An overview of recent work in learning to plan and hierarchical, multi-task reinforcement learning

Kiran Vodrahalli

Contents

1 Introduction

We would like to teach agents to execute complicated tasks, involving many sequential steps, and requiring a lot of precision. For instance, consider Zork, the popular text-based game from the 1980s. In the game, one must navigate through several mazes, acquire various items to solve puzzles, gain reward by gathering treasure, and avoid monsters and various other pitfalls. These kinds of settings are particularly challenging for reinforcement learning to solve. One approach to making such settings more tractable is to introduce prior knowledge: For instance, perhaps if one knows in advance that executing the task can be summarized into a simple, high-level outline, one might be able to use such information while learning to make the problem more tractable with respect to sample complexity. Our goal in this paper is to review methodology for making use of such information (broadly construed as high-level logic), which can be implemented as a finite-state automaton (FSA) or a small Markov Decision Process (MDP), in imitation learning and reinforcement learning algorithms.

In particular, we review papers focusing on "learning to plan", hierarchical reinforcement learning, multi-task learning, and meta learning in order to construct a broad view of the field and to identify new points of interest.

2 Reinforcement Learning and Imitation Learning

First we define the problem setting. In reinforcement learning, our goal is to learn a policy for agents to maximize reward in an environment $\mathcal{E}$. As is standard, we assume a Markov Decision Process model for the environment, instantiated by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$, representing state space, action space, reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and probabilistic transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. At each time step t, the agent is presented with some observations E_t which correspond to a state $s \in \mathcal{S}$, and we can imagine that each agent has an internal map $f_{\text{obs}} : \mathcal{E} \to \mathcal{S}$. For now, we assume f_{obs} is a bijection (we will later relax this assumption). Then, the agent may choose to take an action $a \in \mathcal{A}$, after which the transition function $\mathcal{P}$ will be applied and the agent will end up in a new state, while also receiving some reward according to the reward function $\mathcal{R}$. The goal is to learn a (potentially stochastic) policy $\pi : \mathcal{S} \to \mathcal{A}$ which maximizes reward in expectation: There are several variants of this goal. One can consider maximizing expected reward in a finite time setting, one can consider maximizing an expected *discounted reward* ($\mathbb{E}_\pi\left[\sum_{t=1}^{\infty} \gamma^t R_t\right]$, where $\gamma \in (0, 1)$ is a discount factor), or one can consider maximizing expected *average reward* (all time-steps are treated equally).

In the imitation learning setting, we do not get to see a reward function directly. Instead, we are provided with an *expert* oracle: At each state in the environment, we are allowed to see what the expert would have done. The goal is to learn a policy which *imitates* the expert. This problem can be solved with supervised learning; other approaches involve efficient data collection strategies which are then used to train in a supervised manner.

3 Recent Work on Learning to Plan

In recent years, there has been a lot of attention devoted to the "learning to plan" paradigm (Tamar et al. [2016], Srinivas et al. [2018]). One typically draws planning problems from a distribution over an environment (for instance, GridWorld – a simple world of blocks and empty space, where the goal is to start from some location and end up at another location while avoiding the obstacles. In this case, draws from the GridWorld planning problem distribution involve different obstacle configurations, and different start and end states.) Then the goal is to train a planning module on these different instances of tasks with hopes of generalizing to unseen tasks from the same distribution.

In Value Iteration Networks (VINs) (Tamar et al. [2016]), the authors are able to define a neural network architecture which is fully end-to-end differentiable while containing a planning module which approximates the well-known value iteration algorithm for solving known MDPs. The key idea is to learn a fake MDP which approximately describes the true MDP, and to then learn an *approximate value iteration algorithm* using the fake MDP. One uses the output of the approximate value iteration as an input features for the ultimate policy learned, in addition to information filtered by an attention mechanism about the current state. In particular, we learn functions f_P and f_R which map from observations of the environment to a fake transition matrix and a fake reward matrix. Then, by lifting the same state space and action space to the fake space, we complete the fake MDP. We can then run an approximate value iteration algorithm on top of this fake MDP, which outputs estimates of value for every state in the state space. We filter the value estimates with an attention mechanism, only focusing on a few values at a time for actual use in the policy. The resulting filtered features are then fed into a parametrized policy model, which can be trained using either imitation learning losses (e.g., there is an expert present) or reinforcement learning losses (one must learn from the environment instead). One special property of the value iteration network procedure is that the whole process can be made end-to-end-differentiable, which is desirable in the literature for implementation simplicity. We can approximate k iterations of value iteration by using a k-deep convolutional neural network. Note that the value iteration updates look like:

$$Q_n(s,a) := R(s,a) + \sum_{s'} \gamma P(s'|s,a) V_n(s')$$

$$V_{n+1}(s) = \max_a Q_n(s,a)$$

Essentially, we can take $\gamma P(s'|s,a)$ to be the convolution kernel, and the second-step to be max-pooling! For each action a, we have a separate filter $\gamma P(s'|s,a)$ in the case where the state space can be thought of as movement on a 2-dimensional grid, we essentially recover the exact setup of convolutional nets as used in computer vision.

A recent follow-up work, the Universal Planning Network (Srinivas et al. [2018]), is a more general riff on this general setup. Instead of using value iteration as the planning module, the UPN trains a "gradient descent planner" (GDP), which attempts to learn representations of the underlying control problem which are useful for planning.

4 Hierarchical and Multi-task RL

Hierarchical reinforcement learning, first introduced by Parr and Russell [1998], is the idea of adding prior knowledge and structure by supposing that the task one must solve can be broken down into sub-tasks. For instance, one can consider creating a finite-state controller H (known as a HAM in Parr and Russell [1998]) and take its cross-product with an MDP M to create a new MDP. This step allows us to augment the original MDP with a high-level control structure. Ideally, these new constraints help sample efficiency in learning problems.

Another common setup in reinforcement and imitation learning problems nowadays is the multi-task framework. In this framework, one typically has several tasks one wants to solve, in a typically shared environment, with either shared or related actions. One canonical example is a robot arm doing various tasks: Perhaps it has to pick up a variety of different kinds of objects (the variety of objects adding one level of the plurality of tasks), and perhaps do various things with them: Place them into containers, use them to transport water, etc.

Recent work has applied both of these frameworks in conjunction to yield algorithms which solve these kinds of tasks in practice and in simulation. In particular, in the deep reinforcement learning setting, one typically parametrizes policies with neural net parameters, allowing parameter sharing to easily occur across tasks with simple update rules derived from gradient descent. We will now see a few examples of this setup.

Shu et al. [2017] examines an approach for reducing the complexity of reinforcement learning through human-interpretable descriptions of various tasks. In particular, sub-policies are described with simple English, and the overall approach is validated in Minecraft. Each task is described with two words: an action, and an item. The hierarchical setup is as follows: They have a global policy π_k defined at stage k (which consists of a certain set of tasks $\mathcal{G}_k$ – this grows over time), which has four sub-policies: a base policy for the previously learned tasks, an instruction policy which communicates between the global policy and the base policy, an augmented flat policy which allows the global policy to execute actions, and a switch policy which decides whether to use the global policy or the flat policy. Here, the hierarchy takes on a very specific structure in order to reduce the overall complexity of the model. The instruction policy and the switch policy are in particular special in this paper, as they are agumented by a stochastic temporal grammar (STG). A stochastic temporal grammar encodes the idea that different tasks may have different temporal relations; e.g., one must pick up an object before moving it. In particular, the STG is used as a prior on which to base decisions about whether or not to use a base policy or to instead switch and use a primitive action at any given point. The STG is modeled as a finite state Markov chain, which is learned by maximum likelihood estimation.

Andreas et al. [2016] touches upon our main motivation of using "logical" priors to reduce the sample complexity of reinforcement learning by envisioning a multi-task learning setup with several tasks all taking place in the same state and action space. They further make the assumption that each task can be expressed as a sequence of simple sub-tasks from some common vocabulary of sub-tasks. The goal then becomes to identify which correct sequence of sub-tasks corresponds with each task in the environment. In particular, they make use of the notion that an algorithm designer can "sketch" the main idea of what the task looks like: For instance, (b_1, b_2, b_1, b_3), where each b_i is a symbol referring to some unknown sub-task. Note that the whole vocabulary of sub-tasks $\mathcal{B}$ is consistent across the tasks, and these sketches are provided as information with each task. The idea is that the *structure* of the sub-policies (e.g., the pattern 1-2-1-3) will help reduce the problem to learning sub-policies. Compare this setup with the hierarchical RL setup described in the previous section: In this paper, the authors only consider directed sequences of policies which come from some pool $\mathcal{B}$. The shared sub-task framework is also a departure from the hierarchical setup. One could imagine employing a hierarchical RL setup on top of the sub-tasks as well by sharing parameters across the subtasks, and effectively creating sub-sub-tasks.

The procedure for learning in Andreas et al. [2016] is follows: Tasks are output according to some curriculum (roughly speaking, simple tasks first, then more complicated tasks). At each iteration, one samples a task from the curriculum, and updates a policy using a policy gradient approach with advantage estimation. Note that the advantage must be separately estimated for each task separately, and thus each update also updates the advantage estimate, which is also parameterized according to some parameters. Thus, overall, one learns the sub-policies $b \in \mathcal{B}$ to enact. In order to rollout the policies for the actual tasks in the multi-task environment (this is also necessary to do in training in order to recover the training data), one simply composes the sub-policies in sequence given by the sketch: After sub-policy b_i in the sketch is executed, control is passed to the next sub-policy in the sketch, and so on.

5 A Logical Approach

We may want to provide even more specification about how to solve problems in order to reduce the sample complexity even further. For instance, instead of merely providing the structure of a set of sub-tasks which acts as a correct global policy, we may want to directly tell an actor how to accomplish a goal at a high-level.

A work by Kaplan et al. [2017] focuses on using natural language grounding in order to beat Atari games. In particular, the idea is one can provide structure through natural language directions. If one can learn a good representation of the langauge as well as the game state, then perhaps one can track completion of the tasks assigned via natural language and augment the reward of an environment (say, Montezuma's Revenge in Atari) via a task-completion sub-task reward. This approach yields improved performance on Montezuma's Revenge, which is considered the hardest Atari game. The key feature of this approach relies on the quality of the representations of the text and the game.

However, one may not want to work with embeddings directly, and instead rely on a purer logic structure. In Li et al. [2017b], the authors combine formal methods using logic (in particular, linear temporal logic, or LTL) with hierarchical reinforcement learning in order to reduce the sample complexity of RL tasks. In particular, the authors can specify tasks with complex logic which help define intrinsic rewards for learning. The benefit of using logic directly also shows via the ability to construct new skills from existing ones, by simply using the composition of the logical formulae in an appropriate way.

Linear temporal logic is a logic system which describes temporal operators augmented by standard Boolean logic. The idea is to be able to specify concepts like "always", "then", "next", "eventually", and "until" in a robust manner. The base logic propositions are augmented by a robustness function $\rho(s, \phi)$, which measures the level of satisfaction of a particular trajectory s with respect to an LTL formula ϕ as a real value. In previous work, the same authors developed an algorithm for finding a policy by searching over policies to find one with high robustness (Li et al. [2017a]). Notably, it is possible to directly translate from an LTL specification to an FSA formula with existing software packages. One can then look at the product of the resulting FSA with the MDP and apply standard reinforcement learning algorithms on top of it. This methodology works on top of a single environment – no planning is involved.

6 Future Work

Future work involves marrying these two areas together: We would like to learn to plan in a multi-task setting while additionally using logic to make use of hierarchical information.

References

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. *ArXiv e-prints*, 2016.

Russell Kaplan, Christopher Sauer, and Alexander Sosa. Beating atari with natural language guided reinforcement learning. *arXiv preprint arXiv:1704.05539*, 2017.

Xiao Li, Yao Ma, and Calin Belta. A policy search method for temporal logic specified reinforcement learning tasks. *arXiv preprint arXiv:1709.09611*, 2017a.

Xiao Li, Yao Ma, and Calin Belta. Automata guided hierarchical reinforcement learning for zero-shot skill composition. *arXiv preprint arXiv:1711.00129*, 2017b.

Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pages 1043–1049, 1998.

Tianmin Shu, Caiming Xiong, and Richard Socher. Hierarchical and interpretable skill acquisition in multi-task reinforcement learning. *arXiv preprint arXiv:1712.07294*, 2017.

Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Universal planning networks. *ArXiv e-prints*, 2018.

Aviv Tamar, YI WU, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *Advances in Neural Information Processing Systems 29*, pages 2154–2162, 2016.