source tasks, that teach an agent to navigate to an adjacent room. They are shown grouped by the agent's room for clarity, but each task is independent.

The ninth source task is a dynamic source task that changes based on the current target task. This task initializes the agent in the same room as the goal of the target task, and sets the goal tile to be the same as the target task. As with the static sources, all corridors to other rooms are blocked off. An example of a target task and its corresponding dynamic source task can be seen in Figure 8.1. These sources, together with the target task, form the action space of the CMDP. Note that these tasks were intentionally designed to give rise to a natural and interpretable curriculum for each target task: use the static sources to navigate to the goal room, and follow with the dynamic source task to complete the path.

Reward Function

When a task is selected, it is trained on until convergence. We consider a task converged when the steps taken to reach the goal averaged over the last 5 episodes is less than the

Manhattan distance between the start $\mathbf{s}$ and goal positions $\mathbf{g}$ plus a slack term δ:

$$\text{converged} := \big(\text{steps taken} < ||\mathbf{s} - \mathbf{g}||_1 + \delta\big) \tag{8.2}$$

We set δ to 0 when the start and goal positions are in the same room, 5 when they are in adjacent rooms, and 10 when the rooms are diagonally across. The slack term is introduced as a simple way of accounting for the extra steps needed to navigate around walls. The cost of learning a task is the number of steps needed to learn to convergence. Therefore, the reward given at each CMDP step is the negative of the steps taken to converge on the selected task.

Architecture and Learning Parameters

We use DQN [77] to learn the CMDP. The neural network uses a two-stream architecture, where the features relating to the task/goal space pass through a single hidden layer, while the agent knowledge features pass through three hidden layers. Each hidden layer has 128 units followed by a tanh activation function. The two streams are subsequently merged via element-wise multiplication, and pass through a final hidden layer to produce action-values. A diagram of this network can be seen in Figure 8.3. The learning rate is 5e-4, the replay buffer size is 5000, the batch size is 64, the exploration fraction is 0.05, and the target network is updated every 50 steps. To speed up training, we also capped the number of CMDP actions the teacher agent could take at 5, and trained on the target task thereafter. We arrived at these parameters after informal experimentation with a handful of settings, but they were not extensively optimized.

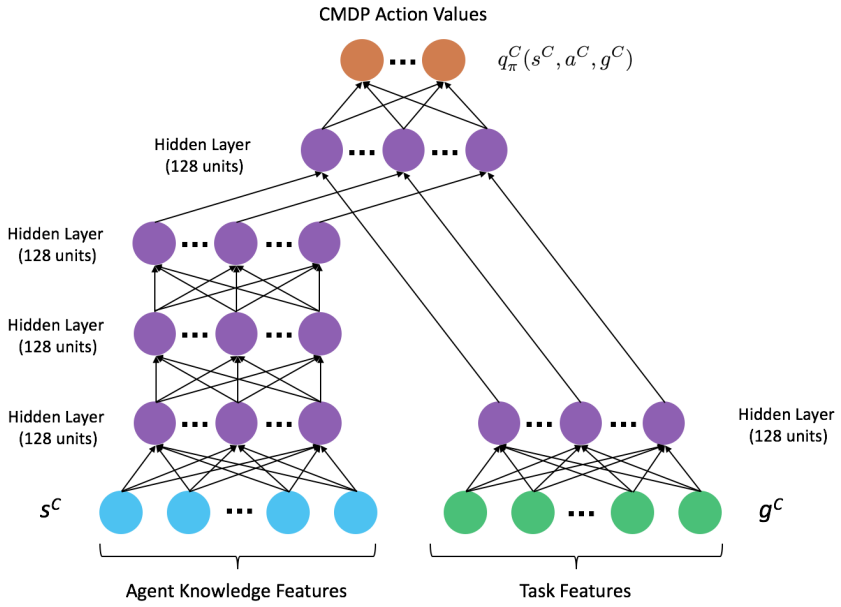


Figure 8.3: The two-stream network architecture used for the teacher CMDP agent. The agent knowledge features s^C are the weights $\boldsymbol{\theta}$ of the student agent's action-value function. The task features is the length 4 vector corresponding to the start and end coordinates of the task as described in Section 8.1.

8.4 Experimental Results

We consider two types of generalization that may be possible in navigational task CMDPs: interpolation and extrapolation. In the interpolation case, we randomly shuffle all the 9900 possible target tasks, and present them to the CMDP agent one by one. Each CMDP episode takes place on a new target task. This situation is similar to the lifelong learning setting, where each task encountered is new, though there may be similarities to tasks seen previously. The results are shown in Figure 8.4. As the CMDP learns and the coverage of tasks in the 4 rooms increase, the curricula produced gradually improve, until they pass the baseline of training on the target task after having seen just 300 of the 9900 possible target tasks. Thus, the results show that the curriculum policy learned jointly over state of knowledge and task representations is able to interpolate and produce curricula for novel unseen tasks. Furthermore, this process requires training on only a small fraction of the

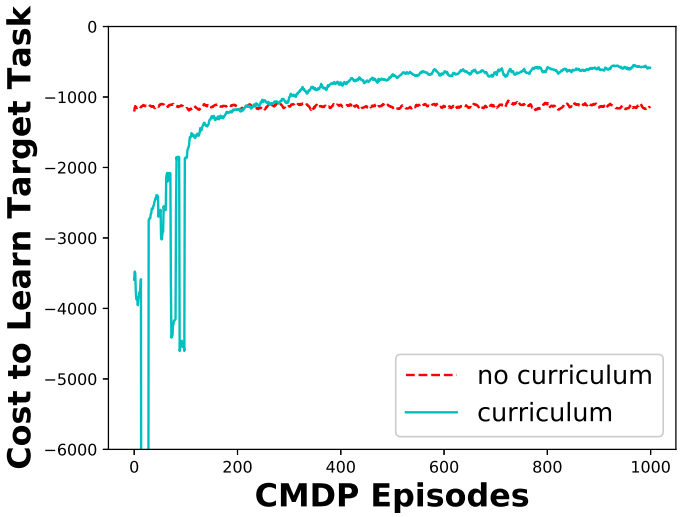


Figure 8.4: CMDP learning curves for the interpolation experiments. The x-axis represents CMDP episodes, where each episode is an entire run of a curriculum. The y-axis is the cost of that curriculum in game steps. The curriculum curve converges to a cost that is statistically signficantly better than the no curriculum curve, using a 2-tail t-test with $p < 0.05$.

total possible tasks.

In the interpolation case, while each CMDP episode presented a new task, after a certain number of episodes, very similar tasks had already been seen. In particular, by the end of training, all possible useful combinations of source tasks were already seen, and each novel target task could have a curriculum designed for it using experience from this same set of source tasks. In the next experiment – the extrapolation case – we explicitly split the set of target tasks into a training set and a test set. The test set contains all tasks that start in the top left room, and end in the bottom right room, while the training set contains tasks with all other possible start and end goal pairs. See Figure 8.5 for examples of tasks in the training and test sets. The extrapolation case is more challenging, because in the previous interpolation setting, generalization was expected because goals were represented with similar features and the set of training tasks covered pairings from all the different rooms. However, in this case, the curriculum of navigating from the top left room to the bottom right room has not been seen before. We train on tasks in the training set for the

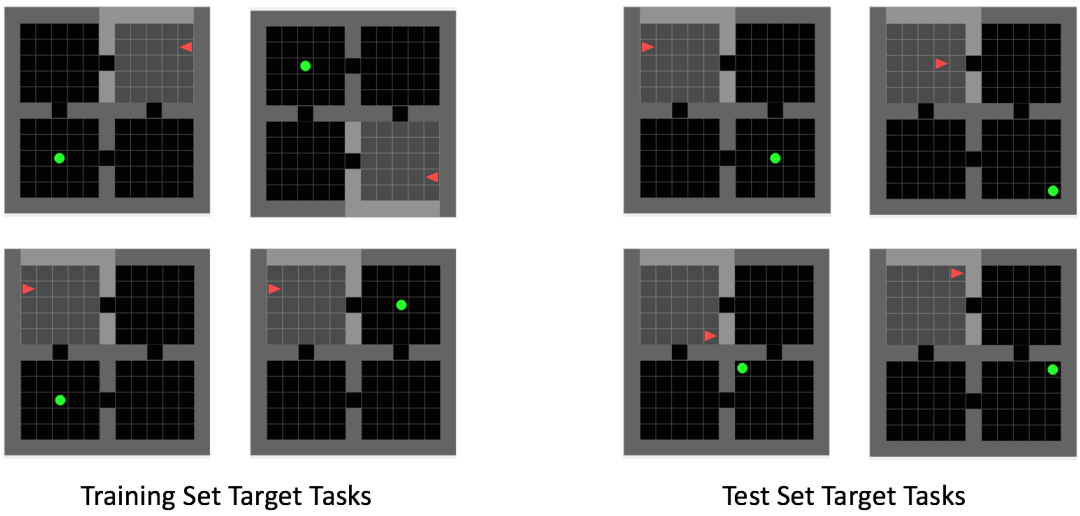


Figure 8.5: Examples of target tasks in the training and test sets for the extrapolation experiments.

first 200 CMDP episodes, and subsequently evaluate on the test set. The results are shown in Figure 8.6, with the curriculum graph offset to reflect time spent training on the training set. The results again show that curricula learned on one set of tasks can transfer to produce curricula for new unseen tasks. In addition, the extrapolation experiment shows that this generalization is possible to target tasks that require an entirely new curriculum. Examples of curricula seen in the training and test sets can be seen in Figure 8.7.

8.5 Summary

Most existing work on automated curriculum learning has relied on heuristics, or has limited the types of source tasks that can be used in a curriculum, because learning a full curriculum directly from experience can be computationally expensive. One way this expense can be amortized is by learning curricula that can generalize to new agents or target tasks. In this chapter, I consider one of these cases, and show how curriculum policies can be combined with universal value functions to generalize curricula to novel unseen navigational tasks. A

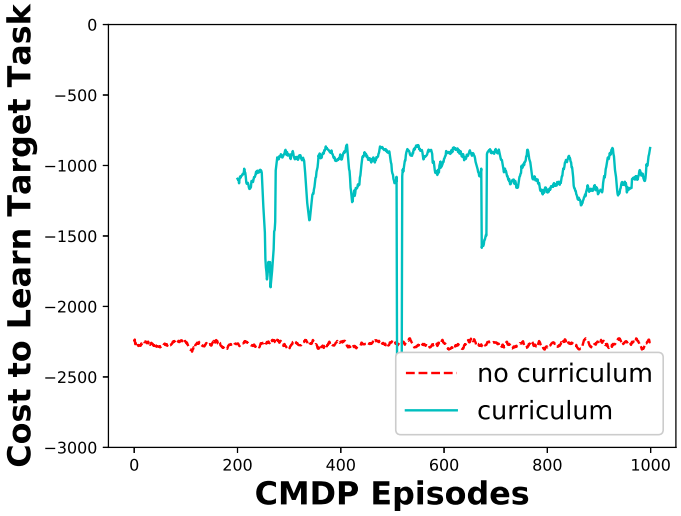
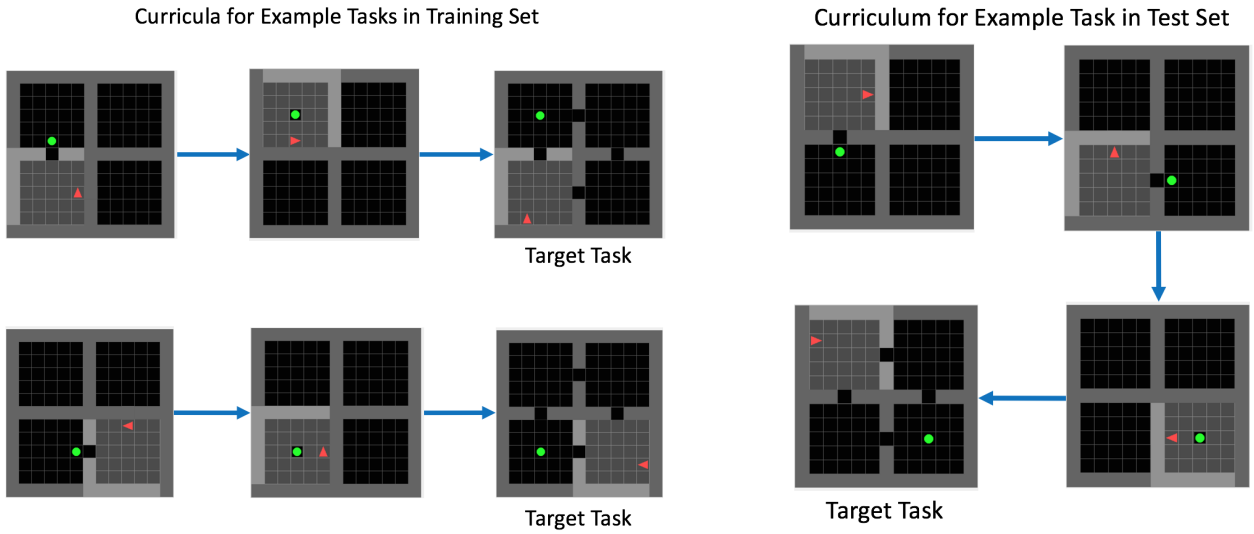


Figure 8.6: CMDP learning curves for the extrapolation experiments. The x-axis represents a CMDP episode, where each episode is an entire run of a curriculum. The y-axis is the cost of that curriculum in game steps. Taking all the points along the curve, the curriculum curve was statistically significantly better than no curriculum, using a 2-tail t-test with $p < 0.05$.



(a) (b)

Figure 8.7: Examples of curricula seen in the (a) training set and (b) test set. Tasks in the test set were the only ones that benefitted from a curriculum that directed the agent from the top left room to the bottom right. All other combinations of start and end rooms were seen in the training set.

universal value function is a value function defined over states and goals. In a curriculum MDP, states correspond to the agent's knowledge (as done in Chapter 7) while goals correspond to target tasks. Using this model and a two-stream neural network architecture, I showed that curriculum policies can both interpolate to new tasks that have similar curricula to seen tasks, and also extrapolate to new navigational tasks that use totally new curricula. This result opens the door to using more learning from experience in curriculum design. In human and animal training, as well as more recently in supervised machine learning, curricula have been adapted to train multiple types of learners for different target tasks. This work provides a similar result for the reinforcement learning setting.

9. Taxonomy of CL Methods and Related Work

Over the past few years, several groups have been studying how curricula can be generated automatically to train reinforcement learning agents, and many approaches to do so now exist. These methods for curriculum generation have separately been introduced for areas such as robotics, multi-agent systems, human-computer and human-robot interaction, and intrinsically motivated learning. This body of work, however, is largely disconnected. In addition, many landmark results in reinforcement learning, from TD-Gammon [131] to AlphaGo [110] have implicitly used curricula to guide training. In some domains, researchers have successfully used methodologies that align with our definition of curriculum learning without explicitly describing it that way (e.g., self-play). Given the many landmark results that have utilized ideas from curriculum learning, we think it is very likely that future landmark results will also rely on curricula, perhaps more so than researchers currently expect. Thus, having a common basis for discussion of ideas in this area is likely to be useful for future AI challenges.

In this chapter, I describe a taxonomy of curriculum learning for reinforcement learning approaches. I systematically survey methods that address each of the 3 main elements of curriculum learning – task generation, sequencing, and transfer learning – and focus in particular on sequencing methods. The central assumption in curriculum learning is that source tasks can be generated and organized to improve learning. Therefore, I organize sequencing methods by the ways in which the source task MDPs are allowed to differ from the target task MDP. During this discussion, I describe how the contributions and methods

This chapter is based on a survey that was published in the Journal of Machine Learning Research [84]. It was done in collaboration with Bei Peng, Jivko Sinapov, Matteo Leonetti, Matthew E. Taylor, and Peter Stone. My collaborators assisted in surveying some of the papers and in writing the article.

presented in this thesis fit in this taxonomy. Following this discussion, I describe how curriculum learning compares to other techniques for improving sample complexity in RL, and how curriculum learning is used in related areas such as supervised learning and for human training. This chapter addresses Contribution 6 from Chapter 1 of this thesis.

9.1 Dimensions of Categorization

Curriculum learning methods make different assumptions about where tasks come from, how they are represented and sequenced, and how they are evaluated. I propose to categorize curriculum learning approaches along the following seven dimensions, organized by attributes (in bold) and the values (in italics) they can take. I use these dimensions to create a taxonomy of surveyed work in Sections 9.2 to 9.4.

1. **Intermediate task generation**: *target / automatic / domain experts / naive users.* In curriculum learning, the primary challenge is how to sequence a set of tasks to improve learning speed. However, finding a good curriculum depends on first having useful source tasks to select from. Most methods assume the set of possible source tasks is fixed and given ahead of time. In the simplest case, only samples from the *target* task are used. When more than one intermediate task is used, typically they are manually designed by humans. I distinguish such tasks as designed by either *domain experts*, who have knowledge of the agent and its learning algorithm, or *naive users*, who do not have this information. On the other hand, some works consider *automatically* creating tasks online using a set of rules or generative process. These approaches may still rely on some human input to control/tune hyper-parameters, such as the number of tasks generated, or to verify that generated tasks are actually solvable.

2. **Curriculum representation**: *single / sequence / graph.* As I discussed previously, the most general form of a curriculum is a directed acyclic graph over subsets of

samples. However, in practice, simplified versions of this representation are often used. In the simplest case, a curriculum is an ordering over samples from a *single* task. When multiple tasks can be used in a curriculum, curricula are often created at the task-level. These curricula can be represented as a linear chain, or *sequence*. In this case, there is exactly one source for each intermediate task in the curriculum. It is up to the transfer learning algorithm to appropriately retain and combine information gathered from previous tasks in the chain. More generally, they can be represented as a full directed acyclic *graph* of tasks. This form supports transfer learning methods that transfer from many-to-one, one-to-many, and many-to-many tasks.

3. **Transfer method**: *policies / value function / task model / partial policies / shaping reward / other / no transfer*. Curriculum learning leverages ideas from transfer learning to transfer knowledge between tasks in the curriculum. As such, the transfer learning algorithm used affects how the curriculum will be produced. The type of knowledge transferred can be low-level knowledge, such as an entire *policy*, an *(action-)value function*, or a full *task model*, which can be used to directly initialize the learner in the target task. It can also be high-level knowledge, such as *partial policies* (e.g. options) or *shaping rewards*. This type of information may not fully initialize the learner in the target task, but it could be used to guide the agent's learning process in the target task. I use partial policies as an umbrella term to represent closely related ideas such as options, skills, and macro-actions. When samples from a single task are sequenced, *no transfer* learning algorithm is necessary. Finally, I use *other* to refer to other types of transfer learning methods. I categorize papers along this dimension based on what is transferred between tasks in the curriculum in each paper's experimental results.

4. **Curriculum sequencer**: *automatic / domain experts / naive users*. Curriculum learning is a three-part method, consisting of task generation, sequencing, and transfer learning. While much of the attention of this chapter is on automated sequencing

approaches, many works consider the other parts of this method, and assume the sequencing is done by a human or oracle. Thus, I identify and categorize the type of sequencing approach used in each work similar to task generation: it can be done *automatically* by a sequencing algorithm, or manually by humans that are either *domain experts* or *naive users*.

5. **Curriculum adaptivity**: *static / adaptive*. Another design question when creating a curriculum is whether it should be generated in its entirety before training, or dynamically adapted during training. I refer to the former type as *static* and to the latter as *adaptive*. Static approaches use properties of the domain and possibly of the learning agent, to generate a curriculum before any task is learned. Adaptive methods, on the other hand, are influenced by properties that can only be measured during learning, such as the learning progress by the agent on the task it is currently facing. For example, learning progress can be used to guide whether subsequent tasks should be easier or harder, as well as how relevant a task is for the agent at a particular point in the curriculum.

6. **Evaluation metric**: *time to threshold / asymptotic / jumpstart / total reward*. I discussed four metrics to quantify the effectiveness of learned curricula in Section 3.3. When calculating these metrics, one can choose whether to treat time spent generating the curriculum and training on the curriculum as a sunk cost, or whether to account for both of these for performance. Specifically, there are three ways to measure the cost of learning and training via a curriculum. 1) The cost of generating and using the curriculum is treated as a sunk cost, and the designer is only concerned with performance on the target task after learning. This case corresponds to the weak transfer setting. 2) The cost of training on intermediate tasks in the curriculum is accounted for, when comparing to training directly on the target task. This case is most common when it is hard to evaluate the cost of generating the curriculum itself,

for example if it was hand-designed by a human. 3) Lastly, the most comprehensive case accounts for the cost of generating the curriculum as well as training via the curriculum. I will refer to the last two as strong transfer, and indicate it by bolding the corresponding metric. Note that achieving asymptotic performance improvements implies strong transfer.

7. **Application area**: *toy / sim robotics / real robotics / video games / other*. Curriculum learning methods have been tested in a wide variety of domains. *Toy* domains consist of environments such as grid worlds, cart-pole, and other low dimensional environments. *Sim robotics* environments simulate robotic platforms, such as in MuJoCo. *Real robotics* papers test their method on physical robotic platforms. *Video games* consist of game environments such as Starcraft or the Arcade Learning Environment (Atari). Finally, *other* is used for custom domains that do not fit in these categories. I list these so that readers can better understand the scalability and applicability of different approaches, and use these to inform what methods would be suitable for their own problems.

In the next 3 sections, I describe our systematic work surveying each of the three central elements of curriculum learning: task generation (Section 9.2), sequencing (Section 9.3), and transfer learning (Section 9.4). For each of these subproblems, I provide a table that categorizes work surveyed according to the dimensions outlined in this section. The bulk of attention will be devoted to the subproblem most commonly associated with curriculum learning: sequencing.

9.2 Task Generation

Task generation is the problem of creating intermediate tasks specifically to be part of a curriculum. In contrast to the life-long learning scenario, where potentially unrelated tasks

are constantly proposed to the agent [134], the aim of task generation is to create a set of tasks such that knowledge transfer through them is beneficial. Therefore, all the generated tasks should be relevant to the final task(s) and avoid *negative transfer*, where using a task for transfer hurts performance. The properties of the research surveyed in this section are reported in Table 9.1.

Very limited work has been dedicated to formally studying this subproblem in the context of reinforcement learning. All known methods assume the domain can be parameterized using some kind of representation, where different instantiations of these parameters create different tasks. For instance, in Chapter 4 (based on our published work [82]), I introduced a number of methods to create intermediate tasks for a specific final task. The methods hinge on a definition of a domain as a set of MDPs identified by a *task descriptor*, which is a vector of parameters specifying the *degrees of freedom* in the domain. For example, in the quick chess example (see Chapter 1), these parameters could be the size of the board, number of pawns, etc. By varying the degrees of freedom and applying task *restrictions*, the methods define different types of tasks. The set of methods determine different kinds of possible tasks, which form a space of tasks in which appropriate intermediate tasks can be chosen.

Da Silva and Reali Costa [21] propose a similar partially automated task generation procedure in their curriculum learning framework, based on Object-Oriented MDPs. Each task is assumed to have a class *environment* parameterized by a number of attributes. A function, which must be provided by the designer, creates simpler versions of the final task by instantiating the attributes with values that make the tasks easier to solve. For example, continuing the quick chess example, the attributes could be the types of pieces, and the values are the number of each type of piece. The presence of different kinds and numbers of objects provide a range of tasks with different levels of difficulty. However, since the generation is mostly random, the designer has to make sure that the tasks are indeed solvable.

Citation	Intermediate Task Generation	Curriculum Representation	Transfer Method	Curriculum Sequencer	Curriculum Adaptivity	Evaluation Metric	Application Area
Da Silva and Reali Costa [21]	automatic	graph	value function	automatic	static	**time to threshold**, total reward	toy, video games
Narvekar et al. [82]	automatic	sequence	value function	domain experts	adaptive	**asymptotic**	video games
Schmidhuber [107]	automatic	sequence	partial policies	automatic	adaptive	**asymptotic**	other
Stone and Veloso [117]	automatic	sequence	other	domain experts	adaptive	time to threshold	other

Table 9.1: The papers discussed in Section 9.2, categorized along the dimensions presented in Section 9.1. Bolded values under evaluation metric indicate strong transfer.

Generating auxiliary intermediate tasks is a problem that has been studied in non-RL contexts as well. For instance, Stone and Veloso [117] consider how to semiautomatically create subproblems to aid in learning to solve difficult *planning* problems. Rather than using a static analysis of the domain's properties, they propose to use a partially completed search trajectory of the target task to identify what makes a problem difficult, and suggest auxiliary tasks. For example, if the task took too long and there are multiple goals in the task, try changing the order of the goals. Other methods they propose include reducing the number of goals, creating tasks to solve difficult subgoals, and changing domain operators and objects available for binding.

Lastly, Schmidhuber [107] introduced Powerplay, a framework that focuses on inventing new problems to train a more and more general problem solver in an unsupervised fashion. The system searches for both a new task and a modification of the current problem solver, such that the modified solver can solve all previous tasks, plus the new one. The search acts on a domain-dependent encoding of the problem and the solver, and has been demonstrated on pattern recognition and control tasks [115]. The generator of the task and new solver is given a limited computational budget, so that it favors the generation of the simplest tasks that could not be solved before. Furthermore, a possible task is to solve all previous tasks, but with a more compact representation of the solver. The resulting iterative process makes the system increasingly more competent at different tasks. The task generation process effectively creates a curriculum, although in this context there are no final tasks, and the system continues to generate pairs of problems and solvers indefinitely, without any

specific goal.

9.3 Sequencing

Given a set of tasks, or samples from them, the goal of sequencing is to order them in a way that facilitates learning. Many different sequencing methods exist, each with their own set of assumptions. One of the fundamental assumptions of curriculum learning is that we can configure the environment to create different tasks. For the practitioner attempting to use curriculum learning, the amount of control one has to shape the environment affects the type of sequencing methods that could be applicable. Therefore, we categorize sequencing methods by the degree to which intermediate tasks may differ. Specifically, they form a spectrum, ranging from methods that simply reorder experience in the final task without modifying any property of the corresponding MDP, to ones that define entirely new intermediate tasks, by progressively adjusting some or all of the properties of the final task.

In this section, I discuss the different sequencing approaches. First, in Section 9.3.1, I consider methods that reorder samples in the target task to derive a curriculum. Experience replay methods are one such example. In Section 9.3.2, I examine multi-agent approaches to curriculum generation, where the cooperation or competition between two (typically evolving) agents induces a sequence of progressively challenging tasks, like a curriculum. Then, in Section 9.3.3, I begin describing methods that explicitly use intermediate tasks, starting with ones that vary in limited ways from the target task. In particular, these methods only change the reward function and/or the initial and terminal state distributions to create a curriculum. In Section 9.3.4, I discuss methods that relax this assumption, and allow intermediate tasks that can vary in any way from the target task MDP. Finally, in Section 9.3.5, we discuss work that explores how humans sequence tasks into a curriculum.

9.3.1 Sample Sequencing

First I consider methods that reorder samples from the final task, but do not explicitly change the domain itself. These ideas are similar to curriculum learning for supervised learning [13], where training examples are presented to a learner in a specific order, rather than completely randomly. Bengio et al. [13] showed that ordering these examples from simple to complex can improve learning speed and generalization ability. An analogous process can be used for reinforcement learning. For example, many current reinforcement learning methods, such as Deep Q Networks (DQN) [77] use a replay buffer to store past state-action-reward experience tuples. At each training step, experience tuples are sampled from the buffer and used to train DQN in minibatches. The original formulation of DQN performed this sampling uniformly randomly. However, as in the supervised setting, samples can be reordered or "prioritized," according to some measure of usefulness or difficulty, to improve learning.

The first to do this type of sample sequencing in the context of deep learning were Schaul et al. [106]. They proposed Prioritized Experience Replay (PER), which prioritizes and replays *important* transitions more. Important transitions are those with high expected learning progress, which is measured by their temporal difference (TD) error. Intuitively, replaying samples with larger TD errors allows the network to make stronger updates. As transitions are learned, the distribution of important transitions changes, leading to an implicit curriculum over the samples.

Alternative metrics for priority/importance have been explored as well. Ren et al. [96] propose to sort samples using a complexity index function, which is a combination of a self-paced prioritized function and a coverage penalty function. The self-paced prioritized function selects samples that would be of appropriate difficulty, while the coverage function penalizes transitions that are replayed frequently. They provide one specific instantiation of these functions, which are used in experiments on the Arcade Learning Environment [12], and show that it performs better than PER in many cases. However, these functions

Citation	Intermediate Task Generation	Curriculum Representation	Transfer Method	Curriculum Sequencer	Curriculum Adaptivity	Evaluation Metric	Application Area
Sample Sequencing (Section 9.3.1)							
Andrychowicz et al. [5]	target	single	no transfer	automatic	adaptive	**asymptotic**	sim robotics
Fang et al. [27]	target	single	no transfer	automatic	adaptive	**asymptotic**	sim robotics
Kim and Choi [62]	target	single	no transfer	automatic	adaptive	**asymptotic**	toy, other
Lee et al. [69]	target	single	no transfer	automatic	adaptive	time to threshold	toy, video games
Ren et al. [96]	target	single	no transfer	automatic	adaptive	**asymptotic**	video games
Schaul et al. [106]	target	single	no transfer	automatic	adaptive	**asymptotic**	video games
Co-learning (Section 9.3.2)							
Baker et al. [7]	automatic	sequence	policies	automatic	adaptive	**asymptotic**, time to threshold	other
Bansal et al. [9]	automatic	sequence	policies	automatic	adaptive	**asymptotic**	sim robotics
Pinto et al. [90]	automatic	sequence	policies	automatic	adaptive	time to threshold	sim robotics
Sukhbaatar et al. [120]	automatic	sequence	policies	automatic	adaptive	time to threshold, **asymptotic**	toy, video games
Vinyals et al. [136]	automatic	sequence	policies	automatic	adaptive	**asymptotic**	video games
Reward and Initial/Terminal State Distribution Changes (Section 9.3.3)							
Asada et al. [6]	domain experts	sequence	value function	automatic	adaptive	**asymptotic**	sim/real robotics
Baranes and Oudeyer [10]	automatic	sequence	partial policies	automatic	adaptive	**asymptotic**	sim/real robotics
Florensa et al. [32]	automatic	sequence	policies	automatic	adaptive	**asymptotic**	sim robotics
Florensa et al. [33]	automatic	sequence	policies	automatic	adaptive	**asymptotic**	sim robotics
Ivanovic et al. [54]	automatic	sequence	policies	automatic	adaptive	**asymptotic**	sim robotics
Racaniere et al. [92]	automatic	sequence	policies	automatic	adaptive	**asymptotic**	toy, video games
Riedmiller et al. [97]	domain experts	sequence	policies	automatic	adaptive	time to threshold	sim/real robotics
Wu and Tian [147]	domain experts	sequence	task model	automatic	both	**asymptotic**	video games
No Restrictions (Section 9.3.4)							
Bassich et al. [11]	domain experts	sequence	policies	automatic	adaptive	**asymptotic**, **time to threshold**	toy
Da Silva and Reali Costa [21]	automatic	graph	value function	automatic	static	**time to threshold**, total reward	toy, video games
Foglino et al. [34]	domain experts	sequence	value function	automatic	static	time to threshold, **asymptotic**, total reward	toy
Foglino et al. [35]	domain experts	sequence	value function	automatic	static	total reward	toy
Foglino et al. [36]	domain experts	sequence	value function	automatic	static	total reward	toy
Jain and Tulabandhula [56]	domain experts	sequence	value function	automatic	adaptive	time to threshold, total reward	toy
Matiisen et al. [74]	domain experts	sequence	policies	automatic	adaptive	**asymptotic**	toy, video games
Narvekar et al. [83]	automatic	sequence	value function	automatic	adaptive	**time to threshold**	toy
Narvekar and Stone [80]	domain experts	sequence	value function, shaping reward	automatic	adaptive	**time to threshold**	toy, video games
Svetlik et al. [123]	domain experts	graph	shaping reward	automatic	static	**asymptotic**, **time to threshold**	toy, video games
Human-in-the-loop Curriculum Generation (Section 9.3.5)							
Hosu and Rebedea [51]	target	single	no transfer	automatic	adaptive	**asymptotic**	video games
Khan et al. [61]	domain experts	sequence	no transfer	naive users	static	N/A	other
MacAlpine and Stone [71]	domain experts	graph	policies	domain experts	static	**asymptotic**	sim robotics
Peng et al. [88]	domain experts	sequence	task model	naive users	static	**time to threshold**	other
Stanley et al. [116]	domain experts	sequence	partial policies	domain experts	adaptive	**asymptotic**	video games

Table 9.2: The papers discussed in Section 9.3, categorized along the dimensions presented in Section 9.1. Bolded values under evaluation metric indicate strong transfer.

must be designed individually for each domain, and designing a broadly applicable domain-independent priority function remains an open problem.

Kim and Choi [62] consider another extension of prioritized experience replay, where the weight/priority of a sample is jointly learned with the main network via a secondary neural network. The secondary network, called ScreenerNet, learns to predict weights according to the error of the sample by the main network. Unlike PER, this approach is memoryless, which means it can directly predict the significance of a training sample even if that particular example was not seen. Thus, the approach could potentially be used to actively request experience tuples that would provide the most information or utility, creating an online curriculum.

Instead of using sample importance as a metric for sequencing, an alternative idea is to restructure the training process based on trajectories of samples experienced. For example, when learning, typically easy to reach states are encountered first, whereas harder to reach states are encountered later on in the learning cycle. However, in practical settings with sparse rewards, these easy to reach states may not provide a reward signal. Hindsight Experience Replay (HER) [5] is one method to make the most of these early experiences. HER is a method that learns from "undesired outcomes," in addition to the desired outcome, by replaying each episode with a goal that was actually achieved rather than the one the agent was trying to achieve. The problem is set up as learning a Universal Value Function Approximator (UVFA) [105], which is a value function $v_\pi(s, g)$ defined over states s and goals g . The agent is given an initial state s_1 and a desired goal state g. Upon executing its policy, the agent may not reach the goal state g, and instead land on some other terminal state s_T. While this trajectory does not help to learn to achieve g, it does help to learn to achieve s_T. Thus, this trajectory is added to the replay buffer with the goal state substituted with s_T, and used with an off-policy RL algorithm. HER forms a curriculum by taking advantage of the implicit curriculum present in exploration, where early episodes are likely to terminate

on easy to reach states, and more difficult to reach states are found later in the training process.

One of the issues with vanilla HER is that all goals in seen trajectories are replayed evenly, but some goals may be more useful at different points of learning. Thus, Fang et al. [27] later proposed Curriculum-guided HER (CHER) to adaptively select goals based on two criteria: curiosity, which leads to the selection of diverse goals, and proximity, which selects goals that are closer to the true goal. Both of these criteria rely on a measure of distance or similarity between goal states. At each minibatch optimization step, the objective selects a subset of goals that maximizes the weighted sum of a diversity and proximity score. They manually impose a curriculum that starts biased towards diverse goals and gradually shifts towards proximity based goals using a weighting factor that is exponentially scaled over time.

Other than PER and HER, there are other works that reorder/resample experiences in a novel way to improve learning. One example is the episodic backward update (EBU) method developed by Lee et al. [69]. In order to speed up the propagation of delayed rewards (e.g., a reward might only be obtained at the end of an episode), Lee et al. [69] proposed to sample a whole episode from the replay buffer and update the values of all transitions within the sampled episode in a backward fashion. Starting from the end of the sampled episode, the max Bellman operator is applied recursively to update the target Q-values until the start of the sampled episode. This process basically reorders all the transitions within each sampled episode from the last timestep of the episode to the first, leading to an implicit curriculum. Updating highly correlated states in a sequence while using function approximation is known to suffer from cumulative overestimation errors. To overcome this issue, a diffusion factor $\beta \in (0, 1)$ was introduced to update the current Q-value using a weighted sum of the new bootstrapped target value and the pre-existing Q-value estimate. Their experimental results show that in 49 Atari games, EBU can achieve the same mean and median human normalized performance of DQN by using significantly fewer samples.

Methods that sequence experience samples have wide applicability and found broad success in many applications, since they can be applied directly on the target task without needing to create intermediate tasks that alter the environment. In the following sections, we consider sequencing approaches that progressively alter how much intermediate tasks in the curriculum may differ.

9.3.2 Co-learning

Co-learning is a multi-agent approach to curriculum learning, in which the curriculum emerges from the interaction of several agents (or multiple versions of the same agent) in the same environment. These agents may act either cooperatively or adversarially to drive the acquisition of new behaviors, leading to an implicit curriculum where both sets of agents improve over time. Self-play is one methodology that fits into this paradigm, and many landmark results such as TD-Gammon [131] and more recently AlphaGo [110] and AlphaStar [136] fall into this category. Rather than describing every work that uses self-play or co-learning, I describe a few papers that focus on how the objectives of the multiple agents can be set up to facilitate co-learning.

Sukhbaatar et al. [120] proposed a novel method called asymmetric self-play that allows an agent to learn about the environment without any external reward in an unsupervised manner. This method considers two agents, a teacher and a student, using the paradigm of "the teacher proposing a task, and the student doing it." The two agents learn their own policies simultaneously by maximizing interdependent reward functions for goal-based tasks. The teacher's task is to navigate to an environment state that the student will use either as 1) a goal, if the environment is resettable, or 2) as a starting state, if the environment is reversible. In the first case, the student's task is to reach the teacher's final state, while in the second case, the student starts from the teacher's final state with the aim of reverting the environment to its original initial state. The student's goal is to minimize the number

of actions it needs to complete the task. The teacher, on the other hand, tries to maximize the difference between the actions taken by the student to execute the task, and the actions spent by the teacher to set up the task. The teacher, therefore, tries to identify a state that strikes a balance between being the simplest goal (in terms of number of teacher actions) for itself to find, and the most difficult goal for the student to achieve. This process is iterated to automatically generate a curriculum of intrinsic exploration.

Another example of jointly training a pair of agents adversarially for policy learning in single-agent RL tasks is Robust Adversarial RL (RARL) by Pinto et al. [90]. Unlike asymmetric self-play [120], in which the teacher defines the goal for the student, RARL trains a protagonist and an adversary, where the protagonist learns to complete the original RL task while being robust to the disturbance forces applied by the adversarial agent. RARL is targeted at robotic systems that are required to generalize effectively from simulation, and learn robust policies with respect to variations in physical parameters. Such variations are modeled as disturbances controlled by an adversarial agent, and the adversarial agent's goal is to learn the optimal sequence of destabilizing actions via a zero-sum game training procedure. The adversarial agent tries to identify the hardest conditions under which the protagonist agent may be required to act, increasing the agent's robustness. Learning takes place in turns, with the protagonist learning against a fixed antagonist's policy, and then the antagonist learning against a fixed protagonist's policy. Each agent tries to maximize its own return, and the returns are zero-sum. The set of "destabilizing actions" available to the antagonist is assumed to be domain knowledge, and given to the adversary ahead of time.

For multi-agent RL tasks, several works have shown how simple interaction between multiple learning agents in an environment can result in emergent curricula. Such ideas were explored early on in the context of evolutionary algorithms by Rosin and Belew [100]. They showed that competition between 2 groups of agents, dubbed hosts and parasites, could lead to an "arms race," where each group drives the other to acquire increasingly complex skills

and abilities. Similar results have been shown in the context of RL agents by Baker et al. [7]. They demonstrated that increasingly complex behaviors can emerge in a physically grounded task. Specifically, they focus on a game of hide and seek, where there are two teams of agents. One team must hide with the help of obstacles and other items in the environment, while the other team needs to find the first team. They were able to show that as one team converged on a successful strategy, the other team was pressured to learn a counter-strategy. This process was repeated, inducing a curriculum of increasingly competitive agents.

A similar idea was explored by Bansal et al. [9]. They proposed to use multi-agent curriculum learning as an alternative to engineering dense shaping rewards. Their method interpolates between dense "exploration" rewards, and sparse multi-agent competitive rewards, with the exploration reward gradually annealed over time. In order to prevent the adversarial agent from getting too far ahead of the learning agent and making the task impossible, the authors propose to additionally sample older versions of the opponent. Lastly, in order to increase robustness, the stochasticity of the tasks is increased over time.

Curriculum learning approaches have also been proposed for cooperative multi-agent systems [138, 149]. In these settings, there is a natural curriculum created by starting with a small number of agents, and gradually increasing them in subsequent tasks. The schedule with which to increase the number of agents is usually manually defined, and the emphasis instead is on how to perform transfer when the number of agents change. Therefore, I discuss these approaches in more detail in Section 9.4.

Finally, while self-play has been successful in a wide variety of domains, including solving games such as Backgammon [131] and Go [110], such an approach alone was not sufficient for producing strong agents in a complex, multi-agent, partially-observable game like Starcraft. One of the primary new elements of Vinyals et al. [136] was the introduction of a Starcraft League, a group of agents that have differing strategies learned from a combination of imitation learning from human game data and reinforcement learning. Rather than have

every agent in the league maximize their own probability of winning against all other agents like in standard self play, there were some agents that did this, and some whose goal was to optimize against the main agent being trained. In effect, these agents were trained to exploit weaknesses in the main agent and help it improve. Training against different sets of agents over time from the league induced a curriculum that allowed the main agents to achieve grandmaster status in the game.

9.3.3 Reward and Initial/Terminal State Distribution Changes

Thus far, the curriculum consisted of ordering experience from the target task or modifying agents in the target environment. In this and the next section, I begin to examine approaches that explicitly create different MDPs for intermediate tasks, by changing some aspect of the MDP. First I consider approaches that keep the state and action spaces the same, as well as the environment dynamics, but allow the reward function and initial/terminal state distributions to vary.

One of the earliest examples of this type of method was *learning from easy missions*. Asada et al. [6] proposed this method to train a robot to shoot a ball into a goal based on vision inputs. The idea was to create a series of tasks, where the agent's initial state distribution starts close to the goal state, and is progressively moved farther away in subsequent tasks, inducing a curriculum of tasks. In this work, each new task starts one "step" farther away from the goal, where steps from the goal is measured using a domain specific heuristic: a state is closer to the terminal state if the goal in the camera image gets larger. The heuristic implicitly requires that the state space can be categorized into "substates," such as goal size or ball position, where the ordering of state transitions in a substate to a goal state is known. Thus, each substate has a dimension for making the task simpler or more complex. Source tasks are manually created to vary along these dimensions of difficulty.

Recently, Florensa et al. [32] proposed more general methods for performing this re-

verse expansion. They proposed reverse curriculum generation, an algorithm that generates a distribution of starting states that get increasingly farther away from the goal. The method assumes at least one goal state is known, which is used as a seed for expansion. Nearby starting states are generated by taking a random walk from existing starting states by selecting actions with some noise perturbation. In order to select the next round of starting states to expand from, they estimate the expected return for each of these states, and select those that produce a return between a manually set minimum and maximum interval. This interval is tuned to expand states where progress is possible, but not too easy. A similar approach by Ivanovic et al. [54] considered combining the reverse expansion phase for curriculum generation with physics-based priors to accelerate learning by continuous control agents.

An opposite "forward" expansion approach has also been considered by Florensa et al. [33]. This method allows an agent to automatically discover different goals in the state space, and thereby guide exploration of the space. They do this discovery with a Generative Adversarial Network (GAN) [42], where the generator network proposes goal regions (parameterized subsets of the state space) and the discriminator evaluates whether the goal region is of appropriate difficulty for the current ability of the agent. Goal regions are specified using an indicator reward function, and policies are conditioned on the goal in addition to the state, like in a universal value function approximator [105]. The agent trains on tasks suggested by the generator. In detail, the approach consists of 3 parts: 1) First, goal regions are labelled according to whether they are of appropriate difficulty. Appropriate goals are those that give a return between hyperparameters R_{min} and R_{max}. Requiring at least R_{min} ensures there is a signal for learning progress. Requiring less than R_{max} ensures that it is not too easy. 2) They use the labeled goals to train a Goal GAN. 3) Goals are sampled from the GAN as well as a replay buffer, and used for training to update the policy. The goals generated by the GAN shift over time to reflect the difficulty of the tasks, and gradually

move from states close to the starting state to those farther away.

Racaniere et al. [92] also consider an approach to automatically generate a curriculum of goals for the agent, but for more complex goal-conditioned tasks in dynamic environments where the possible goals vary between episodes. The idea was to train a "setter" model to propose a curriculum of goals for a "solver" agent to attempt to achieve. In order to help the setter balance its goal predictions, they proposed three objectives which lead to a combination of three losses to train the setter model: *goal validity* (the goal should be valid or achievable by the current solver), *goal feasibility* (the goal should match the feasibility estimates for the solver with current skill), and *goal coverage* (encourage the setter to choose more diverse goals to encourage exploration in the space of goals). In addition, a "judge" model was trained to predict the reward the current solver agent would achieve on a goal (the feasibility of a goal) proposed by the setter. Their experimental results demonstrate the necessity of all three criteria for building useful curricula of goals. They also show that their approach is more stable and effective than the goal GAN method [33] on complex tasks.

An alternative to modifying the initial or terminal state distribution is to modify the reward function. Riedmiller et al. [97] introduce SAC-X (Scheduled Auxiliary Control), an algorithm for scheduling and executing auxiliary tasks that allow the agent to efficiently explore its environment and also make progress towards solving the final task. Auxiliary tasks are defined to be tasks where the state, action, and transition function are the same as the original MDP, but where the reward function is different. The rewards they use in auxiliary tasks correspond to changes in raw or high level sensory input, similar to Jaderberg et al. [55]. However, while Jaderberg et al. [55] only used auxiliary tasks for improving learning of the state representation, here they are used to guide exploration, and are sequenced. The approach is a hierarchical RL method: they need to 1) learn intentions, which are policies for the auxiliary tasks, and 2) learn the scheduler, which sequences intention policies and auxiliary tasks. To learn the intentions, they learn to maximize the action-value function

of each intention from a starting state distribution that comes as a result of following each of the other intention policies. This process makes the policies compatible. The scheduler can be thought of as a meta-agent that performs sequencing, whose goal is to maximize the return on the target task MDP. The scheduler selects intentions, whose policy is executed on the extrinsic task, and is used to guide exploration.

Heuristic-based methods have also been designed to sequence tasks that differ in their reward functions. One such approach is SAGG-RIAC (Self-Adaptive Goal Generation - Robust Intelligent Adaptive Curiosity) [10]. They define *competence* as the distance between the achieved final state and the goal state, and *interest* as the change in competence over time for a set of goals. A region of the task space is deemed more *interesting* than others, if the latest tasks in the region have achieved a high increase in competence. The approach repeatedly selects goals by first picking a region with a probability proportional to its interest, and then choosing a goal at random within that region. With a smaller probability the system also selects a goal at random over the whole task set or a goal close to a previously unsuccessful task. The bias towards interesting regions causes the goals to be more dense in regions where the competence increases the fastest, creating a curriculum. Because of the stochastic nature of the goal generating process, however, not every task is necessarily beneficial in directly increasing the agent's ability on the target task, but contributes to updating the competence and interest measures. Since the intermediate tasks are generated online as the agent learns, in this approach both sequencing and generation result from the same sampling process.

Finally, Wu and Tian [147] also consider changing the transition dynamics and the reward functions of the intermediate tasks. They propose a novel framework for training an agent in a partially observable 3D Doom environment. Doom is a First-Person Shooter game, in which the player controls the agent to fight against enemies. In their experiment, they first train the agent on some simple maps with several curricula. Each curriculum consists

of a sequence of progressively more complex environments with varying domain parameters (e.g., the movement speed or initial health of the agent). After learning a capable initial task model, the agent is then trained on more complicated maps and more difficult tasks with a different reward function. They also design an adaptive curriculum learning strategy in which a probability distribution over different levels of curriculum is maintained. When the agent performs well on the current distribution, the probability distribution is shifted towards more difficult tasks.

9.3.4 No Restrictions

Next, there is a class of methods that create a curriculum using intermediate tasks, but make no restrictions on the MDPs of these intermediate tasks. I categorize them in three ways by how they address the task sequencing problem: treating sequencing 1) as an MDP/POMDP, 2) as a combinatorial optimization over sequences, and 3) as learning the connections in a directed acyclic task graph. Because there are no limitations on the types of intermediate tasks allowed, some assumptions are usually made about the transfer learning algorithm, and additional information about the intermediate tasks (such as task descriptors) is typically assumed. Finally, I also discuss work on an auxiliary problem to sequencing: how long to spend on each task.

MDP-based Sequencing

The first formalization of the sequencing problem is as a Markov Decision Process. These methods formulate curriculum generation as an interaction between 2 types of MDPs. The first is the standard MDP, which models a *learning agent* (i.e., the student) interacting with a task. The second is a higher level meta-MDP for the *curriculum agent* (i.e., the teacher), whose goal is to select tasks for the learning agent.

Narvekar et al. [83] denote the meta-MDP as a curriculum MDP (CMDP), where the state space $\mathcal{S}$ is the set of policies the learning agent can represent (this CMDP is described in Chapter 7). These CMDP states can be represented parametrically using the weights of the learning agent. The action space $\mathcal{A}$ is the set of tasks the learning agent can train on next. Learning a task updates the learning agent's policy, and therefore leads to a transition in the CMDP via a transition function p. Finally, the reward function r is the time in steps or episodes that it took to learn the selected task. Under this model, a curriculum agent typically starts in an initial state corresponding to a random policy for the learning agent. The goal is to reach a terminal state, which is defined as a policy that can achieve some desired performance threshold on the target task, as fast as possible.

Matiisen et al. [74] consider a similar framework, where the interaction is defined as a POMDP. The state and action spaces of the meta-POMDP are the same as in Narvekar et al. [83], but access to the internal parameters of the learning agent is not available. Instead, an observation of the current score of the agent on each intermediate task is given. The reward is the change in the score on the task from this timestep to the previous timestep when the same task was trained on. Thus, while Narvekar et al. [83] focused on minimizing time to threshold performance on the target task, the design of Matiisen et al. [74] aims to maximize the sum of performance in all tasks encountered.

While both approaches are formalized as POMDPs, learning on these POMDPs is computationally expensive. Thus, both propose heuristics to guide the selection of tasks. Narvekar et al. [83] take a sample-based approach (which we describe in detail in Chapter 6), where a small amount of experience samples gathered on the target and intermediate tasks are compared to identify relevant intermediate tasks. The task that causes the greatest change in policy as evaluated on the target task samples is selected. In contrast, Matiisen et al. [74] select tasks where the absolute value of the slope of the learning curve is highest. Thus it selects tasks where the agent is making the most progress or where the agent is

forgetting the most about tasks it has already learned. Initially tasks are sampled randomly. As one task starts making progress, it will be sampled more, until the learning curve plateaus. Then another will be selected, and the cycle will repeat until all the tasks have been learned.

Subsequently, Narvekar and Stone [80] explored whether learning was possible in a curriculum MDP, thus avoiding the need for heuristics in task sequencing. This approach is described in detail in Chapter 7. They showed that you can represent a CMDP state using the weights of the knowledge transfer representation. For example, if the agent uses value function transfer, the CMDP state is represented using the weights of the value function. By utilizing function approximation over this state space, they showed it is possible to learn a policy over this MDP, termed a curriculum policy, which maps from the current status of learning progress of the agent, to the task it should learn next. In addition, the approach addresses the question of how long to train on each intermediate task. While most works have trained on intermediate tasks until learning plateaus, this is not always necessary. Narvekar and Stone [80] showed that training on each intermediate task for a few episodes, and letting the curriculum policy reselect tasks that require additional time, results in faster learning. However, while learning a curriculum policy is possible, doing so independently for each agent and task is still very computationally expensive.

Combinatorial Optimization and Search

A second way of approaching sequencing is as a combinatorial optimization problem: given a fixed set of tasks, find the permutation that leads to the best curriculum, where best is determined by one of the CL metrics introduced in Section 3.3. Finding the optimal curriculum is a computationally difficult black-box optimization problem. Thus, typically fast approximate solutions are preferred.

One such popular class of methods are metaheuristic algorithms, which are heuristic methods that are not tied to specific problem domains, and thus can be used as black boxes.

Foglino et al. [34] adapt and evaluate four representative metaheuristic algorithms to the task sequencing problem: beam search [87], tabu search [40], genetic algorithms [41], and ant colony optimization [22]. The first two are trajectory-based, which start at a guess of the solution, and search the neighborhood of the current guess for a better solution. The last two are population-based, which start with a set of candidate solutions, and improve them as a group towards areas of increasing performance. They evaluate these methods for 3 different objectives: time to threshold, maximum return (asymptotic performance), and cumulative return. Results showed that the trajectory-based methods outperformed their population-based counterparts on the domains tested.

While metaheuristic algorithms are broadly applicable, it is also possible to create specific heuristic search methods targeted at particular problems, such as task sequencing with a specific transfer metric objective. Foglino et al. [35] introduce one such heuristic search algorithm, designed to optimize for the cumulative return. Their approach begins by computing transferability between all pairs of tasks, using a simulator to estimate the cumulative return attained by using one task as a source for another. The tasks are then sorted according to their potential of being a good source or target, and iteratively chained in curricula of increasing length. The algorithm is anytime, and eventually exhaustively searches the space of all curricula with a predefined maximum length.

Jain and Tulabandhula [56] propose 4 different online search methods to sequence tasks into a curriculum. Their methods also assume a simulator is available to evaluate learning on different tasks, and use the learning trajectory of the agent on tasks seen so far to select new tasks. The 4 approaches are: 1) Learn each source task for a fixed number of steps, and add the one that gives the most reward. The intuition is that high reward tasks are the easiest to make progress on. 2) Calculate a transferability matrix for all pairs of tasks, and create a curriculum by chaining tasks backwards from the target tasks greedily with respect to it. 3) Extract a feature vector for each task as in Narvekar et al. [82], and learn

a regression model to predict transferability using the feature vector. 4) Extract pair wise feature vectors between pairs of tasks, and learn a regression model to predict transferability.

Finally, instead of treating the entire problem as a black box, it has also been treated as a gray box. Foglino et al. [36] propose such an approach, formulating the optimization problem as the composition of a white box scheduling problem and black box parameter optimization. The scheduling formulation partially models the effects of a given sequence, assigning a utility to each task, and a penalty to each pair of tasks, which captures the effect on the objective of learning two tasks one after the other. The white-box scheduling problem is an integer linear program, with a single optimal solution that can be computed efficiently. The quality of the solution, however, depends on the parameters of the model, which are optimized by a black-box optimization algorithm. This external optimization problem searches the optimal parameters of the internal scheduling problem, so that the output of the two chained optimizers is a curriculum that maximizes cumulative return.

Graph-based Sequencing

Another class of approaches explicitly treats the curriculum sequencing problem as connecting nodes with edges into a directed acyclic task graph. Typically, the task-level curriculum formulation is used, where nodes in the graph are associated with tasks. A directed edge from one node to another implies that one task is a source task for another.

Existing work has relied on heuristics and additional domain information to determine how to connect different task nodes in the graph. For instance, Svetlik et al. [123] assume the set of tasks is known in advance, and that each task is represented by a task feature descriptor. These features encode properties of the domain. For example, in a domain like Ms. Pac-Man, features could be the number of ghosts or the type of maze. The approach consists of three parts. First, a binary feature vector is extracted from the feature vector to represent non-zero elements. This binary vector is used to group subsets of tasks that

share similar elements. Second, tasks within each group are connected into subgraphs using a novel heuristic called *transfer potential*. Transfer potential is defined for discrete state spaces, and trades off the applicability of a source task against the cost needed to learn it. Applicability is defined as the number of states that a value function learned in the source can be applied to a target task. The cost of a source task is approximated as the size of its state space. Finally, once subgraphs have been created, they are linked together using directed edges from subgraphs that have a set of binary features to subgraphs that have a superset of those features.

Da Silva and Reali Costa [21] follow a similar procedure, but formalize the idea of task feature descriptors using an object-oriented approach. The idea is based on representing the domain as an object-oriented MDP, where states consist of a set of objects. A task OO-MDP is specified by the set of specific objects in this task, and the state, action, transition, and reward functions of the task. With this formulation, source tasks can be generated by selecting a smaller set of objects from the target task to create a simpler task. To create the curriculum graph, they adapt the idea of transfer potential to the object-oriented setting: instead of counting the number of states that the source task value function is applicable in, they compare the sets of objects between the source and target tasks. While the sequencing is automated, human input is still required to make sure the tasks created are solvable.

Auxiliary Problems

Finally, I discuss an additional approach that tackles an auxiliary problem to sequencing: how long to spend on each intermediate task in the curriculum. Most existing work trains on intermediate tasks until performance plateaus. However, as we mentioned previously, Narvekar and Stone [80] showed that this is unnecessary, and that better results can be obtained by training for a few episodes, and reselecting or changing tasks dynamically as needed.

Bassich et al. [11] consider an alternative method for this problem based on *progression* functions. Progression functions specify the pace at which the difficulty of the task should change over time. The method relies on the existence of a task-generation function, which maps a desired complexity $c_t \in [0, 1]$ to a task of that complexity. The most complex task, for which $c_t = 1$, is the final task. After every episode, the progression function returns the difficulty of the task that the agent should face at that time. The authors define two types of progression functions: fixed progressions, for which the learning pace is predefined before learning takes place; and adaptive progressions, which adjust the learning pace online based on the performance of the agent. Linear and exponential progressions are two examples of fixed progression functions, and increase the difficulty of the task linearly and exponentially, respectively, over a prespecified number of time steps. The authors also introduce an adaptive progression based on a friction model from physics, which increases c_t as the agent's performance is increasing, and slows down the learning pace if performance decreases. Progression functions allow the method to change the task at every episode, solving the problem of deciding how long to spend in each task, while simultaneously creating a continually changing curriculum.

9.3.5 Human-in-the-Loop Curriculum Generation

Thus far, all the methods discussed in Section 9.3 create a curriculum *automatically* using a sequencing algorithm, which either reorders samples from the final task or progressively alters how much intermediate tasks in the curriculum may differ. Bengio et al. [13] and Taylor [125] both emphasize the importance of better understanding how *humans* approach designing curricula. Humans may be able to design good curricula by considering which intermediate tasks are "too easy" or "too hard," given the learner's current ability to learn, similar to how humans are taught with the zone of proximal development [137]. These insights could then be leveraged when designing automated curriculum learning systems. Therefore, in this

section, we consider curriculum sequencing approaches that are done *manually* by humans who are either *domain experts*, who have specialized knowledge of the problem domain, or *naive users*, who do not necessarily know about the problem domain and/or machine learning.

One example of having domain experts manually generate the curriculum is the work done by Stanley et al. [116], in which they explore how to keep video games interesting by allowing agents to change and to improve through interaction with the player. They use the NeuroEvolving Robotic Operatives (NERO) game, in which simulated robots start the game with no skills and have to learn complicated behaviors in order to play the game. The human player takes the role of a trainer and designs a curriculum of training scenarios to train a team of simulated robots for military combat. The player has a natural interface for setting up training exercises and specifying desired goals. An ideal curriculum would consist of exercises with increasing difficulty so that the agent can start with learning basic skills and gradually building on them. In their experiments, the curriculum is designed by several NERO programmers who are familiar with the game domain. They show that the simulated robots could successfully be trained to learn different sophisticated battle tactics using the curriculum designed by these domain experts. It is unclear whether a human player who is not familiar with the game can also design a good curriculum.

A more recent example is by MacAlpine and Stone [71]. They use a very extensive manually constructed curriculum to train agents to play simulated robot soccer. The curriculum consists of a training schedule over 19 different learned behaviors. It encompasses skills such as moving to different positions on the field with different speeds and rotation, variable distance kicking, and accessory skills such as getting up when fallen. Optimizing these skills independently can lead to problems at the intersection of these skills. For example, optimizing for speed in a straight walk can lead to instability if the robot needs to turn or kick due to changing environment conditions. Thus, the authors of this work hand-designed

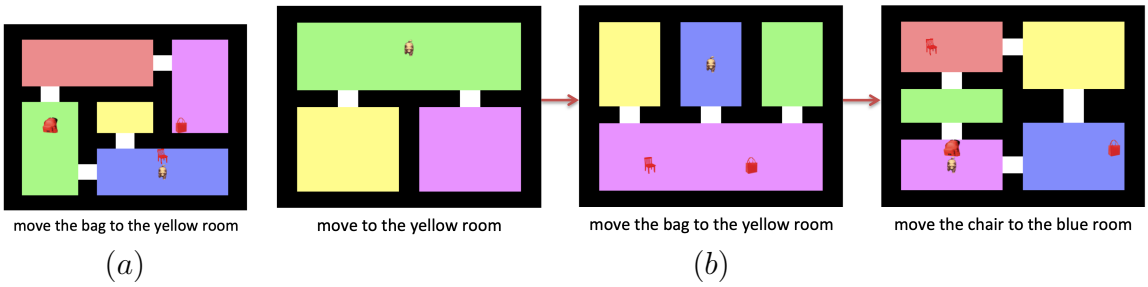


Figure 9.1: One example of curricula designed by human users. (a) Given final task. (b) A curriculum designed by one human participant.

a curriculum to train related skills together using an idea called overlapping layered learning. This curriculum is designed using their domain knowledge of the task and agents.

While domain experts usually generate good curricula to facilitate learning, most existing work does not explicitly explore their curriculum design process. It is unclear what kind of design strategies people follow when sequencing tasks into a curriculum. Published research on Interactive Reinforcement Learning [46, 63, 64, 70, 72, 118, 119, 133] has shown that RL agents can successfully speed up learning using human feedback, demonstrating the significant role can humans play in teaching an agent to learn a (near-) optimal policy. This large body of work mainly focuses on understanding how human teachers want to teach the agent and how to incorporate these insights into the standard RL framework. Similarly, the way we define curriculum design strategies still leaves a lot to be defined by human teachers. As pointed out by Bengio et al. [13], the notion of simple and complex tasks is often based on human intuition, and there is value in understanding how humans identify "simple" tasks. Along these lines, some work has been done to study whether curriculum design is a prominent teaching strategy that naive users choose to teach the agent and how they approach designing curricula.

To study the teaching strategies followed by naive users, Khan et al. [61] conduct behavioral studies in which human participants need to teach a robot the concept of whether an object can be grasped with one hand. In their experiment, participants are provided with

31 cards with photos of common objects (e.g., food, furniture, and animals) for them to select. The experiment consists of two subtasks. In the first subtask, participants sort the objects on the table based on their subjective ratings of their graspability. In the second subtask, participants pick up the cards from the table and show them to the robot while teaching the robot the concept of graspability, using as few cards as possible. While teaching the robot the object's graspability, participants can either use any natural language or say either "graspable" or "not graspable," depending on one of the two conditions they are randomly assigned. They observe that participants follow three distinct teaching strategies, one of which is consistent with the curriculum learning principle, i.e., starting simple and gradually increasing the difficulty of the task. Furthermore, they propose a novel theoretical framework as a potential explanation for the teaching strategy that follows the curriculum learning principle, which shows that it is the result of minimizing per-iteration expected error of the learner.

Peng et al. [88] also explore how naive users design a curriculum of tasks for an agent, but in a more complex sequential decision-making task. Specifically, a simple simulated home environment is used, where the agent must learn to perform tasks in a variety of environments. The tasks are specified via text commands and the agent is trained to perform the task via reinforcement and punishment feedback from a human trainer. It uses the goal-directed Strategy-Aware Bayesian Learning (SABL) algorithm [70] for learning from human feedback. In the user study, participants are asked to design a set of training assignments for the agent to help it quickly learn to complete the given final assignment (shown in Figure 9.1a). A set of source tasks are provided for human participants to select and sequence. One example of curricula designed by human participants is shown in Figure 9.1b. Their empirical results show that, compared to directly learning the pre-specified final task from scratch, non-expert humans can successfully design curricula that result in better overall agent performance on learning both the entire curriculum and the final task. They also

discover that humans are more likely to select commands for intermediate tasks that include concepts that are important for the final task, and that doing so results in curricula that lead to better overall agent performance. Furthermore, they demonstrate that by taking advantage of this type of non-expert guidance, their curriculum-learning algorithm can be adapted to learn the human-generated curricula more efficiently.

There is also some work that does not explicitly ask humans to design a curriculum, but uses human data to help generate the curriculum. One example is the work done by Hosu and Rebedea [51], in which they propose a deep RL method that combines online agent experiences with offline human experiences to train the agent more efficiently. In some sparse-reward Atari games such as Montezuma's Revenge and Private Eye, the agent needs to execute a long sequence of specific actions to receive the first positive reward from the environment, which makes the exploration problem much harder. Thus, the commonly used ϵ-greedy strategy could not find any game paths to reach a first state with positive reward, preventing the neural network from learning relevant features to good states. Inspired by curriculum learning and the human starts evaluation metric used for testing Atari agents, they use checkpoints sampled from a human player's game experience as starting points for the learning process. The main intuition behind this approach is that at least some of the checkpoints will be an "easier" starting point, which is closer to some states with positive reward that the agent can benefit from. While this method belongs to the class of sequencing approaches, as discussed in Section 9.3.1, that reorders samples in the final task to derive a curriculum, it additionally considers more informative sample data generated by naive human users in order to build a more efficient curriculum.

9.4 Transfer Learning

While sequencing, as covered in Section 9.3, is the core component of curriculum learning, the whole premise of CL depends on an agent's ability to transfer knowledge among tasks.

In this subsection, I provide a brief survey of this area.

In curriculum learning, transfer learning methods are used to allow the agent to reuse knowledge learned from one intermediate task for another within the curriculum. It is worth noting that when creating a curriculum using only samples from the target task (discussed in Section 9.3.1), there is no transfer as there is only a single task (the target task) and correspondingly no change in the environment. However, when creating a curriculum using multiple intermediate tasks, which may differ in state/action space, reward function, or transition function from the final task, transfer learning is needed to extract and pass on reusable knowledge acquired in one intermediate task to the next. The type of knowledge transferred also directly affects the type of learner that is applicable to the learning process.

Transferred knowledge can be low-level, such as an entire policy, a value function, a full task model, or some training instances, which can be directly used to initialize the learner in the target task. The knowledge can also be high-level, such as partial policies or options, skills, shaping rewards, or subtask definitions. This type of information may not fully initialize the learner in the target task, but it could be used to guide the agent's learning process in the target task. In this subsection, we discuss different transfer learning approaches used in curricula.

In policy transfer, a policy learned in a source or intermediate task is used to initialize the policy in the target task. When transferring policies between different tasks, the tasks may differ in some aspect of the MDP, such as starting states [32], reward functions [33, 97], or transition functions [19]. For instance, Clegg et al. [19] demonstrate that an arm-like manipulator can successfully learn the control policy for a simulated dressing task, by transferring policies between tasks with different transition functions. In a dressing task, the goal is to insert a robotic arm into a garment (such as a shirt) and achieve a desired position. To achieve this goal, they first train a sphere to move through a funnel-like geometry to reach some target location. They then directly apply the learned policy to a different scenario in

Citation	Intermediate Task Generation	Curriculum Representation	Transfer Method	Curriculum Sequencer	Curriculum Adaptivity	Evaluation Metric	Application Area
Clegg et al. [19]	domain experts	sequence	policies	domain experts	static	**asymptotic**, time to threshold	sim robotics
Fujii et al. [38]	domain experts	sequence	partial policies	domain experts	static	**asymptotic**	real robotics
Karpathy and Van De Panne [60]	domain experts/target	sequence/single	partial policies /no transfer	domain experts/automatic	static/adaptive	time to threshold	sim robotics
Rusu et al. [101]	domain experts	sequence	policies	domain experts	static	**asymptotic**	video games
Shao et al. [109]	domain experts	sequence	task model	domain experts	static	**asymptotic**, total reward	video games
Sinapov et al. [112]	automatic	sequence	value function	automatic	static	jump start	video games
Tessler et al. [132]	domain experts	sequence	partial policies	domain experts	static	**asymptotic**	video games
Vezhnevets et al. [135]	automatic	sequence	partial policies	automatic	static	**asymptotic**, total reward	video games
Wang et al. [138]	domain experts	sequence	policies	domain experts	static	**asymptotic**	video games
Yang and Asada [148]	domain experts	sequence	partial policies	automatic	adaptive	**asymptotic**, **time to threshold**	real robotics
Yang et al. [149]	domain experts	sequence	policies	domain experts	static	**asymptotic**, time to threshold	toy, other
Zimmer et al. [150]	domain experts	sequence	partial policies	domain experts	static	**asymptotic**, total reward	sim robotics

Table 9.3: The papers discussed in Section 9.4, categorized along the dimensions presented in Section 9.1. Bolded values under evaluation metric indicate strong transfer.

which a manipulator with arbitrary shape navigates through a simulated garment. The main trick is to train multiple spheres using a curriculum learning strategy and then aggregate them to control the manipulator in the dressing task.

In Shao et al. [109], a learned task model is transferred between tasks, which is used to initialize the policy network. Thus, it is similar to transferring policies. Their work aims to solve the problem of multi-agent decision making in StarCraft micromanagement, where the goal is to control a group of units to destroy the enemy under certain terrain conditions. A parameter sharing multi-agent gradient-descent Sarsa(λ) (PS-MAGDS) method is proposed to train the units to learn an optimal policy, which is parametrized by a feed-forward neural network. PS-MAGDS simply extends the traditional Sarsa(λ) to multiple units by sharing parameters of the policy network among units to encourage cooperative behaviors. A reward function including small immediate rewards is also designed to accelerate the learning process. When using transfer learning in their experiments, the agents are first trained in some small scale source scenarios using PS-MAGDS. The well-trained model is then used to initialize the policy network to learn micromanagement in the target scenarios. To scale the combat to a large scale scenario, they combine curriculum learning and transfer learning where the agents are trained with a sequence of progressively more complex micromanagement tasks. The difficulty of the micromanagement task is controlled by changing the number and type of units.

Value function transfer is another common method for transferring low-level knowledge between intermediate tasks within a curriculum. In most existing work [21, 83, 112], value function transfer is achieved by using the parameters of a value function learned in one intermediate task to initialize the value function in the next intermediate task in the curriculum, such that the agent learns the final task with some initial policy that is better than random exploration. For example, Sinapov et al. [112] focus on addressing the task selection problem in curriculum learning using value function transfer, under the assumption that no samples from the final tasks are available (this idea is described in detail in Chapter 5). They propose to use meta-data (i.e., a fixed-length feature vector that describes the task) associated with each task to identify suitable intermediate tasks. The main idea is to use such meta-data to learn the benefits of transfer between different 'source-target' task pairs, and have this generalize to new unseen task pairs to guide task selection.

When transferring low-level policies or value functions across tasks, there are several challenges that arise, particularly in the modern context of deep reinforcement learning. First is the problem of catastrophic forgetting, where knowledge from previously learned tasks is lost as information on a new task is incorporated. This effect occurs because the weights of the neural network optimized for a first task must be changed to meet the objectives of a new task, often resulting in poorer performance on the original task. Typically, in the curriculum setting, we only care about performance in the final tasks. However, if information from two orthogonal tasks needs to be combined (such as two independent skills), this challenge needs to be addressed. One approach is progressive neural networks [101], which trains a new network "column" for each new task, and leverages lateral connections to previously learned network columns to achieve transfer. When training subsequent columns, parameters from previous columns are frozen, which prevents catastrophic forgetting. The limitation is that the number of parameters grows with the number of tasks, and at inference time, the task label is needed to know which column to extract output from.

A second problem is the case where the state and action spaces differ between tasks. One alternative is to transfer higher-level knowledge across tasks, such as partial policies or options. A partial policy is a policy that is not necessarily defined for all states in the state space of an MDP. We use partial policies as an umbrella term to represent closely related ideas such as options, skills, and macro-actions. Yang and Asada [148] transfer learned control parameters between tasks, which are similar to partial policies. To solve the impedance learning problem for high-speed robotic assembly, they allow the system to learn impedance parameters associated with different dynamic motions separately, rather than to learn all the control parameters simultaneously. For instance, they first learn only the parameters associated with quasistatic motion by driving the system slowly, leaving other parameters unlearned. After the quasistatic parameters have been learned, they then slightly increase the motion speed, and use the learned values to initialize the quasistatic parameters when learning other parameters. Another example of transferring partial policies between tasks is the work done by Zimmer et al. [150]. Their main idea is to progressively increase the dimensionality of the tackled problem by increasing the (continuous) state and action spaces of the MDP, while an agent is learning a policy. The agent first learns to solve the source task with reduced state and action spaces until the increase in performance stagnates. Then, the partial policy learned by the agent is used as an initialization to learn the full policy in the target task with full state and action spaces. A developmental layer (like a dropout layer) is added to the network to filter dimensions of the states and actions.

Similarly, Fujii et al. [38] transfer options between tasks. To train mobile robots to learn collision avoidance behaviors in multi-robot systems more efficiently, they develop a multi-layered RL mechanism. Rather than gradually increasing the level of task complexity based on the learner's performance as in Yang and Asada [148], their learning process consists of four stages like a curriculum in which each stage learns a pre-defined controller. Each controller learns an option to solve a pre-defined sub-task. For instance, the first controller

learns to move toward a specific goal. Then the output (goal-directed behavior) of the first controller is used as input for the second controller, which aims to learn to avoid the collision to a single robot, and so on.

Vezhnevets et al. [135] also transfer high-level macro-actions between tasks, which are simpler instances of options. In their experiment, the agent is trained with a curriculum where the goal state is first set to be very close to the start state and is then moved further away during learning process. Although the task gets progressively harder, the temporally abstracted macro-actions remain the same. The macro-actions learned early on can also be easily adapted using their proposed architecture. Specifically, a deep recurrent neural network architecture is used to maintain a multi-step action plan. The network learns when to commit to the action plan to generate macro-actions and when to update the plan based on observations.

Another mode for transfer is skills. Tessler et al. [132] propose a deep RL method that effectively retains and transfers learned skills to solve lifelong learning in MineCraft. In their work, a set of N skills are trained a priori on various sub-tasks, which are then reused to solve the harder composite task. In their MineCraft experiment, the agent's action space includes the original primitive actions as well as the set of pre-learned skills (e.g., navigate and pickup). A hierarchical architecture is developed to learn a policy that determines when to execute primitive actions and when to reuse pre-learned skills, by extending the vanilla DQN architecture [77]. The skills could be sub-optimal when they are directly reused for more complex tasks, and this hierarchical architecture allows the agent to learn to refine the policy by using primitive actions. They also show the potential for reusing the pre-learned skill to solve related tasks without performing any additional learning.

Rather than selectively reusing pre-learned skills, Karpathy and Van De Panne [60] focus on learning motor skills in an order of increasing difficulty. They decompose the acquisition of skills into a two-level curriculum: a *high-level* curriculum specifies the order

in which different motor skills should be learned, while the *low-level* curriculum defines the learning process for a specific skill. The high-level curriculum orders the skills in a way such that each skill is relatively easy to learn, using the knowledge of the previously learned skills. For instance, the Acrobot first learns the Hop (easy to learn from scratch) and Flip (similar to hopping very slowly) skills, and then learns the more complex Hop-Flip skill. The learned skill-specific task parameters for easier skills will highly constrain the states that the Acrobat could be in, making it easier to learn more complex skills. For example, the Hop-Flip skills begin from a hopping gait of some speed, which can be reached by repeatedly executing the previously learned Hop skill.

In multi-agent settings, several specific methods have been designed for curricula that progressively scale the number of agents between tasks. In these settings, the state and action spaces often scale based on the number of agents present. One common assumption in many of these methods is that the state space can be factored into elements for the environment s^{env}, the agent s^{n}, and all other agents s^{-n}. For example, Yang et al. [149] propose CM3, which takes a two-stage approach. In the first stage, a single agent is trained without the presence of other agents. This training is done by inducing a new MDP that removes all dependencies on agent interactions (i.e., removing s^{-n}) and training a network on this subspace. Then in the second stage, cooperation is learned by adding the parameters for the other agents into the network.

Wang et al. [138] propose 3 different approaches for multi-agent settings. The first is buffer reuse, which saves the replay buffers from all previous tasks, and samples experience from all of them to train in the current task. Samples from lower dimensional tasks are padded with zeros. The second is curriculum distillation, which adds a distillation loss based on KL divergence between policies/q-values between tasks. The third is transferring the model using a new network architecture called Dynamic Agent-number Network (DyAN). In this architecture, the state space elements related to the agent and environment go through a

fully connected network, while the observations for each teammate agent are passed through a graph neural network (GNN) and then aggregated. These networks are subsequently combined to produce q-values or policies.

9.5 Related Paradigms in Reinforcement Learning

One of the central challenges in applying reinforcement learning to real world problems is sample complexity. Due to issues such as a sparse reward signal or complex dynamics, difficult problems can take an RL agent millions of episodes to learn a good policy, with many suboptimal actions taken during the course of learning. Many different approaches have been proposed to deal with this issue. To name a few, one method is imitation learning [104], which uses demonstrations from a human as labels for supervised learning to bootstrap the learning process. Another example is off-policy learning [49], which uses existing data from an observed behavior policy, to estimate the value of a desired target policy. Model-based approaches [121] first learn a model of the environment, which can then be used for planning the optimal policy.

Each of these methods come with their advantages and disadvantages. For imitation learning, the assumption is that human demonstrations are available. However, these are not always easy to obtain, especially when a good policy for the task is not known. In off-policy learning, in order to make full use of existing data, it is assumed that the behavior policy has a nonzero probability of selecting each action, and typically that every action to be evaluated or the target policy has been seen at least once. Finally, model-based approaches typically first learn a model of the environment, and then use it for planning. However, any inaccuracies in the learned model can compound as the planning horizon increases. Curriculum learning takes a different approach, and makes a different set of assumptions. The primary assumption is that the environment can be configured to create different subtasks, and that it is easier for the agent to discover *on its own* reusable pieces of knowledge in these subtasks that can

be used for solving a more challenging task.

Within reinforcement learning, there are also several paradigms that consider learning on a set of tasks so as to make learning more efficient. Multitask learning, lifelong/continuous learning, active learning, and meta-learning are four such examples.

In *multitask learning*, the goal is to learn how to solve *sets* of prediction or decision making tasks. Formally, given a set of tasks $m_1, m_2, \ldots m_n$, the goal is to *co-learn* all of these tasks, by optimizing the performance over all n tasks simultaneously. Typically, this optimization is facilitated by learning over some shared basis space. For example, Caruana [17] considers multitask learning for supervised learning problems, and shares layers of a neural network between tasks. In supervised learning, these tasks are different classification or regression problems. Similar ideas have been applied in a reinforcement learning context by Wilson et al. [145]. In reinforcement learning, different tasks correspond to different MDPs.

Lifelong learning and *continual learning* can be viewed as an online version of multitask learning. Tasks are presented one at a time to the learner, and the learner must use shared knowledge learned from previous tasks to more efficiently learn the presented task. As in multitask learning, typically the goal is to optimize performance over all tasks given to the learner. Lifelong and continual learning have been examined in both the supervised setting [102] and the reinforcement learning setting [2, 98]. The distinguishing feature of curriculum learning compared to these works is that in curriculum learning, we have full control over the *order* in which tasks are selected. Indeed, we may have control over the *creation* of tasks as well. In addition, the goal is to optimize performance for a specific target task, rather than all tasks. Thus, source tasks in curriculum learning are designed solely to improve performance on the target task—we are not concerned with optimizing performance in a source.

In *active learning*, the learner chooses which task or example to learn or ask about

next, from a given set of tasks. Typically, active learning has been examined in a semi-supervised learning setting: a small amount of labeled data exists whereas a larger amount of unlabeled data is present. The labeled data is used to learn a classifier to infer labels for unlabeled data. Unlabeled data that the classifier is not confident about is requested for a label from a human user. For example, Ruvolo and Eaton [103] consider active learning in a lifelong learning setting, and show how a learner can actively select tasks to improve learning speed for all tasks in a set, or for a specific target task. The selection of which task to be learned next is similar to the *sequencing* aspect of curriculum learning. However, the full method of curriculum learning is much broader, as it also encompasses creating the space of tasks to consider. Ruvolo and Eaton [103] and similar active learning work typically assume the set of tasks to learn and select from are already given. In addition, typically active learning has been examined for supervised prediction tasks, whereas we are concerned with reinforcement learning tasks.

Finally, in *meta-learning* [31], the goal is to train an agent on a variety of tasks such that it can quickly adapt to a new task within a small number of gradient descent steps. Typically, the agent is not given information identifying the task it is training on. In contrast, in curriculum learning, the learning agent may or may not have information identifying the task. However, the process that designs the curriculum by sequencing tasks usually does have this information. Like in the lifelong setting, there is no significance attached to the order in which tasks are presented to the learner. In addition, the objective in meta-learning is to train for fast adaptability, rather than for a specific final task as is the case in curriculum learning.

9.6 Curricula in Supervised Machine Learning

In addition to reinforcement learning, curriculum learning has been examined for supervised learning. In this section, I highlight a few examples of work that draw parallels to the RL

setting.

Bengio et al. [13] first formalized the idea of curriculum learning in the context of supervised machine learning. They conducted case studies examining when and why training with a curriculum can be beneficial for machine learning algorithms, and hypothesized that a curriculum serves as both a continuation method and a regularizer. A continuation method is an optimization method for non-convex criteria, where a smoothed version of the objective is optimized first, with the smoothing gradually reduced over training iterations. Typically, "easy" examples in a curriculum correspond to a smoother objective. Using a simple shape recognition and language domain, they showed that training with a curriculum can improve both learning speed and performance.

While many papers before Bengio et al. [13] *used* the idea of a curriculum to improve training of machine learning algorithms, most work considering how to systematically *learn* a curriculum came after. One recent example is work by Graves et al. [43]. They introduced measures of *learning progress*, which indicate how well the learner is currently improving from the training examples it is being given. They introduce 2 main measures based on 1) rate of increase in prediction accuracy and 2) rate of increase of network complexity. These serve as the reward to a non-stationary multi-armed bandit algorithm, which learns a stochastic policy for selecting tasks. These signals of learning progress could in theory be applied or adapted to the reinforcement learning setting as well. Graves et al. [43] also make an interesting observation, which is that using a curriculum is similar to changing the step size of the learning algorithm. Specifically, in their experiments, they found that a random curriculum still serves as a strong baseline, because all tasks in the set provide a gradient[10]. Easier tasks provide a stronger gradient while harder tasks provide a gradient closer to 0. Thus, choosing easy, useful tasks allows the algorithm to take larger steps and converge faster.

[10]Note however that in the reinforcement learning setting, because the policy affects the distribution of states an agent encounters, random training can be significantly worse.

More recently, Fan et al. [26] frame curriculum learning as "Learning to Teach," where a teacher agent learned to train a learning agent using a curriculum. The process is formulated as an MDP between these two interacting agents, similar to the MDP approaches discussed in Section 9.3.4: the teacher agent selects the training data, loss function, and hypothesis space, while the learning agent trains given the parameters specified by the teacher. The state space of the MDP is represented as a combination of features of the data, features of the student model, and features that represent the combination of both data and learner models. The reward signal is the accuracy on a held-out development set. Training a teacher agent can be computationally expensive. They amortize this cost by using a learned teacher agent to teach a new student with the same architecture. For example, they train the teacher using the first half of MNIST, and use the learned teacher to train a new student from the second half of MNIST. Another way they amortize the cost is to train a new student with a different architecture (e.g., changing from ResNet32 to ResNet110). Similar ideas have been explored in the reinforcement learning setting. However, the test set distribution is different from the training set distribution, which makes performing these kind of evaluations more challenging. However, showing that the cost for training a teacher can be amortized is an important direction for future work.

Finally, Jiang et al. [58] explore the idea of self-paced curriculum learning for supervised learning, which unifies and takes advantage of the benefits of self-paced learning and curriculum learning. In their terminology, curriculum learning uses prior knowledge, but does not adapt to the learner. Specifically, a curriculum is characterized by a ranking function, which orders a dataset of samples by priority. This function is usually derived by predetermined heuristics, and cannot be adjusted by feedback from the learner. In contrast, self-paced learning (SPL) adjusts to the learner, but does not incorporate prior knowledge and leads to overfitting. In SPL, the curriculum design is implicitly embedded as a regularization term into the learning objective. However, during learning, the training loss

usually dominates over the regularization, leading to overfitting. This paper proposes a framework that unifies these two ideas into a concise optimization problem, and discusses several concrete implementations. The idea is to replace the regularization term in SPL with a self-paced function, such that the weights lie within a predetermined curriculum region. In short, the curriculum region induces *a weak ordering* over the samples, and the self-paced function determines the actual learning scheme within that ordering. The idea has parallels to a task-level curriculum for RL, where the curriculum induces a weak ordering over samples from all tasks, and with the learning algorithm determining the actual scheme within that ordering.

9.7 Algorithmically Designed Curricula in Education

Curriculum learning has also been widely used for building effective Intelligent Tutoring Systems (ITS) for human education [15, 23, 44, 52, 53]. An ITS system involves a student interacting with an intelligent tutor (a computer-based system), with the goal of helping the student to master all skills quickly, using as little learning content as possible. Given that students have different learning needs, styles, and capabilities, the intelligent tutor should be able to provide customized instructions to them. To achieve this goal, one common strategy is called *curriculum sequencing*, which aims to provide the learning materials in a meaningful order that maximizes learning of the students with different knowledge levels. The main problem this strategy must solve is to find the most effective lesson to propose next, given the student's current learning needs and capabilities.

Reinforcement learning is one of the machine learning techniques that has been used with intelligent tutors to partially automate construction of the student model and to automatically compute an optimal teaching policy [146]. One advantage of using RL methods in tutoring is that the model can learn adaptive teaching actions based on each individual student's performance in real time, without needing to encode complex pedagogical rules that

the system requires to teach effectively (e.g., how to sequence the learning content, when and how to provide an exercise). Another advantage is that it is a general domain-independent technique that can be applied in any ITS.

As a concrete example, Iglesias et al. [52, 53] adapt Q-learning [141] to an adaptive and intelligent educational system to allow it to automatically learn how to teach each student. They formulate the learning problem as an RL problem, where the state is defined as the description of the student's knowledge, indicating whether the student has learned each knowledge item. The set of actions the intelligent tutor can execute includes selecting and showing a knowledge item to the student. A positive reward is given when all required content has been learned, otherwise no reward is given. The system evaluates the student's knowledge state through tests, which shows how much the student knows about each knowledge item. The Q-value estimates the usefulness of executing an action when the student is in a particular knowledge state. Then, the tutoring problem can be solved using the traditional Q-learning algorithm.

Green et al. [44] propose using a multi-layered Dynamic Bayes Net (DBN) to model the teaching problem in an ITS system. The main idea is to model the dynamics of a student's skill acquisition using a DBN, which is normally used in RL to represent transition functions for state spaces. More specifically, they formulate the problem as a factored MDP, where the state consists of one factor for each skill, corresponding to the student's proficiency on that particular skill. The actions are to either provide a hint or to pose a problem about a particular skill to the student. From a history of teacher-student interaction, the teacher can model the student's proficiency state, with the goal of teaching the student to achieve the highest possible proficiency value on each skill, using as few problems and hints as possible. Subsequently, the learned DBN model is used by a planning algorithm to search for the optimal teaching policy, mapping proficiency states of student knowledge to the most effective problem or hint to pose next.

To allow the automated teacher to select a sequence of pedagogical actions in cases where learner's knowledge may be unobserved, a different problem formulation is posed by Rafferty et al. [93]. They formulate teaching as a partially observable Markov decision process (POMDP), where the learner's knowledge state is considered as a hidden state, corresponding to the learner's current understanding of the concept being taught. The actions the automated teacher can select is a sequence of pedagogical choices, such as examples or short quizzes. The learner's next knowledge state is dependent on her current knowledge state and the pedagogical action the teacher chooses. Changes in the learner's knowledge state reflect learning. In this framework, the automated teacher makes some assumptions about student learning, which is referred to as the learner model: it specifies the space of possible knowledge states and how the knowledge state changes. Then the teacher can update its beliefs about the learner's current knowledge state based on new observations, given this learner model. Using this POMDP framework, they explore how different learner models affect the teacher's selection of pedagogical actions.

While most approaches seek to solely maximize overall learning gains, Ramachandran and Scassellati [95] propose an RL-based approach that uses a personalized social robot to tutor children, that maximizes learning gains and sustained engagement over the student-robot interaction. The main goal of the social robot is to learn the ordering of questions presented to a child, based on difficulty level and the child's engagement level in real time. To represent the idea that children with different knowledge levels need a different curriculum, each child is categorized into a given group based on knowledge level at the start of the one-on-one tutoring interaction. An optimal teaching policy is then learned specific to each group. In particular, their approach consists of a training phase and an interaction phase. In the training phase, participants are asked to complete a tutoring exercise. A pretest and post-test will be used to evaluate the participant's relative learning gains, which will also be used as the reward function to learn an optimal policy during the training phase. Subsequently, in

the interaction phase, the child's real-time engagement will be detected, serving as another reward signal for the RL algorithm to further optimize the teaching policy.

Non-RL-based algorithms have been considered as well. Ballera et al. [8] leverage the roulette wheel selection algorithm (RWSA) to perform personalized topic sequencing in e-learning systems. RWSA is typically used in genetic algorithms to arrange the chromosomes based on their fitness function, such that individuals with higher fitness value will have higher probability of being selected [41]. Similarly, in an e-learning system, a chromosome is denoted by a lesson. Each lesson has a fitness value that dynamically changes based on the student's learning performance. This fitness value indicates how well the topic was learned by the student, depending on three performance parameters: exam performance, study performance, and review performance of the learner. A lower fitness value means that the student has a poorer understanding of the topic. Thus, a reversed mechanism of RWSA is implemented, so as to select the lessons with lower fitness values more often for reinforcement. Then, this reversed RWSA algorithm is combined with linear ranking algorithm to sort the lessons.

9.8 Summary

In this chapter, I introduced a taxonomy of curriculum learning methods for reinforcement learning, that classified approaches according to 7 different attributes. These attributes spanned categories such as how the curriculum is generated, how it is represented, and how it is evaluated. I then systematically surveyed existing work on each of 3 elements of curriculum learning – task generation (Section 9.2), sequencing (Section 9.3), and transfer learning (Section 9.4) – with a particular focus on sequencing methods. In particular, I split sequencing methods into five categories, based on the assumptions they make about intermediate tasks in the curriculum. The simplest of these methods are sample sequencing methods (Section 9.3.1), which reorder samples from the final task itself, but do not explicitly

change the domain. These methods were followed by co-learning methods (Section 9.3.2), where a curriculum emerges from the interaction of several agents in the same environment. Next I considered methods that explicitly changed the MDP to produce intermediate tasks. Some of these methods assumed that the environment dynamics stay the same, but that the initial/terminal state distribution and reward function can change (Section 9.3.3). Others made no restrictions on the differences allowed from the target task MDP (Section 9.3.4). I also discussed how humans approach sequencing, to shed light on manually designed curricula in existing work (Section 9.3.5).

The work in this thesis contributes in several areas to this body of work, as I highlighted throughout this chapter. The task generation methods described in Chapter 4 are one of only a few methods designed to address this subproblem. Other methods to address this subproblem (Section 9.2) rely on similar ideas to the task descriptor feature vector for creating tasks. Chapter 5 then uses the task descriptor to perform source task selection for a target task, and learn a model of transferability. I discuss how this fits in with other transfer learning methods that make different assumptions about what is available in the target task, and what is transferred in Section 9.4.

Chapters 6 and 7 introduce full curriculum sequencing methods, that fall within the approaches that make no assumptions on the way the target task MDP can be modified to create source tasks for the curriculum (Section 9.3.4). Chapter 8 extends the method from Chapter 7 to learn curriculum policies that generalize to new tasks – an idea that, to the best of my knowledge, has not been considered by previous work.

Following this discussion, I described how curriculum learning relates to other approaches to improve sample complexity in reinforcement learning, and approaches that also consider learning multiple sets of tasks (Section 9.5). Finally, I also discussed how curriculum learning is used in supervised learning (Section 9.6), as well as for teaching humans in education (Section 9.7). Our survey of this literature has helped identify several open

problems and directions for future work, which I will discuss in the next chapter (Chapter 10).

10. Conclusion and Future Work

Reinforcement learning is a branch of machine learning that considers how an agent should act in an environment, using only a numeric reward signal. Due to the typically sparse nature of this signal and challenges in the environment (such as a large state space, partial observability, or adversaries), learning in complex tasks can be very slow, taking millions of episodes. One way to accelerate this process is to train the agents through a *curriculum*, which allows them to acquire simple skills in easy tasks, and use those skills to aid in solving a more difficult target task. This idea is inspired by human learning, where people gradually acquire complex cognitive and motor skills by training through a curriculum.

While manually designed curricula have been used to train artificial agents for over 2 decades, the question of how to automatically design a curriculum has only recently begun to receive attention. This thesis therefore sought to answer the following question:

Can reinforcement learning agents benefit from learning via a curriculum, and how can an autonomous curriculum design agent automatically create a curriculum tailored to both the abilities of individual learning agents and the task in question?

To answer this question, we first formalized the idea of a curriculum, and the methodology of curriculum learning. Over the past few years, several groups have looked at different ways of organizing experience from tasks, and the definitions we provide encompass these different techniques. We then tackled the main components of curriculum learning: how to create useful source tasks, how to evaluate how good a source task is for another target task, and how to automatically sequence these tasks into a curriculum. We also showed that our

methods produce curricula that are tailored to each agent's learning abilities, and showed how the curricula could be generalized for new unseen tasks.

10.1 Contributions

In summary, this thesis made the following contributions to the field of curriculum learning in reinforcement learning:

1. **Problem Definition**

   In Chapter 3, I formalized the concept of a curriculum as a directed acyclic graph over sets of samples. I showed how this general-purpose definition encompasses special cases such as curricula composed of sequences of tasks. I also introduced the method of curriculum learning, which consists of 3 parts – task generation, sequencing, and transfer learning – and discussed how curriculum learning approaches can be evaluated by adapting metrics from transfer learning.

2. **Methods for Creating Source Tasks**

   In Chapter 4, I presented several methods that modify the target task MDP to produce relevant source tasks. Some of these methods use a parameterized model of the domain to create simpler instances of the domain, while others use an agent's experience tuples from trajectories on the target task to create agent-specific source tasks.

3. **Method to Evaluate Task Transferability**

   In Chapter 5, I discussed an approach that learns a task transferability model between source and target task pairs. The method uses only a feature vector describing both tasks, and learns to predict the expected jumpstart in the target task, after first training on the source task.

4. **Methods for Sequencing Tasks into a Curriculum**

   In Chapter 6, I presented our first method for sequencing tasks into a curriculum. This method repeatedly obtains samples of the agent's policy on the target task, and uses a heuristic that looks for the maximum policy change on those samples to select which task should go next in the curriculum.

   In Chapter 7, I discussed our second method for sequencing tasks, which formulates curriculum generation as an interaction between 2 MDPs – one for the student RL agent, and one for the teacher curriculum agent. This approach uses reinforcement learning to learn a policy over the teacher MDP, which specifies what a student agent should train to optimize a desired performance metric.

   For both methods, I showed that an individualized curriculum was learned for each agent.

5. **Methods for Adapting a Curriculum Created for one Task to a Different Task**

   In Chapter 8, I discussed how the CMDP approach from Chapter 7 could be combined with universal value functions. With this combination, I showed that a curriculum policy could be trained on one set of tasks, and generalize to produce curricula for new and unseen navigation-based target tasks.

6. **A Taxonomy of Curriculum Learning Approaches for RL**

   In Chapter 9, I presented a taxonomy to classify curriculum learning methods based on 7 different attributes. These attributes encompassed properties such as how the curriculum is generated, how it is represented, and how it is evaluated. I also presented a systematic survey of methods addressing each of the 3 elements of curriculum learning, with a particular focus on sequencing methods. Finally, I also briefly discussed how curriculum learning for RL methods compare to other methods to improve sample

complexity in RL, how curriculum learning has been used in supervised learning, and how it has been used for teaching humans in education.

7. **Empirical Validation**

   Throughout this thesis, we have evaluated our methods on domains ranging from gridworlds, to domains such as Ms. Pac-Man and Half Field Offense. Domains were chosen based on whether they had the properties relevant to show the benefits of curriculum-based strategies, with the primary one being customizability (to create a diverse set of source and target tasks).

10.2 Future Work

In this section, I describe open problems and ideas for future work. These ideas include extensions of the methods presented throughout this thesis, as well as related directions that I expect will be useful for the field of curriculum learning in reinforcement learning in general.

10.2.1 Human Studies

As laid out in Chapter 1, human learning is the inspiration for using curriculum-based strategies to improve learning in artificial agents. Humans learn and refine complex motor and cognitive skills by training via a curriculum. Whether it's coaching an athlete in sports, or designing a therapy to help stroke patients recover control of a limb, the training process is organized through a series of drills that incrementally build up their abilities. However, how best to design such a curriculum is a challenging open problem, even for human learning and education [20, 78, 79, 93]. Currently, curricula across many domains are developed largely based on tradition and intuition, centered around the ill-defined notion of "practice."

The focus of this thesis has been on understanding how curricula can be automatically

designed for artificial agents. But this topic also begs the question: can a similar process be used to design curricula for human learning? Our ongoing work [39] – being done in collaboration with Keya Ghonasgi, Reuth Mirsky, Bharath Masetty, Ashish Deshpande, and Peter Stone – considers learning curricula for human motor learning tasks, such as learning to regain control of a limb after a stroke by devising a therapy (i.e., a curriculum) or improving proficiency in a motor task (such as in sports or games).

As a step towards this goal, we are proposing to adapt the CMDP curriculum learning model presented in Chapter 7 designed for RL agents to represent the learning process of humans. This model was inspired by human learning, and designed to improve training for autonomous reinforcement learning agents. In this work, we take the complementary view and aim to train human agents by adapting curriculum learning methods designed for RL agents to the human setting. In the CMDP model, this effect is achieved by replacing the RL student agent with a human learner.

There are two initial challenges to adapting the CMDP formulation and directly using it for human student agents:

1. **Representing the Human's State of Knowledge.** The knowledge state of an RL student is usually defined as the weights of the student's policy. Based on these weights, one can predict the student's behavior in each possible task. However, for humans, there is no perfect way to fully capture the student's knowledge state, as we do not have access to the brain's parameters. Instead, we can only infer their internal state by measuring features of their performance on an instance of the task (such as using a diagnostic or evaluative "test").

2. **Learning Never Stops.** In RL agents, when we wish to evaluate the state of knowledge of the student, we can turn off its learning process and run the student's policy many times without it using these new experiences to improve its policy. We can also evaluate the effects of different curricula by resetting the student's state to some base-

line level for every new curriculum. However, when evaluating people we cannot "turn off" their learning mechanisms, and hence any evaluation must be considered as part of the training process. Moreover, once a learner has trained on one sequence of source tasks, the knowledge state cannot be reset to test a new curriculum.

In order to test automated curriculum learning for human agents, we also need a domain that is both flexible enough to create many different learning scenarios (i.e., has many potential source tasks), and challenging enough to induce motor learning in humans quantifiable via measured features of performance. In our ongoing work [39], we have introduced a new game called "Reach Ninja," which is inspired by the popular phone game Fruit Ninja [47]. It uses a webcam to track a player's hand movement as they reach for various objects on the screen, rewarding them for hitting certain objects and penalizing them for hitting others. The game has several features that make it challenging to learn, and we have observed that training on the game over time does improve a participant's motor skill and performance.

The next step is to evaluate whether a good curriculum can be learned by adapting the curriculum MDP model from Chapter 7. One way to implement this idea is to directly apply the CMDP process as done previously to human students. However, training directly in this way is computationally expensive, and would be even more expensive since it would require human time. Therefore, we propose to first learn a curriculum policy for this domain using an RL agent, where the cost of simulation is much cheaper. We will use this curriculum as a baseline, and evaluate how well it works for a human.

Although at first glance it seems unlikely that humans and artificial agents could benefit from similar curricula, there could be some overlap. We hypothesize that a curriculum is based on 2 elements. First, there is progression of difficulty that is intrinsic to the domain – for example, playing chess with more types of pieces is harder than playing with fewer types. Such domain-specific qualities would influence curricula for all types of learners in

this domain. The second is an agent-specific element, where the curriculum is modified based on how a specific agent learns. We hypothesize that the RL-agent curriculum will capture some elements of the difficulty that are intrinsic to the domain, and thus will serve as a good starting point for adaptation for the second element. If the starting point is found to be useful, performing this adaptation will be an important direction for future work. Tailoring the curriculum for human learners will require understanding how a human's state of knowledge changes as they interact with tasks, and can be thought of as a sim-to-real problem, where the curriculum policy is learned in simulation on RL agents, and must be applied to real human learners.

10.2.2 Fully Automated Task Creation

Task creation is an important component of the methodology of curriculum learning. Whether tasks are created "on-demand" or all in advance, the quality of the pool of tasks generated directly affects the quality of curricula that can be produced. In addition, the *quantity* of tasks produced affect the search space and efficiency of curriculum sequencing algorithms. Despite its importance to curriculum learning, very limited work (see Section 9.2) has been done on the problem of automatically generating tasks.

In Chapter 4, I introduced a set of methods for semi-automatically creating tasks. However, these methods have hyper-parameters that control both how many and what types of tasks are created. These parameters are usually manually tuned to keep the space of tasks a reasonable size while retaining quality source tasks. For example, MistakeLearning (Algorithm 4) uses the parameter ϵ which controls how far back to rewind, while TaskSimplification (Algorithm 1) can simplify each dimension of the degrees of freedom by a variable amount. Reducing the amount of manual input required by these methods remains an important area for future work.

10.2.3 Transferring and Combining Different Types of Knowledge

Between each pair of tasks in a curriculum, knowledge must be transferred from one task to the next (or in the case of a graph, knowledge must be transferred from nodes that have a directed edge into the current node). In Chapters 6 and 7, we have assumed that the type of knowledge transferred has been fixed. For example, in Chapter 6, value function transfer was used to transfer information between every pair of tasks in the curriculum. In Chapter 7, we showed that the CMDP approach can be used to transfer different types of knowledge (both value functions and shaping rewards). However, as in Chapter 6, only one or the other is used in a particular run while learning the CMDP, because it affects how the CMDP state space is represented. To the best of my knowledge, all existing work also sequences curricula using a single, fixed transfer method.

However, this limitation opens the question of whether different tasks could benefit from extracting different types of knowledge. For instance, it may be useful to extract an option from one task, and a model from another. Thus, in addition to deciding *which* task to transfer from (which is the typical question in sequencing), we could also ask *what* to extract and transfer from that task. Past transfer learning literature has shown that many forms of transfer are possible, and the best type of knowledge to extract may very well differ based on task. In addition, new methods will need to be developed to effectively combine these different types of knowledge.

10.2.4 Generalizing Curricula to Different Agents

As we discussed in Chapter 8, a limitation of many curriculum learning approaches is that the time to *generate* a curriculum can be greater than the time to learn the target task outright. This shortcoming stems from the fact that curricula are typically learned independently for each agent and target task. However, in areas such as human education, curricula are used to train multiple students in multiple subjects.

In Chapter 8, I discussed how to deal with one half of this limitation: generalizing curricula to new tasks, by combining CMDPs with universal value functions. Another way would be to create a curriculum that can generalize or adapt to new agents. One approach to do so could be to encode a representation of the learner, rather than the task as done in Chapter 8. This representation would need to capture the class of policies that learner could represent, in a space that is comparable with classes of policies for other learners. Another option could be meta-learning (in the style of MAML [31]), where we learn a curriculum policy over a distribution of agents, in a way that is able to quickly adapt to new agent types. Such an approach could also be considered for quickly adapting to new target tasks.

10.2.5 Extending CMDPs to Black Box Agents

The CMDP formulation for curriculum generation presented in Chapter 7 represents the CMDP state space using the agent's state of knowledge. For RL agents, I described how the state of knowledge can be represented by the weights $\boldsymbol{\theta}$ of the RL agent's value function or policy. However, sometimes we may not have access to the agent's internal parameters. For example, when designing a curriculum for multi-agent or ad hoc teamwork settings, we may not have access to these parameters. Even in the cases where we do have such access, the agent may have too many parameters $\boldsymbol{\theta}$ for efficient learning, such as if the agent uses a very large neural network. In both of these cases, the agent is effectively a black box.

One alternative is to instead rely on observations of the agent's state of knowledge – i.e., actions from instances of its policy. For example, we can evaluate the agent's policy on some set of states from the target task (and optionally also the source tasks). These states can be thought of as "test states" – similar to how humans evaluate their knowledge through a finite set of questions on a test. A key open question is how to select an appropriate set of test states. The main property we desire is that as the agent trains through a curriculum, the policy should change to reflect accumulation of knowledge on these test states. As part

of ongoing work, we are looking into using states from expert demonstrations of the target task as test states.

10.2.6 Sim-to-Real Curriculum Learning

As I discussed earlier, generating curricula can be very expensive. However, using a curriculum once learned is faster than learning a task tabula rasa. Therefore, if the curriculum can be generated in a (simulation) environment where the cost is cheap compared to the (real) environment where the actual agent is deployed, the cost can also be justified.

This idea is similar to the sim-to-real problem, where policies learned in simulation (such as a simulated robot) need to be transferred to the real world (a physical robot), where the environment dynamics are different. Common approaches to address the sim-to-real problem include grounding the simulator by making it more closely resemble the real world [28], transforming the agent's actions to make their effects more similar to reality [50], and injecting noise or training on multiple simulation environments to learn more robust policies [94].

The motivation behind these approaches is that the policy learned in simulation is not optimized for the real world. Similarly, when training an agent in simulation using a curriculum, the exact weights of the policy learned after the curriculum in simulation would not apply in the real world. However, an interesting question is whether the semantics of the curriculum tasks might. Therefore, the physical robot could go through the same training regimen that was already optimized in simulation, but learn using the physics and dynamics of the real world. Also, I would like to note that the "real world" does not have to be limited to robots. As I described in Section 10.2.1, a similar approach can be used for human learning, where humans are the "real world" setting.

10.2.7 Combining Task Generation and Sequencing

The curriculum learning method can be thought of as consisting of 3 parts: task generation, sequencing, and transfer learning. In this thesis and most existing work, each of these pieces has been tackled independently. For example, sequencing methods typically assume the tasks are prespecified, or a task generation method exists. However, an interesting question is whether the task generation and task sequencing phases can be done simultaneously, by directly *generating* the next task in the curriculum.

Some very preliminary work has been done in this direction in the context of video game level generation. For example, Green et al. [45] used an evolutionary algorithm to generate maps for a gridworld, where each tile had a different element. The generator was optimized to maximize the loss of a deep RL agent's network, inducing a training curriculum.

Combining task generation and sequencing introduces additional challenges, such as specifying the space of possible maps/tasks, ensuring those maps are solvable, and creating maps that are challenging, but not too difficult to solve. In addition, training the generator can be very expensive. However, it promises an end-to-end solution that could reduce the amount of human intervention needed to design curricula.

10.2.8 Theoretical Analysis

There have been many practical applications of curricula to speed up learning in both supervised and reinforcement learning. However, despite empirical evidence that curricula are beneficial, there is a lack of theoretical results analyzing why they are useful, and how they should be created. An initial analysis in the context of supervised learning was done by Weinshall and Amir [142] and Weinshall et al. [143]. They analyzed whether reordering samples in linear regression and binary classification problems could improve the ability to learn new concepts. They did this analysis by formalizing the idea of an Ideal Difficulty Score (IDS), which is the loss of the example with respect to the optimal hypothesis, and the

Local Difficulty Score (LDS), which is the loss of the example with respect to the current hypothesis. These are 2 ways to classify the difficulty of a sample, which can be used as a means to sequence samples. They showed that the convergence of an algorithm like stochastic gradient descent monotonically decreases with the IDS, and monotonically increases with the LDS. An open question is whether similar grounded metrics for difficulty of tasks can be identified in reinforcement learning, and what kind of convergence guarantees we can draw from them.

10.3 Concluding Remarks

Humans learn concepts and develop complex motor skills by training via a curriculum. This thesis explored whether similar ideas could be used to improve the efficiency of training reinforcement learning agents. Concretely, this thesis formalized the idea of a curriculum, and the method of curriculum learning in reinforcement learning. It presented several methods that address the main new components of curriculum learning – generating source tasks and automatically sequencing them into a curriculum. Together, the formalizations create a common basis for discussion in this field, and the methods devised offer a new paradigm for training RL agents. As alluded to in this thesis, curriculum-based strategies have been key to many successful reinforcement learning applications, and we expect advances in this field will play an important role in future ones as well.

A. Acronyms

term	meaning
CC	Correlation Coefficient
CHER	Curriculum Hindsight Experience Replay
CL	Curriculum Learning
CM3	Cooperative Multi-goal Multi-stage Multi-agent
CMAC	Cerebellar Model Arithmetic Computer
CMDP	Curriculum Markov Decision Process
DBN	Dynamic Bayes Net
DCG	Discounted Cumulative Gain
DQN	Deep Q-Network
DyAN	Dynamic Agent-number Network
EBU	Episodic Backward Update
GAN	Generative Adversarial Network
GNN	Graph Neural Network
HER	Hindsight Experience Replay
HFO	Half Field Offense
IDS	Ideal Difficulty Score
ITS	Intelligent Tutoring System
KL	KullbackLeibler
LDS	Local Difficulty Score
LR	Linear Regression
MAML	Model Agnostic Meta-learning
MDP	Markov Decision Process

term	**meaning**
NDCG	Normalized Discounted Cumulative Gain
NERO	NeuroEvolving Robotic Operatives
NN	Neural Network
OO-MDP	Object-oriented Markov Decision Process
PER	Prioritized Experience Replay
POMDP	Partially Observable Markov Decision Process
PPO	Proximal Policy Optimization
PS-MAGDS	Parameter Sharing Multi-agent Gradient Descent
RARL	Robust Adversarial Reinforcement Learning
RWSA	Roulette Wheel Selection Algorithm
RL	Reinforcement Learning
SABL	Strategy-Aware Bayesian Learning
SAC-X	Scheduled Auxiliary Control
SAGG-RIAC	Self-adaptive Goal Generation - Robust Intellligent Adaptive Curiosity
SPL	Self-paced Learning
TD	Temporal Difference
TL	Transfer Learning
UVFA	Universal Value Function Approximator

Bibliography

[1] H. Akiyama and T. Nakashima. *HELIOS 2012: RoboCup 2012 Soccer Simulation 2D League Champion*, volume 7500. Springer, 2012. 44

[2] H. B. Ammar, E. Eaton, P. Ruvolo, and M. E. Taylor. Online multi-task learning for policy gradient methods. In *International Conference on Machine Learning (ICML)*, pages 1206–1214, 2014. 152

[3] H. B. Ammar, E. Eaton, M. E. Taylor, D. C. Mocanu, K. Driessens, G. Weiss, and K. Tuyls. An automated measure of mdp similarity for transfer in reinforcement learning. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014. 50

[4] H. B. Ammar, E. Eaton, J. M. Luna, and P. Ruvolo. Autonomous cross-domain knowledge transfer in lifelong policy gradient reinforcement learning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3345–3351, 2015. 14

[5] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5048–5058, 2017. 124, 125

[6] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23(2-3): 279–303, 1996. 19, 124, 130

[7] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch. Emergent tool use from multi-agent autocurricula. In *International Conference on Learning Representations (ICLR)*, 2020. 124, 129

[8] M. Ballera, I. A. Lukandu, and A. Radwan. Personalizing e-learning curriculum using reversed roulette wheel selection algorithm. In *International Conference on Education Technologies and Computers (ICETC)*, pages 91–97. IEEE, 2014. 159

[9] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch. Emergent complexity via multi-agent competition. In *International Conference on Learning Representations (ICLR)*, 2018. 124, 129

[10] A. Baranes and P.-Y. Oudeyer. Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73, 2013. 124, 133

[11] A. Bassich, F. Foglino, M. Leonetti, and D. Kudenko. Curriculum learning with a progression function. https://arxiv.org/abs/2008.00511, 2020. 124, 140

[12] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. 123

[13] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *International Conference on Machine Learning (ICML)*, pages 41–48, 2009. 123, 140, 142, 154

[14] M. Bowling, N. Burch, M. Johanson, and O. Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, January 2015. 1

[15] E. Brunskill and S. Russell. Partially observable sequential decision making for problem

selection in an intelligent tutoring system. In *Poster at International Conference on Educational Data Mining (EDM)*. Citeseer, 2011. 156

[16] P. Burrow and S. M. Lucas. Evolution versus temporal difference learning for learning to play ms. pac-man. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 53–60. IEEE, 2009. 37

[17] R. Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997. 152

[18] M. Chevalier-Boisvert, D. Bahdanau, S. Lahlou, L. Willems, C. Saharia, T. H. Nguyen, and Y. Bengio. BabyAI: First steps towards grounded language learning with a human in the loop. In *International Conference on Learning Representations*, 2019. 105

[19] A. Clegg, W. Yu, Z. Erickson, J. Tan, C. K. Liu, and G. Turk. Learning to navigate cloth using haptics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 2799–2805, 2017. 145, 146

[20] B. Clement, D. Roy, P.-Y. Oudeyer, and M. Lopes. Multi-armed bandits for intelligent tutoring systems. *Journal of Educational Data Mining*, 7(2), 2015. 165

[21] F. L. Da Silva and A. Reali Costa. Object-oriented curriculum generation for reinforcement learning. In *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, 2018. 120, 121, 124, 139, 147

[22] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: An autocatalytic optimizing process. *Technical Report*, 1991. 137

[23] S. Doroudi, K. Holstein, V. Aleven, and E. Brunskill. Sequence matters but how exactly? a method for evaluating activity sequences from data. *Grantee Submission*, 2016. 156

[24] J. L. Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993. 19

[25] A. Fachantidis, I. Partalas, G. Tsoumakas, and I. Vlahavas. Transferring task models in reinforcement learning agents. *Neurocomputing*, 107:23–32, 2013. 14

[26] Y. Fan, F. Tian, T. Qin, X.-Y. Li, and T.-Y. Liu. Learning to teach. In *International Conference on Learning Representations (ICLR)*, 2018. 155

[27] M. Fang, T. Zhou, Y. Du, L. Han, and Z. Zhang. Curriculum-guided hindsight experience replay. In *Advances in Neural Information Processing Systems (NIPS)*, pages 12602–12613, 2019. 124, 126

[28] A. Farchy, S. Barrett, P. MacAlpine, and P. Stone. Humanoid robots learning to walk faster: From the real world to simulation and back. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, pages 39–46, 2013. 171

[29] F. Fernández, J. García, and M. Veloso. Probabilistic policy reuse for inter-task transfer learning. *Robotics and Autonomous Systems*, 58(7):866–871, 2010. 14

[30] N. Ferns, P. Panangaden, and D. Precup. Bisimulation metrics for continuous markov decision processes. *SIAM Journal on Computing*, 40(6):1662–1714, 2011. 74

[31] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning (ICML)*, pages 1126–1135. JMLR. org, 2017. 153, 170

[32] C. Florensa, D. Held, M. Wulfmeier, M. Zhang, and P. Abbeel. Reverse curriculum generation for reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2017. 31, 124, 130, 145

[33] C. Florensa, D. Held, X. Geng, and P. Abbeel. Automatic goal generation for reinforcement learning agents. In *International Conference on Machine Learning (ICML)*, pages 1514–1523, 2018. 124, 131, 132, 145

[34] F. Foglino, C. Coletto Christakou, and M. Leonetti. An optimization framework for task sequencing in curriculum learning. In *International Conference on Developmental Learning (ICDL-EPIROB)*, 2019. 124, 137

[35] F. Foglino, C. Coletto Christakou, R. Luna Gutierrez, and M. Leonetti. Curriculum learning for cumulative return maximization. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2019. 124, 137

[36] F. Foglino, M. Leonetti, S. Sagratella, and R. Seccia. A gray-box approach for curriculum learning. In *World Congress on Global Optimization*, 2019. 124, 138

[37] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002. 56

[38] T. Fujii, Y. Arai, H. Asama, and I. Endo. Multilayered reinforcement learning for complicated collision avoidance problems. In *International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2186–2191. IEEE, 1998. 146, 148

[39] K. Ghonasgi, R. Mirsky, B. Masetty, S. Narvekar, A. Haith, A. Deshpande, and P. Stone. Leveraging reinforcement learning for human motor skill acquisition. In *Social AI for Human-Robot Interactions of Human-Care Service Robots Workshop in the International Conference on Intelligent Robots and Systems (IROS)*, 2020. 166, 167

[40] F. Glover and M. Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998. 137

[41] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. 137, 159

[42] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2672–2680, 2014. 131

[43] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu. Automated curriculum learning for neural networks. In *International Conference on Machine Learning (ICML)*, 2017. 154

[44] D. T. Green, T. J. Walsh, P. R. Cohen, and Y.-H. Chang. Learning a skill-teaching curriculum with dynamic Bayes nets. In *Innovative Applications of Artificial Intelligence (IAAI)*, 2011. 156, 157

[45] M. C. Green, B. Sergent, P. Shandilya, and V. Kumar. Evolutionarily-curated curriculum learning for deep reinforcement learning agents. In *AAAI Reinforcement Learning in Games Workshop*, 2019. 172

[46] S. Griffith, K. Subramanian, J. Scholz, C. Isbell, and A. L. Thomaz. Policy shaping: Integrating human feedback with reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2625–2633, 2013. 142

[47] Halfbrick Studios. Fruit ninja, Apr 2010. 167

[48] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1): 10–18, 2009. 57

[49] J. Hanna, P. Thomas, P. Stone, and S. Niekum. Data-efficient policy evaluation

through behavior policy search. In *International Conference on Machine Learning (ICML)*, August 2017. 151

[50] J. P. Hanna and P. Stone. Grounded action transformation for robot learning in simulation. In *Association for the Advancement of Artificial Intelligence*, pages 3834–3840, 2017. 171

[51] I.-A. Hosu and T. Rebedea. Playing Atari games with deep reinforcement learning and human checkpoint replay. In *Workshop on Evaluating General-Purpose AI (EGPAI)*, 2016. 124, 144

[52] A. Iglesias, P. Martínez, and F. Fernández. An experience applying reinforcement learning in a web-based adaptive and intelligent educational system. *Informatics in Education*, 2:223–240, 2003. 156, 157

[53] A. Iglesias, P. Martínez, R. Aler, and F. Fernández. Learning teaching strategies in an adaptive and intelligent educational system through reinforcement learning. *Applied Intelligence*, 31(1):89–106, 2009. 156, 157

[54] B. Ivanovic, J. Harrison, A. Sharma, M. Chen, and M. Pavone. Barc: Backward reachability curriculum for robotic reinforcement learning. In *International Conference on Robotics and Automation (ICRA)*, pages 15–21. IEEE, 2019. 124, 131

[55] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *International Conference on Learning Representations (ICLR)*, 2017. 132

[56] V. Jain and T. Tulabandhula. Faster reinforcement learning using active simulators. In *NIPS Workshop on Teaching Machines, Robots, and Humans*, 2017. 124, 137

[57] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002. 53

[58] L. Jiang, D. Meng, Q. Zhao, S. Shan, and A. G. Hauptmann. Self-paced curriculum learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2015. 155

[59] S. Kalyanakrishnan, Y. Liu, and P. Stone. Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. In *RoboCup-2006: Robot Soccer World Cup X*, volume 4434 of *Lecture Notes in Artificial Intelligence*, pages 72–85. Springer Verlag, Berlin, 2007. ISBN 978-3-540-74023-0. 40, 41

[60] A. Karpathy and M. Van De Panne. Curriculum learning for motor skills. In *Canadian Conference on Artificial Intelligence*, pages 325–330. Springer, 2012. 146, 149

[61] F. Khan, B. Mutlu, and X. Zhu. How do humans teach: On curriculum learning and teaching dimension. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1449–1457, 2011. 124, 142

[62] T.-H. Kim and J. Choi. Screenernet: Learning self-paced curriculum for deep neural networks. *arXiv preprint arXiv:1801.00904*, 2018. 124, 125

[63] W. B. Knox and P. Stone. Interactively shaping agents via human reinforcement: The TAMER framework. In *International Conference on Knowledge Capture*, 2009. 142

[64] W. B. Knox and P. Stone. Reinforcement learning from simultaneous human and MDP reward. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 475–482, 2012. 142

[65] G. Konidaris and A. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems*, 2009. 35, 75

[66] A. Lazaric. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pages 143–173. Springer, 2012. 1, 14

[67] A. Lazaric and M. Restelli. Transfer from multiple MDPs. In *Advances in Neural Information Processing Systems (NIPS)*, 2011. 14

[68] A. Lazaric, M. Restelli, and A. Bonarini. Transfer of samples in batch reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 544–551, 2008. 14, 50

[69] S. Y. Lee, C. Sungik, and S.-Y. Chung. Sample-efficient deep reinforcement learning via episodic backward update. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 2110–2119, 2019. 124, 126

[70] R. Loftin, B. Peng, J. MacGlashan, M. L. Littman, M. E. Taylor, J. Huang, and D. L. Roberts. Learning behaviors via human-delivered discrete feedback: modeling implicit feedback strategies to speed up learning. *Autonomous Agents and Multi-Agent Systems*, 30(1):30–59, 2016. 142, 143

[71] P. MacAlpine and P. Stone. Overlapping layered learning. *Artificial Intelligence*, 254: 21–43, 2018. 19, 22, 124, 141

[72] J. MacGlashan, M. K. Ho, R. Loftin, B. Peng, G. Wang, D. L. Roberts, M. E. Taylor, and M. L. Littman. Interactive learning from policy-dependent human feedback. In *International Conferences on Machine Learning (ICML)*, 2017. 142

[73] S. Mannor, I. Menache, A. Hoze, and U. Klein. Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pages 560–567, 2004. 33

[74] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher-student curriculum learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2017. 124, 135

[75] A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 361–368, 2001. 33

[76] I. Menache, S. Mannor, and N. Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *13th European Conference on Machine Learning*, pages 295–306. Springer, 2002. 33

[77] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015. 1, 13, 109, 123, 149

[78] T. Mu, S. Wang, E. Andersen, and E. Brunskill. Combining adaptivity with progression ordering for intelligent tutoring systems. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, pages 1–4, 2018. 165

[79] T. Mu, A. Jetten, and E. Brunskill. Towards suggesting actionable interventions for wheel-spinning students. In *Proceedings of The 13th International Conference on Educational Data Mining (EDM)*, 2020. 165

[80] S. Narvekar and P. Stone. Learning curriculum policies for reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2019. 82, 124, 136, 139

[81] S. Narvekar and P. Stone. Generalizing curricula for reinforcement learning. In *4th Lifelong Learning Workshop at ICML*, 2020. 103

[82] S. Narvekar, J. Sinapov, M. Leonetti, and P. Stone. Source task creation for curriculum learning. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, Singapore, 2016. 27, 120, 121, 137

[83] S. Narvekar, J. Sinapov, and P. Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 147, page 149, 2017. xvi, 22, 24, 68, 91, 95, 124, 135, 147

[84] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(181):1–50, 2020. 19, 115

[85] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning (ICML)*, volume 99, pages 278–287, 1999. 16

[86] T. Nguyen, T. Silander, and T. Y. Leong. Transferring expectations in model-based reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2555–2563, 2012. 50

[87] P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *The International Journal Of Production Research*, 26(1):35–62, 1988. 137

[88] B. Peng, J. MacGlashan, R. Loftin, M. L. Littman, D. L. Roberts, and M. E. Taylor. Curriculum design for machine learners in sequential decision tasks. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(4):268–277, 2018. 124, 143

[89] G. B. Peterson. A day of great illumination: B. F. Skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, 82(3):317–328, 2004. 1

[90] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta. Robust adversarial reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 2817–2826, 2017. 124, 128

[91] J. R. Quinlan et al. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, volume 92, pages 343–348. World Scientific, 1992. 57

[92] S. Racaniere, A. Lampinen, A. Santoro, D. Reichert, V. Firoiu, and T. Lillicrap. Automated curriculum generation through setter-solver interactions. In *International Conference on Learning Representations (ICLR)*, 2019. 124, 132

[93] A. N. Rafferty, E. Brunskill, T. L. Griffiths, and P. Shafto. Faster teaching via pomdp planning. *Cognitive Science*, 40(6):1290–1332, 2016. 158, 165

[94] A. Rajeswaran, S. Ghotra, B. Ravindran, and S. Levine. Epopt: Learning robust neural network policies using model ensembles. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017. 171

[95] A. Ramachandran and B. Scassellati. Adapting difficulty levels in personalized robot-child tutoring interactions. In *Workshop at the AAAI Conference on Artificial Intelligence*, 2014. 158

[96] Z. Ren, D. Dong, H. Li, and C. Chen. Self-paced prioritized curriculum learning with coverage penalty in deep reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, 29(6):2216–2226, 2018. 123, 124

[97] M. Riedmiller, R. Hafner, T. Lampe, M. Neunert, J. Degrave, T. van de Wiele, V. Mnih, N. Heess, and J. T. Springenberg. Learning by playing solving sparse reward tasks from scratch. In *International Conference on Machine Learning (ICML)*, pages 4344–4353, 2018. 124, 132, 145

[98] M. B. Ring. Child: A first step towards continual learning. *Machine Learning*, 28(1): 77–104, 1997. 152

[99] D. Robles and S. M. Lucas. A simple tree search method for playing ms. pac-man.

In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 249–255. IEEE, 2009. 37

[100] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary computation*, 5(1):1–29, 1997. 128

[101] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016. 146, 147

[102] P. Ruvolo and E. Eaton. ELLA: An efficient lifelong learning algorithm. In *International Conference on Machine Learning (ICML)*, 2013. 152

[103] P. Ruvolo and E. Eaton. Active task selection for lifelong machine learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2013. 153

[104] S. Schaal. Learning from demonstration. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1040–1046, 1997. 151

[105] T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal value function approximators. In *International Conference on Machine Learning (ICML)*, 2015. 104, 105, 125, 131

[106] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2016. 21, 123, 124

[107] J. Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in Psychology*, 4:313, 2013. 121

[108] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 11

[109] K. Shao, Y. Zhu, and D. Zhao. Starcraft micromanagement with reinforcement learning and curriculum transfer learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2018. 146

[110] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016. 1, 13, 115, 127, 129

[111] O. Simsek and A. G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pages 751–758, 2004. 33

[112] J. Sinapov, S. Narvekar, M. Leonetti, and P. Stone. Learning inter-task transferability in the absence of target task samples. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 725–733, 2015. 50, 146, 147

[113] B. F. Skinner. Reinforcement today. *American Psychologist*, 13(3):94, 1958. 1

[114] V. Soni and S. Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *American Association for Artificial Intelligence (AAAI)*, 2006. 14

[115] R. K. Srivastava, B. R. Steunebrink, and J. Schmidhuber. First experiments with powerplay. *Neural Networks*, 41:130 – 136, 2013. Special Issue on Autonomous Learning. 121

[116] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. Evolving neural network agents in the nero video game. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, Piscataway, NJ, 2005. 124, 141

[117] P. Stone and M. Veloso. Learning to solve complex planning problems: Finding useful auxiliary problems. In *AAAI Fall Symposium on Planning and Learning*, pages 137–141, 1994. 121

[118] H. B. Suay and S. Chernova. Effect of human guidance and state space size on interactive reinforcement learning. In *International Conference on Robot and Human Interactive Communication (RO-MAN)*, pages 1–6, 2011. 142

[119] K. Subramanian, C. L. Isbell Jr, and A. L. Thomaz. Exploration from demonstration for interactive reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 447–456, 2016. 142

[120] S. Sukhbaatar, Z. Li, I. Kostrikov, G. Synnaeve, A. Szlam, and R. Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. In *International Conference on Learning Representations (ICLR)*, 2018. 124, 127, 128

[121] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. 9, 11, 12, 38, 42, 54, 151

[122] R. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999. 31, 33

[123] M. Svetlik, M. Leonetti, J. Sinapov, R. Shah, N. Walker, and P. Stone. Automatic curriculum graph generation for reinforcement learning agents. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 2590–2596, 2017. xiv, xvi, xxi, 14, 16, 22, 24, 91, 97, 98, 99, 100, 101, 124, 138

[124] I. Szita and A. Lõrincz. Learning to play using low-complexity rule-based policies: Illustrations through ms. pac-man. *Journal of Artificial Intelligence Research*, 30:659–684, 2007. 37

[125] M. E. Taylor. Assisting transfer-enabled machine learning algorithms: Leveraging human knowledge for curriculum design. In *The AAAI Spring Symposium on Agents that Learn from Human Teachers*, 2009. 140

[126] M. E. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 53–59, New York, NY, 2005. ACM Press. xvi, 13, 14

[127] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009. xvi, 1, 14, 16, 17

[128] M. E. Taylor, P. Stone, and Y. Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(1):2125–2167, 2007. 14, 15

[129] M. E. Taylor, G. Kuhlmann, and P. Stone. Autonomous transfer for reinforcement learning. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008. 14

[130] M. E. Taylor, N. Carboni, A. Fachantidis, I. Vlahavas, and L. Torrey. Reinforcement learning agents providing advice in complex video games. *Connection Science*, 26(1): 45–63, 2014. 37, 38, 97

[131] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995. 115, 127, 129

[132] C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor. A deep hierarchical

approach to lifelong learning in minecraft. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1553–1561, 2017. 146, 149

[133] A. L. Thomaz and C. Breazeal. Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *Association for the Advancement of Artificial Intelligence (AAAI)*, volume 6, pages 1000–1005, 2006. 142

[134] S. Thrun. Lifelong learning algorithms. In S. Thrun and L. Pratt, editors, *Learning to Learn*, pages 181–209. Kluwer Academic Publishers, Norwell, MA, USA, 1998. 120

[135] A. Vezhnevets, V. Mnih, S. Osindero, A. Graves, O. Vinyals, J. Agapiou, et al. Strategic attentive writer for learning macro-actions. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3486–3494, 2016. 146, 149

[136] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, pages 1–5, 2019. 124, 127, 129

[137] L. S. Vygotsky. *Mind in Society: Development of Higher Psychological Processes*. Harvard University Press, 1978. 140

[138] W. Wang, T. Yang, Y. Liu, J. Hao, X. Hao, Y. Hu, Y. Chen, C. Fan, and Y. Gao. From few to more: Large-scale dynamic multiagent curriculum learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 7293–7300, 2020. 129, 146, 150

[139] Y. Wang. Inducing model trees for continuous classes. In *Proceedings of the European Conference on Machine Learning*, 1997. 57

[140] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992. 11

[141] C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989. 157

[142] D. Weinshall and D. Amir. Theory of curriculum learning, with convex loss functions. *arXiv preprint arXiv:1812.03472*, 2018. 172

[143] D. Weinshall, G. Cohen, and D. Amir. Curriculum learning by transfer learning: Theory and experiments with deep networks. In *International Conference on Machine Learning (ICML)*, pages 5235–5243, 2018. 172

[144] E. Wiewiora, G. W. Cottrell, and C. Elkan. Principled methods for advising reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 792–799, 2003. 16

[145] A. Wilson, A. Fern, S. Ray, and P. Tadepalli. Multi-task reinforcement learning: a hierarchical bayesian approach. In *International Conference on Machine Learning (ICML)*, pages 1015–1022. ACM, 2007. 152

[146] B. P. Woolf. *Building Intelligent Interactive Tutors: Student-centered Strategies for Revolutionizing e-Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. 156

[147] Y. Wu and Y. Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *International Conference on Learning Representations (ICLR)*, 2017. 19, 124, 133

[148] B.-H. Yang and H. Asada. Progressive learning and its application to robot impedance learning. *IEEE Transactions on Neural Networks*, 7(4):941–952, 1996. 146, 148

[149] J. Yang, A. Nakhaei, D. Isele, K. Fujimura, and H. Zha. Cm3: Cooperative multi-goal multi-stage multi-agent reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2020. 129, 146, 150

[150] M. Zimmer, Y. Boniface, and A. Dutech. Developmental reinforcement learning through sensorimotor space enlargement. In *International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 33–38. IEEE, 2018. 146, 148