Meta Learning for Control

by

Yan Duan

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Pieter Abbeel, Chair
Professor Peter L. Bartlett
Professor Stuart Russell
Assistant Professor Anne Collins
Doctor John Schulman

Fall 2017

Meta Learning for Control

Abstract

Meta Learning for Control

by

Yan Duan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Pieter Abbeel, Chair

In this thesis, we discuss meta learning for control: policy learning algorithms that can themselves generate algorithms that are highly customized towards a certain domain of tasks. The generated algorithms can be orders of magnitudes faster than human-designed, general purpose algorithms. We begin with a thorough review of existing policy learning algorithms for control, which motivates the need for better algorithms that can solve complicated tasks with affordable sample complexity. Then, we discuss two formulations of meta learning. The first formulation is meta learning for reinforcement learning, where the task is specified through a reward function, and the agent needs to improve its performance by acting in the environment, receiving scalar reward signals, and adjusting its strategy according to the information it receives. The second formulation is meta learning for imitation learning, where the task is specified through an expert demonstration of the task, and the agent needs to mimic the behavior of the expert to achieve good performance under new situations of the same task, as measured by the underlying objective of the expert (which is not directly given to the agent). We present practical algorithms for both formulations, and show that these algorithms can acquire sophisticated learning behaviors on par with learning algorithms designed by human experts, and can scale to complex, high-dimensional tasks. We also analyze their current limitations, including challenges associated with long horizons and imperfect demonstrations, which suggest important venues for future work. Finally, we conclude with several promising future directions of meta learning for control.

ACKNOWLEDGMENTS

I am extremely fortunate to be advised by Pieter Abbeel throughout my undergraduate and graduate research. His constant support and advice is what made this thesis possible.

I've also had the great fortune to work with John Schulman, who has shaped many of my ideas leading to this work, and closely collaborated with me on RL^2, a meta learning framework for reinforcement learning. Thanks to my other major collaborators for the work in this thesis, including Peter Chen, Rein Houthooft, Ilya Sutskever, Marcin Andrychowicz, Wojciech Zaremba, Bradly Stadie, and Jonathan Ho. Thanks to Peter Bartlett, Stuart Russell, and Anne Collins for serving on my quals and thesis committee. I am thankful to UC Berkeley and OpenAI for providing me the research freedom I needed, allowing me to exchange ideas with an extremely talented group of researchers. I would like to thank my supervisor at OpenAI, Wojciech Zaremba, as well as Ilya Sutskever, John Schulman, and Greg Brockman. I also collaborated with a number of colleagues at Berkeley and OpenAI on several projects not included in this document, including Chelsea Finn, Sergey Levine, Haoran Tang, Carlos Florensa, Josh Tobin, Sandy Huang, Prafulla Dhariwal, Cathy Wu, and Thanard Kurutach. Thanks to my collaborators during my undergraduate study, including Arjun Singh, Sachin Patil, and Ken Goldberg.

Finally, this thesis is dedicated to my wife, Zhaoyu Wang, and my parents, Qionghua Zhang and Luqiao Duan, for all the years of their love and support.

CONTENTS

1

INTRODUCTION

1.1 REINFORCEMENT LEARNING

Reinforcement learning (RL) studies algorithms for sequential decision problems, where an *agent* continually interacts with an *environment*, and should optimize its performance over time according to a desired metric.

Mathematically, we assume decisions are made at discrete intervals. At each time step t, the agent receives the current state s_t (or the current observation $o_t = f_{obs}(s_t)$, which makes the problem partially observable) from the environment, and computes an action a_t. Then, the environment receives and executes this action, and advances the state to s_{t+1} according to a transition probability function, $P(s_{t+1}|s_t, a_t)$. It also computes a scalar reward signal, r_t according to a reward function $R(s_t, a_t)$. Both s_{t+1} and r_t are sent back to the agent, and the loop continues. Some problems may optionally have a finite horizon T, after which the sequential process terminates. Additionally, a real-valued discount factor γ ($0 < \gamma \leqslant 1$) may be given, which injects a preference of rewards received sooner rather than later. The objective is to optimize the expected sum of discounted rewards over time:

$$\eta = \mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$$

where we allow $T = \infty$ to incorporate both the finite- and infinite-horizon cases.

Reinforcement learning is a general framework that can be used to study a wide range of problems. In robotic applications, the state can be the current positions and velocities of all objects in the system, the observation can include sensory information such as RGB images, point clouds, and joint angles of the robot, the action can be the torques applied to each joint, and the reward signal can be designed based on the task at hand: for example, to learn a controller that can make the robot move forward, the reward can be

the displacement of the robot in the forward direction. Computer games also constitute a natural category, where the state can include the current memory and CPU state, the observation can include the raw pixels of the current screen (or a concatenation of past few frames), the action can be the keyboard, mouse, or joystick inputs, and the reward can be the score of the game.

For a more thorough overview of reinforcement learning and its history, we refer the reader to R. S. Sutton and Barto (1998).

1.2 DEEP LEARNING

Deep learning employs powerful function approximators such as neural networks in order to learn representations from data. This stands in sharp contrast to traditional approaches in machine learning, which have typically required hand-crafted features. Originally conceived in the 70s and 80s (Werbos, 1974; Parker, 1985; LeCun, 1985; D. Williams and Hinton, 1986), deep learning has grown increasingly popular in recent years, achieving state-of-the-art performance in speech recognition, image classification, machine translation, and many other applications. We refer the reader to LeCun et al. (2015) for a high-level overview of the underlying techniques and recent advances in deep learning, and to Goodfellow et al. (2016) for a more comprehensive account of the subject.

1.3 DEEP REINFORCEMENT LEARNING

Deep reinforcement learning (Deep RL) studies reinforcement learning algorithms that make use of expressive function approximators such as neural networks. This allows the algorithm to scale up to high-dimensional sensory inputs and complex control logic, without requiring manual feature engineering or limiting oneself to simple, insufficiently expressive models. Its success can be traced back to the 90s, with the work by Tesauro (1994) demonstrating a neural-network-learned strategy achieving superior performance on the backgammon game. Recently, advances in deep learning have led to significant progress in Deep RL, with impressive applications such as learning to play Atari games from raw pixels (Mnih et al., 2013; Mnih et al., 2015), mastering the game of Go (Silver et al., 2016), acquiring advanced manipulation skills (Levine et al., 2016), and learning high-dimensional locomotion controllers (Schulman et al., 2015; Schulman et al., 2016; T. P. Lillicrap et al., 2016). In Chapter 2 we extensively benchmark recently proposed deep reinforcement learning algorithms for continuous control, and we refer the reader

to Y. Li (2017) for a more recent survey of the algorithms and applications of deep reinforcement learning.

1.4 META LEARNING

The success of deep learning is inseparable from the availability of vast amount of annotated data. For example, state-of-the-art image classifiers (Krizhevsky et al., 2012; Zeiler and Fergus, 2014; Szegedy et al., 2015; He et al., 2016) are typically trained on ImageNet (Deng et al., 2009; Russakovsky et al., 2015), a publicly available dataset consisting of millions of images labeled with corresponding object categories; Baidu's Deep Speech 2 system (Amodei et al., 2016) utilizes over 10,000 hours of speech data; top neural machine translation models (Wu et al., 2016) are trained on hundreds of millions of sentence pairs.

While deep learning systems have achieved great performance in the big data regime, there has been growing interest in reducing the amount of data required. For instance, as the largest e-commerce retailer in the U.S., Amazon now has over 300 million products in its inventory, and the number is growing at a pace of hundreds of thousands more per day (ScrapeHero, 2017). For an object detection system at this scale (which can be used for product recommendations), it is impractical to collect a large number of annotated examples per category, and it is essential to be able to adapt to new categories using very few samples.

As another example, personalized recommendation systems need to build up each user's profile from as little history per user as possible, so that they can tailor towards each user's interest from early on.

In these cases, what we want is not a static classifier, but instead an algorithm that can efficiently train new classifiers orders of magnitude more efficiently than typical deep learning systems. Such an algorithm likely needs to incorporate domain-specific information, so that it does not need to rely on new data to provide such knowledge. On the other hand, we want to avoid manually designing new algorithms for each such domain.

Meta learning, which learns a learning algorithm from data, is a promising approach towards resolving this dilemma. By training on a large number of related tasks, where each task may only have a small amount of annotated data, meta learning automatically produces a domain-specific algorithm that captures the common knowledge among training tasks, and can be used to solve new tasks quickly.

Pioneered by Ellis (1965); S. J. Russell (1987); Schmidhuber (1987); Thrun and Pratt

(1998); Naik and Mammone (1992); Schmidhuber et al. (1996); Baxter (2000), meta learning is not a new idea. However, compared to these early approaches, where the class of algorithms that may be meta-learned are often constrained, modern formulation of meta learning can take advantage of deep neural networks, which are expressive, scalable, and amenable for optimization. Here, a parameterized model specifies the algorithm at interest, and the parameters can be optimized with respect to an objective related to the algorithm's actual task performance.

Formally, a meta learning problem consists of a distribution over tasks $\mathfrak{T}$. Typically we have a collection of training tasks, $\mathfrak{T}_{\text{train}}$, and test tasks, $\mathfrak{T}_{\text{test}}$, which are drawn from $\mathfrak{T}$. At *meta-training* time[1], the parameters of a parameterized algorithm π_θ (often called a "fast" algorithm) are optimized with respect to a (meta-)training loss: $\mathbb{E}_{\mathcal{T}\sim\mathfrak{T}_{\text{train}}}[\mathcal{L}_{\text{train}}(\mathcal{T}, \pi_\theta)]$. At *meta-test* time, the algorithm is evaluated by a (meta-)test loss: $\mathbb{E}_{\mathcal{T}\sim\mathfrak{T}_{\text{test}}}[\mathcal{L}_{\text{test}}(\mathcal{T}, \pi_\theta)]$.

The above formulation is very general, and we will look at some examples below.

1.4.1 *Meta Learning for Supervised Learning*

In meta learning for supervised learning, also known as *few-shot learning*, the distribution over tasks consists of different mini-datasets, where each dataset may differ by the input samples, or their corresponding labels. Two commonly used benchmark datasets are (1) Omniglot (Lake et al., 2011), a dataset of thousands of handwritten characters, with 20 samples per character, and (2) ImageNet (Deng et al., 2009; Russakovsky et al., 2015) or a smaller version of it, MiniImageNet (Vinyals et al., 2016b), where different subsets of the object categories give rise to different mini-datasets.

After a specific task $\mathcal{T}$ is chosen, the algorithm is given a number of training examples consisting of input samples and their corresponding labels. The algorithm should then make predictions about test samples in $\mathcal{T}$. Typically, the training loss is chosen to be a differentiable surrogate, such as the cross-entropy loss, whereas the evaluation metric on test data is typically the classification accuracy.

The literature on few-shot learning exhibits great diversity. Overall we can divide them into three categories: metric-based, optimization-based, and fully generic models.

Metric-based models: This class of algorithms is based on learning an adaptive metric to find a small subset of training data most relevent to the test data, and predict the test label based on their corresponding labels. One specific instantiation makes use of a *siamese network* (Koch, 2015), which takes a pair of input samples and predicts whether

1 The prefix "meta-" is used to distinguish from the training and test phases that may be present within each specific task.

they are from the same class. At test time, as new sample arrives, it is compared with each training sample using the learned network, and the label of the training sample with the highest score is chosen. Several extensions to this framework are made. Shyam et al. (2017) use differentiable visual attentions based on DRAW (Gregor et al., 2015) to perform pairwise comparisons. Another work by Mehrotra and Dukkipati (2017) uses residual networks (He et al., 2016) to structure the siamese network. It also makes use of generative adversarial networks for regularization. At the time of writing, it is the current state of the art among all siamese-network-based architectures on Omniglot and MiniImageNet.

Another instantiation of this model structures the algorithm as performing weighted nearest neighbor in a embedding space, where the embedding can adapt itself based on the data available through a meta-learned mechanism. Vinyals et al. (2016b) use cosine distance as the similarity metric. More recently, Snell et al. (2017) propose to use squared Euclidean distance between the embeddings of the sample and the cluster center of each class, and observe improved performance.

Optimization-based models: Since most deep neural networks for supervised learning are trained via gradient descent, it is natural to incorporate similar elements into the fast algorithm. Munkhdalai and H. Yu (2017) propose an architecture that contains both slow weights, which are learned during the training process of meta learning, and fast weights, which are computed per task during testing. The fast weights are computed conditioned on gradient information with respect to an auxiliary loss. Ravi and Larochelle (2017) propose to learn the initial weights and an update rule by utilizing an LSTM (Hochreiter and Schmidhuber, 1997) , which receives gradient information as its input. Finn et al. (2017a) propose a simplified model that only learns the initial weights while using a fixed learning rate. More recently, Z. Li et al. (2017) propose an alternative model that learns both the initial weight and an elementwise learning rate. At the time of writing, it is the current state of the art among all optimization-based methods.

Fully generic models: Observe that the training-evaluation loop can be considered as a generic sequential prediction problem: first, the training samples and their labels are received sequentially; then, the model makes predictions about test samples as they arrive. Based on this observation, people have considered using generic recurrent architectures to represent the fast algorithm, without imposing specific assumptions on how it should behave. Early work by Baxter (1993); Hochreiter et al. (2001); Younger et al. (2001) train recurrent neural networks using backpropagation or evolution. However they only evaluate on low-dimensional synthetic datasets. More recently, Santoro et al. (2016) utilize memory-augmented architectures such as neural Turing machines, and ob-

serve improved performance over LSTM on more modern datasets including Omniglot and MiniImageNet. Mishra et al. (2017) further propose to use temporal convolutions and attentions, which have demonstrated great performance on generative modeling (Oord et al., 2016b; Oord et al., 2016a) and machine translation (Kalchbrenner et al., 2016). This simple model outperforms all other attempts to incorporate human-designed algorithmic components, and is the current state of the art.

1.4.2 *Meta Learning for Optimization*

It has long been observed that the choice of optimization algorithms and hyperparameters has significant effect on the overall convergence speed in training neural networks (Rumelhart et al., 1985). Modern update rules that incorporate first- and second-order statistics, such as momentum (Nesterov, 1983; Tseng, 1998), Rprop (Riedmiller and Braun, 1993), Adagrad (Duchi et al., 2011), RMSProp (Tieleman and Hinton, 2012), and Adam (D. Kingma and Ba, 2014), can often outperform plain gradient descent, although their merits have been recently debated (A. C. Wilson et al., 2017). Rather than hand-designing optimizers, we can also apply meta learning to optimization, where the goal is to obtain optimizers that can converge faster and to better solutions.

In this scenario, a task $\mathcal{T}$ consists of a particular neural network architecture, an initial set of parameters ϕ_0, and a loss function $\mathcal{L}(\phi)$, which is often assumed to be differentiable with respect to ϕ. The algorithm π_θ receives the current parameters ϕ_t, the current loss $\mathcal{L}(\phi_t)$, and its gradient $\frac{\partial L_{\mathcal{T}}(\phi)}{\partial \phi}(\phi_t)$ when available, and iteratively computes the new parameters ϕ_{t+1}. The goal is to minimize the final loss, $\mathcal{L}(\phi_T)$, which is typically chosen to be the meta-test loss. At meta-training time, a more shaped loss function is usually used, such as a weighted sum of all intermediate values: $\sum_{t=1}^{T} w_t \mathcal{L}(\phi_t)$, where greater emphasis can be imposed on later time steps as training proceeds.

Limited by computing resources, earlier work on meta learning for optimization typically consider simple forms of update rules, such as linear functions of hand-selected features (S. Bengio et al., 1992; Y. Bengio et al., 1990; Chalmers, 1990) or small neural networks (Naik and Mammone, 1992; Runarsson and Jonsson, 2000).

Benefiting from significant improvements in hardware, recent work has experimented with much larger-scale models. Andrychowicz et al. (2016) train an LSTM to control per-parameter update given each parameter's gradient information, with weight sharing across all parameters. The trained optimizer is evaluated on problems such as image classification and neural style transfer (Gatys et al., 2015). Wichrowska et al. (2017) further improve the scalability and generalizability of Andrychowicz et al. (2016) by employ-

ing a hierarchical recurrent architecture. Y. Chen et al. (2017) also consider optimizing non-differentiable black-box objectives using a similar formulation.

In addition to using gradient descent to learn a neural network optimizer, several alternatives have been considered. K. Li and Malik (2017) uses guided policy search (Levine and Abbeel, 2014) to train an LSTM. Bello et al. (2017) apply ideas presented in Baker et al. (2016); Zoph and Le (2016) to search among update rules expressible by simple programs.

1.4.3 *Other Applications*

In addition to the more widely studied applications of supervised learning and optimization, meta learning has also been applied to active learning (Woodward and Finn, 2017; Contardo et al., 2017), where a meta-learned algorithm decides which samples should be labeled next to receive incremental supervision, and generative models (Edwards and Storkey, 2017; Bartunov and Vetrov, 2016; Bornschein et al., 2017), where the meta-learned algorithm should be able to generate data from the same distribution after seeing only a few samples.

1.5 META LEARNING FOR CONTROL

Similar to supervised learning, learning algorithms for control often suffer from high sample complexity, which in this context is defined as the amount of experience that needs to be collected to achieve learning. This challenge is complicated by the following factors specific to control:

- **Exploration**: Rather than having access to a static dataset, the agent needs to take actions in the environment to build up its experience. To achieve the best sample complexity, the agent needs to efficiently explore different regions of the environment to extract most relevant information about the task. There has been vast literature on exploration (Kearns and S. Singh, 2002; Brafman and Tennenholtz, 2002; Jaksch et al., 2010; Ghavamzadeh et al., 2015; Sun et al., 2011; Kolter and A. Y. Ng, 2009; Pazis and Parr, 2013; Osband et al., 2014; R. S. Sutton and Barto, 1998; Schmidhuber, 2010; Oudeyer and Kaplan, 2009). However, practical algorithms are usually based on simple heuristics that aims to be generally applicable (B. C. Stadie et al., 2015; Oh et al., 2015; Osband et al., 2016; Montúfar et al., 2016; Mohamed and Rezende, 2015; Houthooft et al., 2016; Tang et al., 2017; Fortunato et al., 2017; Pathak et al., 2017; Fu et al., 2017; Plappert et al., 2017; M. Bellemare et al., 2016; Ostrovski

et al., 2017), which are far from optimal or how human explores different strategies when learning a new skill.

- **Credit Assignment**: Rather than having ground truth labels, the agent does not directly receive supervision about which actions it should take. Instead, a reward signal is provided that scores only the current state and action. Credit assignment is the process of figuring out which action(s) are contributing to the overall reward received by the agent (Hull, 1943; Minsky, 1961; R. S. Sutton and Barto, 1998). This is a challenging problem because all the past actions contribute to the current state. Usually, the choice of an RL algorithm prescribes a specific strategy for credit assignmentl, such as learning a baseline or a critic (R. J. Williams, 1992b; Konda and Tsitsiklis, 2000; Kimura, Kobayashi, et al., 2000; Greensmith et al., 2004; Wawrzyński, 2009; Hafner and Riedmiller, 2011).
- **Hierarchy**: As the problem complexity increases, it becomes substantially harder for the agent to efficiently explore its environment and assign credits to its past. Hierarchical RL (HRL) can greatly reduce the complexity by making the search space more structured. However, *tabula rasa* HRL faces a classic chicken-and-egg issue: without solving the task first, how can one know what a good hierarchical structure would be? Indeed, although recent attempts of generic hierarchical RL algorithms show that the agent can learn to make decisions hierarchically in high-dimensional tasks (A. Vezhnevets et al., 2016; Bacon et al., 2017; A. S. Vezhnevets et al., 2017; Dilokthanakul et al., 2017), we have yet to observe significant improvement in learning speed consistent across many environments.

Inspired by recent progress in meta learning for supervised learning, it is natural to consider similar techniques for control, or more specifically reinforcement learning and imitation learning. The central theme of this thesis is that a meta learning approach can potentially combine the representation power of deep neural networks, as well as the ability to incorporate domain-specific knowledge about how the learned algorithm should explore the environment, assign credits to past experience, make decisions hierarchically, and infer intentions from raw demonstrations.

The main contributions of this thesis are the following:

- In Chapter 2, we review recent advances of deep reinforcement learning algorithms for control. The evaluation results of recently published algorithms on a set of challenging benchmark tasks suggests that better algorithms need to be developed for improved exploration, hierarchical RL, and reduced sample complexity. Although we can continue developing better algorithms ourselves (and there has been great recent work in this direction, such as Z. Wang et al. (2016); Hessel et al. (2017); Wu

et al. (2017); Metz et al. (2017); Schulman et al. (2017)), fully solving all the challenges listed above would very likely require injecting more prior knowledge into how the algorithm behaves, which motivates the need for meta learning. This work was previously published as Y. Duan et al. (2016a).

- In Chapter 3, we propose a meta learning framework for reinforcement learning, RL^2. Following the terminology in Section 1.4, here each task $\mathcal{T}$ consists of a complete specification of an MDP. Both the meta-train loss $\mathcal{L}_{\text{train}}$ and the meta-test loss $\mathcal{L}_{\text{test}}$ are the performance of the agent across a fixed number of episodes on the given task. This work was previously published as (Y. Duan et al., 2016b).
- In Chapter 4, we propose a meta learning framework for imitation learning. In this framework, each task $\mathcal{T}$ consists of a full specification of an MDP except the reward, as well as a single demonstration. Both $\mathcal{L}_{\text{train}}$ and $\mathcal{L}_{\text{test}}$ are the performance of the agent on a single episode when conditioning on the demonstration. However, we cannot train on $\mathcal{L}_{\text{train}}$ directly since the reward is not given; instead, imitation learning algorithms develop various proxy objectives that can be optimized. This work was previously published as (Y. Duan et al., 2017).
- Finally, in Chapter 5, we conclude with possible future directions of meta learning for control.

2

BENCHMARKING DEEP REINFORCEMENT LEARNING

2.1 OVERVIEW

Reinforcement learning addresses the problem of how agents should learn to take actions to maximize cumulative reward through interactions with the environment. The traditional approach for reinforcement learning algorithms requires carefully chosen feature representations, which are usually hand-engineered.

Recently, significant progress has been made by combining advances in deep learning for learning feature representations (Krizhevsky et al., 2012; Hinton et al., 2012) with reinforcement learning, tracing back to much earlier work of Tesauro (1995) and Bertsekas and Tsitsiklis (1995). Notable examples are training agents to play Atari games based on raw pixels (Guo et al., 2014; Mnih et al., 2015; Schulman et al., 2015) and to acquire advanced manipulation skills using raw sensory inputs (Levine et al., 2016; T. P. Lillicrap et al., 2016; Watter et al., 2015a). Impressive results have also been obtained in training deep neural network policies for 3D locomotion and manipulation tasks (Schulman et al., 2015; Schulman et al., 2016; Heess et al., 2015b).

Along with this recent progress, the Arcade Learning Environment (ALE) (M. G. Bellemare et al., 2013) has become a popular benchmark for evaluating algorithms designed for tasks with high-dimensional state inputs and discrete actions. However, these algorithms do not always generalize straightforwardly to tasks with continuous actions, leading to a gap in our understanding. For instance, algorithms based on Q-learning quickly become infeasible when naive discretization of the action space is performed, due to the curse of dimensionality (Bellman, 1957; T. P. Lillicrap et al., 2016). In the continuous control domain, where actions are continuous and often high-dimensional, we argue that the existing control benchmarks fail to provide a comprehensive set of challenging problems (see Section 2.7 for a review of existing benchmarks). Benchmarks have played a significant role in other areas such as computer vision and speech recognition.

Examples include MNIST (LeCun et al., 1998), Caltech101 (Fei-Fei et al., 2006), CIFAR (Krizhevsky and Hinton, 2009), ImageNet (Deng et al., 2009), PASCAL VOC (Everingham et al., 2010), BSDS500 (Martin et al., 2001), SWITCHBOARD (Godfrey et al., 1992), TIMIT (Garofolo et al., 1993), Aurora (Hirsch and Pearce, 2000), and VoiceSearch (D. Yu et al., 2007). The lack of a standardized and challenging testbed for reinforcement learning and continuous control makes it difficult to quantify scientific progress. Systematic evaluation and comparison will not only further our understanding of the strengths of existing algorithms, but also reveal their limitations and suggest directions for future research.

We attempt to address this problem and present a benchmark consisting of 31 continuous control tasks. These tasks range from simple tasks, such as cart-pole balancing, to challenging tasks such as high-DOF locomotion, tasks with partial observations, and hierarchically structured tasks. Furthermore, a range of reinforcement learning algorithms are implemented on which we report novel findings based on a systematic evaluation of their effectiveness in training deep neural network policies. The benchmark and reference implementations are available at https://github.com/rllab/rllab, allowing for the development, implementation, and evaluation of new algorithms and tasks.

2.2 PRELIMINARIES

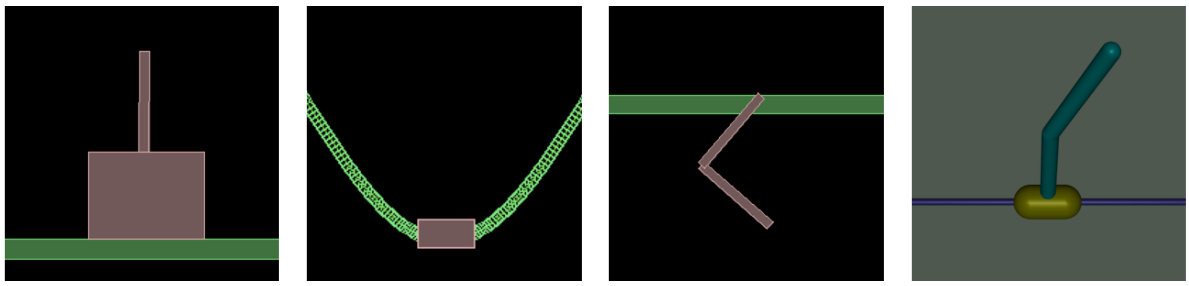
In this section, we define the notation used in subsequent sections.

The implemented tasks conform to the standard interface of a finite-horizon discounted Markov decision process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, P, r, \rho_0, \gamma, T)$, where $\mathcal{S}$ is a (possibly infinite) set of states, $\mathcal{A}$ is a set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_{\geqslant 0}$ is the transition probability distribution, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, $\rho_0 : \mathcal{S} \to \mathbb{R}_{\geqslant 0}$ is the initial state distribution, $\gamma \in (0, 1]$ is the discount factor, and T is the horizon.

For partially observable tasks, which conform to the interface of a partially observable Markov decision process (POMDP), two more components are required, namely Ω, a set of observations, and $\mathcal{O} : \mathcal{S} \times \Omega \to \mathbb{R}_{\geqslant 0}$, the observation probability distribution.

Most of our implemented algorithms optimize a stochastic policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{\geqslant 0}$. Let $\eta(\pi)$ denote its expected discounted reward: $\eta(\pi) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \right]$, where $\tau = (s_0, a_0, \ldots)$ denotes the whole trajectory, $s_0 \sim \rho_0(s_0)$, $a_t \sim \pi(a_t|s_t)$, and $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$.

For deterministic policies, we use the notation $\mu_\theta : \mathcal{S} \to \mathcal{A}$ to denote the policy instead. The objective for it has the same form as above, except that now we have $a_t = \mu(s_t)$.



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

Figure 1: Illustration of basic tasks: (a) Cart-Pole Balancing and Cart-Pole Swing Up; (b) Mountain Car; (c) Acrobot Swing Up; and (d) Double Inverted Pendulum Balancing.

2.3 TASKS

The tasks in the presented benchmark can be divided into four categories: basic tasks, locomotion tasks, partially observable tasks, and hierarchical tasks. For each task, we provide a brief description as well as motivation for including it in the testbed. More detailed specifications are given in the supplementary materials and in the source code.

We choose to implement all tasks using physics simulators rather than symbolic equations, since the former approach is less error-prone and permits easy modification of each task. Tasks with simple dynamics are implemented using Box2D (Catto, 2011), an open-source, freely available 2D physics simulator. Tasks with more complicated dynamics, such as locomotion, are implemented using MuJoCo (Todorov et al., 2012), a 3D physics simulator with better modeling of contacts.

2.3.1 *Basic Tasks*

We implement five basic tasks that have been widely analyzed in reinforcement learning and control literature.

Cart-Pole Balancing: This classic task in dynamics and control theory has been originally described by Stephenson (1908), and first studied in a learning context by Donaldson (1960), Widrow (1964), and Michie and Chambers (1968). An inverted pendulum is mounted on a pivot point on a cart. The cart itself is restricted to linear movement, achieved by applying horizontal forces. Due to the system's inherent instability, continuous cart movement is needed to keep the pendulum upright.

Cart-Pole Swing Up: A slightly more complex version of the previous task has been

proposed by Kimura and Kobayashi (1999) in which the system should not only be able to balance the pole, but first succeed in swinging it up into an upright position. This task extends the working range of the inverted pendulum to 360°. This is a nonlinear extension of the previous task (Doya, 2000).

Mountain Car: We implement a continuous version of the classic task described by Moore (1990). A car has to escape a valley by repetitive application of tangential forces. Because the maximal tangential force is limited, the car has to alternately drive up along the two slopes of the valley in order to build up enough inertia to overcome gravity. This brings a challenge of exploration, since before first reaching the goal among all trials, a locally optimal solution exists, which is to drive to the point closest to the target and stay there for the rest of the episode.

Acrobot Swing Up: In this widely-studied task an under-actuated, two-link robot has to swing itself into an upright position (DeJong and Spong, 1994; Murray and Hauser, 1991; Doya, 2000). It consists of two joints of which the first one has a fixed position and only the second one can exert torque. The goal is to swing the robot into an upright position and stabilize around that position. The controller not only has to swing the pendulum in order to build up inertia, similar to the Mountain Car task, but also has to decelerate it in order to prevent it from tipping over.

Double Inverted Pendulum Balancing: This task extends the Cart-Pole Balancing task by replacing the single-link pole by a two-link rigid structure. As in the former task, the goal is to stabilize the two-link pole near the upright position. This task is more difficult than single-pole balancing, since the system is even more unstable and requires the controller to actively maintain balance (Furuta et al., 1978).

2.3.2 *Locomotion Tasks*

In this category, we implement six locomotion tasks of varying dynamics and difficulty. The goal for all the tasks is to move forward as quickly as possible. These tasks are more challenging than the basic tasks due to high degrees of freedom. In addition, a great amount of exploration is needed to learn to move forward without getting stuck at local optima. Since we penalize for excessive controls as well as falling over, during the initial stage of learning, when the robot is not yet able to move forward for a sufficient distance without falling, apparent local optima exist including staying at the origin or diving forward cautiously.

Swimmer (Purcell, 1977; Coulom, 2002; Levine and Koltun, 2013; Schulman et al., 2015): The swimmer is a planar robot with 3 links and 2 actuated joints. Fluid is sim-

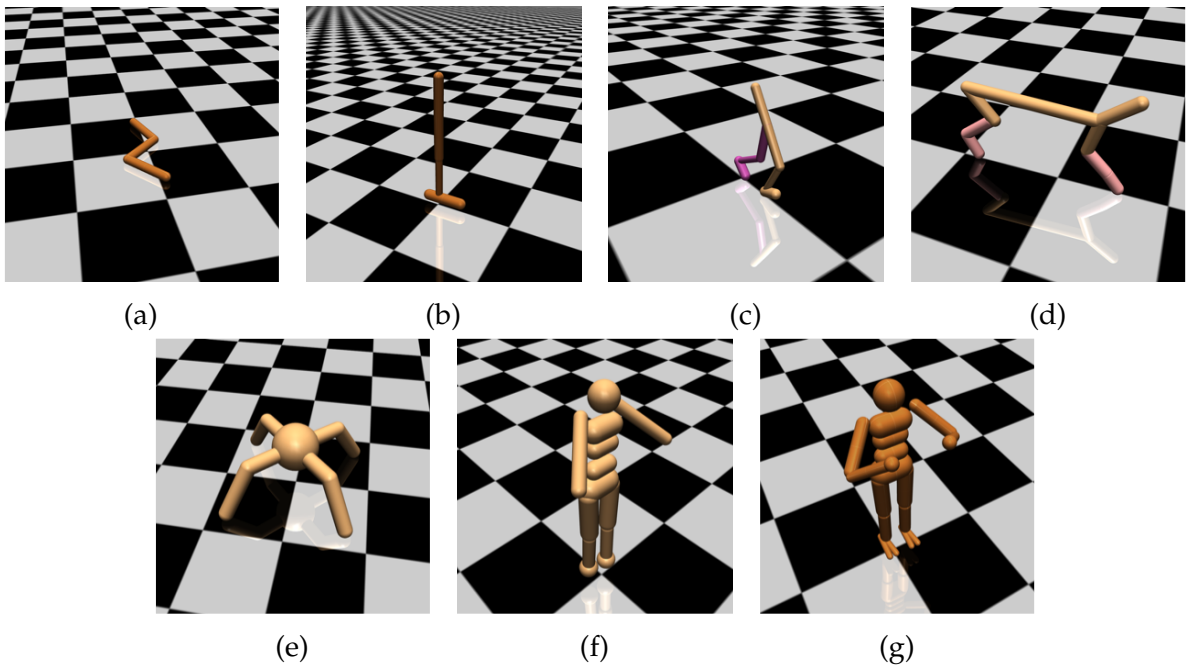


Figure 2: Illustration of locomotion tasks: (a) Swimmer; (b) Hopper; (c) Walker; (d) Half-Cheetah; (e) Ant; (f) Simple Humanoid; and (g) Full Humanoid.

ulated through viscosity forces, which apply drag on each link, allowing the swimmer to move forward. This task is the simplest of all locomotion tasks, since there are no irrecoverable states in which the swimmer can get stuck, unlike other robots which may fall down or flip over. This places less burden on exploration.

Hopper (Murthy and Raibert, 1984; Erez et al., 2011; Levine and Koltun, 2013; Schulman et al., 2015): The hopper is a planar monopod robot with 4 rigid links, corresponding to the torso, upper leg, lower leg, and foot, along with 3 actuated joints. More exploration is needed than the swimmer task, since a stable hopping gait has to be learned without falling. Otherwise, it may get stuck in a local optimum of diving forward.

Walker (Raibert and Hodgins, 1991; Erez et al., 2011; Levine and Koltun, 2013; Schulman et al., 2015): The walker is a planar biped robot consisting of 7 links, corresponding to two legs and a torso, along with 6 actuated joints. This task is more challenging than hopper, since it has more degrees of freedom, and is also prone to falling.

Half-Cheetah (Wawrzyński, 2007; Heess et al., 2015b): The half-cheetah is a planar biped robot with 9 rigid links, including two legs and a torso, along with 6 actuated joints.

Ant (Schulman et al., 2016): The ant is a quadruped with 13 rigid links, including four legs and a torso, along with 8 actuated joints. This task is more challenging than the previous tasks due to the higher degrees of freedom.

Simple Humanoid (Tassa et al., 2012; Schulman et al., 2016): This is a simplified humanoid model with 13 rigid links, including the head, body, arms, and legs, along with 10 actuated joints. The increased difficulty comes from the increased degrees of freedom as well as the need to maintain balance.

Full Humanoid (Tassa et al., 2012): This is a humanoid model with 19 rigid links and 28 actuated joints. It has more degrees of freedom below the knees and elbows, which makes the system higher-dimensional and harder for learning.

2.3.3 *Partially Observable Tasks*

In real-life situations, agents are often not endowed with perfect state information. This can be due to sensor noise, sensor occlusions, or even sensor limitations that result in partial observations. To evaluate algorithms in more realistic settings, we implement three variations of partially observable tasks for each of the five basic tasks described in Section 2.3.1, leading to a total of $5 \times 3 = 15$ additional tasks. These variations are described below.

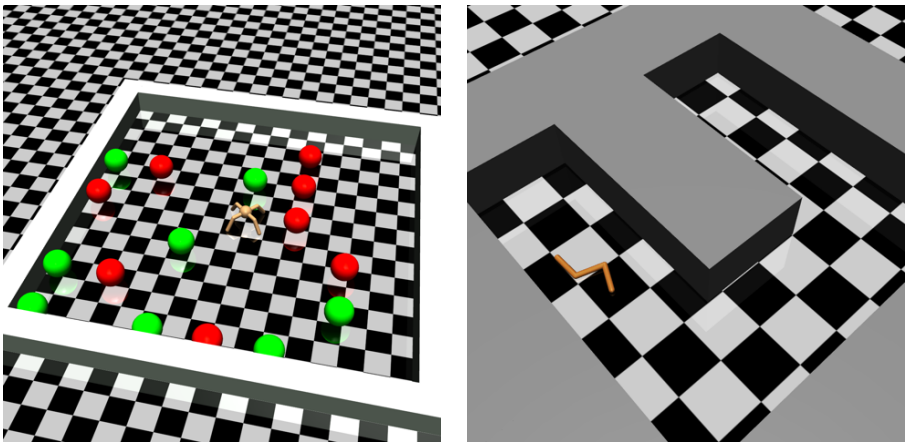
Limited Sensors: For this variation, we restrict the observations to only provide positional information (including joint angles), excluding velocities. An agent now has to learn to infer velocity information in order to recover the full state. Similar tasks have been explored in Gomez and Miikkulainen (1998); Schäfer and Udluft (2005); Heess et al. (2015a); Wierstra et al. (2007).

Noisy and Delayed Observations: In this case, sensor noise is simulated through the addition of Gaussian noise to the observations. We also introduce a time delay between taking an action and the action being in effect, accounting for physical latencies (Hester and Stone, 2013). Agents now need to learn to integrate both past observations and past actions to infer the current state. Similar tasks have been proposed in Bakker (2001).

System Identification: For this category, the underlying physical model parameters are varied across different episodes (Szita et al., 2003). The agents must learn to generalize across different models, as well as to infer the model parameters from its observation and action history.

2.3.4 *Hierarchical Tasks*

Many real-world tasks exhibit hierarchical structure, where higher level decisions can reuse lower level skills (Parr and S. Russell, 1998; R. S. Sutton et al., 1999; Dietterich, 2000). For instance, robots can reuse locomotion skills when exploring the environment. We propose several tasks where both low-level motor controls and high-level decisions are needed. Each of these two components operates on a different time scale and calls for a natural hierarchy in order to efficiently learn the task.



(a) (b)

Figure 3: Illustration of hierarchical tasks: (a) Locomotion + Food Collection; and (b) Locomotion + Maze.

Locomotion + Food Collection: For this task, the agent needs to learn to control either the swimmer or the ant robot to collect food and avoid bombs in a finite region. The agent receives range sensor readings about nearby food and bomb units. It is given a positive reward when it reaches a food unit, or a negative reward when it reaches a bomb.

Locomotion + Maze: For this task, the agent needs to learn to control either the swimmer or the ant robot to reach a goal position in a fixed maze. The agent receives range sensor readings about nearby obstacles as well as its goal (when visible). A positive reward is given only when the robot reaches the goal region.

2.4 ALGORITHMS

In this section, we briefly summarize the algorithms implemented in our benchmark, and note any modifications made to apply them to general parametrized policies. We implement a range of gradient-based policy search methods, as well as two gradient-free methods for comparison with the gradient-based approaches.

2.4.1 *Batch Algorithms*

Most of the implemented algorithms are batch algorithms. At each iteration, N trajectories $\{\tau_i\}_{i=1}^N$ are generated, where $\tau_i = \{(s_t^i, a_t^i, r_t^i)\}_{t=0}^T$ contains data collected along the ith trajectory. For on-policy gradient-based methods, all the trajectories are sampled under the current policy. For gradient-free methods, they are sampled under perturbed versions of the current policy.

REINFORCE (R. J. Williams, 1992a): This algorithm estimates the gradient of expected return $\nabla_\theta \eta(\pi_\theta)$ using the likelihood ratio trick:

$$\widehat{\nabla_\theta \eta(\pi_\theta)} = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi(a_t^i | s_t^i; \theta)(R_t^i - b_t^i),$$

where $R_t^i = \sum_{t'=t}^T \gamma^{t'-t} r_{t'}^i$ and b_t^i is a baseline that only depends on the state s_t^i to reduce variance. Hereafter, an ascent step is taken in the direction of the estimated gradient. This process continues until θ_k converges.

Truncated Natural Policy Gradient (TNPG) (Kakade, 2002; Peters et al., 2003; Bagnell and Schneider, 2003; Schulman et al., 2015): Natural Policy Gradient improves upon REINFORCE by computing an ascent direction that approximately ensures a small change in the policy distribution. This direction is derived to be $I(\theta)^{-1}\nabla_\theta \eta(\pi_\theta)$, where $I(\theta)$ is the Fisher information matrix (FIM). We use the step size suggested by Peters and Schaal (2008): $\alpha = \sqrt{\delta_{KL} \left(\nabla_\theta \eta(\pi_\theta)^T I(\theta)^{-1} \nabla_\theta \eta(\pi_\theta)\right)^{-1}}$. Finally, we replace $\nabla_\theta \eta(\pi_\theta)$ and $I(\theta)$ by their empirical estimates.

For neural network policies with tens of thousands of parameters or more, generic Natural Policy Gradient incurs prohibitive computation cost by forming and inverting the empirical FIM. Instead, we study Truncated Natural Policy Gradient (TNPG) in this chapter, which computes the natural gradient direction without explicitly forming the matrix inverse, using a conjugate gradient algorithm that only requires computing $I(\theta)v$ for arbitrary vector v. TNPG makes it practical to apply natural gradient in policy search

setting with high-dimensional parameters, and we refer the reader to Schulman et al. (2015) for more details.

Reward-Weighted Regression (RWR) (Peters and Schaal, 2007; Kober and Peters, 2009): This algorithm formulates the policy optimization as an Expectation-Maximization problem to avoid the need to manually choose learning rate, and the method is guaranteed to converge to a locally optimal solution. At each iteration, this algorithm optimizes a lower bound of the log-expected return: $\theta = \arg\max_{\theta'} \mathcal{L}(\theta')$, where

$$\mathcal{L}(\theta) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T} \log \pi(a_t^i | s_t^i; \theta) \rho(R_t^i - b_t^i)$$

Here, $\rho : \mathbb{R} \to \mathbb{R}_{\geqslant 0}$ is a function that transforms raw returns to nonnegative values. Following M. P. Deisenroth et al. (2013), we choose ρ to be $\rho(R) = R - R_{\min}$, where $R_{\min}$ is the minimum return among all trajectories collected in the current iteration.

Relative Entropy Policy Search (REPS) (Peters et al., 2010): This algorithm limits the loss of information per iteration and aims to ensure a smooth learning progress (M. P. Deisenroth et al., 2013). At each iteration, we collect all trajectories into a dataset $\mathcal{D} = \{(s_i, a_i, r_i, s_i')\}_{i=1}^{M}$, where M is the total number of samples. Then, we first solve for the dual parameters $[\eta^*, \nu^*] = \arg\min_{\eta', \nu'} g(\eta', \nu')$ s.t. $\eta > 0$, where

$$g(\eta, \nu) = \eta \delta_{KL} + \eta \log \left(\frac{1}{M} \sum_{i=1}^{M} e^{\delta_i(\nu)/\eta} \right).$$

Here $\delta_{KL} > 0$ controls the step size of the policy, and $\delta_i(\nu) = r_i + \nu^T(\phi(s_i') - \phi(s_i))$ is the sample Bellman error. We then solve for the new policy parameters:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{M} \sum_{i=1}^{M} e^{\delta_i(\nu^*)/\eta^*} \log \pi(a_i | s_i; \theta).$$

Trust Region Policy Optimization (TRPO) (Schulman et al., 2015): This algorithm allows more precise control on the expected policy improvement than TNPG through the introduction of a surrogate loss. At each iteration, we solve the following constrained optimization problem (replacing expectations with samples):

$$\begin{aligned} \text{maximize}_{\theta} \quad & \mathbb{E}_{s \sim \rho_{\theta_k}, a \sim \pi_{\theta_k}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A_{\theta_k}(s, a) \right] \\ \text{s.t.} \quad & E_{s \sim \rho_{\theta_k}} [D_{KL}(\pi_{\theta_k}(\cdot|s) \| \pi_\theta(\cdot|s))] \leqslant \delta_{KL} \end{aligned}$$

where $\rho_\theta = \rho_{\pi_\theta}$ is the discounted state-visitation frequencies induced by π_θ, $A_{\theta_k}(s, a)$, known as the advantage function, is estimated by the empirical return minus the baseline, and δ_{KL} is a step size parameter which controls how much the policy is allowed to change per iteration. We follow the procedure described in the original paper for solving the optimization, which results in the same descent direction as TNPG with an extra line search in the objective and KL constraint.

Cross Entropy Method (CEM) (Rubinstein, 1999; Szita and Lőrincz, 2006): Unlike previously mentioned methods, which perform exploration through stochastic actions, CEM performs exploration directly in the policy parameter space. At each iteration, we produce N perturbations of the policy parameter: $\theta_i \sim \mathcal{N}(\mu_k, \Sigma_k)$, and perform a rollout for each sampled parameter. Then, we compute the new mean and diagonal covariance using the parameters that correspond to the top q-quantile returns.

Covariance Matrix Adaption Evolution Strategy (CMA-ES) (Hansen and Ostermeier, 2001): Similar to CEM, CMA-ES is a gradient-free evolutionary approach for optimizing nonconvex objective functions. In our case, this objective function equals the average sampled return. In contrast to CEM, CMA-ES estimates the covariance matrix of a multivariate normal distribution through incremental adaption along evolution paths, which contain information about the correlation between consecutive updates.

2.4.2 *Online Algorithms*

Deep Deterministic Policy Gradient (DDPG) (T. P. Lillicrap et al., 2016): Compared to batch algorithms, the DDPG algorithm continuously improves the policy as it explores the environment. It applies gradient descent to the policy with minibatch data sampled from a replay pool, where the gradient is computed via

$$\widehat{\nabla_\theta \eta(\mu_\theta)} = \sum_{i=1}^{B} \nabla_a Q_\phi(s_i, a)\big|_{a=\mu_\theta(s_i)} \nabla_\theta \mu_\theta(s_i)$$

where B is the batch size. The critic Q is trained via gradient descent on the ℓ^2 loss of the Bellman error $L = \frac{1}{B}\sum_{i=1}^{B}(y_i - Q_\phi(s_i, a_i))^2$, where $y_i = r_i + \gamma Q'_{\phi'}(s'_i, \mu'_{\theta'}(s'_i))$. To improve stability of the algorithm, we use target networks for both the critic and the policy when forming the regression target y_i. We refer the reader to T. P. Lillicrap et al. (2016) for a more detailed description of the algorithm.

2.4.3 *Recurrent Variants*

We implement direct applications of the aforementioned batch-based algorithms to recurrent policies. The only modification required is to replace $\pi(a_t^i|s_t^i)$ by $\pi(a_t^i|o_{1:t}^i, a_{1:t-1}^i)$, where $o_{1:t}^i$ and $a_{1:t-1}$ are the histories of past and current observations and past actions. Recurrent versions of reinforcement learning algorithms have been studied in many existing works, such as Bakker (2001), Schäfer and Udluft (2005), Wierstra et al. (2007), and Heess et al. (2015a).

2.5 EXPERIMENT SETUP

In this section, we elaborate on the experimental setup used to generate the results.

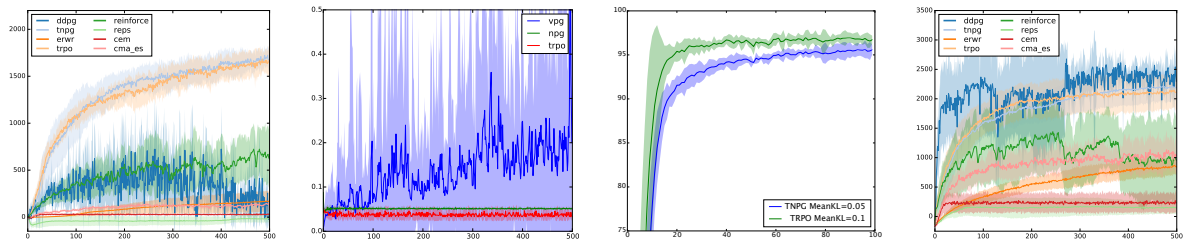
Performance Metrics: For each report unit (a particular algorithm running on a particular task), we define its performance as $\frac{1}{\sum_{i=1}^{I} N_i} \sum_{i=1}^{I} \sum_{n=1}^{N_i} R_{in}$, where I is the number of training iterations, N_i is the number of trajectories collected in the ith iteration, and R_{in} is the undiscounted return for the nth trajectory of the ith iteration,

Hyperparameter Tuning: For the DDPG algorithm, we used the hyperparametes reported in T. P. Lillicrap et al. (2016). For the other algorithms, we follow the approach in Mnih et al. (2015), and we select two tasks in each category, on which a grid search of hyperparameters is performed. Each choice of hyperparameters is executed under five random seeds. The criterion for the best hyperparameters is defined as $\text{mean}(\text{returns}) - \text{std}(\text{returns})$. This metric selects against large fluctuations of performance due to overly large step sizes.

For the other tasks, we try both of the best hyperparameters found in the same category, and report the better performance of the two. This gives us insights into both the maximum possible performance when extensive hyperparameter tuning is performed, and the robustness of the best hyperparameters across different tasks.

Policy Representation: For basic, locomotion, and hierarchical tasks and for batch algorithms, we use a feed-forward neural network policy with 3 hidden layers, consisting of 100, 50, and 25 hidden units with tanh nonlinearity at the first two hidden layers, which map each state to the mean of a Gaussian distribution. The log-standard deviation is parameterized by a global vector independent of the state, as done in Schulman et al. (2015). For all partially observable tasks, we use a recurrent neural network with a single hidden layer consisting of 32 LSTM hidden units (Hochreiter and Schmidhuber, 1997).

For the DDPG algorithm which trains a deterministic policy, we follow T. P. Lillicrap et al. (2016). For both the policy and the Q function, we use the same architecture of



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

Figure 4: Performance as a function of the number of iterations; the shaded area depicts the mean $\pm$ the standard deviation over five different random seeds: (a) Performance comparison of all algorithms in terms of the average reward on the Walker task; (b) Comparison between REINFORCE, TNPG, and TRPO in terms of the mean KL-divergence on the Walker task; (c) Performance comparison on TNPG and TRPO on the Swimmer task; (d) Performance comparison of all algorithms in terms of the average reward on the Half-Cheetah task.

a feed-forward neural network with 2 hidden layers, consisting of 400 and 300 hidden units with relu activations.

Baseline: For all gradient-based algorithms except REPS, we can subtract a baseline from the empirical return to reduce variance of the optimization. We use a linear function as the baseline with a time-varying feature vector.

2.6 RESULTS

The main evaluation results are presented in Table 1. The tasks on which the grid search is performed are marked with (*). In each entry, the pair of numbers shows the mean and standard deviation of the normalized cumulative return using the best possible hyperparameters.

REINFORCE: Despite its simplicity, REINFORCE is an effective algorithm in optimizing deep neural network policies in most basic and locomotion tasks. Even for high-DOF tasks like Ant, REINFORCE can achieve competitive results. However we observe that REINFORCE sometimes suffers from premature convergence to local optima as noted by Peters and Schaal (2008), which explains the performance gaps between REINFORCE and TNPG on tasks such as Walker (Fig. 4a). By visualizing the final policies, we can see that REINFORCE results in policies that tend to jump forward and fall over to maximize short-term return instead of acquiring a stable walking gait to maximize long-term

Table 1: Performance of the implemented algorithms in terms of average return over all training iterations for five different random seeds (same across all algorithms). The results of the best-performing algorithm on each task, as well as all algorithms that have performances that are not statistically significantly different (Welch's t-test with $p < 0.05$), are highlighted in boldface.[a] In the tasks column, the partially observable variants of the tasks are annotated as follows: LS stands for limited sensors, NO for noisy observations and delayed actions, and SI for system identifications. The notation N/A denotes that an algorithm has failed on the task at hand, e.g., CMA-ES leading to out-of-memory errors in the Full Humanoid task.

Task	Random	REINFORCE	TNPG	RWR	REPS	TRPO	CEM	CMA-ES	DDPG[b]
Cart-Pole Balancing	77.1 ± 0.0	4693.7 ± 14.0	**3986.4 ± 748.9**	**4861.5 ± 12.3**	565.6 ± 137.6	**4869.8 ± 37.6**	4815.4 ± 4.8	2440.4 ± 568.3	4634.4 ± 87.8
Inverted Pendulum*	−153.4 ± 0.2	13.4 ± 18.0	**209.7 ± 55.5**	84.7 ± 13.8	−113.3 ± 4.6	**247.2 ± 76.1**	38.2 ± 25.7	−40.1 ± 5.7	40.0 ± 244.6
Mountain Car	−415.4 ± 0.0	−67.1 ± 1.0	**-66.5 ± 4.5**	−79.4 ± 1.1	−275.6 ± 166.3	**-61.7 ± 0.9**	−66.0 ± 2.4	−85.0 ± 7.7	−288.4 ± 170.3
Acrobot	−1904.5 ± 1.0	−508.1 ± 91.0	−395.8 ± 121.2	−352.7 ± 35.9	−1001.5 ± 10.8	−326.0 ± 24.4	−436.8 ± 14.7	−785.6 ± 13.1	**-223.6 ± 5.8**
Double Inverted Pendulum*	149.7 ± 0.1	4116.5 ± 65.2	**4455.4 ± 37.6**	3614.8 ± 368.1	446.7 ± 114.8	**4412.4 ± 50.4**	2566.2 ± 178.9	1576.1 ± 51.3	2863.4 ± 154.0
Swimmer*	−1.7 ± 0.1	92.3 ± 0.1	**96.0 ± 0.2**	60.7 ± 5.5	3.8 ± 3.3	**96.0 ± 0.2**	68.8 ± 2.4	64.9 ± 1.4	85.8 ± 1.8
Hopper	8.4 ± 0.0	714.0 ± 29.3	**1155.1 ± 57.9**	553.2 ± 71.0	86.7 ± 17.6	**1183.3 ± 150.0**	63.1 ± 7.8	20.3 ± 14.3	267.1 ± 43.5
2D Walker	−1.7 ± 0.0	506.5 ± 78.8	**1382.6 ± 108.2**	136.0 ± 15.9	−37.0 ± 38.1	**1353.8 ± 85.0**	84.5 ± 19.2	77.1 ± 24.3	318.4 ± 181.6
Half-Cheetah	−90.8 ± 0.3	1183.1 ± 69.2	**1729.5 ± 184.6**	376.1 ± 28.2	34.5 ± 38.0	**1914.0 ± 120.1**	330.4 ± 274.8	441.3 ± 107.6	**2148.6 ± 702.7**
Ant*	13.4 ± 0.7	548.3 ± 55.5	**706.0 ± 127.7**	37.6 ± 3.1	39.0 ± 9.8	**730.2 ± 61.3**	49.2 ± 5.9	17.8 ± 15.5	326.2 ± 20.8
Simple Humanoid	41.5 ± 0.2	128.1 ± 34.0	**255.0 ± 24.5**	93.3 ± 17.4	28.3 ± 4.7	**269.7 ± 40.3**	60.6 ± 12.9	28.7 ± 3.9	99.4 ± 28.1
Full Humanoid	13.2 ± 0.1	262.2 ± 10.5	**288.4 ± 25.2**	46.7 ± 5.6	41.7 ± 6.1	**287.0 ± 23.4**	36.9 ± 2.9	N/A ± N/A	119.0 ± 31.2
Cart-Pole Balancing (LS)*	77.1 ± 0.0	420.9 ± 265.5	**945.1 ± 27.8**	68.9 ± 1.5	898.1 ± 22.1	**960.2 ± 46.0**	227.0 ± 223.0	68.0 ± 1.6	
Inverted Pendulum (LS)	−122.1 ± 0.1	−13.4 ± 3.2	**0.7 ± 6.1**	−107.4 ± 0.2	−87.2 ± 8.0	**4.5 ± 4.1**	−81.2 ± 33.2	−62.4 ± 3.4	
Mountain Car (LS)	−83.0 ± 0.0	−81.2 ± 0.6	**-65.7 ± 9.0**	−81.7 ± 0.1	−82.6 ± 0.4	**-64.2 ± 9.5**	**-68.9 ± 1.3**	**-73.2 ± 0.6**	
Acrobot (LS)*	−393.2 ± 0.0	−128.9 ± 11.6	**-84.6 ± 2.9**	−235.9 ± 5.3	−379.5 ± 1.4	**-83.3 ± 9.9**	−149.5 ± 15.3	−159.9 ± 7.5	
Cart-Pole Balancing (NO)*	101.4 ± 0.1	616.0 ± 210.8	**916.3 ± 23.0**	93.8 ± 1.2	99.6 ± 7.2	606.2 ± 122.2	181.4 ± 32.1	104.4 ± 16.0	
Inverted Pendulum (NO)	−122.2 ± 0.1	6.5 ± 1.1	**11.5 ± 0.5**	−110.0 ± 1.4	−119.3 ± 4.2	**10.4 ± 2.2**	−55.6 ± 16.7	−80.3 ± 2.8	
Mountain Car (NO)	−83.0 ± 0.0	−74.7 ± 7.8	**-64.5 ± 8.6**	−81.7 ± 0.1	−82.9 ± 0.1	**-60.2 ± 2.0**	−67.4 ± 1.4	−73.5 ± 0.5	
Acrobot (NO)*	−393.5 ± 0.0	**-186.7 ± 31.3**	**-164.5 ± 13.4**	−233.1 ± 0.4	−258.5 ± 14.0	**-149.6 ± 8.6**	−213.4 ± 6.3	−236.6 ± 6.2	
Cart-Pole Balancing (SI)*	76.3 ± 0.1	431.7 ± 274.1	**980.5 ± 7.3**	69.0 ± 2.8	702.4 ± 196.4	**980.3 ± 5.1**	746.6 ± 93.2	71.6 ± 2.9	
Inverted Pendulum (SI)	−121.8 ± 0.2	−5.3 ± 5.6	**14.8 ± 1.7**	−108.7 ± 4.7	−92.8 ± 23.9	**14.1 ± 0.9**	−51.8 ± 10.6	−63.1 ± 4.8	
Mountain Car (SI)	−82.7 ± 0.0	−63.9 ± 0.2	**-61.8 ± 0.4**	−81.4 ± 0.1	−80.7 ± 2.3	**-61.6 ± 0.4**	−63.9 ± 1.0	−66.9 ± 0.6	
Acrobot (SI)*	−387.8 ± 1.0	**-169.1 ± 32.3**	**-156.6 ± 38.9**	−233.2 ± 2.6	−216.1 ± 7.7	**-170.9 ± 40.3**	−250.2 ± 13.7	−245.0 ± 5.5	
Swimmer + Gathering	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Ant + Gathering	−5.8 ± 5.0	−0.1 ± 0.1	−0.4 ± 0.1	−5.5 ± 0.5	−6.7 ± 0.7	−0.4 ± 0.0	−4.7 ± 0.7	N/A ± N/A	−0.3 ± 0.3
Swimmer + Maze	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Ant + Maze	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	N/A ± N/A	0.0 ± 0.0

[a] Except for the hierarchical tasks

[b] Recurrent variant of DDPG (Heess et al., 2015a) not included, since it requires significant modifications to DDPG.

return. In Fig. 4b, we can observe that even with a small learning rate, steps taken by REINFORCE can sometimes result in large changes to policy distribution, which may explain the fast convergence to local optima.

TNPG and TRPO: Both TNPG and TRPO outperform other batch algorithms by a large margin on most tasks, confirming that constraining the change in the policy distribution results in more stable learning (Peters and Schaal, 2008).

Compared to TNPG, TRPO offers better control over each policy update by performing a line search in the natural gradient direction to ensure an improvement in the surrogate loss function. We observe that hyperparameter grid search tends to select conservative step sizes (δ_{KL}) for TNPG, which alleviates the issue of performance collapse caused by a large update to the policy. By contrast, TRPO can robustly enforce constraints with larger a δ_{KL} value and hence speeds up learning in some cases. For instance, grid search on the Swimmer task reveals that the best step size for TNPG is $\delta_{\mathrm{KL}} = 0.05$, whereas TRPO's best step-size is larger: $\delta_{\mathrm{KL}} = 0.1$. As shown in Fig. 4c, this larger step size enables slightly faster learning.

RWR: RWR is the only gradient-based algorithm we implemented that does not require any hyperparameter tuning. It can solve some basic tasks to a satisfactory degree, but fails to solve more challenging tasks such as locomotion. We observe empirically that RWR shows fast initial improvement followed by significant slow-down, as shown in Fig. 4d.

REPS: Our main observation is that REPS is especially prone to early convergence to local optima in case of continuous states and actions. Its final outcome is greatly affected by the performance of the initial policy, an observation that is consistent with the original work of Peters et al. (2010). This leads to a bad performance on average, although under particular initial settings the algorithm can perform on par with others. Moreover, the tasks presented here do not assume the existence of a stationary distribution, which is assumed in Peters et al. (2010). In particular, for many of our tasks, transient behavior is of much greater interest than steady-state behavior, which agrees with previous observation by Hoof et al. (2015),

Gradient-free methods: Surprisingly, even when training deep neural network policies with thousands of parameters, CEM achieves very good performance on certain basic tasks such as Cart-Pole Balancing and Mountain Car, suggesting that the dimension of the searching parameter is not always the limiting factor of the method. However, the performance degrades quickly as the system dynamics becomes more complicated. We also observe that CEM outperforms CMA-ES, which is remarkable as CMA-ES estimates the full covariance matrix. For higher-dimensional policy parameterizations, the compu-

tational complexity and memory requirement for CMA-ES become noticeable. On tasks with high-dimensional observations, such as the Full Humanoid, the CMA-ES algorithm runs out of memory and fails to yield any results, denoted as N/A in Table 1.

DDPG: Compared to batch algorithms, we found that DDPG was able to converge significantly faster on certain tasks like Half-Cheetah due to its greater sample efficiency. However, it was less stable than batch algorithms, and the performance of the policy can degrade significantly during training. We also found it to be more susceptible to scaling of the reward. In our experiment for DDPG, we rescaled the reward of all tasks by a factor of 0.1, which seems to improve the stability.

Partially Observable Tasks: We experimentally verify that recurrent policies can find better solutions than feed-forward policies in Partially Observable Tasks but recurrent policies are also more difficult to train. As shown in Table 1, derivative-free algorithms like CEM and CMA-ES work considerably worse with recurrent policies. Also we note that the performance gap between REINFORCE and TNPG widens when they are applied to optimize recurrent policies, which can be explained by the fact that a small change in parameter space can result in a bigger change in policy distribution with recurrent policies than with feedforward policies.

Hierarchical Tasks: We observe that all of our implemented algorithms achieve poor performance on the hierarchical tasks, even with extensive hyperparameter search and 500 iterations of training. It is an interesting direction to develop algorithms that can automatically discover and exploit the hierarchical structure in these tasks.

2.7 RELATED WORK

In this section, we review existing benchmarks of continuous control tasks. The earliest efforts of evaluating reinforcement learning algorithms started in the form of individual control problems described in symbolic form. Some widely adopted tasks include the inverted pendulum (Stephenson, 1908; Donaldson, 1960; Widrow, 1964), mountain car (Moore, 1990), and Acrobot (DeJong and Spong, 1994). These problems are frequently incorporated into more comprehensive benchmarks.

Some reinforcement learning benchmarks contain low-dimensional continuous control tasks, such as the ones introduced above, including RLLib (Abeyruwan, 2013), MMLF (Metzen and Edgington, 2011), RL-Toolbox (Neumann, 2006), JRLF (Kochenderfer, 2006), Beliefbox (Dimitrakakis et al., 2007), Policy Gradient Toolbox (Peters, 2002), and ApproxRL (Busoniu, 2010). A series of RL competitions has also been held in recent years (Dutech et al., 2005; Dimitrakakis et al., 2014), again with relatively low-dimensional ac-

tions. In contrast, our benchmark contains a wider range of tasks with high-dimensional continuous state and action spaces.

Previously, other benchmarks have been proposed for high-dimensional control tasks. Tdlearn (Dann et al., 2014) includes a 20-link pole balancing task, DotRL (Papis and Wawrzyński, 2013) includes a variable-DOF octopus arm and a 6-DOF planar cheetah model, PyBrain (Schaul et al., 2010) includes a 16-DOF humanoid robot with standing and jumping tasks, RoboCup Keepaway (Stone et al., 2005) is a multi-agent game which can have a flexible dimension of actions by varying the number of agents, and SkyAI (Yamaguchi and Ogasawara, 2010) includes a 17-DOF humanoid robot with crawling and turning tasks. Other libraries such as CL-Square (Riedmiller et al., 2012) and RLPark (Degris et al., 2013) provide interfaces to actual hardware, e.g., Bioloid and iRobot Create. In contrast to these aforementioned testbeds, our benchmark makes use of simulated environments to reduce computation time and to encourage experimental reproducibility. Furthermore, it provides a much larger collection of tasks of varying difficulty.

2.8 DISCUSSION

In this chapter, a benchmark of continuous control problems for reinforcement learning is presented, covering a wide variety of challenging tasks. We implemented several reinforcement learning algorithms, and presented them in the context of general policy parameterizations. Results show that among the implemented algorithms, TNPG, TRPO, and DDPG are effective methods for training deep neural network policies. Still, the poor performance on the proposed hierarchical tasks calls for new algorithms to be developed. Implementing and evaluating existing and newly proposed algorithms will be our continued effort. By providing an open-source release of the benchmark, we encourage other researchers to evaluate their algorithms on the proposed tasks.

2.9 TASK SPECIFICATIONS

Below we provide some specifications for the task observations, actions, and rewards. Please refer to the benchmark source code (https://github.com/rllab/rllab) for complete specification of physics parameters.

2.9.1 *Basic Tasks*

Cart-Pole Balancing: The observation consists of the cart position x, pole angle θ, the cart velocity $\dot{x}$, and the pole velocity $\dot{\theta}$. The 1D action consists of the horizontal force applied to the cart body. The reward function is given by $r(s, a) := 10 - (1 - \cos(\theta)) - 10^{-5}\|a\|_2^2$. The episode terminates when $|x| > 2.4$ or $|\theta| > 0.2$.

Cart-Pole Swing Up: Same observation and action as in balancing. The reward function is given by $r(s, a) := \cos(\theta)$. The episode terminates when $|x| > 3$, with a penalty of -100.

Mountain Car: The observation is given by the horizontal position x and the horizontal velocity $\dot{x}$ of the car. The reward is given by $r(s, a) := -1 + \text{height}$, with height the car's vertical offset. The episode terminates when the car reaches a target height of 0.6. Hence the goal is to reach the target as soon as possible.

Acrobot Swing Up: The observation includes the two joint angles, θ_1 and θ_2, and their velocities, $\dot{\theta}_1$ and $\dot{\theta}_2$. The action is the torque applied at the second joint. The reward is defined as $r(s, a) := -\|\text{tip}(s) - \text{tip}_{\text{target}}\|_2$, where $\text{tip}(s)$ computes the Cartesian position of the tip of the robot given the joint angles. No termination condition is applied.

Double Inverted Pendulum Balancing: The observation includes the cart position x, joint angles (θ_1 and θ_2), and joint velocities ($\dot{\theta}_1$ and $\dot{\theta}_2$). We encode each joint angle as its sine and cosine values. The action is the same as in cart-pole tasks. The reward is given by $r(s, a) = 10 - 0.01x_{\text{tip}}^2 - (y_{\text{tip}} - 2)^2 - 10^{-3} \cdot \dot{\theta}_1^2 - 5 \cdot 10^{-3} \cdot \dot{\theta}_2^2$, where $x_{\text{tip}}, y_{\text{tip}}$ are the coordinates of the tip of the pole. No termination condition is applied. The episode is terminated when $y_{\text{tip}} \leqslant 1$.

2.9.2 *Locomotion Tasks*

Swimmer: The 13-dim observation includes the joint angles, joint velocities, as well as the coordinates of the center of mass. The reward is given by $r(s, a) = v_x - 0.005\|a\|_2^2$, where v_x is the forward velocity. No termination condition is applied.

Hopper: The 20-dim observation includes joint angles, joint velocities, the coordinates of center of mass, and constraint forces. The reward is given by $r(s, a) := v_x - 0.005 \cdot \|a\|_2^2 + 1$, where the last term is a bonus for being "alive." The episode is terminated when $z_{\texttt{body}} < 0.7$ where $z_{\texttt{body}}$ is the z-coordinate of the body, or when $|\theta_y| < 0.2$, where θ_y is the forward pitch of the body.

Walker: The 21-dim observation includes joint angles, joint velocities, and the coordinates of center of mass. The reward is given by $r(s, a) := v_x - 0.005 \cdot \|a\|_2^2$. The episode is

terminated when $z_{body} < 0.8$, $z_{body} > 2.0$, or when $|\theta_y| > 1.0$.

Half-Cheetah: The 20-dim observation includes joint angles, joint velocities, and the coordinates of the center of mass. The reward is given by $r(s, a) = v_x - 0.05 \cdot \|a\|_2^2$. No termination condition is applied.

Ant: The 125-dim observation includes joint angles, joint velocities, coordinates of the center of mass, a (usually sparse) vector of contact forces, as well as the rotation matrix for the body. The reward is given by $r(s, a) = v_x - 0.005 \cdot \|a\|_2^2 - C_{contact} + 0.05$, where $C_{contact}$ penalizes contacts to the ground, and is given by $5 \cdot 10^{-4} \cdot \|F_{contact}\|_2^2$, where $F_{contact}$ is the contact force vector clipped to values between -1 and 1. The episode is terminated when $z_{body} < 0.2$ or when $z_{body} > 1.0$.

Simple Humanoid: The 102-dim observation includes the joint angles, joint velocities, vector of contact forces, and the coordinates of the center of mass. The reward is given by $r(s, a) = v_x - 5 \cdot 10^{-4}\|a\|_2^2 - C_{contact} - C_{deviation} + 0.2$, where $C_{contact} = 5 \cdot 10^{-6} \cdot \|F_{contact}\|$, and $C_{deviation} = 5 \cdot 10^{-3} \cdot (v_y^2 + v_z^2)$ to penalize deviation from the forward direction. The episode is terminated when $z_{body} < 0.8$ or when $z_{body} > 2.0$.

Full Humanoid: The 142-dim observation includes the joint angles, joint velocities, vector of contact forces, and the coordinates of the center of mass. The reward and termination condition is the same as in the Simple Humanoid model.

2.9.3 *Partially Observable Tasks*

Limited Sensors: The full description is included in the main text.

Noisy Observations and Delayed Actions: For all tasks, we use a Gaussan noise with $\sigma = 0.1$. The time delay is as follows: Cart-Pole Balancing 0.15 sec, Cart-Pole Swing Up 0.15 sec, Mountain Car 0.15 sec, Acrobot Swing Up 0.06 sec, and Double Inverted Pendulum Balancing 0.06 sec. This corresponds to 3 discretization frames for each task.

System Identifications: For Cart-Pole Balancing and Cart-Pole Swing Up, the pole length is varied uniformly between, 50% and 150%. For Mountain Car, the width of the valley varies uniformly between 75% and 125%. For Acrobot Swing Up, each of the pole length varies uniformly between 50% and 150%. For Double Inverted Pendulum Balancing, each of the pole length varies uniformly between 83% and 167%. Please refer to the benchmark source code for reference values.

Table 2: Experiment Setup

	Basic & Locomotion	Partially Observable	Hierarchical
Sim. steps per Iter.	50,000	50,000	50,000
Discount(λ)	0.99	0.99	0.99
Horizon	500	100	500
Num. Iter.	500	300	500

2.9.4 *Hierarchical Tasks*

Locomotion + Food Collection: During each episode, 8 food units and 8 bombs are placed in the environment. Collecting a food unit gives $+1$ reward, and collecting a bomb gives -1 reward. Hence the best cumulative reward for a given episode is 8.

Locomotion + Maze: During each episode, a $+1$ reward is given when the robot reaches the goal. Otherwise, the robot receives a zero reward throughout the episode.

2.10 EXPERIMENT PARAMETERS

For all batch gradient-based algorithms, we use the same time-varying feature encoding for the linear baseline:

$$\phi_{s,t} = \text{concat}(s, s \odot s, 0.01t, (0.01t)^2, (0.01t)^3, 1)$$

where s is the state vector and $\odot$ represents element-wise product.

Table 2 shows the experiment parameters for all four categories. We will then detail the hyperparameter search range for the selected tasks and report best hyperparameters, shown in Table 3, Table 4, Table 5, Table 6, Table 7, and Table 8.

Table 3: Learning Rate α for REINFORCE

	Search Range	Best
Cart-Pole Swing Up	$[1 \times 10^{-4}, 1 \times 10^{-1}]$	5×10^{-3}
Double Inverted Pendulum	$[1 \times 10^{-4}, 1 \times 10^{-1}]$	5×10^{-3}
Swimmer	$[1 \times 10^{-4}, 1 \times 10^{-1}]$	1×10^{-2}
Ant	$[1 \times 10^{-4}, 1 \times 10^{-1}]$	5×10^{-3}

Table 4: Step Size δ_{KL} for TNPG

	Search Range	Best
Cart-Pole Swing Up	$[1 \times 10^{-3}, 5 \times 10^{0}]$	5×10^{-2}
Double Inverted Pendulum	$[1 \times 10^{-3}, 5 \times 10^{0}]$	3×10^{-2}
Swimmer	$[1 \times 10^{-3}, 5 \times 10^{0}]$	1×10^{-1}
Ant	$[1 \times 10^{-3}, 5 \times 10^{0}]$	3×10^{-1}

Table 5: Step Size δ_{KL} for TRPO

	Search Range	Best
Cart-Pole Swing Up	$[1 \times 10^{-3}, 5 \times 10^{0}]$	5×10^{-2}
Double Inverted Pendulum	$[1 \times 10^{-3}, 5 \times 10^{0}]$	1×10^{-3}
Swimmer	$[1 \times 10^{-3}, 5 \times 10^{0}]$	5×10^{-2}
Ant	$[1 \times 10^{-3}, 5 \times 10^{0}]$	8×10^{-2}

Table 6: Step Size δ_{KL} for REPS

	Search Range	Best
Cart-Pole Swing Up	$[1 \times 10^{-3}, 5 \times 10^{0}]$	1×10^{-2}
Double Inverted Pendulum	$[1 \times 10^{-3}, 5 \times 10^{0}]$	8×10^{-1}
Swimmer	$[1 \times 10^{-3}, 5 \times 10^{0}]$	3×10^{-1}
Ant	$[1 \times 10^{-3}, 5 \times 10^{0}]$	8×10^{-1}

Table 7: Initial Extra Noise for CEM

	Search Range	Best
Cart-Pole Swing Up	$[1 \times 10^{-3}, 1]$	1×10^{-2}
Double Inverted Pendulum	$[1 \times 10^{-3}, 1]$	1×10^{-1}
Swimmer	$[1 \times 10^{-3}, 1]$	1×10^{-1}
Ant	$[1 \times 10^{-3}, 1]$	1×10^{-1}

Table 8: Initial Standard Deviation for CMA-ES

	Search Range	Best
Cart-Pole Swing Up	$[1 \times 10^{-3}, 1 \times 10^{3}]$	1×10^{3}
Double Inverted Pendulum	$[1 \times 10^{-3}, 1 \times 10^{3}]$	3×10^{-1}
Swimmer	$[1 \times 10^{-3}, 1 \times 10^{3}]$	1×10^{-1}
Ant	$[1 \times 10^{-3}, 1 \times 10^{3}]$	1×10^{-1}

3

RL^2: FAST REINFORCEMENT LEARNING VIA SLOW REINFORCEMENT LEARNING

3.1 OVERVIEW

Deep reinforcement learning has achieved many impressive results, including playing Backgammon (Tesauro, 1995), playing Atari games from raw pixels (Guo et al., 2014; Mnih et al., 2015; Schulman et al., 2015), mastering the game of Go (Silver et al., 2016), and acquiring advanced manipulation and locomotion skills (Levine et al., 2016; T. P. Lillicrap et al., 2016; Watter et al., 2015b; Heess et al., 2015b; Schulman et al., 2015; Schulman et al., 2016). However, many of the successes come at the expense of high sample complexity. For example, the state-of-the-art Atari results require tens of thousands of episodes of experience (Mnih et al., 2015) per game. To master a game, one would need to spend nearly 40 days playing it with no rest. In contrast, humans and animals are capable of learning a new task in a very small number of trials. Continuing the previous example, the human player in Mnih et al. (2015) can perform well on a game after just a few trials – although they may be eventually caught up by the AI. We argue that the reason for this sharp contrast is largely due to the lack of a good prior, which results in these deep RL agents needing to rebuild their knowledge about the world from scratch.

Although Bayesian reinforcement learning provides a solid framework for incorporating prior knowledge into the learning process (Strens, 2000; Ghavamzadeh et al., 2015; Kolter and A. Y. Ng, 2009), exact computation of the Bayesian update is intractable in all but the simplest cases. Thus, practical reinforcement learning algorithms often incorporate a mixture of Bayesian and domain-specific ideas to bring down sample complexity and computational burden. Notable examples include guided policy search with unknown dynamics (Levine and Abbeel, 2014) and PILCO (M. Deisenroth and Rasmussen, 2011). These methods can learn a task using a few minutes to a few hours of real experience, compared to days or even weeks required by previous methods (Schulman et al.,

2015; Schulman et al., 2016; T. P. Lillicrap et al., 2016). However, these methods tend to make assumptions about the environment (e.g., instrumentation for access to the state at learning time), or become computationally intractable in high-dimensional settings (Wahlström et al., 2015).

Rather than hand-designing domain-specific reinforcement learning algorithms, we take a different approach in this chapter: we view the learning process of the agent itself as an objective, which can be optimized using standard reinforcement learning algorithms. The objective is averaged across all possible MDPs according to a specific distribution, which reflects the prior that we would like to distill into the agent. We structure the agent as a recurrent neural network, which receives past rewards, actions, and termination flags as inputs in addition to the normally received observations. Furthermore, its internal state is preserved across episodes, so that it has the capacity to perform learning in its own hidden activations. The learned agent thus also acts as the learning algorithm, and can adapt to the task at hand when deployed.

There has been significant prior work using recurrent neural networks to solve partially observable MDPs (Wierstra et al., 2007; Heess et al., 2015a; Mnih et al., 2016) – and in fact, many of the previously studied problems can be reformulated under RL^2. However, our emphasis here is on proposing a generally applicable transformation, which reduces learning RL algorithms to ordinary reinforcement learning.

We evaluate this approach on two sets of classical problems, multi-armed bandits and tabular MDPs. These problems have been extensively studied, and there exist algorithms that achieve asymptotically optimal performance. We demonstrate that our method, named RL^2, can achieve performance comparable with these theoretically justified algorithms. Next, we evaluate RL^2 on a vision-based navigation task implemented using the ViZDoom environment (Kempka et al., 2016), showing that RL^2 can also scale to high-dimensional problems. Videos of our experiments are available at `https://bit.ly/nips2017-rl2`.

3.2 METHOD

3.2.1 *Preliminaries*

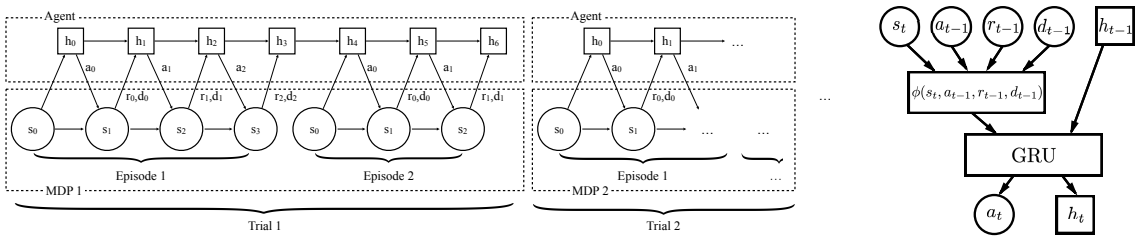
We define a discrete-time finite-horizon discounted Markov decision process (MDP) by a tuple $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \rho_0, \gamma, T)$, in which $\mathcal{S}$ is a state set, $\mathcal{A}$ an action set, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_+$ a transition probability distribution, $r : \mathcal{S} \times \mathcal{A} \to [-R_{max}, R_{max}]$ a bounded reward function, $\rho_0 : \mathcal{S} \to \mathbb{R}_+$ an initial state distribution, $\gamma \in [0, 1]$ a discount factor,

and T the horizon. In policy search methods, we typically optimize a stochastic policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+$ parametrized by θ. The objective is to maximize its expected discounted return, $\eta(\pi_\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)]$, where $\tau = (s_0, a_0, \ldots)$ denotes the whole trajectory, $s_0 \sim \rho_0(s_0)$, $a_t \sim \pi_\theta(a_t|s_t)$, and $s_{t+1} \sim \mathcal{P}(s_{t+1}|s_t, a_t)$.

3.2.2 *Formulation*

We now describe our formulation, which casts learning an RL algorithm as a reinforcement learning problem, and hence the name RL^2.

We assume knowledge of a set of MDPs, denoted by $\mathcal{M}$, and a distribution over them: $\rho_{\mathcal{M}} : \mathcal{M} \to \mathbb{R}_+$. We only need to sample from this distribution. We use n to denote the total number of episodes allowed to spend with a specific MDP. We define a *trial* to be such a series of episodes of interaction with a fixed MDP.



(a) Procedure of agent-environment interaction.

(b) Policy architecture.

Figure 5: Illustration of the main components.

This process of interaction between an agent and the environment is illustrated in Fig. 5a. Here, each trial happens to consist of two episodes, hence $n = 2$. For each trial, a separate MDP is drawn from $\rho_{\mathcal{M}}$, and for each episode, a fresh s_0 is drawn from the initial state distribution specific to the corresponding MDP. Upon receiving an action a_t produced by the agent, the environment computes reward r_t, steps forward, and computes the next state s_{t+1}. If the episode has terminated, it sets termination flag d_t to 1, which otherwise defaults to 0. Together, the next state s_{t+1}, action a_t, reward r_t, and termination flag d_t, are concatenated to form the input to the policy (or the *fast learner*)[1], which, conditioned on the hidden state h_{t+1}, generates the next hidden state h_{t+2} and action a_{t+1}. At the end of an episode, the hidden state of the fast learner is preserved to the next episode, but not preserved between trials.

1 To make sure that the inputs have a consistent dimension, we use placeholder values for the initial input to the fast learner.

The objective under this formulation is to maximize the expected total discounted reward accumulated during a single trial rather than a single episode. Maximizing this objective is equivalent to minimizing the cumulative pseudo-regret (Bubeck and Cesa-Bianchi, 2012). Since the underlying MDP changes across trials, as long as different strategies are required for different MDPs, the agent must act differently according to its belief over which MDP it is currently in. Hence, the agent is forced to integrate all the information it has received, including past actions, rewards, and termination flags, and adapt its strategy continually. Hence, we have set up an end-to-end optimization process, where the agent is encouraged to learn a "fast" reinforcement learning algorithm.

For clarity of exposition, we have defined the "inner" problem (of which the agent sees n each trials) to be an MDP rather than a POMDP. However, the method can also be applied in the partially-observed setting without any conceptual changes. In the partially observed setting, the agent is faced with a sequence of POMDPs, and it receives an observation o_t instead of state s_t at time t. The visual navigation experiment in Section 3.3.3, is actually an instance of this POMDP setting.

3.2.3 *Architecture*

We represent the fast learner as a general recurrent neural network as illustrated in Fig. 5b. Each timestep, it receives the tuple (s, a, r, d) as input, which is embedded using a function $\phi(s, a, r, d)$ and provided as input to an RNN.

To alleviate the difficulty of training RNNs due to vanishing and exploding gradients (Y. Bengio et al., 1994), we use Gated Recurrent Units (GRUs) (Cho et al., 2014a) which have been demonstrated to have good empirical performance (Chung et al., 2014; Józefowicz et al., 2015). The output of the GRU is fed to a fully connected layer followed by a softmax function, which forms the distribution over actions.

We have also experimented with alternative architectures which explicitly reset part of the hidden state each episode of the sampled MDP, but we did not find any improvement over the simple architecture described above.

3.2.4 *Learning the Fast Learner*

After formulating the task as a reinforcement learning problem, we can readily use standard off-the-shelf (slow) RL algorithms to optimize the fast learner. We use Proximal Policy Optimization (PPO) (Schulman et al., 2017), because of its excellent empirical performance, and because it does not require excessive hyperparameter tuning. To reduce

variance in the stochastic gradient estimation, we use a baseline which is also represented as an RNN using GRUs as building blocks. We optionally apply Generalized Advantage Estimation (GAE) (Schulman et al., 2016) to further reduce the variance.

3.3 EVALUATION

We designed experiments to answer the following questions:

- Can RL^2 learn algorithms that achieve good performance on MDP classes with special structure, relative to existing algorithms tailored to this structure that have been proposed in the literature?
- Can RL^2 scale to high-dimensional tasks?
- How does RL^2 compare with alternative methods?

For the first question, we evaluate RL^2 on two sets of tasks, multi-armed bandits (MAB) and tabular MDPs. These problems have been studied extensively in the reinforcement learning literature, and this body of work includes algorithms with guarantees of asymptotic optimality. We demonstrate that our approach achieves comparable performance to these theoretically justified algorithms.

For the second question, we evaluate RL^2 on a vision-based navigation task. Our experiments show that the fast learner makes effective use of the learned visual information and also short-term information acquired from previous episodes.

For the third question, we compare RL^2 with a recent meta-learning algorithm, MAML (Finn et al., 2017a). While RL^2 puts no assumptions about how the fast learning algorithm should operate and learns it entirely end-to-end, MAML fixes the fast learning algorithm to be policy gradient, and learns a good initialization of the policy weights so that it can quickly adapt. We compare RL^2 to MAML on all RL tasks in Finn et al. (2017a).

3.3.1 *Multi-armed bandits*

Multi-armed bandit problems are a subset of MDPs where the agent's environment is stateless. Specifically, there are k arms (actions), and at every time step, the agent pulls one of the arms, say i, and receives a reward drawn from an unknown distribution: our experiments take each arm to be a Bernoulli distribution with parameter p_i. The goal is to maximize the total reward obtained over a fixed number of time steps. The key challenge is balancing exploration and exploitation—"exploring" each arm enough times to estimate its distribution (p_i), but eventually switching over to "exploitation" of the best arm. Despite the simplicity of multi-arm bandit problems, their study has led to

a rich theory and a collection of algorithms with optimality guarantees.

Using RL^2, we can train an RNN fast learner to solve bandit problems by training it on a given distribution $\rho_{\mathcal{M}}$. If the learning is successful, the resulting fast learner should be able to perform competitively with the theoretically optimal algorithms. We randomly generated bandit problems by sampling each parameter p_i from the uniform distribution on $[0, 1]$. After training the RNN fast learner with RL^2, we compared it against the following strategies:

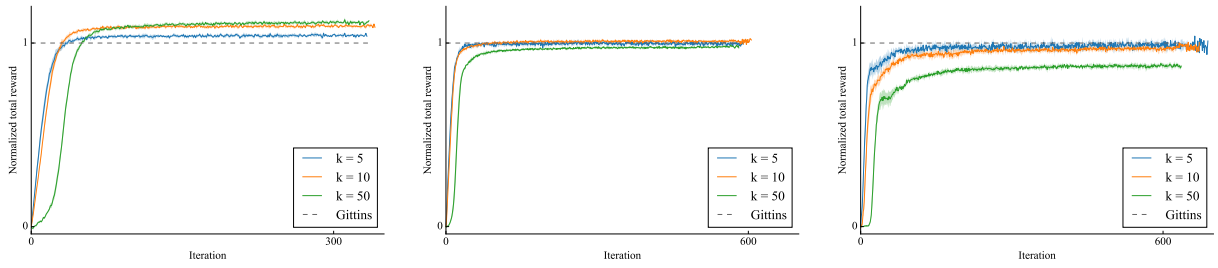
- Random: this is a baseline strategy, where the agent pulls a random arm each time.
- Dynamic programming (DP): when both the horizon and the number of arms are small, we can solve for the optimal strategy exactly using dynamic programming. However as the experiments suggest, this method scales very poorly.
- Gittins index (J. C. Gittins, 1979): this method gives the Bayes optimal solution in the discounted infinite-horizon case, by computing an index separately for each arm, and taking the arm with the largest index. While this chapter shows it is sufficient to independently compute an index for each arm (hence avoiding combinatorial explosion with the number of arms), it doesn't show how to tractably compute these individual indices exactly. We follow the practical approximations described in J. Gittins et al. (2011), Chakravorty and Mahajan (2013), and Whittle (1982), and choose the best-performing approximation for each setup.
- UCB1 (Auer, 2002): this method estimates an upper-confidence bound, and pulls the arm with the largest value of $\text{ucb}_i(t) = \hat{\mu}_i(t-1) + c\sqrt{\frac{2\log t}{T_i(t-1)}}$, where $\hat{\mu}_i(t-1)$ is the estimated mean parameter for the ith arm, $T_i(t-1)$ is the number of times the ith arm has been pulled, and c is a tunable hyperparameter (Audibert and Munos, 2011). We initialize the statistics with exactly one success and one failure, which corresponds to a $\text{Beta}(1, 1)$ prior.
- Thompson sampling (TS) (Thompson, 1933): this is a simple method which, at each time step, samples a list of arm means from the posterior distribution, and choose the best arm according to this sample. It has been demonstrated to compare favorably to UCB1 empirically (Chapelle and L. Li, 2011). We also experiment with an optimistic variant (OTS) (May et al., 2012), which samples N times from the posterior, and takes the one with the highest probability.
- ϵ-Greedy: in this strategy, the agent chooses the arm with the best empirical mean with probability $1 - \epsilon$, and chooses a random arm with probability ϵ. We use the same initialization as UCB1.
- Greedy: this is a special case of ϵ-Greedy with $\epsilon = 0$.

The Bayesian methods, Gittins index and Thompson sampling, take advantage of the

distribution $\rho_{\mathcal{M}}$; and we provide these methods with the true distribution. For each method with hyperparameters, we maximize the score with a separate grid search for each of the experimental settings. The hyperparameters used for all algorithms are provided in the Appendix.

Table 9: MAB Results. Each grid cell records the total reward averaged over 1000 different instances of the bandit problem. We consider $k \in \{5, 10, 50\}$ bandits and $n \in \{10, 100, 500\}$ episodes of interaction. We highlight the best-performing algorithms in each setup according to the computed mean, and we also highlight the other algorithms in that row whose performance is not significantly different from the best one (determined by a one-sided t-test with $p = 0.05$). For dynamic programming, the solver runs out of memory for $n \geqslant 100$ after running for over 8 hours.

Setup	Random	DP	Gittins	TS	OTS	UCB1	ϵ-Greedy	Greedy	RL^2
$n = 10, k = 5$	5.0	**6.7**	**6.6**	5.7	6.5	**6.7**	**6.6**	**6.6**	**6.7**
$n = 10, k = 10$	5.0	**6.7**	**6.6**	5.5	6.2	**6.7**	**6.6**	**6.6**	**6.7**
$n = 10, k = 50$	5.1	**6.7**	6.5	5.2	5.5	**6.6**	6.5	6.5	**6.8**
$n = 100, k = 5$	49.9	n/a	**78.3**	74.7	**77.9**	**78.0**	75.4	74.8	**78.7**
$n = 100, k = 10$	49.9	n/a	**82.8**	76.7	81.4	82.4	77.4	77.1	**83.5**
$n = 100, k = 50$	49.8	n/a	**85.2**	64.5	67.7	84.3	78.3	78.0	**84.9**
$n = 500, k = 5$	249.8	n/a	**405.8**	**402.0**	**406.7**	**405.8**	388.2	380.6	**401.6**
$n = 500, k = 10$	249.0	n/a	**437.8**	429.5	**438.9**	**437.1**	408.0	395.0	432.5
$n = 500, k = 50$	249.6	n/a	**463.7**	427.2	437.6	457.6	413.6	402.8	438.9



(a) $n = 10$ (b) $n = 100$ (c) $n = 500$

Figure 6: RL^2 learning curves for multi-armed bandits. Performance is normalized such that Gittins index scores 1, and random policy scores 0.

The results are summarized in Table 9, and the learning curves are shown in Fig. 6. We observe that our approach achieves performance that is almost as good as the the reference methods, which were (human) designed specifically to perform well on multi-armed bandit problems. It is worth noting that the published algorithms are mostly designed to minimize asymptotic regret (rather than finite horizon regret), hence there tends to be a little bit of room to outperform them in the finite horizon settings.

Ablation study: We observe that there is a noticeable gap between Gittins index and RL^2 in the most challenging scenario. This raises the question whether better architectures or better (slow) RL algorithms should be explored. To determine the bottleneck, we trained the same architecture using supervised learning, using the trajectories generated by the Gittins index approach as training data. We found that the fast learner, when executed in test domains, achieved the same level of performance as the Gittins index approach, suggesting that there is room for improvement by using better RL algorithms. Further analysis on the bandit tasks is available in the Appendix.

3.3.2 *Tabular MDPs*

The bandit problem provides a natural and simple setting to investigate whether the fast learner learns to trade off between exploration and exploitation. However, the problem itself involves no sequential decision making, and does not fully characterize the challenges in solving MDPs. Hence, we perform further experiments using randomly generated tabular MDPs, where there is a finite number of possible states and actions—small enough that the transition probability distribution can be explicitly given as a table. We compare our approach with the following methods:

- Random: the agent chooses an action uniformly at random for each time step;
- PSRL (Strens, 2000; Osband et al., 2013): it generalizes Thompson sampling to MDPs, where at the beginning of each episode, we sample an MDP from the posterior distribution, and take actions according to the optimal policy for the entire episode. Similarly, we include an optimistic variant (OPSRL), which has also been explored in Osband and Van Roy (2017).
- BEB (Kolter and A. Y. Ng, 2009): this is a model-based optimistic algorithm that adds an exploration bonus to (thus far) infrequently visited states and actions.
- UCRL2 (Jaksch et al., 2010): this algorithm computes, at each iteration, the optimal policy against an optimistic MDP under the current belief, using an extended value iteration procedure.
- ϵ-Greedy: this algorithm takes actions optimal against the MAP estimate according

to the current posterior, which is updated once per episode.

- Greedy: a special case of ϵ-Greedy with $\epsilon = 0$.

Table 10: Random MDP Results

Setup	Random	PSRL	OPSRL	UCRL2	BEB	ϵ-Greedy	Greedy	RL^2
$n = 10$	100.1	138.1	144.1	146.6	150.2	132.8	134.8	**156.2**
$n = 25$	250.2	408.8	425.2	424.1	427.8	377.3	368.8	**445.7**
$n = 50$	499.7	904.4	**930.7**	918.9	917.8	823.3	769.3	**936.1**
$n = 75$	749.9	1417.1	**1449.2**	1427.6	1422.6	1293.9	1172.9	1428.8
$n = 100$	999.4	1939.5	**1973.9**	1942.1	1935.1	1778.2	1578.5	1913.7

The distribution over MDPs is constructed with $|\mathcal{S}| = 10$, $|\mathcal{A}| = 5$. The rewards follow a Gaussian distribution with unit variance, and the mean parameters are sampled independently from Normal$(1, 1)$. The transitions are sampled from a flat Dirichlet distribution. This construction matches the commonly used prior in Bayesian RL methods. We set the horizon for each episode to be $T = 10$, and an episode always starts on the first state.

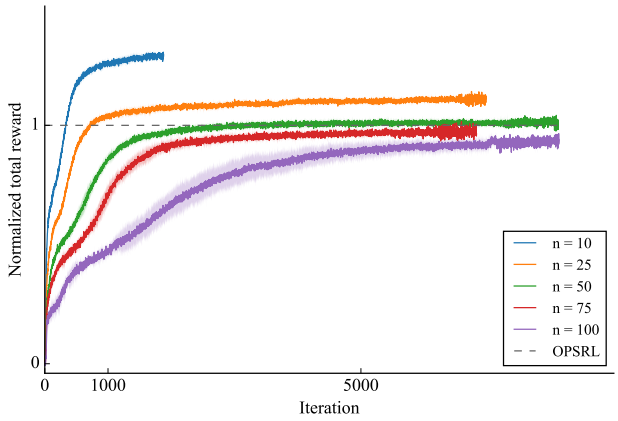


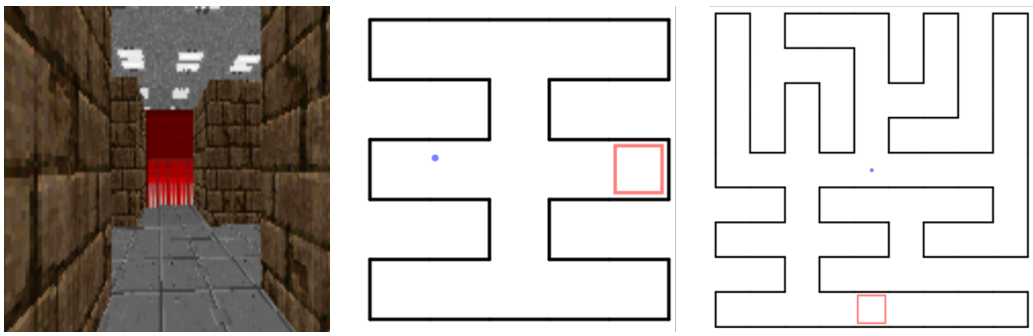
Figure 7: RL^2 learning curves for tabular MDPs. Performance is normalized such that OPSRL scores 1, and random policy scores 0.

The results are summarized in Table 10, and the learning curves are shown in Fig. 7.

We follow the same evaluation procedure as in the bandit case. We experiment with $n \in \{10, 25, 50, 75, 100\}$. For fewer episodes, our approach surprisingly outperforms existing methods by a large margin. The advantage is reversed as n increases, suggesting that the reinforcement learning problem in the outer loop becomes more challenging to solve. We think that the advantage for small n comes from the need for more aggressive exploitation: since there are 140 degrees of freedom to estimate in order to characterize the MDP, and by the 10th episode, we will not have enough samples to form a good estimate of the entire dynamics. By directly optimizing the RNN in this setting, our approach should be able to cope with this shortage of samples, and decides to exploit sooner compared to the reference algorithms.

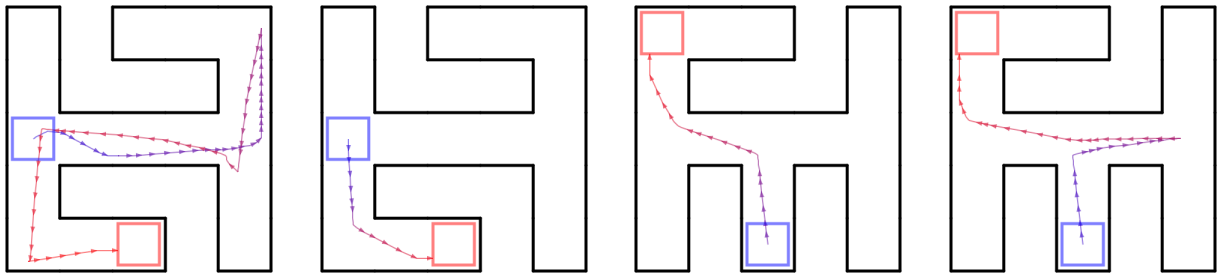
3.3.3 *Visual Navigation*

The previous two tasks both only involve very low-dimensional state spaces. To evaluate the feasibility of scaling up RL^2, we further experiment with a challenging vision-based task, where the agent is asked to navigate a randomly generated maze to find a randomly placed target. The agent receives a 30×40 RGB image of the current field of view, and the actions consist of {Turn 5° left, turn 5° right, move 10cm forward}. The agent receives a $+1$ reward when it reaches the target, -0.001 when it hits the wall, and -0.04 per time step to encourage it to reach targets faster. It can interact with the maze for multiple episodes, during which the maze structure and target position are held fixed. The optimal strategy is to explore the maze efficiently during the first episode, and after locating the target, act optimally against the current maze and target based on the collected information. An illustration of the task is given in Fig. 8.



(a) Sample observation (b) Layout of a 5x5 maze (c) Layout of a 9x9 maze

Figure 8: Visual navigation. The target block is shown in red, and occupies an entire grid in the maze layout.



(a) Good (1st) (b) Good (2nd) (c) Bad (1st) (d) Bad (2nd)

Figure 9: Visualization of the agent's behavior. In each scenario, the agent starts at the center of the blue block, and the goal is to reach anywhere in the red block.

Visual navigation alone is a challenging task for reinforcement learning. The agent only receives very sparse rewards during training, and does not have the primitives for efficient exploration at the beginning of training. It also needs to make efficient use of memory to decide how it should explore the space, without forgetting about where it has already explored. Previously, Oh et al. (2016) have studied similar vision-based navigation tasks in Minecraft. However, they use higher-level actions for efficient navigation. Similar high-level actions in our task would each require around 5 low-level actions combined in the right way. In contrast, our RL^2 agent needs to learn these higher-level actions from scratch.

We use a simple training setup, where we use small mazes of size 5×5, with 2 episodes of interaction, each with horizon up to 250. Here the size of the maze is measured by the number of grid cells along each wall in a discrete representation of the maze. During

each trial, we sample 1 out of 1000 randomly generated configurations of map layout and target positions. During testing, we evaluate on 1000 separately generated configurations. In addition, we also study its extrapolation behavior along two axes, by (1) testing on large mazes of size 9×9 (see Fig. 8c) and (2) running the agent for up to 5 episodes in both small and large mazes. For the large maze, we also increase the horizon per episode by 4x due to the increased size of the maze.

Table 11: Results for visual navigation. In 12c, we measure the proportion of mazes where the trajectory length in the second episode does not exceed the trajectory length in the first episode.

(a) Avg. length of successful trajectories

Episode	Small	Large
1	52.4 ± 1.3	180.1 ± 6.0
2	39.1 ± 0.9	151.8 ± 5.9
3	42.6 ± 1.0	169.3 ± 6.3
4	43.5 ± 1.1	162.3 ± 6.4
5	43.9 ± 1.1	169.3 ± 6.5

(b) %Success

Episode	Small	Large
1	99.3%	97.1%
2	99.6%	96.7%
3	99.7%	95.8%
4	99.4%	95.6%
5	99.6%	96.1%

(c) %Improved

Small	Large
91.7%	71.4%

The results are summarized in Table 11, and the learning curves are shown in Fig. 10. We observe that there is a significant reduction in trajectory lengths between the first two episodes in both the smaller and larger mazes, suggesting that the agent has learned how to use information from past episodes. It also achieves reasonable extrapolation behavior in further episodes by maintaining its performance, although there is a small drop in the rate of success in the larger mazes. We also observe that on larger mazes, the ratio of improved trajectories is lower, likely because the agent has not learned how to act optimally in the larger mazes.

Still, even on the small mazes, the agent does not learn to perfectly reuse prior information. An illustration of the agent's behavior is shown in Fig. 9. The intended behavior, which occurs most frequently, as shown in 9a and 9b, is that the agent should remember the target's location, and utilize it to act optimally in the second episode. However, occasionally the agent forgets about where the target was, and continues to explore in the second episode, as shown in 9c and 9d. We believe that better reinforcement learning

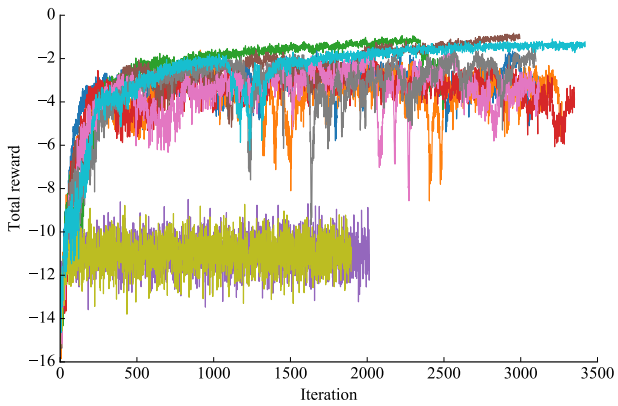


Figure 10: RL^2 learning curves for visual navigation. Each curve shows a different random initialization of the RNN weights. Performance varies greatly across different initializations.

techniques used as the outer-loop algorithm will improve these results in the future.

3.3.4 *Comparison with MAML*

We evaluate RL^2 and MAML on the following tasks:

- Controlling a 2D point robot to navigate to a random goal;
- Controlling a planar cheetah robot to run forward at various speeds;
- Controlling a planar cheetah robot to either run forward or backward as fast as possible.
- Controlling a quadruped robot (ant) to run forward at various speeds;
- Controlling a quadruped robot (ant) to either run forward or backward.

Figure 11: Comparison between RL^2 and MAML.

Results are shown in Fig. 11. We find that RL^2 significantly outperforms MAML. While MAML may require 40 or more rollouts to learn the task, RL^2 only needs 1. This suggests that the tasks in Finn et al. (2017a) are somewhat simplistic: since the reward signal is dense and conveys enough information about the task, the fast algorithm learned by

RL^2 can infer the task after a single time step, and achieves optimal performance immediately (i.e. as soon as the beginning of the very first rollout). In contrast, MAML requires many rollouts. It is worth noting that the tasks investigated in the main body of our paper require much richer behaviors, including balancing exploration and exploitation, performing credit assignment, and emergence of hierarchical skills.

We also note that the final performance achieved by RL^2 is better than what MAML achieves. We want to emphasize that the MAML results shown are the exact numbers from Finn et al. (2017a), and we used the exact same environments. There are a few possible explanations for MAML not achieving the same final performance: (i) MAML simply needs more iterations than were considered in the original MAML paper; this wouldn't be too surprising as vanilla policy gradients are known to not be the most sample-efficient RL algorithms, and MAML is ultimately relying on policy gradients for the adaptation (and trying to seek an initial policy from which policy gradients can be maximally effective, which simply still might not be as effective as other algorithms, including the adaptation mechanism RL^2 discovers); (ii) Related, it could be that MAML needs more meta-training to find a better initialization; (iii) Finally, it's worth noting that the earlier saturation of the MAML learning curves might relate to it not being as able to manage the exploration / exploitation trade-off, and reducing variance too quickly.

3.4 RELATED WORK

The concept of using prior experience to speed up reinforcement learning algorithms has been explored in the past in various forms. Earlier studies have investigated automatic tuning of hyper-parameters, such as learning rate and temperature (Ishii et al., 2002; Schweighofer and Doya, 2003), as a form of meta-learning. A. Wilson et al. (2007) use hierarchical Bayesian methods to maintain a posterior over possible models of dynamics, and apply optimistic Thompson sampling according to the posterior. Many works in hierarchical reinforcement learning propose to extract reusable skills from previous tasks to speed up exploration in new tasks (S. P. Singh, 1992; Perkins, Precup, et al., 1999). We refer the reader to Taylor and Stone (2009) for a more thorough survey on the multi-task and transfer learning aspects.

The formulation of searching for a best-performing algorithm, whose performance is averaged over a given distribution over MDPs, have been investigated in the past in more limited forms (Maes et al., 2011; Castronovo et al., 2012). There, they propose to learn an algorithm to solve multi-armed bandits using program search, where the search space consists of simple formulas composed from hand-specified primitives, which needs to

be tuned for each specific distribution over MDPs. In comparison, our approach allows for entirely end-to-end training without requiring such domain knowledge.

More recently, Fu et al. (2015) propose a model-based approach on top of iLQG with unknown dynamics (Levine and Abbeel, 2014), which uses samples collected from previous tasks to build a neural network prior for the dynamics, and can perform one-shot learning on new, but related tasks thanks to reduced sample complexity. There has been a growing interest in using deep neural networks for multi-task learning and transfer learning (Parisotto et al., 2015; Rusu et al., 2015; Rusu et al., 2016a; C. Devin et al., 2016; Rusu et al., 2016b).

In the broader context of machine learning, there has been a lot of interest in one-shot learning for object classification (Vilalta and Drissi, 2002; Fei-Fei et al., 2006; Larochelle et al., 2008; Lake et al., 2011; Koch, 2015). Our work draws inspiration from a particular line of work (Younger et al., 2001; Santoro et al., 2016; Vinyals et al., 2016a), which formulates meta-learning as an optimization problem, and can thus be optimized end-to-end via gradient descent. While these work applies to the supervised learning setting, our work applies in the more general reinforcement learning setting. The reinforcement learning setting requires a richer concept to be learned: our agent must not only learn to exploit existing information, but also learn to explore, a problem that is usually not a factor in supervised learning. Another line of work (Hochreiter et al., 2001; Younger et al., 2001; Andrychowicz et al., 2016; K. Li and Malik, 2016) studies meta-learning over the optimization process. There, the meta-learner makes explicit updates to a parametrized model. In comparison, we do not use a directly parametrized policy; instead, the recurrent neural network agent acts as the meta-learner and the resulting policy simultaneously.

Our formulation essentially constructs a partially observable MDP (POMDP) which is solved in the outer loop, where the underlying MDP is unobserved by the agent. This reduction of an unknown MDP to a POMDP can be traced back to dual control theory (Feldbaum, 1960), where "dual" refers to the fact that one is controlling both the state and the state estimate. Feldbaum pointed out that the solution can in principle be computed with dynamic programming, but doing so is usually impractical. POMDPs with such structure have also been studied under the name "mixed observability MDPs" (Ong et al., 2010). However, the method proposed there suffers from the usual challenges of solving POMDPs in high dimensions.

Apart from the various multiple-episode tasks we investigate in this chapter, previous literature on training RNN policies have used similar tasks that require memory to test if long-term dependency can be learned. Recent examples include the Labyrinth (Mnih

et al., 2016) and the water maze experiment (Heess et al., 2015a). These tasks can be reinterpreted under the RL^2 framework as instances of meta-learning. However, these works did not identify the shared structure among the tasks, and study them in isolation. In comparison, we propose a common framework that brings these different tasks together.

Simultaneous to our work, J. X. Wang et al. (2016) have independently proposed and studied a similar meta-learning setting. The two studies are nicely complementary, in the sense that their work provides insightful connections between meta-learning and neuroscience, whereas our work contributes more depth towards understanding the strength and limitations of such an implementation of meta-learning for RL.

3.5 DISCUSSION

This chapter suggests a different approach for designing better reinforcement learning algorithms: instead of acting as the designers ourselves, learn the algorithm end-to-end using standard reinforcement learning techniques. That is, the "fast" RL algorithm is a computation whose state is stored in the RNN activations, and the RNN's weights are learned by a general-purpose "slow" reinforcement learning algorithm. Our method, RL^2, has demonstrated competence comparable with theoretically optimal algorithms in small-scale settings. We have further shown its potential to scale to high-dimensional tasks.

In the experiments, we have identified opportunities to improve upon RL^2: the outer-loop reinforcement learning algorithm was shown to be an immediate bottleneck, and we believe that for settings with extremely long horizons, better architecture may also be required for the fast learner. Although we have used generic methods and architectures for the outer-loop algorithm and the fast learner, doing this also ignores the underlying episodic structure. We expect algorithms and architectures that exploit the problem structure to significantly boost the performance.

3.6 DETAILED EXPERIMENT SETUP

Common to all experiments: as mentioned in the Experiment section, we use placeholder values when necessary. For example, at $t = 0$ there is no previous action, reward, or termination flag. Since all of our experiments use discrete actions, we use the embedding of the action 0 as a placeholder for actions, and 0 for both the rewards and termination flags. To form the input to the GRU, we use the values for the rewards and termination flags as-is, and embed the states and actions as described separately below for each

experiments. These values are then concatenated together to form the joint embedding.

For the neural network architecture, We use rectified linear units throughout the experiments as the hidden activation, and we apply weight normalization without data-dependent initialization (Salimans and D. P. Kingma, 2016) to all weight matrices. The hidden-to-hidden weight matrix uses an orthogonal initialization (Saxe et al., 2013), and all other weight matrices use Xavier initialization (Glorot and Y. Bengio, 2010). We initialize all bias vectors to 0. Unless otherwise mentioned, the policy and the baseline uses separate neural networks with the same architecture until the final layer, where the number of outputs differ.

All experiments are implemented using TensorFlow (Abadi et al., 2016) and rllab (Y. Duan et al., 2016a). We use the implementations of classic algorithms provided by the TabulaRL package (Osband, 2016).

3.6.1 *Multi-armed bandits*

The parameters for TRPO are shown in Table 13. Since the environment is stateless, we use a constant embedding 0 as a placeholder in place of the states, and a one-hot embedding for the actions.

Table 13: Hyperparameters for TRPO: multi-armed bandits

Discount	0.99
GAE λ	0.3
Policy Iters	Up to 1000
#GRU Units	256
Mean KL	0.01
Batch size	250000

3.6.2 *Tabular MDPs*

The parameters for TRPO are shown in Table 14. We use a one-hot embedding for the states and actions separately, which are then concatenated together.

Table 14: Hyperparameters for TRPO: tabular MDPs

Discount	0.99
GAE λ	0.3
Policy Iters	Up to 10000
#GRU Units	256
Mean KL	0.01
Batch size	250000

3.6.3 *Visual Navigation*

The parameters for TRPO are shown in Table 15. For this task, we use a neural network to form the joint embedding. We rescale the images to have width 40 and height 30 with RGB channels preserved, and we recenter the RGB values to lie within range $[-1, 1]$. Then, this preprocessed image is passed through 2 convolution layers, each with 16 filters of size 5×5 and stride 2. The action is first embedded into a 256-dimensional vector where the embedding is learned, and then concatenated with the flattened output of the final convolution layer. The joint vector is then fed to a fully connected layer with 256 hidden units.

Unlike previous experiments, we let the policy and the baseline share the same neural network. We found this to improve the stability of training baselines and also the end performance of the policy, possibly due to regularization effects and better learned features imposed by weight sharing. Similar weight-sharing techniques have also been explored in Mnih et al. (2016).

Table 15: Hyperparameters for TRPO: visual navigation

Discount	0.99
GAE λ	0.99
Policy Iters	Up to 5000
#GRU Units	256
Mean KL	0.01
Batch size	50000

3.7 HYPERPARAMETERS FOR BASELINE ALGORITHMS

3.7.1 *Multi-armed bandits*

There are 3 algorithms with hyperparameters: UCB1, Optimistic Thompson Sampling (OTS), and ϵ-Greedy. We perform a coarse grid search to find the best hyperparameter for each of them. More specifically:

- **UCB1**: We test $c \in \{0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. The best found parameter for each setting is given in Table 16.

Table 16: Best hyperparameter for UCB1

Setting	Best c
$n = 10, k = 5$	0.1
$n = 10, k = 10$	0.1
$n = 10, k = 50$	0.1
$n = 100, k = 5$	0.2
$n = 100, k = 10$	0.2
$n = 100, k = 50$	0.2
$n = 500, k = 5$	0.2
$n = 500, k = 10$	0.2
$n = 500, k = 50$	0.2

- **Optimistic Thompson Sampling (OTS)**: The hyperparameter is the number of posterior samples. We use up to 20 samples. The best found parameter for each setting is given in Table 17.

Table 17: Best hyperparameter for OTS

Setting	Best #samples
$n = 10, k = 5$	15
$n = 10, k = 10$	14
$n = 10, k = 50$	19
$n = 100, k = 5$	8
$n = 100, k = 10$	20
$n = 100, k = 50$	16
$n = 500, k = 5$	7
$n = 500, k = 10$	20
$n = 500, k = 50$	20

- ϵ**-Greedy**: The hyperparameter is the ϵ parameter. We test $\epsilon \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. The best found parameter for each setting is given in Table 18.

Table 18: Best hyperparameter for ϵ-Greedy

Setting	Best ϵ
$n = 10, k = 5$	0.0
$n = 10, k = 10$	0.0
$n = 10, k = 50$	0.0
$n = 100, k = 5$	0.0
$n = 100, k = 10$	0.0
$n = 100, k = 50$	0.1
$n = 500, k = 5$	0.1
$n = 500, k = 10$	0.1
$n = 500, k = 50$	0.1

3.7.2 *Tabular MDPs*

There are 4 algorithms with hyperparameters: Optimistic PSRL (OPSRL), BEB, ϵ-Greedy, UCRL2. Details are given below.

- **Optimistic PSRL (OPSRL)**: The hyperparameter is the number of posterior samples. We use up to 20 samples. The best found parameter for each setting is given in Table 19.

Table 19: Best hyperparameter for OPSRL

Setting	Best #samples
$n = 10$	14
$n = 25$	14
$n = 50$	14
$n = 75$	14
$n = 100$	17

- **BEB**: We search for the scaling factor in front of the exploration bonus, in the log-linear span of $[\log(0.0001), \log(1.0)]$ with 21 way points. The actual searched parameters are 0.0001, 0.000158, 0.000251, 0.000398, 0.000631, 0.001, 0.001585, 0.002512, 0.003981, 0.00631, 0.01, 0.015849, 0.025119, 0.039811, 0.063096, 0.1, 0.158489, 0.251189, 0.398107, 0.630957, 1.0. The best found parameter for each setting is given in Table 20.

Table 20: Best hyperparameter for BEB

Setting	Best scaling
$n = 10$	0.002512
$n = 25$	0.001585
$n = 50$	0.001585
$n = 75$	0.001585
$n = 100$	0.001585

- **ϵ-Greedy**: We test $\epsilon \in \{0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. The best found parameter for each setting is given in Table 21.

Table 21: Best hyperparameter for ϵ-Greedy

Setting	Best ϵ
$n = 10$	0.1
$n = 25$	0.1
$n = 50$	0.1
$n = 75$	0.1
$n = 100$	0.1

- **UCRL2**: We search for the scaling factor of exploration bonus among the same values as BEB. The best found parameter for each setting is given in Table 22.

Table 22: Best hyperparameter for UCRL2

Setting	Best scaling
$n = 10$	0.398107
$n = 25$	0.398107
$n = 50$	0.398107
$n = 75$	0.398107
$n = 100$	0.398107

3.8 FURTHER ANALYSIS ON MULTI-ARMED BANDITS

In this section, we provide further analysis of the behavior of RL^2 agent in comparison with the baseline algorithms, on the multi-armed bandit task. Certain algorithms such as UCB1 are designed not in the Bayesian context; instead they are tailored to be robust in adversarial cases. To highlight this aspect, we evaluate the algorithms on a different metric, namely the percentage of trials where the best arm is recovered. We treat the best arm chosen by the policy to be the arm that has been pulled most often, and the

ground truth best arm is the arm with the highest mean parameter. In addition, we split the set of all possible bandit tasks into simpler and harder tasks, where the difficulty is measured by the ϵ-gap between the mean parameter of the best arm and the second best arm. We compare the percentage of recovering the best arm separately according to the ϵ gap, as shown in Table 23.

Table 23: Percentage of tasks where the best arm is chosen most frequently, with $k = 5$ arms and $n = 500$ episodes of interaction.

Setup	Random	Gittins	TS	OTS	UCB1	UCB1*	ϵ-Greedy	Greedy	RL^2
$\epsilon \in [0, 0.01]$	21.5%	51.1%	**53.1%**	52.8%	50.9%	**56.5%**	37.3%	38.3%	46.1%
$\epsilon \in [0.01, 0.05]$	19.3%	59.5%	71.2%	67.4%	62.5%	**76.3%**	42.3%	41.3%	55.1%
$\epsilon \in [0.05, 0.1]$	17.7%	71.2%	91.5%	84.0%	78.9%	**94.6%**	46.1%	45.7%	67.4%
$\epsilon \in [0.1, 0.3]$	20.1%	92.7%	99.2%	95.3%	93.5%	**99.9%**	58.1%	58.4%	87.1%
$\epsilon \in [0.3, 0.5]$	20.4%	99.6%	**100.0%**	99.5%	**99.8%**	**100.0%**	85.4%	84.6%	99.0%

Note that there are two columns associated with the UCB1 algorithm, where UCB1 (without "*") is evaluated with $c = 0.2$, the parameter that gives the best performance as evaluated by the average total reward, and UCB1* uses $c = 1.0$. Surprisingly, although using $c = 1.0$ performs the best in terms of recovering the best arm, its performance is significantly worse than using $c = 0.2$ when evaluated under the average total reward (369.2 ± 2.2 vs. 405.8 ± 2.2). This also explains that although RL^2 does not perform the best according to this metric (which is totally expected, since it is not optimized under this metric), it achieves comparable average total reward as other best-performing methods.

3.9 ADDITIONAL BASELINE EXPERIMENTS

We performed comparison with two additional baselines:

- B1: Train RL^2 agent as-is, but reset the hidden states after each episode, both in training and testing;
- B2: Train RL^2 agent on single episodes, and test it on multiple episodes without resetting hidden state in-between.

The results are shown in Table 24, Table 25, Table 26, and Table 28. In summary, RL^2 performs much better than B1, which in turn performs better than B2. These results can be explained as follows:

- **MAB**: For these environments, a single episode corresponds to a single pull of arms, and B2 reduces to random guessing so it won't do well. Yet surprisingly, B1 performs much better than random. This is because the policy observes the last action taken and the last reward, and it effectively has a single-step memory, which can be exploited (for instance, if the last pull resulted in a reward of 1, it makes sense to pull that arm again). However, such single-step memory is still very limited, and this baseline performs much worse than RL^2 as the number of episodes increases.
- **Tabular MDP**: B2 does slightly better than random by exploiting reward information in a single episode. The relative advantage of B1 compared to B2 is less significant here, since single-step memory is not as informative when each episode spans over multiple steps. RL^2 still outperforms these baselines significantly.
- **Maze**: RL^2 significantly outperforms both baselines. Interestingly, even in this case B1 can still slightly make use of its single-step memory, though it is much less effective than explicitly preserving hidden states. B2 is completely unable of making use of previous episodes.

Table 24: Multi-Armed Bandits

Setup	Random	Gittins	TS	OTS	UCB1	ϵ-Greedy	Greedy	RL^2	B1	B2
$n = 10, k = 5$	5.0	**6.6**	5.7	6.5	**6.7**	**6.6**	**6.6**	**6.7**	6.4	4.9
$n = 10, k = 10$	5.0	**6.6**	5.5	6.2	**6.7**	**6.6**	**6.6**	**6.7**	**6.6**	5.1
$n = 10, k = 50$	5.1	6.5	5.2	5.5	**6.6**	6.5	6.5	**6.8**	6.6	5.2
$n = 100, k = 5$	49.9	**78.3**	74.7	**77.9**	**78.0**	75.4	74.8	**78.7**	68.7	50.4
$n = 100, k = 10$	49.9	**82.8**	76.7	81.4	82.4	77.4	77.1	**83.5**	72.7	48.4
$n = 100, k = 50$	49.8	**85.2**	64.5	67.7	84.3	78.3	78.0	**84.9**	78.0	49.2
$n = 500, k = 5$	249.8	**405.8**	**402.0**	**406.7**	**405.8**	388.2	380.6	**401.6**	340.6	251.9
$n = 500, k = 10$	249.0	**437.8**	429.5	**438.9**	**437.1**	408.0	395.0	432.5	371.0	249.2
$n = 500, k = 50$	249.6	**463.7**	427.2	437.6	457.6	413.6	402.8	438.9	384.1	248.5

Table 25: Random MDPs

Setup	Random	PSRL	OPSRL	UCRL2	BEB	ϵ-Greedy	Greedy	RL^2	B1	B2
$n = 10$	100.1	138.1	144.1	146.6	150.2	132.8	134.8	**156.2**	123.1	111.6
$n = 25$	250.2	408.8	425.2	424.1	427.8	377.3	368.8	**445.7**	316.4	278.3
$n = 50$	499.7	904.4	**930.7**	918.9	917.8	823.3	769.3	**936.1**	646.9	557.8
$n = 75$	749.9	1417.1	**1449.2**	1427.6	1422.6	1293.9	1172.9	1428.8	975.6	832.0
$n = 100$	999.4	1939.5	**1973.9**	1942.1	1935.1	1778.2	1578.5	1913.7	1306.2	1114.8

Table 26: Visual navigation (small mazes)

Ep.	RL^2	B1	B2
1	52.4	51.2	**49.4**
2	**39.1**	47.0	51.1
3	**42.6**	50.4	50.3
4	**43.5**	50.1	51.0
5	**43.9**	49.8	50.6

(a) Average length of successful trajectories

Ep.	RL^2	B1	B2
1	99.3%	99.1%	98.4%
2	99.6%	99.3%	99.1%
3	99.7%	99.4%	99.6%
4	99.4%	99.6%	99.4%
5	99.6%	99.4%	99.2%

(b) %Success

RL^2	B1	B2
91.7%	85.0%	82.9%

(c) %Improved

Table 28: Visual navigation (test out-of-distribution generalization on large mazes)

Ep.	RL^2	B1	B2
1	**180.1**	189.8	190.4
2	**151.8**	172.2	191.4
3	**169.3**	176.2	190.2
4	**162.3**	184.3	193.8
5	**169.3**	175.6	191.5

(a) Average length of successful trajectories

Ep.	RL^2	B1	B2
1	97.1%	96.3%	95.9%
2	96.7%	96.8%	94.6%
3	95.8%	96.6%	96.2%
4	95.6%	96.2%	95.5%
5	96.1%	96.9%	95.5%

(b) %Success

RL^2	B1	B2
71.4%	69.3%	63.3%

(c) %Improved

4

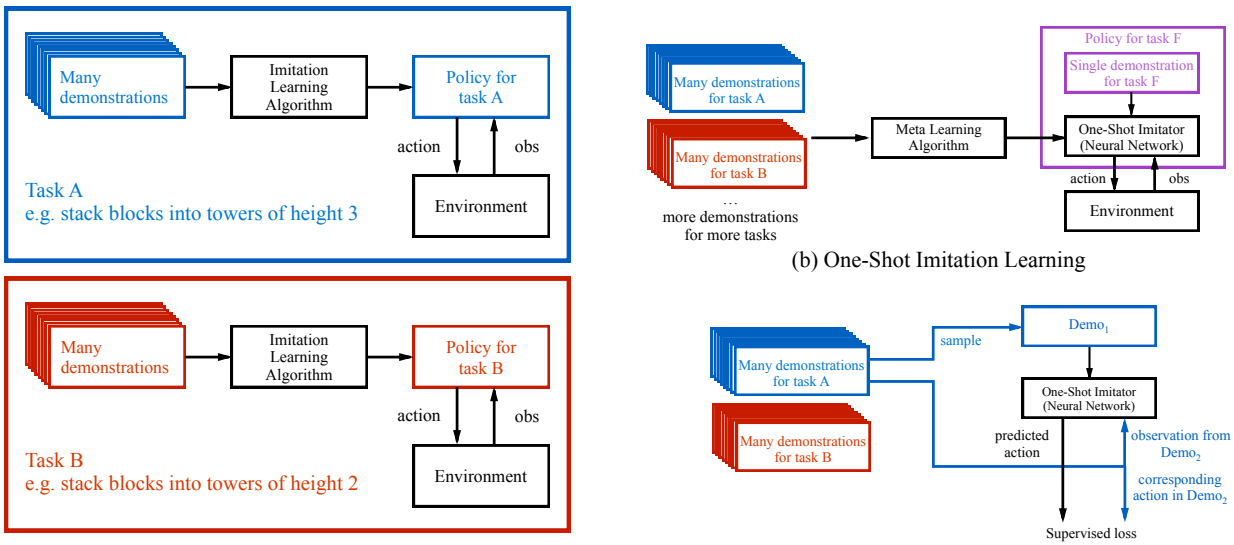
ONE-SHOT IMITATION LEARNING

4.1 OVERVIEW

We are interested in robotic systems that are able to perform a variety of complex useful tasks, e.g. tidying up a home or preparing a meal. The robot should be able to learn new tasks without long system interaction time. To accomplish this, we must solve two broad problems. The first problem is that of dexterity: robots should learn how to approach, grasp and pick up complex objects, and how to place or arrange them into a desired configuration. The second problem is that of communication: how to communicate the *intent* of the task at hand, so that the robot can replicate it in a broader set of initial conditions.

Demonstrations are an extremely convenient form of information we can use to teach robots to overcome these two challenges. Using demonstrations, we can unambiguously communicate essentially any manipulation task, and simultaneously provide clues about the specific motor skills required to perform the task. We can compare this with an alternative form of communication, namely natural language. Although language is highly versatile, effective, and efficient, natural language processing systems are not yet at a level where we could easily use language to precisely describe a complex task to a robot. Compared to language, using demonstrations has two fundamental advantages: first, it does not require the knowledge of language, as it is possible to communicate complex tasks to humans that don't speak one's language. And second, there are many tasks that are extremely difficult to explain in words, even if we assume perfect linguistic abilities: for example, explaining how to swim without demonstration and experience seems to be, at the very least, an extremely challenging task.

Indeed, learning from demonstrations have had many successful applications. However, so far these applications have either required careful feature engineering, or a significant amount of system interaction time. This is far from what what we desire: ideally,



(a) Traditional Imitation Learning

(b) One-Shot Imitation Learning

(c) Training the One-Shot Imitator

Figure 12: (a) Traditionally, policies are task-specific. For example, a policy might have been trained through an imitation learning algorithm to stack blocks into towers of height 3, and then another policy would be trained to stack blocks into towers of height 2, etc. (b) In this chapter, we are interested in training networks that are *not* specific to one task, but rather can be told (through a single demonstration) what the current new task is, and be successful at this new task. For example, when it is conditioned on a single demonstration for task F, it should behave like a good policy for task F. (c) We can phrase this as a supervised learning problem, where we train this network on a set of training tasks, and with enough examples it should generalize to unseen, but related tasks. To train this network, in each iteration we sample a demonstration from one of the training tasks, and feed it to the network. Then, we sample another pair of observation and action from a second demonstration of the same task. When conditioned on both the first demonstration and this observation, the network is trained to output the corresponding action.

we hope to demonstrate a certain task only once or a few times to the robot, and have it instantly generalize to new situations of the same task, without long system interaction time or domain knowledge about individual tasks.

In this chapter we explore the one-shot imitation learning setting illustrated in Fig. 12, where the objective is to maximize the expected performance of the learned policy when faced with a new, previously unseen, task, and having received as input only one demonstration of that task. For the tasks we consider, the policy is expected to achieve good

performance without any additional system interaction, once it has received the demonstration.

We train a policy on a broad distribution over tasks, where the number of tasks is potentially infinite. For each training task we assume the availability of a set of successful demonstrations. Our learned policy takes as input: (i) the current observation, and (ii) one demonstration that successfully solves a different instance of the same task (this demonstration is fixed for the duration of the episode). The policy outputs the current controls. We note that any pair of demonstrations for the same task provides a supervised training example for the neural net policy, where one demonstration is treated as the input, while the other as the output.

To make this model work, we made essential use of soft attention (Bahdanau et al., 2015) for processing both the (potentially long) sequence of states and action that correspond to the demonstration, and for processing the components of the vector specifying the locations of the various blocks in our environment. The use of soft attention over both types of inputs made strong generalization possible. In particular, on a family of block stacking tasks, our neural network policy was able to perform well on novel block configurations which were not present in any training data. Videos of our experiments are available at `http://bit.ly/nips2017-oneshot`.

4.2 METHOD

4.2.1 *Problem Formalization*

We denote a distribution of tasks by $\mathbb{T}$, an individual task by $t \sim \mathbb{T}$, and a distribution of demonstrations for the task t by $\mathbb{D}(t)$. A policy is symbolized by $\pi_\theta(a|o, d)$, where a is an action, o is an observation, d is a demonstration, and θ are the parameters of the policy. A demonstration $d \sim \mathbb{D}(t)$ is a sequence of observations and actions : $d = [(o_1, a_1), (o_2, a_2), \ldots, (o_T, a_T)]$. We assume that the distribution of tasks $\mathbb{T}$ is given, and that we can obtain successful demonstrations for each task. We assume that there is some scalar-valued evaluation function $R_t(d)$ (e.g. a binary value indicating success) for each task, although this is not required during training. The objective is to maximize the expected performance of the policy, where the expectation is taken over tasks $t \in \mathbb{T}$, and demonstrations $d \in \mathbb{D}(t)$.

4.2.2 *Example Settings*

To clarify the problem setting, we describe two concrete examples, which we will also later study in the experiments.

4.2.2.1 *Particle Reaching*

The particle reaching problem is a very simple family of tasks. In each task, we control a point robot to reach a specific landmark, and different tasks are identified by different landmarks. As illustrated in Fig. 13, one task could be to reach the orange square, and another task could be to reach the green triangle. The agent receives its own 2D location, as well as the 2D locations of each of the landmarks. Within each task, the initial position of the agent, as well as the positions of all the landmarks, can vary across different instances of the task.

Without a demonstration, the robot does not know which landmark it should reach, and will not be able to accomplish the task. Hence, this setting already gets at the essence of one-shot imitation, namely to communicate the task via a demonstration. After learning, the agent should be able to identify the target landmark from the demonstration, and reach the same landmark in a new instance of the task.

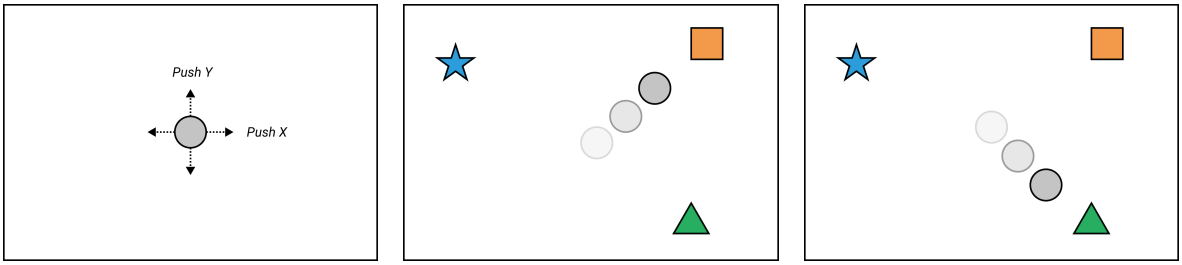


Figure 13: The robot is a point mass controlled with 2-dimensional force. The family of tasks is to reach a target landmark. The identity of the landmark differs from task to task, and the model has to figure out which target to pursue based on the demonstration. (left) illustration of the robot; (middle) the task is to reach the orange box, (right) the task is to reach the green triangle.

4.2.2.2 *Block Stacking*

We now consider a more challenging set of tasks, which requires more advanced manipulation skills, and where different tasks share a compositional structure, which allows

us to investigate nontrivial generalization to unseen tasks. In the block stacking tasks family, the goal is to control a 7-DOF Fetch robotic arm [10] (see Fig. 14 for a simulated model of the robot) to stack various numbers of cube-shaped blocks into configurations specified by the user. Each configuration consists of a list of blocks arranged into towers of different heights, and can be identified by a string such as `ghij` or `ab cd ef gh`, as illustrated in Fig. 15 and Fig. 16. Each of these configurations correspond to a different task. In a typical task, an observation is a list of (x, y, z) object positions relative to the gripper, and information if gripper is opened or closed. The number of objects may vary across different task instances.

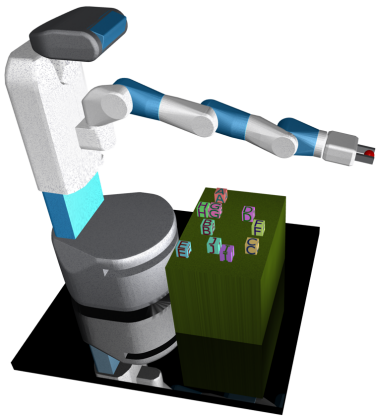


Figure 14: The tasks are to control a Fetch robotic arm to stack blocks into various layouts. This figure shows an example of the initial state, where blocks are randomly placed on the table.

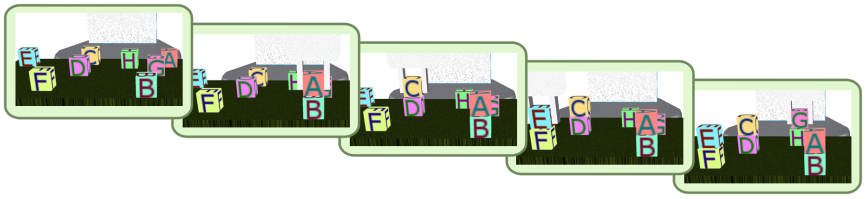


Figure 15: An entire episode can take up to several thousand time-steps. We define a *stage* as a single operation of stacking one block on top of another. This figure shows a trajectory of stacking 4 towers of height 2 each, where block A is on top of block B, block C is on top of block D, block E is on top of block F, and block G is on top of block H. This task has 4 stages, and is identified as `ab cd ef gh`.

[10] http://fetchrobotics.com/

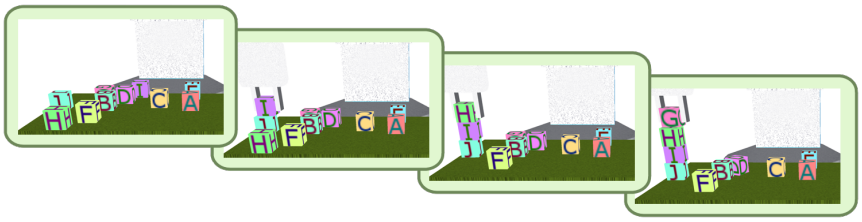


Figure 16: Trajectory of stacking 1 block tower of height 4, where block G is on top of block H, block H is on top of block I, and block I is on top of block J. This task has 3 stages, and is identified as `ghij`.

4.2.3 *Algorithm*

In order to train the neural network policy, we make use of imitation learning algorithms such as behavioral cloning and DAGGER (Ross et al., 2011), which only require demonstrations rather than reward functions to be specified. This has the potential to be more scalable, since it is often easier to demonstrate a task than specifying a well-shaped reward function (A. Y. Ng et al., 1999).

We start by collecting a set of demonstrations for each task, where we add noise to the actions in order to have wider coverage in the trajectory space. In each training iteration, we sample a list of tasks (with replacement). For each sampled task, we sample a demonstration as well as a small batch of observation-action pairs. The policy is trained to regress against the desired actions when conditioned on the current observation and the demonstration, by minimizing an ℓ_2 or cross-entropy loss based on whether actions are continuous or discrete. A high-level illustration of the training procedure is given in Fig. 12(c). Across all experiments, we use Adamax (D. Kingma and Ba, 2014) to perform the optimization with a learning rate of 0.001.

4.3 ARCHITECTURE

While, in principle, a generic neural network could learn the mapping from demonstration and current observation to appropriate action, we found it important to use an appropriate architecture. Our architecture for learning block stacking is one of the main contributions of this paper, and we believe it is representative of what architectures for one-shot imitation learning of more complex tasks could look like in the future. Although the particle task is simpler, we also found architectural decisions to be important, and we consider several choices below to be evaluated in Section 4.4.1.

4.3.1 *Architecture for Particle Reaching*

We consider three architectures for this problem:

- **Plain LSTM:** The first architecture is a simple LSTM (Hochreiter and Schmidhuber, 1997) with 512 hidden units. It reads the demonstration trajectory, the output of which is then concatenated with the current state, and fed to a multi-layer perceptron (MLP) to produce the action.
- **LSTM with attention:** In this architecture, the LSTM outputs a weighting over the different landmarks from the demonstration sequence. Then, it applies this weighting in the test scene, and produces a weighted combination over landmark positions given the current state. This 2D output is then concatenated with the current agent position, and fed to an MLP to produce the action.
- **Final state with attention:** Rather than looking at the entire demonstration trajectory, this architecture only looks at the final state in the demonstration (which is already sufficient to communicate the task), and produce a weighting over landmarks. It then proceeds like the previous architecture.

Notice that these three architectures are increasingly more specialized to the specific particle reaching setting, which suggests a potential trade-off between expressiveness and generalizability. We will quantify this tradeoff in Section 4.4.1.

4.3.2 *Architecture for Block Stacking*

For the block stacking task, it is desirable that the policy architecture has the following properties:

1. It should be easy to apply to task instances that have varying number of blocks.
2. It should naturally generalize to different permutations of the same task. For instance, the policy should perform well on task `dcba`, even if it is only trained on task `abcd`.
3. It should accommodate demonstrations of variable lengths.

Our proposed architecture consists of three modules: the demonstration network, the context network, and the manipulation network. An illustration of the architecture is shown in Fig. 17. We will describe the main operations performed in each module below, and a full specification is available in the Appendix.

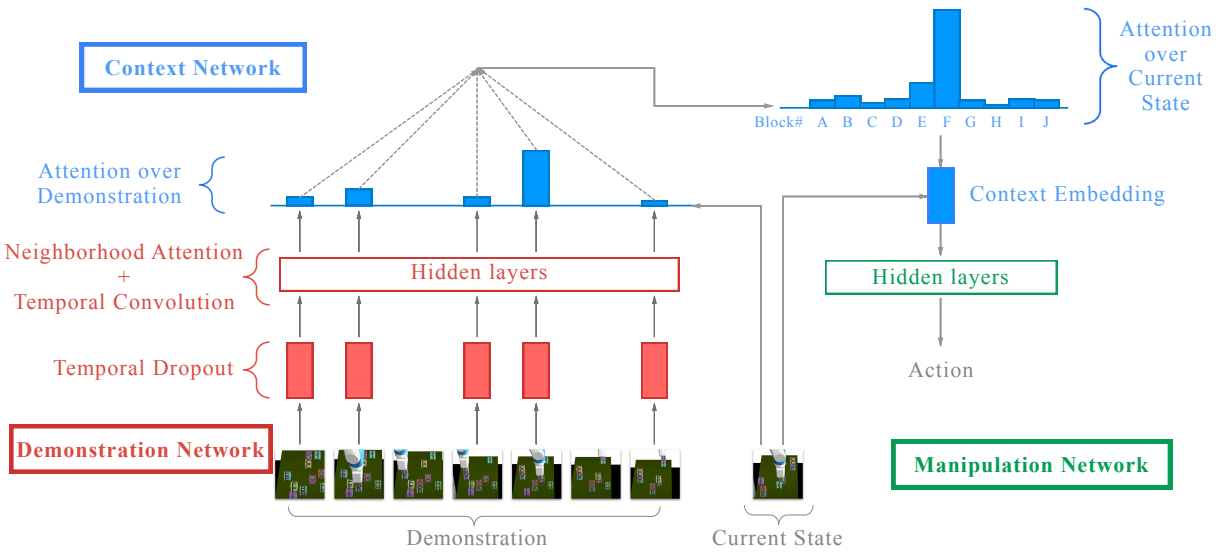


Figure 17: Illustration of the network architecture.

4.3.2.1 *Demonstration Network*

The demonstration network receives a demonstration trajectory as input, and produces an embedding of the demonstration to be used by the policy. The size of this embedding grows linearly as a function of the length of the demonstration as well as the number of blocks in the environment.

Temporal Dropout: For block stacking, the demonstrations can span hundreds to thousands of time steps, and training with such long sequences can be demanding in both time and memory usage. Hence, we randomly discard a subset of time steps during training, an operation we call *temporal dropout*, analogous to (Srivastava et al., 2014; Krueger et al., 2016). We denote p as the proportion of time steps that are thrown away. In our experiments, we use $p = 0.95$, which reduces the length of demonstrations by a factor of 20. During test time, we can sample multiple downsampled trajectories, use each of them to compute downstream results, and average these results to produce an ensemble estimate. In our experience, this consistently improves the performance of the policy.

Neighborhood Attention: After downsampling the demonstration, we apply a sequence of operations, composed of dilated temporal convolution (F. Yu and Koltun, 2016) and neighborhood attention. We now describe this second operation in more detail.

Since our neural network needs to handle demonstrations with variable numbers of

blocks, it must have modules that can process variable-dimensional inputs. Soft attention is a natural operation which maps variable-dimensional inputs to fixed-dimensional outputs. However, by doing so, it may lose information compared to its input. This is undesirable, since the amount of information contained in a demonstration grows as the number of blocks increases. Therefore, we need an operation that can map variable-dimensional inputs to outputs with comparable dimensions. Intuitively, rather than having a single output as a result of attending to all inputs, we have as many outputs as inputs, and have each output attending to all other inputs in relation to its own corresponding input.

We start by describing the soft attention module as specified in (Bahdanau et al., 2015). The input to the attention includes a query q, a list of context vectors $\{c_j\}$, and a list of memory vectors $\{m_j\}$. The ith attention weight is given by $w_i \leftarrow v^T \tanh(q + c_i)$, where v is a learned weight vector. The output of attention is a weighted combination of the memory content, where the weights are given by a softmax operation over the attention weights. Formally, we have $\text{output} \leftarrow \sum_i m_i \frac{\exp(w_i)}{\sum_j \exp(w_j)}$. Note that the output has the same dimension as a memory vector. The attention operation can be generalized to multiple query heads, in which case there will be as many output vectors as there are queries.

Now we turn to neighborhood attention. We assume there are B blocks in the environment. We denote the robot's state as s_{robot}, and the coordinates of each block as $(x_1, y_1, z_1), \ldots, (x_B, y_B, z_B)$. The input to neighborhood attention is a list of embeddings $h_1^{\text{in}}, \ldots, h_B^{\text{in}}$ of the same dimension, which can be the result of a projection operation over a list of block positions, or the output of a previous neighborhood attention operation. Given this list of embeddings, we use two separate linear layers to compute a query vector and a context embedding for each block: $q_i \leftarrow \text{Linear}(h_i^{\text{in}})$, and $c_i \leftarrow \text{Linear}(h_i^{\text{in}})$. The memory content to be extracted consists of the coordinates of each block, concatenated with the input embedding. The ith query result is given by the following soft attention operation: $\text{result}_i \leftarrow \text{SoftAttn}(\text{query: } q_i, \text{context: } \{c_j\}_{j=1}^B, \text{memory: } \{((x_j, y_j, z_j), h_j^{\text{in}}))\}_{j=1}^B)$.

Intuitively, this operation allows each block to query other blocks in relation to itself (e.g. find the closest block), and extract the queried information. The gathered results are then combined with each block's own information, to produce the output embedding per block. Concretely, we have $\text{output}_i \leftarrow \text{Linear}(\text{concat}(h_i^{\text{in}}, \text{result}_i, (x_i, y_i, z_i), s_{\text{robot}}))$. In practice, we use multiple query heads per block, so that the size of each result_i will be proportional to the number of query heads.

4.3.2.2 *Context network*

The context network is the crux of our model. It processes both the current state and the embedding produced by the demonstration network, and outputs a context embedding, whose dimension does not depend on the length of the demonstration, or the number of blocks in the environment. Hence, it is forced to capture only the relevant information, which will be used by the manipulation network.

Attention over demonstration: The context network starts by computing a query vector as a function of the current state, which is then used to attend over the different time steps in the demonstration embedding. The attention weights over different blocks within the same time step are summed together, to produce a single weight per time step. The result of this temporal attention is a vector whose size is proportional to the number of blocks in the environment. We then apply neighborhood attention to propagate the information across the embeddings of each block. This process is repeated multiple times, where the state is advanced using an LSTM cell with untied weights.

Attention over current state: The previous operations produce an embedding whose size is independent of the length of the demonstration, but still dependent on the number of blocks. We then apply standard soft attention over the current state to produce fixed-dimensional vectors, where the memory content only consists of positions of each block, which, together with the robot's state, forms the *context embedding*, which is then passed to the manipulation network.

Intuitively, although the number of objects in the environment may vary, at each stage of the manipulation operation, the number of relevant objects is small and usually fixed. For the block stacking environment specifically, the robot should only need to pay attention to the position of the block it is trying to pick up (the *source* block), as well as the position of the block it is trying to place on top of (the *target* block). Therefore, a properly trained network can learn to match the current state with the corresponding stage in the demonstration, and infer the identities of the source and target blocks expressed as soft attention weights over different blocks, which are then used to extract the corresponding positions to be passed to the manipulation network. Although we do not enforce this interpretation in training, our experiment analysis supports this interpretation of how the learned policy works internally.

4.3.2.3 *Manipulation network*

The manipulation network is the simplest component. After extracting the information of the source and target blocks, it computes the action needed to complete the current stage

of stacking one block on top of another one, using a simple MLP network.[1] This division of labor opens up the possibility of modular training: the manipulation network may be trained to complete this simple procedure, without knowing about demonstrations or more than two blocks present in the environment. We leave this possibility for future work.

4.4 EXPERIMENTS

4.4.1 *Particle Reaching*

To demonstrate the key concepts that underlie the one-shot imitation learning framework, we conduct experiments with the simple 2D particle reaching task described in Section 4.2.2.1. We consider an increasingly difficult set of task families, where the number of landmarks increases from 2 to 10. For each task family, we collect 10000 trajectories for training, where the positions of landmarks and the starting position of the point robot are randomized. We use a hard-coded expert policy to efficiently generate demonstrations. We add noise to the trajectories by perturbing the computed actions before applying them to the environment, and we use simple behavioral cloning to train the neural network policy. The trained policy is evaluated on new scenarios and conditioned on new demonstration trajectories unseen during training.

We evaluate the performance of the three architectures described in Section 4.3.1. For the LSTM-based architectures, we apply dropout (Srivastava et al., 2014) to the fully connected layers, by zeroing out activations with probability 0.1 during training.

4.4.1.1 *Results*

The results are shown in Fig. 18. We observe that as the architecture becomes more specialized, we achieve much better generalization performance. For this simple task, it appears that conditioning on the entire demonstration hurts generalization performance, and conditioning on just the final state performs the best even without explicit regularization. This makes intuitive sense, since the final state already sufficiently characterizes the task at hand.

1 In principle, one can replace this module with an RNN module. But we did not find this necessary for the tasks we consider.

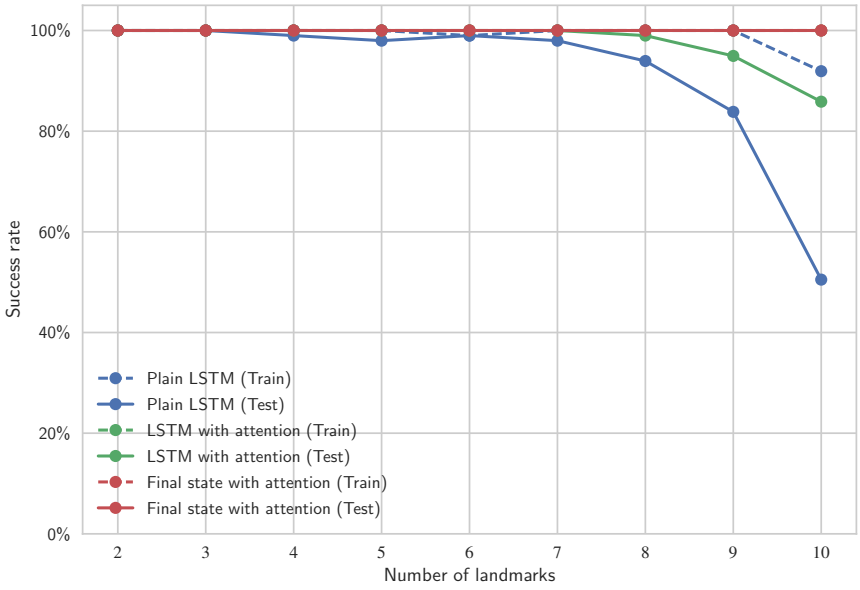
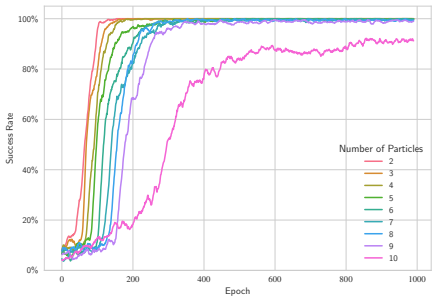


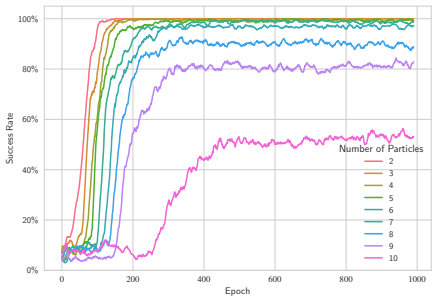
Figure 18: Success rates of different architectures for particle reaching. The "Train" curves show the success rates when conditioned on demonstrations seen during training, and running the policy on initial conditions seen during training, while the "Test" curves show the success rates when conditioned on new trajectories and operating in new situations. Both attention-based architectures achieve perfect training success rates, and the curves are overlapped.

However, the same conclusion does not appear to hold as the task becomes more complicated, as shown by the next set of experiments.

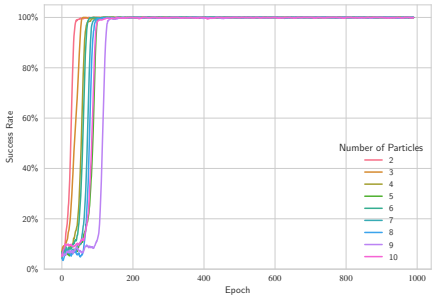
Fig. 19 shows the learning curves for the three architectures designed for the particle reaching tasks, as the number of landmarks is varied, by running the policies over 100 different configurations, and computing success rates over both training and test data. We can clearly observe that both LSTM-based architectures exhibit overfitting as the number of landmarks increases. On the other hand, using attention clearly improves generalization performance, and when conditioning on only the final state, it achieves perfect generalization in all scenarios. It is also interesting to observe that learning undergoes a phase transition. Intuitively, this may be when the network is learning to infer the task from the demonstration. Once this is finished, the learning of control policy is almost trivial.



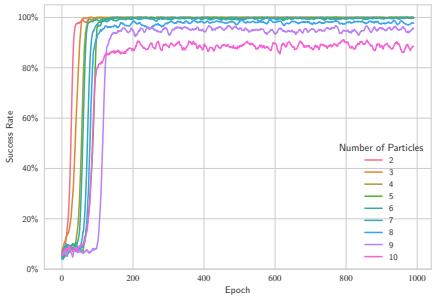
(a) Plain LSTM (Train)



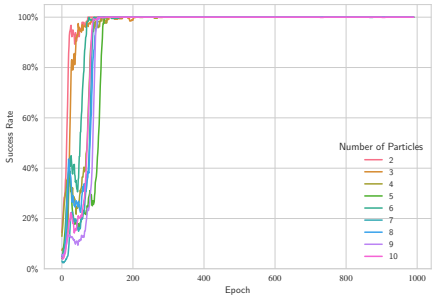
(b) Plain LSTM (Test)



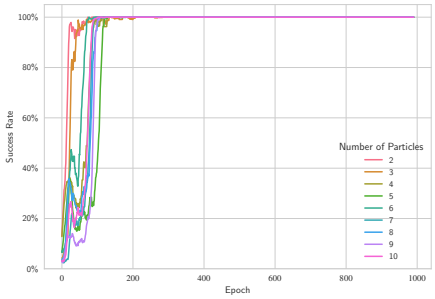
(c) LSTM with attention (Train)



(d) LSTM with attention (Test)



(e) Final state with attention (Train)



(f) Final state with attention (Test)

Figure 19: Learning curves for particle reaching tasks. Shown success rates are moving averages of past 10 epochs for smoother curves. Each policy is trained for up to 1000 epochs, which takes up to an hour using a Titan X Pascal GPU (as can be seen from the plot, most experiments can be finished sooner).

Table 30 and Table 31 show the exact performance numbers for reference.

#Landmarks	Plain LSTM	LSTM with attention	Final state with attention
2	**100.0%**	**100.0%**	**100.0%**
3	**100.0%**	**100.0%**	**100.0%**
4	**100.0%**	**100.0%**	**100.0%**
5	**100.0%**	**100.0%**	**100.0%**
6	99.0%	**100.0%**	**100.0%**
7	**100.0%**	**100.0%**	**100.0%**
8	**100.0%**	**100.0%**	**100.0%**
9	**100.0%**	**100.0%**	**100.0%**
10	91.9%	**100.0%**	**100.0%**

Table 30: Success rates of particle reaching conditioned on seen demonstrations, and running on seen initial configurations.

#Landmarks	Plain LSTM	LSTM with attention	Final state with attention
2	**100.0%**	**100.0%**	**100.0%**
3	**100.0%**	**100.0%**	**100.0%**
4	99.0%	**100.0%**	**100.0%**
5	98.0%	**100.0%**	**100.0%**
6	99.0%	**100.0%**	**100.0%**
7	98.0%	**100.0%**	**100.0%**
8	93.9%	99.0%	**100.0%**
9	83.8%	94.9%	**100.0%**
10	50.5%	85.9%	**100.0%**

Table 31: Success rates of particle reaching conditioned on unseen demonstrations, and running on unseen initial configurations.

4.4.2 *Block Stacking*

The particle reaching tasks nicely demonstrates the challenges in generalization in a simplistic scenario. However, the tasks do not share a compositional structure, making the evaluation of generalization to new tasks challenging. The skills and the information content required for each individual task are also simple. Hence, we conduct further experiments with the block stacking tasks described in Section 4.2.2.2. These experiments are designed to answer the following questions:

- How does training with behavioral cloning compare with DAGGER?
- How does conditioning on the entire demonstration compare to conditioning on the final state, even when it already has enough information to fully specify the task?
- How does conditioning on the entire demonstration compare to conditioning on a "snapshot" of the trajectory, which is a small subset of frames that are most informative?
- Can our framework generalize to tasks that it has never seen during training?
- What are the current limitations of the method?

To answer these questions, we compare the performance of the following architectures:

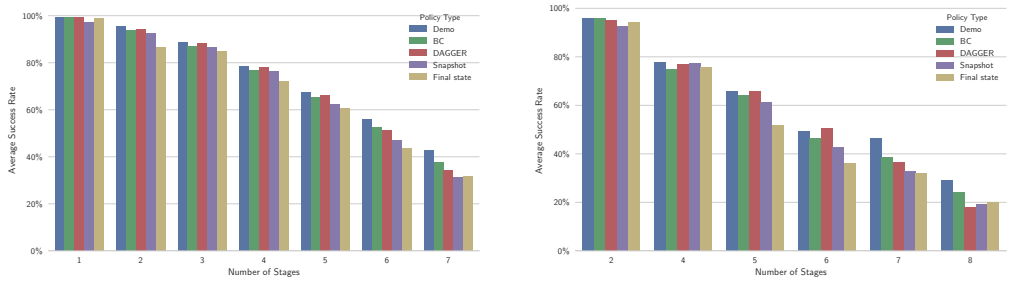
- **BC**: We use the same architecture as previous, but and the policy using behavioral cloning.
- **DAGGER**: We use the architecture described in the previous section, and train the policy using DAGGER.
- **Final state**: This architecture conditions on the final state rather than on the entire demonstration trajectory. For the block stacking task family, the final state uniquely identifies the task, and there is no need for additional information. However, a full trajectory, one which contains information about intermediate stages of the task's solution, can make it easier to train the optimal policy, because it could learn to rely on the demonstration directly, without needing to memorize the intermediate steps into its parameters. This is related to the way in which reward shaping can significantly affect performance in reinforcement learning (A. Y. Ng et al., 1999). A comparison between the two conditioning strategies will tell us whether this hypothesis is valid. We train this policy using DAGGER.
- **Snapshot**: This architecture conditions on a "snapshot" of the trajectory, which includes the last frame of each stage along the demonstration trajectory. This assumes that a segmentation of the demonstration into multiple stages is available at test time, which gives it an unfair advantage compared to the other condition-

ing strategies. Hence, it may perform better than conditioning on the full trajectory, and serves as a reference, to inform us whether the policy conditioned on the entire trajectory can perform as well as if the demonstration is clearly segmented. Again, we train this policy using DAGGER.

We evaluate the policy on tasks seen during training, as well as tasks unseen during training. Note that generalization is evaluated at multiple levels: the learned policy not only needs to generalize to new configurations and new demonstrations of tasks seen already, but also needs to generalize to new tasks. We also perform a thorough breakdown analysis of the failure scenarios as the difficulty of the task varies. Videos of our experiments are available at `http://bit.ly/one-shot-imitation`.

Concretely, we collect 140 training tasks, and 43 test tasks, each with a different desired layout of the blocks. The number of blocks in each task can vary between 2 and 10. We collect 1000 trajectories per task for training, and maintain a separate set of trajectories and initial configurations to be used for evaluation. The trajectories are collected using a hard-coded policy.

4.4.2.1 *Performance Evaluation*



(a) Performance on training tasks.

(b) Performance on test tasks.

Figure 20: Comparison of different conditioning strategies. The darkest bar shows the performance of the hard-coded policy, which unsurprisingly performs the best most of the time. For architectures that use temporal dropout, we use an ensemble of 10 different downsampled demonstrations and average the action distributions. Then for all architectures we use the greedy action for evaluation.

Fig. 20 shows the performance of various architectures. Results for training and test tasks are presented separately, where we group tasks by the number of stages required to complete them. This is because tasks that require more stages to complete are typically more

challenging. In fact, even our scripted policy frequently fails on the hardest tasks. We measure success rate per task by executing the greedy policy (taking the most confident action at every time step) in 100 different configurations, each conditioned on a different demonstration unseen during training. We report the average success rate over all tasks within the same group.

From the figure, we can observe that for the easier tasks with fewer stages, all of the different conditioning strategies perform equally well and almost perfectly. As the difficulty (number of stages) increases, however, conditioning on the entire demonstration starts to outperform conditioning on the final state. One possible explanation is that when conditioned only on the final state, the policy may struggle about which block it should stack first, a piece of information that is readily accessible from demonstration, which not only communicates the task, but also provides valuable information to help accomplish it.

More surprisingly, conditioning on the entire demonstration also seems to outperform conditioning on the snapshot, which we originally expected to perform the best. We suspect that this is due to the regularization effect introduced by temporal dropout, which effectively augments the set of demonstrations seen by the policy during training.

Another interesting finding was that training with behavioral cloning has the same level of performance as training with DAGGER, which suggests that the entire training procedure could work without requiring interactive supervision. In our preliminary experiments, we found that injecting noise into the trajectory collection process was important for behavioral cloning to work well, hence in all experiments reported here we use noise injection. In practice, such noise can come from natural human-induced noise through tele-operation, or by artificially injecting additional noise before applying it on the physical robot.

4.4.2.2 *Evaluating Permutation Invariance*

During training and in the previous evaluations, we only select one task per equivalence class, where two tasks are considered equivalent if they are the same up to permuting different blocks. This is based on the assumption that our architecture is invariant to permutations among different blocks. For example, if the policy is only trained on the task `abcd`, it should perform well on task `dcba`, given a single demonstration of the task `dcba`. We now experimentally verify this property by fixing a training task, and evaluating the policy's performance under all equivalent permutations of it. As Fig. 21 shows, although the policy has only seen the task `abcd`, it achieves the same level of performance on all other equivalent tasks.

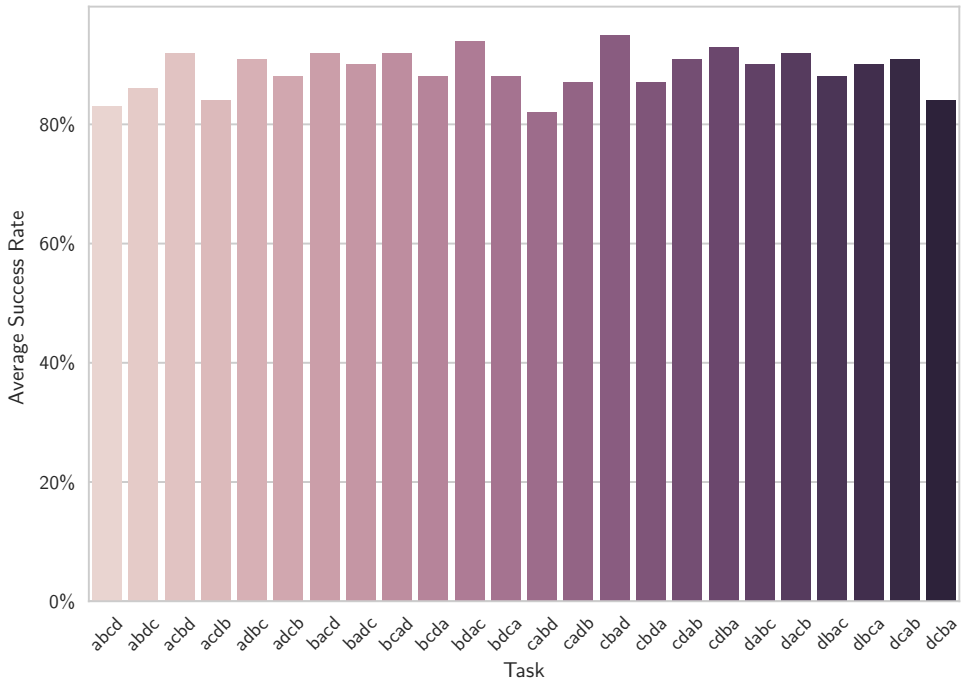


Figure 21: Performance of policy on a set of tasks equivalent up to permutations.

4.4.2.3 *Effect of Ensembling*

We now evaluate the importance of sampling multiple downsampled demonstrations during evaluation, which was introduced in Section 4.3.2.1. Fig. 22 shows the performance across all training and test tasks, as the number of ensembles varies from 1 to 20. We observe that more ensembles helps the most for tasks with fewer stages. On the other hand, it consistently improves performance for the harder tasks, although the gap is smaller. We suspect that this is because the policy has learned to attend to frames in the demonstration trajectory where the blocks are already stacked together. In tasks with only 1 stage, for example, it is very easy for these frames to be dropped in a single downsampled demonstration. On the other hand, in tasks with more stages, it becomes more resilient to missing frames. Using more than 10 ensembles appears to provide no significant improvements, and hence we used 10 ensembles in our main evaluation.

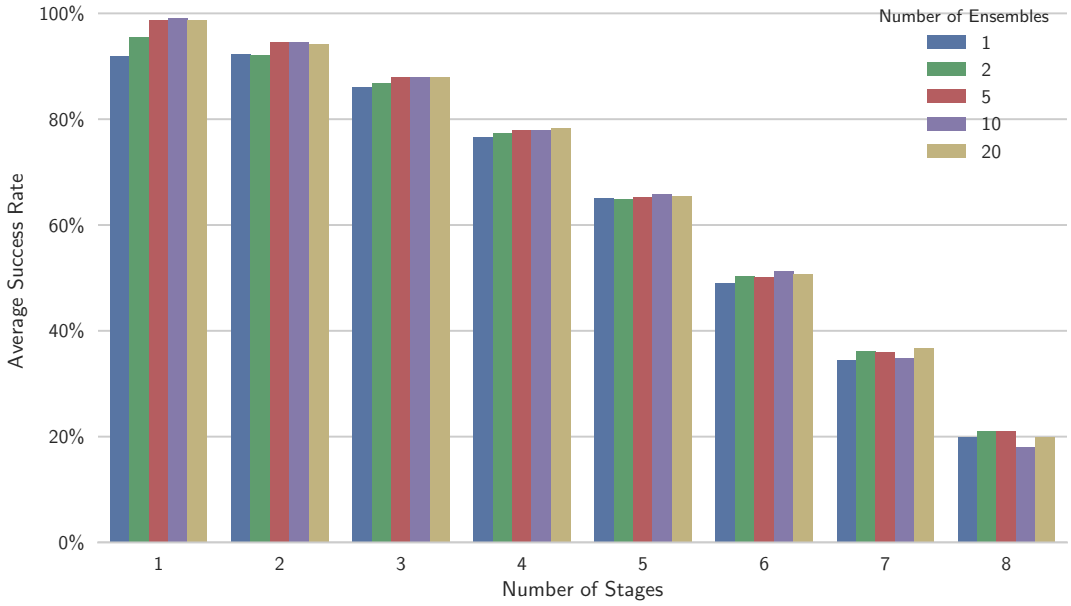


Figure 22: Performance of various number of ensembles.

4.4.2.4 *Breakdown of Failure Cases*

To understand the limitations of the current approach, we perform a breakdown analysis of the failure cases. We consider three failure scenarios: "Wrong move" means that the policy has arranged a layout incompatible with the desired layout. This could be because the policy has misinterpreted the demonstration, or due to an accidental bad move that happens to scramble the blocks into the wrong layout. "Manipulation failure" means that the policy has made an irrecoverable failure, for example if the block is shaken off the table, which the current hard-coded policy does not know how to handle. "Recoverable failure" means that the policy runs out of time before finishing the task, which may be due to an accidental failure during the operation that would have been recoverable given more time. As shown in Fig. 23, conditioning on only the final state makes more wrong moves compared to other architectures. Apart from that, most of the failure cases are actually due to manipulation failures that are mostly irrecoverable.[2] This suggests that

2 Note that the actual ratio of misinterpreted demonstrations may be different, since the runs that have caused a manipulation failure could later lead to a wrong move, were it successfully executed. On the other hand, by visually inspecting the videos, we observed that most of the trajectories categorized as "Wrong Move" are actually due to manipulation failures (except for policy conditioning on the final state,

better manipulation skills need to be acquired to make the learned one-shot policy more reliable.

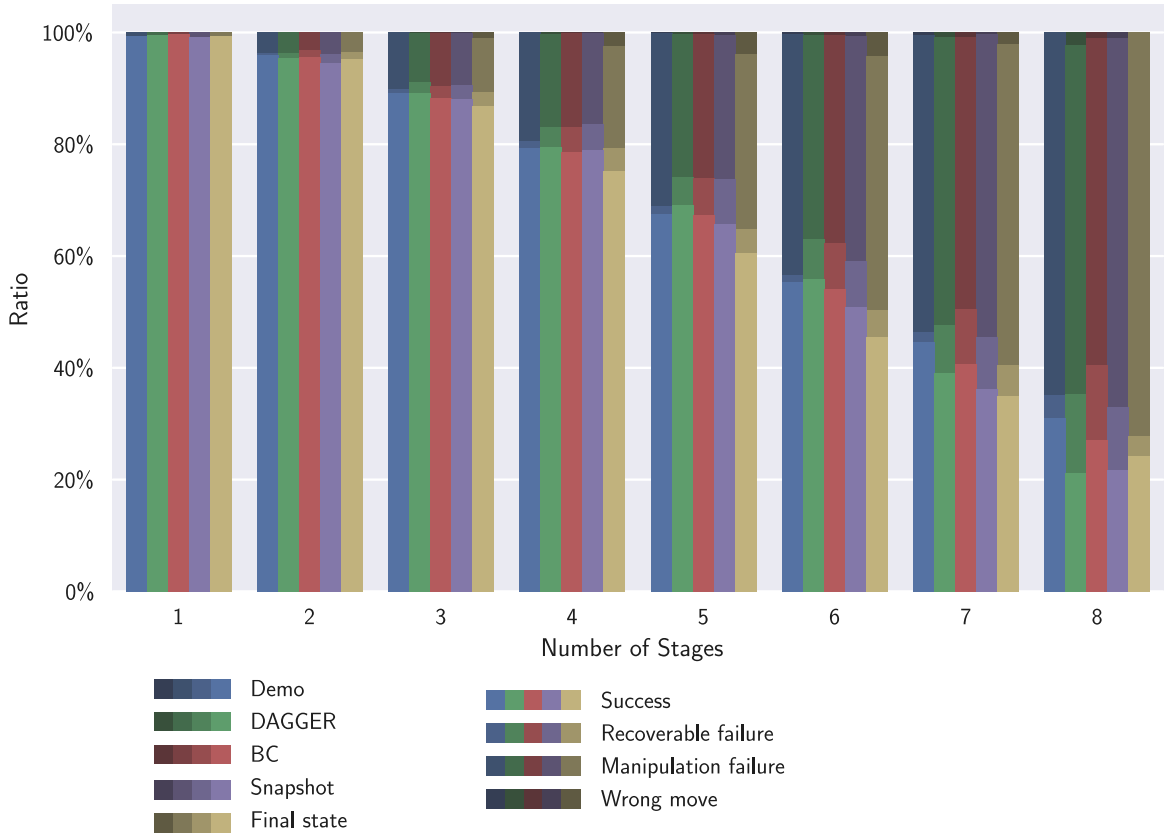
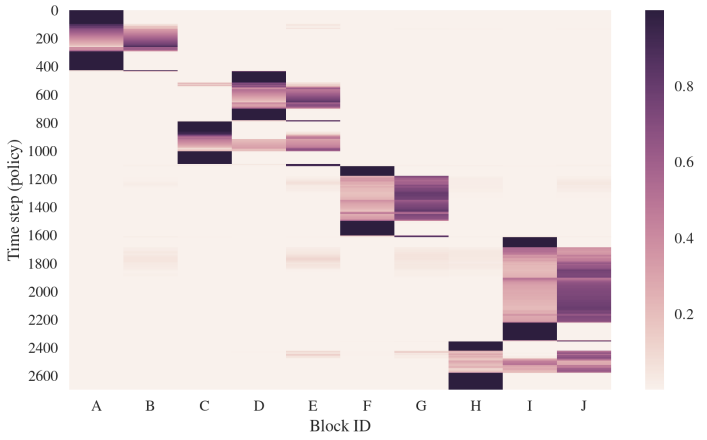


Figure 23: Breakdown of the success and failure scenarios. The area that each color occupies represent the ratio of the corresponding scenario.

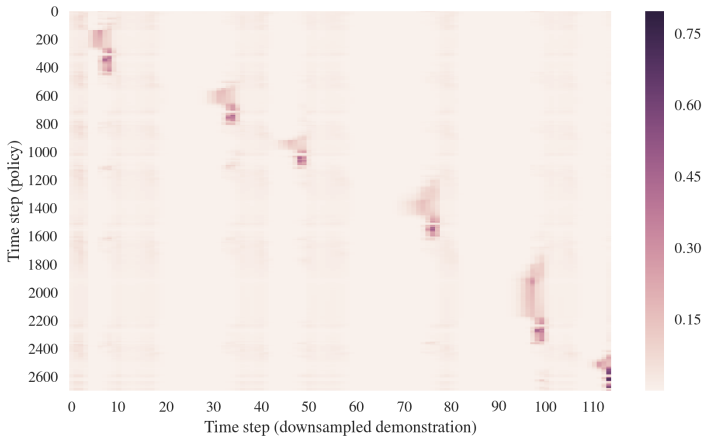
4.4.2.5 *Visualization*

We visualize the attention mechanisms underlying the main policy architecture to have a better understanding about how it operates. There are two kinds of attention we are mainly interested in, one where the policy attends to different time steps in the demonstration, and the other where the policy attends to different blocks in the current state. Fig. 24 shows some of the attention heatmaps.

which does seem to occasionally execute an actual wrong move).



(a) Attention over blocks in the current state.



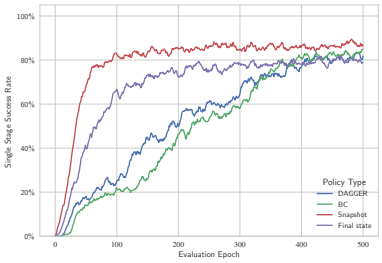
(b) Attention over downsampled demonstration.

Figure 24: Visualizing attentions performed by the policy during an entire execution. The task being performed is `ab cde fg hij`. Note that the policy has multiple query heads for each type of attention, and only one query head per type is visualized. (a) We can observe that the policy almost always focuses on a small subset of the block positions in the current state, which allows the manipulation network to generalize to operations over different blocks. (b) We can observe a sparse pattern of time steps that have high attention weights. This suggests that the policy has essentially learned to segment the demonstrations, and only attend to important key frames. Note that there are roughly 6 regions of high attention weights, which nicely corresponds to the 6 stages required to complete the task.

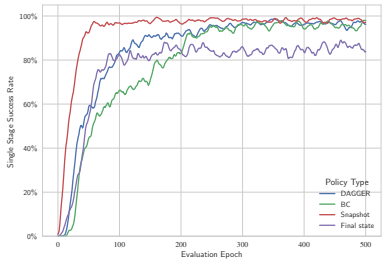
4.4.2.6 *Learning Curves*

Fig. 25 shows the learning curves for different architectures designed for the block stacking tasks. These learning curves do not reflect final performance: for each evaluation point, we sample tasks and demonstrations from training data, reset the environment to the starting point of some particular stage (so that some blocks are already stacked), and only run the policy for up to one stage. If the training algorithm is DAGGER, these sampled trajectories are annotated and added to the training set. Hence this evaluation does not evaluate generalization. We did not perform full evaluation as training proceeds, because it is very time consuming: each evaluation requires tens of thousands of trajectories across over > 100 tasks. However, these figures are still useful to reflect some relative trend.

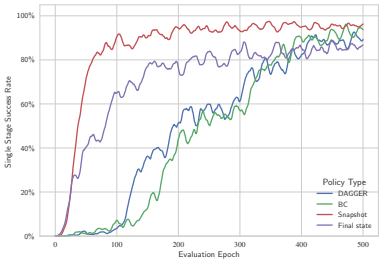
From these figures, we can observe that while conditioning on full trajectories gives the best performance which was shown in the main text, it requires much longer training time, simply because conditioning on the entire demonstration requires more computation. In addition, this may also be due to the high variance of the training process due to downsampling demonstrations, as well as the fact that the network needs to learn to properly segment the demonstration. It is also interesting that conditioning on snapshots seems to learn faster than conditioning on just the final state, which again suggests that conditioning on intermediate information is helpful, not only for the final policy, but also to facilitate training. We also observe that learning happens most rapidly for the initial stages, and much slower for the later stages, since manipulation becomes more challenging in the later stages. In addition, there are fewer tasks with more stages, and hence the later stages are not sampled as frequently as the earlier stages during evaluation.



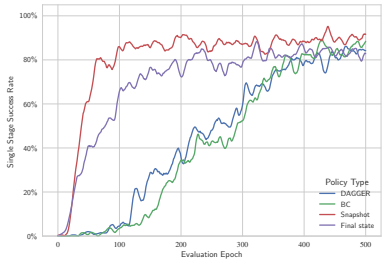
(a) All Stages



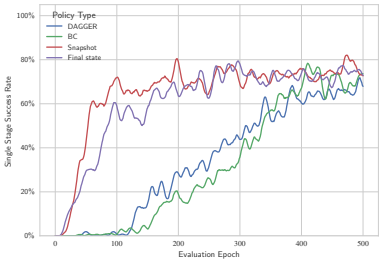
(b) Stage 0



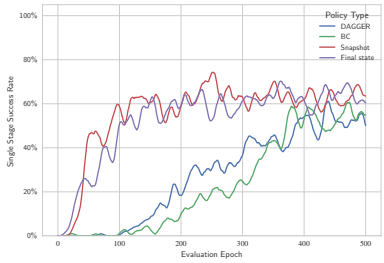
(c) Stage 1



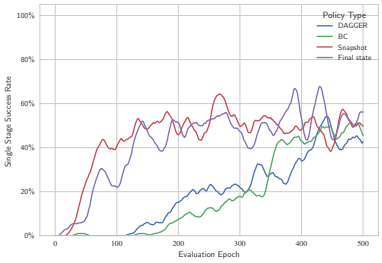
(d) Stage 2



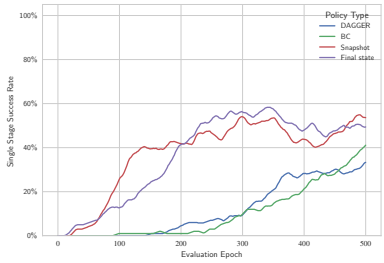
(e) Stage 3



(f) Stage 4



(g) Stage 5



(h) Stage 6

Figure 25: Learning curves of block stacking task. The first plot shows the average success rates over initial configurations of all stages. The subsequent figures shows the breakdown of each stage. For instance, "Stage 3" means that the first 3 stacking operations are already completed, and the policy is evaluated on its ability to perform the 4th stacking operation.

Imitation learning considers the problem of acquiring skills from observing demonstrations. Survey articles include (Schaal, 1999; Calinon, 2009; Argall et al., 2009).

Two main lines of work within imitation learning are behavioral cloning, which performs supervised learning from observations to actions (e.g., (Pomerleau, 1989; Ross et al., 2011)); and inverse reinforcement learning (A. Ng and S. Russell, 2000), where a reward function (Abbeel and A. Ng, 2004; Ziebart et al., 2008; Levine et al., 2011; Finn et al., 2016; Ho and Ermon, 2016) is estimated that explains the demonstrations as (near) optimal behavior. While this past work has led to a wide range of impressive robotics results, it considers each skill separately, and having learned to imitate one skill does not accelerate learning to imitate the next skill.

One-shot and few-shot learning has been studied for image recognition (Vinyals et al., 2016a; Koch, 2015; Santoro et al., 2016; Ravi and Larochelle, 2017), generative modeling (Edwards and Storkey, 2017; Rezende et al., 2016), and learning "fast" reinforcement learning agents with recurrent policies (Y. Duan et al., 2016b; J. X. Wang et al., 2016). Fast adaptation has also been achieved through fast-weights (Ba et al., 2016). Like our algorithm, many of the aforementioned approaches are a form of meta-learning (Thrun and Pratt, 1998; Schmidhuber, 1987; Naik and Mammone, 1992), where the algorithm itself is being learned. Meta-learning has also been studied to discover neural network weight optimization algorithms (S. Bengio et al., 1992; Y. Bengio et al., 1990; Hochreiter et al., 2001; Schmidhuber, 1992; Andrychowicz et al., 2016; K. Li and Malik, 2016). This prior work on one-shot learning and meta-learning, however, is tailored to respective domains (image recognition, generative models, reinforcement learning, optimization) and not directly applicable in the imitation learning setting. Recently, (Finn et al., 2017a) propose a generic framework for meta learning across several aforementioned domains. However they do not consider the imitation learning setting.

Reinforcement learning (R. S. Sutton and Barto, 1998; Bertsekas and Tsitsiklis, 1995) provides an alternative route to skill acquisition, by learning through trial and error. Reinforcement learning has had many successes, including Backgammon (Tesauro, 1995), helicopter control (A. Y. Ng et al., 2003), Atari (Mnih et al., 2015), Go (Silver et al., 2016), continuous control in simulation (Schulman et al., 2015; Heess et al., 2015b; T. P. Lillicrap et al., 2016) and on real robots (Peters and Schaal, 2008; Levine et al., 2016). However, reinforcement learning tends to require a large number of trials and requires specifying a reward function to define the task at hand. The former can be time-consuming and the latter can often be significantly more difficult than providing a demonstration (A. Ng

and S. Russell, 2000).

Multi-task and transfer learning considers the problem of learning policies with applicability and re-use beyond a single task. Success stories include domain adaptation in computer vision (Yang et al., 2007; Mansour et al., 2009; Kulis et al., 2011; Aytar and Zisserman, 2011; L. Duan et al., 2012; Hoffman et al., 2013; Long and J. Wang, 2015; Tzeng et al., 2014; Donahue et al., 2014) and control (Tzeng et al., 2015; Rusu et al., 2016a; Sadeghi and Levine, 2016; Gupta et al., 2017; B. Stadie et al., 2017). However, while acquiring a multitude of skills faster than what it would take to acquire each of the skills independently, these approaches do not provide the ability to readily pick up a new skill from a single demonstration.

Our approach heavily relies on an attention model over the demonstration and an attention model over the current observation. We use the soft attention model proposed in (Bahdanau et al., 2015) for machine translations, and which has also been successful in image captioning (Xu et al., 2015). The interaction networks proposed in (Battaglia et al., 2016; Chang et al., 2017) also leverage locality of physical interaction in learning. Our model is also related to the sequence to sequence model (Sutskever et al., 2014; Cho et al., 2014b), as in both cases we consume a very long demonstration sequence and, effectively, emit a long sequence of actions.

4.6 DISCUSSION

In this chapter, we presented a simple model that maps a single successful demonstration of a task to an effective policy that solves said task in a new situation. We demonstrated effectiveness of this approach on a family of block stacking tasks. There are a lot of exciting directions for future work. We plan to extend the framework to demonstrations in the form of image data, which will allow more end-to-end learning without requiring a separate perception module. We are also interested in enabling the policy to condition on multiple demonstrations, in case where one demonstration does not fully resolve ambiguity in the objective. Furthermore and most importantly, we hope to scale up our method on a much larger and broader distribution of tasks, and explore its potential towards a general robotics imitation learning system that would be able to achieve an overwhelming variety of tasks.

4.7 ADDITIONAL DETAILS ON BLOCK STACKING

4.7.1 *Full Description of Architecture*

We now specify the architecture in pseudocode. We omit implementation details which involve handling a minibatch of demonstrations and observation-action pairs, as well as necessary padding and masking to handle data of different dimensions. We use weight normalization with data-dependent initialization (Salimans and D. P. Kingma, 2016) for all dense and convolution operations.

4.7.1.1 *Demonstration Network*

Assume that the demonstration has T time steps and we have B blocks. Our architecture only make use of the observations in the input demonstration but not the actions. Each observation is a $(3B+2)$-dimensional vector, containing the (x, y, z) coordinates of each block relative to the current position of the gripper, as well as a 2-dimensional gripper state indicating whether it is open or closed.

Module 1 Demonstration Network

Input: Demonstration $d \in \mathbb{R}^{T \times (3B+2)}$

Hyperparameters: $p = 0.95$, $D = 64$

Output: Demonstration embedding $\in \mathbb{R}^{\tilde{T} \times B \times D}$, where $\tilde{T} = \lceil T(1-p) \rceil$ is the length of the downsampled trajectory.

```
d' ← TemporalDropout(d, probability=p)
block_state, robot_state ← Split(d')
h ← Conv1D(block_state, kernel_size=1, channels=D)
for a ∈ {1, 2, 4, 8} do
  // Residual connections
  h' ← ReLU(h)
  attn_result ← NeighborhoodAttention(h')
  h' ← Concat({h', block_state, robot_state}, axis=-1)
  h' ← Conv1D(h', kernel_size=2, channels=D, dilation=a)
  h' ← ReLU(h')
  h ← h + h'
end for
demo_embedding ← h
```

The full sequence of operations is given in Module 1. We first apply temporal dropout as described in the main text. Then we split the observation into information about the block and information about the robot, where the first dimension is time and the second dimension is the block ID. The robot state is broadcasted across different blocks. Hence the shape of outputs should be $\tilde{T} \times B \times 3$ and $\tilde{T} \times B \times 2$, respectively.

Then, we perform a 1×1 convolution over the block states to project them to the same dimension as the per-block embedding. Then we perform a sequence of neighborhood attention operations and 1×1 convolutions, where the input to the convolution is the concatenation of the attention result, the current block position, and the robot's state. This allows each block to query the state of other blocks, and reason about the query result in comparison with its own state and the robot's state. We use residual connections during this procedure.

4.7.1.2 *Context Network*

The pseudocode is shown in Module 2. We perform a series of attention operations over the demonstration, followed by attention over the current state, and we apply them repeatedly through an LSTM with different weights per time step (we found this to be slightly easier to optimize). Then, in the end we apply a final attention operation which produces a fixed-dimensional embedding independent of the length of the demonstration or the number of blocks in the environment.

Module 2 Context Network

Input: Demonstration embedding $h_{in} \in \mathbb{R}^{\tilde{T} \times B \times D}$, current state $s \in \mathbb{R}^{3B+2}$
Hyperparameters: $D = 64, t_{lstm} = 4, H = 2$
Output: Context embedding $\in \mathbb{R}^{2+6H}$

// Split the current state into block state $\in \mathbb{R}^{B \times 3}$ and robot state broadcasted to all blocks $\in \mathbb{R}^{B \times 2}$
`block_state, robot_state` $\leftarrow$ `SplitSingle(s)`
// Initialize LSTM output $\in \mathbb{R}^{B \times D}$ and state (including hidden and cell state) $\in \mathbb{R}^{B \times 2D}$
`output, state` $\leftarrow$ `InitLSTMState(size=B, hidden_dim=D)`
for $t = 1$ to t_{lstm} **do**
  // Temporal attention: every block attend to the same time step
  `x` $\leftarrow$ `output`
  if $t > 1$ **then**
    `x` $\leftarrow$ `ReLU(x)`
  end if
  // Computing query for attention over demonstration $\in \mathbb{R}^{B \times D}$
  `q` $\leftarrow$ `Dense(x, output_dim=D)`
  // Compute result from attention $\in \mathbb{R}^{H \times B \times D}$
  `temp` $\leftarrow$ `SoftAttention(query=q, context=h_in, memory=h_in, num_heads=H)`
  // Reorganize result into shape $B \times (HD)$
  `temp` $\leftarrow$ `Reshape(Transpose(temp, (1, 0, 2)), (B, H*D))`

  // Spatial attention: each block attend to a different block separately
  `x` $\leftarrow$ `output`
  if $t > 1$ **then**
    `x` $\leftarrow$ `ReLU(x)`
  end if
  `x` $\leftarrow$ `Concat({x, temp}, axis=-1)`
  // Computing context for attention over current state $\in \mathbb{R}^{B \times D}$
  `ctx` $\leftarrow$ `Dense(x, output_dim=D)`
  // Computing query for attention over current state $\in \mathbb{R}^{B \times D}$
  `q` $\leftarrow$ `Dense(x, output_dim=D)`
  // Computing memory for attention over current state $\in \mathbb{R}^{B \times (HD+3)}$
  `mem` $\leftarrow$ `Concat({block_state, temp}, axis=-1)`
  // Compute result from attention $\in \mathbb{R}^{B \times H \times (HD+3)}$
  `spatial` $\leftarrow$ `SoftAttention(query=q, context=ctx, memory=mem, num_heads=H)`
  // Reorganize result into shape $B \times H(HD+3)$
  `spatial` $\leftarrow$ `Reshape(spatial, (B, H*(H*D+3)))`
  // Form input to the LSTM cell $\in \mathbb{R}^{B \times (H(HD+3)+HD+8)}$
  `input` $\leftarrow$ `Concat({robot_state, block_state, spatial, temp}, axis=-1)`
  // Run one step of an LSTM with untied weights (meaning that we use different weights per time step
  `output, state` $\leftarrow$ `LSTMOneStep(input=input, state=state)`
end for
// Final attention over the current state, compressing an $O(B)$ representation down to $O(1)$
// Compute the query vector. We use a fixed, trainable query vector independent of the input data, with size $\in \mathbb{R}^{2 \times D}$ (we use two queries, originally intended to have one for the source block and one for the target block)
`q` $\leftarrow$ `GetFixedQuery()`
// Get attention result, which should be of shape $2 \times H \times 3$
`r` $\leftarrow$ `SoftAttention(query=q, context=output, memory=block_state, num_heads=H)` // Form the final context embedding (we pick the first robot state since no need to broadcast here)
`context_embedding` $\leftarrow$ `Concat({robot_state[0], Reshape(r, 2*H*3)})`

4.7.1.3 *Manipulation Network*

Given the context embedding, this module is simply a multilayer perceptron. Pseudocode is given in Module 3.

Module 3 Manipulation Network

Input: Context embedding $h_{in} \in \mathbb{R}^{2+6H}$
Hyperparameters: $H = 2$
Output: Predicted action distribution $\in \mathbb{R}^{|A|}$

```
h ← ReLU(Dense(h_in, output_dim=256))
h ← ReLU(Dense(h, output_dim=256))
action_dist ← Dense(h, output_dim=|A|)
```

4.7.2 *Exact Performance Numbers*

Exact performance numbers are presented for reference:

- Table 32 and Table 33 show the success rates of different architectures on training and test tasks, respectively;
- Table 34 shows the success rates across all tasks as the number of ensembles is varied;
- Table 35 shows the success rates of tasks that are equivalent to `abcd` up to permutations;
- Table 36, Table 37, Table 38, Table 39, and Table 40 show the breakdown of different success and failure scenarios for all considered architectures.

#Stages	Demo	DAGGER	BC	Snapshot	Final state
1	**99.1%**	**99.1%**	**99.1%**	97.2%	98.8%
2	**95.6%**	**94.3%**	93.7%	92.6%	86.7%
3	**88.5%**	**88.0%**	86.9%	86.7%	84.8%
4	**78.6%**	**78.2%**	76.7%	76.4%	71.9%
5	**67.3%**	**65.9%**	65.4%	62.5%	60.6%
6	**55.7%**	51.5%	**52.4%**	47.0%	43.6%
7	**42.8%**	34.3%	**37.5%**	31.4%	31.5%

Table 32: Success rates of different architectures on training tasks of block stacking.

#Stages	Demo	DAGGER	BC	Snapshot	Final state
2	95.8%	94.9%	**95.9%**	92.8%	94.1%
4	**77.6%**	77.0%	74.8%	**77.2%**	75.8%
5	**65.9%**	**65.9%**	64.3%	61.1%	51.9%
6	49.4%	**50.6%**	46.5%	42.6%	35.9%
7	**46.5%**	36.5%	**38.5%**	32.8%	32.0%
8	**29.0%**	18.0%	**24.0%**	19.0%	20.0%

Table 33: Success rates of different architectures on test tasks of block stacking.

#Stages	1 Ens.	2 Ens.	5 Ens.	10 Ens.	20 Ens.
1	91.9%	95.4%	98.8%	**99.1%**	98.7%
2	92.3%	92.2%	94.5%	**94.6%**	94.1%
3	86.0%	86.8%	87.9%	**88.0%**	87.9%
4	76.6%	77.4%	77.9%	78.0%	**78.3%**
5	65.1%	65.0%	65.3%	**65.9%**	65.5%
6	49.0%	50.4%	50.1%	**51.3%**	50.8%
7	34.4%	36.1%	36.0%	34.9%	**36.8%**
8	20.0%	**21.0%**	**21.0%**	18.0%	20.0%

Table 34: Success rates of varying number of ensembles using the DAGGER policy conditioned on full trajectories, across both training and test tasks.

Task ID	Success Rate
abcd	83.0%
abdc	86.0%
acbd	92.0%
acdb	84.0%
adbc	91.0%
adcb	88.0%
bacd	92.0%
badc	90.0%
bcad	92.0%
bcda	88.0%
bdac	94.0%
bdca	88.0%
cabd	82.0%
cadb	87.0%
cbad	95.0%
cbda	87.0%
cdab	91.0%
cdba	93.0%
dabc	90.0%
dacb	92.0%
dbac	88.0%
dbca	90.0%
dcab	91.0%
dcba	84.0%

Table 35: Success rates of a set of tasks that are equivalent up to permutations, using the DAGGER policy conditioned on full trajectories.

#Stages	Success	Recoverable failure	Manipulation failure	Wrong move
1	99.3%	0.0%	0.7%	0.0%
2	95.9%	0.4%	3.7%	0.0%
3	89.1%	0.7%	10.1%	0.1%
4	79.2%	1.2%	19.4%	0.1%
5	67.5%	1.4%	30.9%	0.2%
6	55.2%	1.4%	43.1%	0.3%
7	44.6%	1.7%	53.2%	0.6%
8	30.9%	4.3%	64.9%	0.0%

Table 36: Breakdown of success and failure scenarios for Demo policy.

#Stages	Success	Recoverable failure	Manipulation failure	Wrong move
1	99.4%	0.0%	0.6%	0.0%
2	95.3%	0.9%	3.8%	0.0%
3	89.1%	1.9%	8.8%	0.1%
4	79.5%	3.5%	16.7%	0.3%
5	69.1%	5.0%	25.6%	0.3%
6	55.8%	7.3%	36.4%	0.5%
7	39.0%	8.6%	51.5%	0.8%
8	21.2%	14.1%	62.4%	2.4%

Table 37: Breakdown of success and failure scenarios for DAGGER policy.

#Stages	Success	Recoverable failure	Manipulation failure	Wrong move
1	99.6%	0.0%	0.4%	0.0%
2	95.6%	1.1%	3.2%	0.1%
3	88.1%	2.2%	9.5%	0.2%
4	78.5%	4.5%	16.8%	0.2%
5	67.2%	6.6%	25.7%	0.4%
6	53.9%	8.3%	37.1%	0.6%
7	40.6%	9.8%	48.7%	0.9%
8	27.0%	13.5%	58.4%	1.1%

Table 38: Breakdown of success and failure scenarios for BC policy.

#Stages	Success	Recoverable failure	Manipulation failure	Wrong move
1	99.1%	0.0%	0.9%	0.0%
2	94.5%	1.6%	3.8%	0.1%
3	88.0%	2.5%	9.3%	0.2%
4	78.9%	4.6%	16.2%	0.3%
5	65.6%	8.0%	25.8%	0.6%
6	50.8%	8.3%	40.2%	0.7%
7	36.1%	9.2%	54.2%	0.4%
8	21.6%	11.4%	65.9%	1.1%

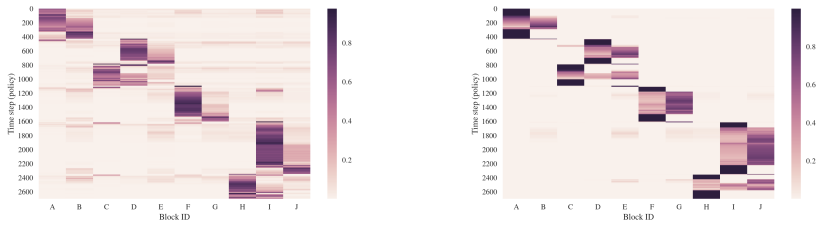
Table 39: Breakdown of success and failure scenarios for Snapshot policy.

#Stages	Success	Recoverable failure	Manipulation failure	Wrong move
1	99.2%	0.0%	0.8%	0.0%
2	95.1%	1.3%	3.6%	0.0%
3	86.7%	2.5%	9.7%	1.1%
4	75.2%	4.0%	18.3%	2.5%
5	60.5%	4.3%	31.2%	4.0%
6	45.5%	4.7%	45.5%	4.3%
7	34.9%	5.6%	57.3%	2.2%
8	24.1%	3.6%	72.3%	0.0%

Table 40: Breakdown of success and failure scenarios for Final state policy.

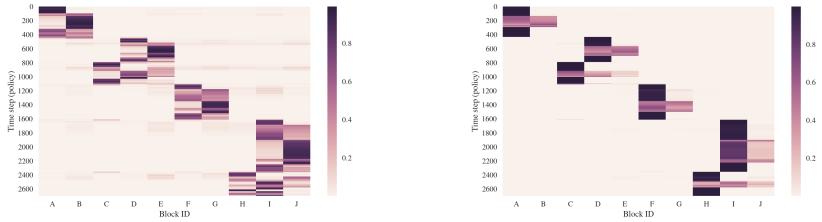
4.7.3 *More Visualizations*

Fig. 26 and Fig. 27 show the full set of heatmaps of attention weights. Interestingly, in Fig. 26, we observe that rather than attending to two blocks at a time, as we originally expected, the policy has learned to mostly attend to only one block at a time. This makes sense because during each of the grasping and the placing phase of a single stacking operation, the policy needs to only pay attention to the single block that the gripper should aim towards. For context, Fig. 28 and Fig. 29 show key frames of the neural network policy executing the task.



(a) Head 0

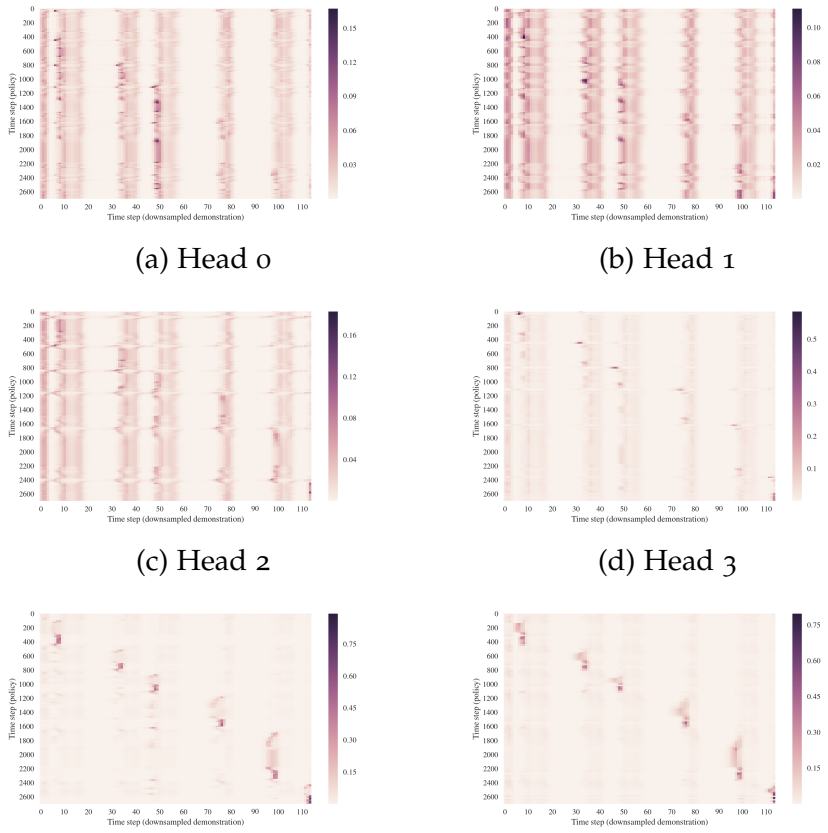
(b) Head 1



(c) Head 2

(d) Head 3

Figure 26: Heatmap of attention weights over different blocks of all 4 query heads.



(a) Head 0

(b) Head 1

(c) Head 2

(d) Head 3

(e) Head 4

(f) Head 5

Figure 27: Heatmap of attention weights over downsampled demonstration trajectory of all 6 query heads. There are 2 query heads per step of LSTM, and 3 steps of LSTM are performed.

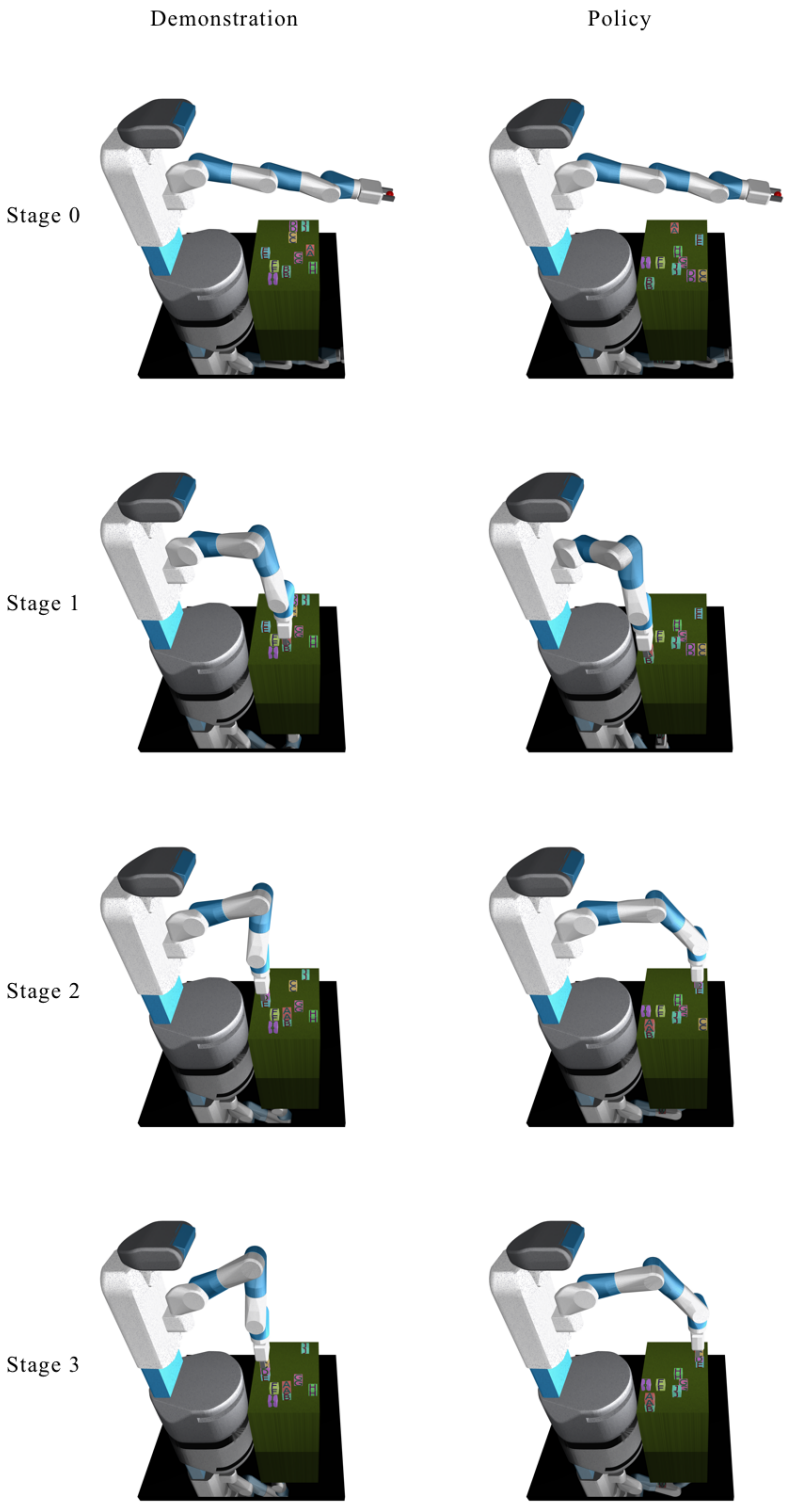


Figure 28: Illustration of the task used for the visualization of attention heatmaps (first half). The task is `ab cde fg hij`. The left side shows the key frames in the demonstration. The right side shows how, after seeing the entire demonstration, tthe policy reproduces the same layout in a new initialization of the same task.

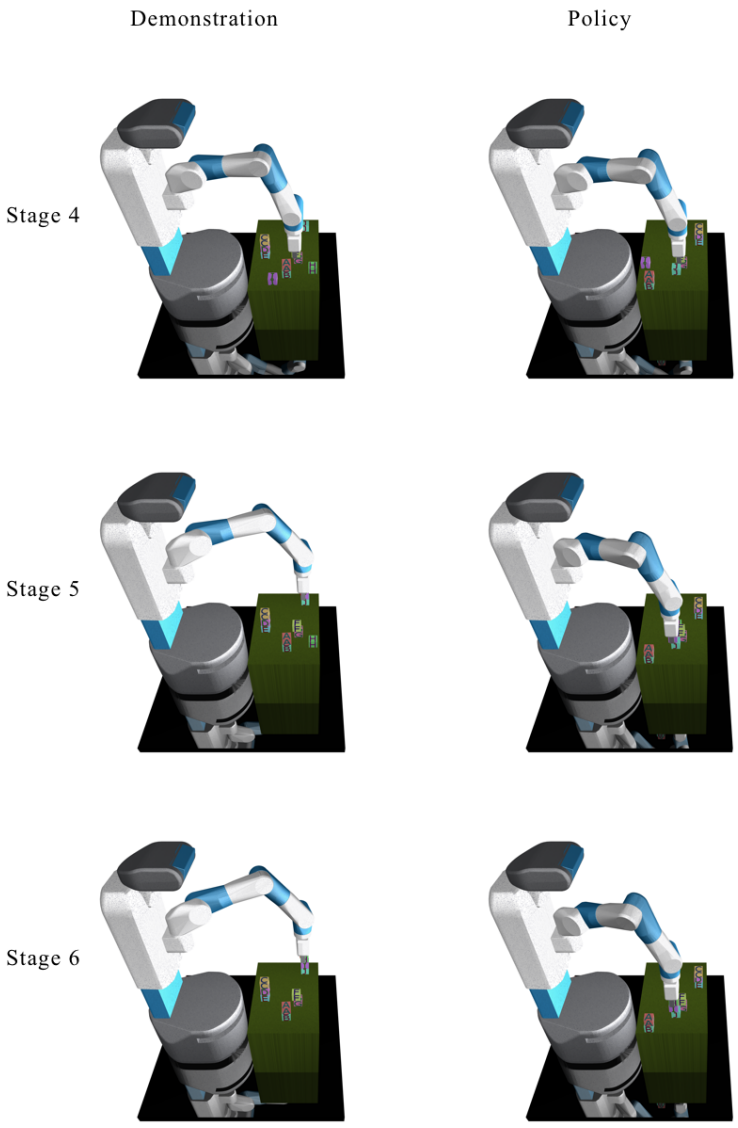


Figure 29: Illustration of the task used for the visualization of attention heatmaps (second half). The task is `ab cde fg hij`. The left side shows the key frames in the demonstration. The right side shows how, after seeing the entire demonstration, tthe policy reproduces the same layout in a new initialization of the same task.

5

CONCLUSION

In this thesis, we have investigated several instantiations of meta learning for control. We start by evaluating current state-of-the-art algorithms on a set of challenging benchmark environments, which motivate the need for better reinforcement learning algorithms. Then, rather than proposing hand-designed algorithms, we advocate a meta learning approach to automate the algorithm design process by structuring the algorithm as a general parameterized model. This model is differentiable so that it can be end-to-end optimized on a set of environments. The two meta learning frameworks that are presented, RL^2 and one-shot imitation learning, demonstrate that meta learning can be applied to complex, high-dimensional control problems given sufficient data. Meanwhile, analyses of these algorithms suggest that a lot more can be done. Below, we point out several important future directions that should be further investigated.

Data collection: As seen in Section 3.3.4, choosing an appropriate distribution of tasks for meta learning is challenging. A very restrictive distribution can render the problem too easy to solve, which can fail to evaluate important aspects of meta learning, such as whether the algorithm can learn to explore. While benchmark environments such as bandits, random MDPs, and visual navigation provide adequate challenge, they are less connected to real-world applications compared to datasets in supervised learning such as ImageNet. There has been much recent progress in this direction, including OpenAI Universe (OpenAI, 2017), a platform that can convert any video game into a reinforcement learning environment, and World of Bits (Shi et al., 2017), a collection of web navigation tasks. However, these environments can be computationally demanding for individual researchers and small research groups. It would be very valuable to develop benchmark datasets that are relevant, challenging, and yet computationally feasible. In addition, exciting opportunities remain in building datasets of tasks for real-world robotic applications.

Another angle is to further explore mechanisms for task specification. In RL^2, the task

is specified via a reward function, while in one-shot imitation learning, the task is specified by first-person expert demonstrations of the task. One can consider many possible alternatives, such as specifying a task through language (see), imperfect demonstrations (from which the learned algorithm should infer the underlying objective, and surpass the performance of the demonstrator), third-person demonstrations (see), or a combination of rewards and demonstrations.

Overcoming underfitting: In RL^2, we have observed that the learned algorithm can be outperformed by human-designed algorithms on the more challenging environments with long horizons, including multi-armed bandits and tabular MDPs. This can be considered a form of underfitting, where the algorithm fails to achieve good performance even on the training tasks (in our case, the algorithm has an infinite supply of training environments). On the other hand, the ablation study in Section 3.3.1 suggests that this underfitting behavior is not bottlenecked by the policy architecture (though (Mishra et al., 2017) indicate that architectural choices do matter), as policies trained via supervised learning achive much better performance. Hence the underfitting mostly happens at an algorithmic level rather than architectural: since the fast algorithm is trained via RL, it inherits the usual challenges such as exploration in long-horizon problems. This is very different from supervised learning, where underfitting is rarely a concern given a sufficiently expressive model. To overcome this issue, better meta learning algorithms need to be developed. Promising directions include incorporating auxiliary objectives, utilizing human demonstrations as additional supervision, and curriculum learning.

Overcoming overfitting: As discussed above, it can be challenging to collect a diverse set of tasks for meta learning. Therefore, it is particularly important to develop meta learning algorithms that can generalize well from a manageable number of training tasks, in the sense that it can perform well on new tasks sampled from the same distribution. One idea is to restrict the set of algorithms expressible by the parameterized model. At one extreme we have methods such as RL^2, TCML (Mishra et al., 2017), and one-shot imitation learning, which use generic recurrent architectures and are, in theory, capable of approximating any algorithm that can run on a turing machine (Siegelmann and Sontag, 1995). At the other extreme, we have methods that merely tune hyperparameters over a set of training tasks (Ishii et al., 2002; Schweighofer and Doya, 2003). There are many possibilities between these two extremes. For example, MAML (Finn et al., 2017a) and MIL (Finn et al., 2017b) restrict the underlying algorithm to be policy gradient or behavior cloning, and only meta-learn an initial set of parameters. There are many other possibilities by making certain components of an existing algorithm more flexible and amenable for optimization. We leave this to the reader as food for thought.

Another idea is to apply techniques common in supervised learning, such as data augmentation and regularization. We can take inspiration from domain randomization (Sadeghi and Levine, 2016; Tobin et al., 2017a; Tobin et al., 2017b; Peng et al., 2017), an effective technique for transferring neural networks from simulation to real world by drastically randomizing the visual and physical features within the virtual world. Combined with techniques such as Neural Style Transfer (Gatys et al., 2015) and Image-to-Image Translation (Isola et al., 2016; Zhu et al., 2017), we can potentially build powerful data augmentation operators over tasks. For regularization, we should ask ourselves the following question: how should we properly regularize the weights of the learned algorithm, so that it impose an implicit preference of simpler algorithms over more complex ones? Moreover, can we define a more proper metric of capacity, analogous to VC dimension (Vapnik and Chervonenkis, 1971), but over families of algorithms?

At an even more advanced level, we want to move beyond generalizing to new tasks from the same distribution. Eventually, we want to deploy learning systems with appropriate prior knowledge baked in, that can deal with distributions of tasks that slowly drift over time. This is a much more practical setting than assuming that the distribution over tasks never change, however it also makes learning more challenging. One promising idea is to jointly learn a dynamic distribution over tasks as well as an adaptive fast learning algorithm with an adversarial interaction between them (Goodfellow et al., 2014; Sukhbaatar et al., 2017).

* * *

Over the course of four billion years, nature has witnessed an endless process of evolution, which has led to the emergence of us humans. In a certain sense, we are the product of the most powerful meta learning algorithm to date. It is an exciting journey ahead of us to develop learning machines that can fully utilize the inductive bias of the world we live in, and thus reaching or even surpassing the pace of human learning.

BIBLIOGRAPHY

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. (2016). "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." In: *OSDI'16 Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation* (cit. on p. 47).

Abbeel, P. and A. Ng (2004). "Apprenticeship learning via inverse reinforcement learning." In: *International Conference on Machine Learning (ICML)* (cit. on p. 79).

Abeyruwan, S. (2013). *RLLib: Lightweight standard and on/off policy reinforcement learning library (C++)*. http://web.cs.miami.edu/home/saminda/rilib.html (cit. on p. 24).

Amodei, D., S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. (2016). "Deep speech 2: End-to-end speech recognition in english and mandarin." In: *International Conference on Machine Learning*, pp. 173–182 (cit. on p. 3).

Andrychowicz, M., M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas (2016). "Learning to learn by gradient descent by gradient descent." In: *Neural Information Processing Systems (NIPS)* (cit. on pp. 6, 45, 79).

Argall, B. D., S. Chernova, M. Veloso, and B. Browning (2009). "A survey of robot learning from demonstration." In: *Robotics and autonomous systems* 57.5, pp. 469–483 (cit. on p. 79).

Audibert, J.-Y. and R. Munos (2011). "Introduction to bandits: Algorithms and theory." In: *ICML Tutorial on bandits* (cit. on p. 36).

Auer, P. (2002). "Using confidence bounds for exploitation-exploration trade-offs." In: *Journal of Machine Learning Research* 3.Nov, pp. 397–422 (cit. on p. 36).

Aytar, Y. and A. Zisserman (2011). "Tabula rasa: Model transfer for object category detection." In: *2011 International Conference on Computer Vision*. IEEE, pp. 2252–2259 (cit. on p. 80).

Ba, J., G. E. Hinton, V. Mnih, J. Z. Leibo, and C. Ionescu (2016). "Using Fast Weights to Attend to the Recent Past." In: *Neural Information Processing Systems (NIPS)* (cit. on p. 79).

Bacon, P.-L., J. Harb, and D. Precup (2017). "The Option-Critic Architecture." In: *AAAI*, pp. 1726–1734 (cit. on p. 8).

Bagnell, J. A. and J. Schneider (2003). "Covariant policy search." In: IJCAI, pp. 1019–1024 (cit. on p. 17).
Bahdanau, D., K. Cho, and Y. Bengio (2015). "Neural machine translation by jointly learning to align and translate." In: *Int. Conf. on Learning Representations (ICLR)* (cit. on pp. 58, 64, 80).
Baker, B., O. Gupta, N. Naik, and R. Raskar (2016). "Designing neural network architectures using reinforcement learning." In: *arXiv preprint arXiv:1611.02167* (cit. on p. 7).
Bakker, B. (2001). "Reinforcement Learning with Long Short-Term Memory." In: *NIPS*, pp. 1475–1482 (cit. on pp. 15, 20).
Bartunov, S. and D. P. Vetrov (2016). "Fast Adaptation in Generative Models with Generative Matching Networks." In: *arXiv preprint arXiv:1612.02192* (cit. on p. 7).
Battaglia, P., R. Pascanu, M. Lai, D. J. Rezende, et al. (2016). "Interaction networks for learning about objects, relations and physics." In: *Advances in Neural Information Processing Systems*, pp. 4502–4510 (cit. on p. 80).
Baxter, J. (1993). "The evolution of learning algorithms for artificial neural networks." In: *Complex systems*, pp. 313–326 (cit. on p. 5).
Baxter, J. (2000). "A model of inductive bias learning." In: *J. Artif. Intell. Res.(JAIR)* 12.149-198, p. 3 (cit. on p. 4).
Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling (2013). "The Arcade Learning Environment: An Evaluation Platform for General Agents." In: *J. Artif. Intell. Res.* 47, pp. 253–279 (cit. on p. 10).
Bellemare, M., S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos (2016). "Unifying count-based exploration and intrinsic motivation." In: *Advances in Neural Information Processing Systems*, pp. 1471–1479 (cit. on p. 7).
Bellman, R. (1957). *Dynamic Programming*. Princeton University Press (cit. on p. 10).
Bello, I., B. Zoph, V. Vasudevan, and Q. V. Le (2017). "Neural optimizer search with reinforcement learning." In: *arXiv preprint arXiv:1709.07417* (cit. on p. 7).
Bengio, S., Y. Bengio, J. Cloutier, and J. Gecsei (1992). "On the optimization of a synaptic learning rule." In: *Optimality in Artificial and Biological Neural Networks*, pp. 6–8 (cit. on pp. 6, 79).
Bengio, Y., S. Bengio, and J. Cloutier (1990). *Learning a synaptic learning rule*. Université de Montréal, Département d'informatique et de recherche opérationnelle (cit. on pp. 6, 79).
Bengio, Y., P. Simard, and P. Frasconi (1994). "Learning long-term dependencies with gradient descent is difficult." In: *IEEE transactions on neural networks* 5.2, pp. 157–166 (cit. on p. 34).

Bertsekas, D. P. and J. N. Tsitsiklis (1995). "Neuro-dynamic programming: an overview." In: *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*. Vol. 1. IEEE, pp. 560–564 (cit. on pp. 10, 79).

Bornschein, J., A. Mnih, D. Zoran, and D. J. Rezende (2017). "Variational Memory Addressing in Generative Models." In: *arXiv preprint arXiv:1709.07116* (cit. on p. 7).

Brafman, R. I. and M. Tennenholtz (2002). "R-max-a general polynomial time algorithm for near-optimal reinforcement learning." In: *Journal of Machine Learning Research* 3.Oct, pp. 213–231 (cit. on p. 7).

Bubeck, S. and N. Cesa-Bianchi (2012). "Regret analysis of stochastic and nonstochastic multi-armed bandit problems." In: *Foundations and Trends in Machine Learning* (cit. on p. 34).

Busoniu, L. (2010). *ApproxRL: A Matlab toolbox for approximate RL and DP*. http://busoniu.net/files/repository/readme-approxrl.html (cit. on p. 24).

Calinon, S. (2009). *Robot programming by demonstration*. EPFL Press (cit. on p. 79).

Castronovo, M., F. Maes, R. Fonteneau, and D. Ernst (2012). "Learning Exploration/Exploitation Strategies for Single Trajectory Reinforcement Learning." In: *EWRL*, pp. 1–10 (cit. on p. 44).

Catto, E. (2011). *Box2D: A 2D physics engine for games* (cit. on p. 12).

Chakravorty, J. and A. Mahajan (2013). "Multi-armed bandits, Gittins index, and its calculation." In: *Methods and Applications of Statistics in Clinical Trials: Planning, Analysis, and Inferential Methods* 2, pp. 416–435 (cit. on p. 36).

Chalmers, D. J. (1990). "The evolution of learning: An experiment in genetic connectionism." In: *Proceedings of the 1990 connectionist models summer school*. San Mateo, CA, pp. 81–90 (cit. on p. 6).

Chang, M. B., T. Ullman, A. Torralba, and J. B. Tenenbaum (2017). "A compositional object-based approach to learning physical dynamics." In: *Int. Conf. on Learning Representations (ICLR)* (cit. on p. 80).

Chapelle, O. and L. Li (2011). "An empirical evaluation of thompson sampling." In: *Advances in neural information processing systems*, pp. 2249–2257 (cit. on p. 36).

Chen, Y., M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and N. Freitas (2017). "Learning to learn without gradient descent by gradient descent." In: *International Conference on Machine Learning*, pp. 748–756 (cit. on p. 7).

Cho, K., B. Van Merriënboer, D. Bahdanau, and Y. Bengio (2014a). "On the properties of neural machine translation: Encoder-decoder approaches." In: *arXiv preprint arXiv:1409.1259* (cit. on p. 34).

Cho, K., B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014b). "Learning phrase representations using RNN encoder-decoder for statistical machine translation." In: *arXiv preprint arXiv:1406.1078* (cit. on p. 80).

Chung, J., C. Gulcehre, K. Cho, and Y. Bengio (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling." In: *arXiv preprint arXiv:1412.3555* (cit. on p. 34).

Contardo, G., L. Denoyer, and T. Artieres (2017). "A Meta-Learning Approach to One-Step Active Learning." In: *arXiv preprint arXiv:1706.08334* (cit. on p. 7).

Coulom, R. (2002). "Reinforcement learning using neural networks, with applications to motor control." Doctoral dissertation. Institut National Polytechnique de Grenoble-INPG (cit. on p. 13).

Dann, C., G. Neumann, and J. Peters (2014). "Policy evaluation with temporal differences: A survey and comparison." In: *J. Mach. Learn. Res.* 15.1, pp. 809–883 (cit. on p. 25).

Degris, T., J. Béchu, A. White, J. Modayil, P. M. Pilarski, and C. Denk (2013). *RLPark*. http://rlpark.github.io (cit. on p. 25).

Deisenroth, M. P., G. Neumann, and J. Peters (2013). "A Survey on Policy Search for Robotics, Foundations and Trends in Robotics." In: *Found. Trends Robotics* 2.1-2, pp. 1–142 (cit. on p. 18).

Deisenroth, M. and C. E. Rasmussen (2011). "PILCO: A model-based and data-efficient approach to policy search." In: *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472 (cit. on p. 31).

DeJong, G. and M. W. Spong (1994). "Swinging up the Acrobot: An example of intelligent control." In: *ACC*, pp. 2158–2162 (cit. on pp. 13, 24).

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (2009). "ImageNet: A Large-Scale Hierarchical Image Database." In: *CVPR*, pp. 248–255 (cit. on pp. 3, 4, 11).

Devin, C., A. Gupta, T. Darrell, P. Abbeel, and S. Levine (2016). "Learning Modular Neural Network Policies for Multi-Task and Multi-Robot Transfer." In: *arXiv preprint arXiv:1609.07088* (cit. on p. 45).

Dietterich, T. G. (2000). "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition." In: *J. Artif. Intell. Res* 13, pp. 227–303 (cit. on p. 16).

Dilokthanakul, N., C. Kaplanis, N. Pawlowski, and M. Shanahan (2017). "Feature Control as Intrinsic Motivation for Hierarchical Reinforcement Learning." In: *arXiv preprint arXiv:1705.06769* (cit. on p. 8).

Dimitrakakis, C., N. Tziortziotis, and A. Tossou (2007). *Beliefbox: A framework for statistical methods in sequential decision making*. http://code.google.com/p/beliefbox/ (cit. on p. 24).

Dimitrakakis, C., G. Li, and N. Tziortziotis (2014). "The reinforcement learning competition 2014." In: *AI Magazine* 35.3, pp. 61–65 (cit. on p. 24).

Donahue, J., Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell (2014). "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition." In: *ICML*, pp. 647–655 (cit. on p. 80).

Donaldson, P. E. K. (1960). "Error decorrelation: a technique for matching a class of functions." In: *Proc. 3th Intl. Conf. Medical Electronics*, pp. 173–178 (cit. on pp. 12, 24).

Doya, K. (2000). "Reinforcement learning in continuous time and space." In: *Neural Comput.* 12.1, pp. 219–245 (cit. on p. 13).

Duan, L., D. Xu, and I. Tsang (2012). "Learning with augmented features for heterogeneous domain adaptation." In: *arXiv preprint arXiv:1206.4660* (cit. on p. 80).

Duan, Y., M. Andrychowicz, B. Stadie, J. Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba (2017). "One-Shot Imitation Learning." In: *Neural Information Processing Systems (NIPS)* (cit. on p. 9).

Duan, Y., X. Chen, R. Houthooft, J. Schulman, and P. Abbeel (2016a). "Benchmarking Deep Reinforcement Learning for Continuous Control." In: *arXiv preprint arXiv:1604.06778* (cit. on pp. 9, 47).

Duan, Y., J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel (2016b). "RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning." In: *arXiv preprint arXiv:1611.02779* (cit. on pp. 9, 79).

Duchi, J., E. Hazan, and Y. Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159 (cit. on p. 6).

Dutech, A., T. Edmunds, J. Kok, M. Lagoudakis, M. Littman, M. Riedmiller, B. Russell, B. Scherrer, R. Sutton, S. Timmer, et al. (2005). "Reinforcement learning benchmarks and bake-offs II." In: *Advances in Neural Information Processing Systems (NIPS)* 17 (cit. on p. 24).

Edwards, H. and A. Storkey (2017). "Towards a Neural Statistician." In: *International Conference on Learning Representations (ICLR)* (cit. on pp. 7, 79).

Ellis, H. C. (1965). "The transfer of learning." In: (cit. on p. 3).

Erez, T., Y. Tassa, and E. Todorov (2011). "Infinite horizon model predictive control for nonlinear periodic tasks." In: *Manuscript under review* 4 (cit. on p. 14).

Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman (2010). "The pascal visual object classes (VOC) challenge." In: *Int. J. Comput. Vision* 88.2, pp. 303–338 (cit. on p. 11).

Fei-Fei, L., R. Fergus, and P. Perona (2006). "One-shot learning of object categories." In: *IEEE transactions on pattern analysis and machine intelligence* 28.4, pp. 594–611 (cit. on pp. 11, 45).

Feldbaum, A. (1960). "Dual control theory. I." In: *Avtomatika i Telemekhanika* 21.9, pp. 1240–1249 (cit. on p. 45).

Finn, C., P. Abbeel, and S. Levine (2017a). "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks." In: *ICML* (cit. on pp. 5, 35, 43, 44, 79, 95).

Finn, C., S. Levine, and P. Abbeel (2016). "Guided cost learning: Deep inverse optimal control via policy optimization." In: *Proceedings of the 33rd International Conference on Machine Learning*. Vol. 48 (cit. on p. 79).

Finn, C., T. Yu, T. Zhang, P. Abbeel, and S. Levine (2017b). "One-Shot Visual Imitation Learning via Meta-Learning." In: *Conference on Robot Learning (CoRL)* (cit. on p. 95).

Fortunato, M., M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, et al. (2017). "Noisy networks for exploration." In: *arXiv preprint arXiv:1706.10295* (cit. on p. 7).

Fu, J., S. Levine, and P. Abbeel (2015). "One-shot learning of manipulation skills with online dynamics adaptation and neural network priors." In: *arXiv preprint arXiv:1509.06841* (cit. on p. 45).

Fu, J., J. D. Co-Reyes, and S. Levine (2017). "EX2: Exploration with Exemplar Models for Deep Reinforcement Learning." In: *arXiv preprint arXiv:1703.01260* (cit. on p. 7).

Furuta, K., T. Okutani, and H. Sone (1978). "Computer control of a double inverted pendulum." In: *Comput. Electr. Eng.* 5.1, pp. 67–84 (cit. on p. 13).

Garofolo, J. S., L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett (1993). "DARPA TIMIT acoustic-phonetic continuous speech corpus CD-ROM. NIST speech disc 1-1.1." In: *NASA STI/Recon Technical Report N* 93 (cit. on p. 11).

Gatys, L. A., A. S. Ecker, and M. Bethge (2015). "A neural algorithm of artistic style." In: *arXiv preprint arXiv:1508.06576* (cit. on pp. 6, 96).

Ghavamzadeh, M., S. Mannor, J. Pineau, A. Tamar, et al. (2015). *Bayesian reinforcement learning: a survey*. World Scientific (cit. on pp. 7, 31).

Gittins, J. C. (1979). "Bandit processes and dynamic allocation indices." In: *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 148–177 (cit. on p. 36).

Gittins, J., K. Glazebrook, and R. Weber (2011). *Multi-armed bandit allocation indices*. John Wiley & Sons (cit. on p. 36).

Glorot, X. and Y. Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks." In: *Aistats*. Vol. 9, pp. 249–256 (cit. on p. 47).

Godfrey, J. J., E. C. Holliman, and J. McDaniel (1992). "SWITCHBOARD: Telephone Speech Corpus for Research and Development." In: *ICASSP*, pp. 517–520 (cit. on p. 11).

Gomez, F. and R. Miikkulainen (1998). "2-d pole balancing with recurrent evolutionary networks." In: *ICANN*, pp. 425–430 (cit. on p. 15).

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep learning*. MIT press (cit. on p. 2).

Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio (2014). "Generative adversarial nets." In: *Advances in Neural Information Processing Systems*, pp. 2672–2680 (cit. on p. 96).

Greensmith, E., P. L. Bartlett, and J. Baxter (2004). "Variance reduction techniques for gradient estimates in reinforcement learning." In: *Journal of Machine Learning Research* 5.Nov, pp. 1471–1530 (cit. on p. 8).

Gregor, K., I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra (2015). "DRAW: A recurrent neural network for image generation." In: *arXiv preprint arXiv:1502.04623* (cit. on p. 5).

Guo, X., S. Singh, H. Lee, R. L. Lewis, and X. Wang (2014). "Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning." In: *Advances in neural information processing systems*, pp. 3338–3346 (cit. on pp. 10, 31).

Gupta, A., C. Devin, Y. Liu, P. Abbeel, and S. Levine (2017). "Learning Invariant Feature Spaces to Transfer Skills with Reinforcement Learning." In: *Int. Conf. on Learning Representations (ICLR)* (cit. on p. 80).

Hafner, R. and M. Riedmiller (2011). "Reinforcement learning in feedback control." In: *Machine learning* 84.1, pp. 137–169 (cit. on p. 8).

Hansen, N. and A. Ostermeier (2001). "Completely derandomized self-adaptation in evolution strategies." In: *Evol. Comput.* 9.2, pp. 159–195 (cit. on p. 19).

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778 (cit. on pp. 3, 5).

Heess, N., J. J. Hunt, T. P. Lillicrap, and D. Silver (2015a). "Memory-based control with recurrent neural networks." In: *arXiv preprint arXiv:1512.04455* (cit. on pp. 15, 20, 22, 32, 46).

Heess, N., G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa (2015b). "Learning continuous control policies by stochastic value gradients." In: *Advances in Neural Information Processing Systems*, pp. 2944–2952 (cit. on pp. 10, 14, 31, 79).

Hessel, M., J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver (2017). "Rainbow: Combining Improvements in Deep Reinforcement Learning." In: *arXiv preprint arXiv:1710.02298* (cit. on p. 8).

Hester, T. and P. Stone (2013). "The open-source TEXPLORE code release for reinforcement learning on robots." In: *RoboCup 2013: Robot World Cup XVII*, pp. 536–543 (cit. on p. 15).

Hinton, G., L. Deng, D. Yu, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. S. G. Dahl, and B. Kingsbury (2012). "Deep Neural Networks for Acoustic Modeling in Speech Recognition." In: *IEEE Signal Process. Mag* 29.6, pp. 82–97 (cit. on p. 10).

Hirsch, H.-G. and D. Pearce (2000). "The Aurora experimental framework for the performance evaluation of speech recognition systems under noisy conditions." In: *ASR2000-Automatic Speech Recognition: Challenges for the new Millenium ISCA Tutorial and Research Workshop (ITRW)* (cit. on p. 11).

Ho, J. and S. Ermon (2016). "Generative adversarial imitation learning." In: *Advances in Neural Information Processing Systems*, pp. 4565–4573 (cit. on p. 79).

Hochreiter, S. and J. Schmidhuber (1997). "Long short-term memory." In: *Neural computation* 9.8, pp. 1735–1780 (cit. on pp. 5, 20, 62).

Hochreiter, S., A. S. Younger, and P. R. Conwell (2001). "Learning to learn using gradient descent." In: *International Conference on Artificial Neural Networks*. Springer, pp. 87–94 (cit. on pp. 5, 45, 79).

Hoffman, J., E. Rodner, J. Donahue, T. Darrell, and K. Saenko (2013). "Efficient learning of domain-invariant image representations." In: *arXiv preprint arXiv:1301.3224* (cit. on p. 80).

Hoof, H. van, J. Peters, and G. Neumann (2015). "Learning of Non-Parametric Control Policies with High-Dimensional State Features." In: *AISTATS*, pp. 995–1003 (cit. on p. 23).

Houthooft, R., X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel (2016). "Vime: Variational information maximizing exploration." In: *Advances in Neural Information Processing Systems*, pp. 1109–1117 (cit. on p. 7).

Hull, C. (1943). "Principles of behavior." In: (cit. on p. 8).

Ishii, S., W. Yoshida, and J. Yoshimoto (2002). "Control of exploitation–exploration meta-parameter in reinforcement learning." In: *Neural networks* 15.4, pp. 665–687 (cit. on pp. 44, 95).

Isola, P., J.-Y. Zhu, T. Zhou, and A. A. Efros (2016). "Image-to-image translation with conditional adversarial networks." In: *arXiv preprint arXiv:1611.07004* (cit. on p. 96).

Jaksch, T., R. Ortner, and P. Auer (2010). "Near-optimal regret bounds for reinforcement learning." In: *Journal of Machine Learning Research* 11.Apr, pp. 1563–1600 (cit. on pp. 7, 38).

Józefowicz, R., W. Zaremba, and I. Sutskever (2015). "An Empirical Exploration of Recurrent Network Architectures." In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pp. 2342–2350. URL: http://jmlr.org/proceedings/papers/v37/jozefowicz15.html (cit. on p. 34).

Kakade, S. M. (2002). "A Natural Policy Gradient." In: *NIPS*, pp. 1531–1538 (cit. on p. 17).

Kalchbrenner, N., L. Espeholt, K. Simonyan, A. v. d. Oord, A. Graves, and K. Kavukcuoglu (2016). "Neural machine translation in linear time." In: *arXiv preprint arXiv:1610.10099* (cit. on p. 6).

Kearns, M. and S. Singh (2002). "Near-optimal reinforcement learning in polynomial time." In: *Machine Learning* 49.2-3, pp. 209–232 (cit. on p. 7).

Kempka, M., M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski (2016). "ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning." In: *arXiv preprint arXiv:1605.02097* (cit. on p. 32).

Kimura, H. and S. Kobayashi (1999). "Stochastic real-valued reinforcement learning to solve a nonlinear control problem." In: *IEEE SMC*, pp. 510–515 (cit. on p. 13).

Kimura, H., S. Kobayashi, et al. (2000). "An analysis of actor-critic algorithms using eligibility traces: reinforcement learning with imperfect value functions." In: *Journal of Japanese Society for Artificial Intelligence* 15.2, pp. 267–275 (cit. on p. 8).

Kingma, D. and J. Ba (2014). "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (cit. on pp. 6, 61).

Kober, J. and J. Peters (2009). "Policy Search for Motor Primitives in Robotics." In: *NIPS*, pp. 849–856 (cit. on p. 18).

Koch, G. (2015). "Siamese neural networks for one-shot image recognition." In: *ICML Deep Learning Workshop* (cit. on pp. 4, 45, 79).

Kochenderfer, M. (2006). *JRLF: Java reinforcement learning framework*. http://mykel.kochenderfer.com/jrlf (cit. on p. 24).

Kolter, J. Z. and A. Y. Ng (2009). "Near-Bayesian exploration in polynomial time." In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, pp. 513–520 (cit. on pp. 7, 31, 38).

Konda, V. R. and J. N. Tsitsiklis (2000). "Actor-critic algorithms." In: *Advances in neural information processing systems*, pp. 1008–1014 (cit. on p. 8).

Krizhevsky, A. and G. Hinton (2009). *Learning multiple layers of features from tiny images*. Tech. rep. (cit. on p. 11).

Krizhevsky, A., I. Sutskever, and G. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks." In: *NIPS*, pp. 1097–1105 (cit. on pp. 3, 10).

Krueger, D., T. Maharaj, J. Kramár, M. Pezeshki, N. Ballas, N. R. Ke, A. Goyal, Y. Bengio, H. Larochelle, A. Courville, et al. (2016). "Zoneout: Regularizing rnns by randomly preserving hidden activations." In: *arXiv preprint arXiv:1606.01305* (cit. on p. 63).

Kulis, B., K. Saenko, and T. Darrell (2011). "What you saw is not what you get: Domain adaptation using asymmetric kernel transforms." In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE, pp. 1785–1792 (cit. on p. 80).

Lake, B. M., R. Salakhutdinov, J. Gross, and J. B. Tenenbaum (2011). "One shot learning of simple visual concepts." In: *Proceedings of the 33rd Annual Conference of the Cognitive Science Society*. Vol. 172, p. 2 (cit. on pp. 4, 45).

Larochelle, H., D. Erhan, and Y. Bengio (2008). "Zero-data Learning of New Tasks." In: *AAAI*. Vol. 1. 2, p. 3 (cit. on p. 45).

LeCun, Y., C. Cortes, and C. Burges (1998). *The MNIST database of handwritten digits* (cit. on p. 11).

LeCun, Y. (1985). "Une procédure d'apprentissage pour réseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks)." In: *Proceedings of Cognitiva 85, Paris, France* (cit. on p. 2).

LeCun, Y., Y. Bengio, and G. Hinton (2015). "Deep learning." In: *Nature* 521.7553, pp. 436–444 (cit. on p. 2).

Levine, S. and V. Koltun (2013). "Guided policy search." In: *ICML*, pp. 1–9 (cit. on pp. 13, 14).

Levine, S., Z. Popovic, and V. Koltun (2011). "Nonlinear inverse reinforcement learning with gaussian processes." In: *Advances in Neural Information Processing Systems (NIPS)* (cit. on p. 79).

Levine, S. and P. Abbeel (2014). "Learning neural network policies with guided policy search under unknown dynamics." In: *Advances in Neural Information Processing Systems*, pp. 1071–1079 (cit. on pp. 7, 31, 45).

Levine, S., C. Finn, T. Darrell, and P. Abbeel (2016). "End-to-end training of deep visuomotor policies." In: *Journal of Machine Learning Research* 17.39, pp. 1–40 (cit. on pp. 2, 10, 31, 79).

Li, K. and J. Malik (2016). "Learning to Optimize." In: *arXiv preprint arXiv:1606.01885* (cit. on pp. 45, 79).

Li, K. and J. Malik (2017). "Learning to Optimize Neural Nets." In: *arXiv preprint arXiv:1703.00441* (cit. on p. 7).

Li, Y. (2017). "Deep reinforcement learning: An overview." In: *arXiv preprint arXiv:1701.07274* (cit. on p. 3).

Li, Z., F. Zhou, F. Chen, and H. Li (2017). "Meta-SGD: Learning to Learn Quickly for Few Shot Learning." In: *arXiv preprint arXiv:1707.09835* (cit. on p. 5).

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2016). "Continuous control with deep reinforcement learning." In: *International Conference on Learning Representations (ICLR)* (cit. on pp. 2, 10, 19, 20, 31, 32, 79).

Long, M. and J. Wang (2015). "Learning transferable features with deep adaptation networks." In: *CoRR, abs/1502.02791* 1, p. 2 (cit. on p. 80).

Maes, F., L. Wehenkel, and D. Ernst (2011). "Automatic discovery of ranking formulas for playing with multi-armed bandits." In: *European Workshop on Reinforcement Learning*. Springer, pp. 5–17 (cit. on p. 44).

Mansour, Y., M. Mohri, and A. Rostamizadeh (2009). "Domain adaptation: Learning bounds and algorithms." In: *arXiv preprint arXiv:0902.3430* (cit. on p. 80).

Martin, D., C. Fowlkes, D. Tal, and J. Malik (2001). "A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics." In: *ICCV*, pp. 416–423 (cit. on p. 11).

May, B. C., N. Korda, A. Lee, and D. S. Leslie (2012). "Optimistic Bayesian sampling in contextual-bandit problems." In: *Journal of Machine Learning Research* 13.Jun, pp. 2069–2106 (cit. on p. 36).

Mehrotra, A. and A. Dukkipati (2017). "Generative Adversarial Residual Pairwise Networks for One Shot Learning." In: *arXiv preprint arXiv:1703.08033* (cit. on p. 5).

Metz, L., J. Ibarz, N. Jaitly, and J. Davidson (2017). "Discrete Sequential Prediction of Continuous Actions for Deep RL." In: *arXiv preprint arXiv:1705.05035* (cit. on p. 9).

Metzen, J. M. and M. Edgington (2011). *Maja Machine Learning Framework*. http://mloss.org/software/view/220/ (cit. on p. 24).

Michie, D. and R. A. Chambers (1968). "BOXES: An experiment in adaptive control." In: *Machine Intelligence* 2, pp. 137–152 (cit. on p. 12).

Minsky, M. (1961). "Steps toward artificial intelligence." In: *Proceedings of the IRE* 49.1, pp. 8–30 (cit. on p. 8).

Mishra, N., M. Rohaninejad, X. Chen, and P. Abbeel (2017). "Meta-Learning with Temporal Convolutions." In: *arXiv preprint arXiv:1707.03141* (cit. on pp. 6, 95).

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis (2015).

"Human-level control through deep reinforcement learning." In: *Nature* 518.7540, pp. 529–533 (cit. on pp. 2, 10, 20, 31, 79).
Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu (2016). "Asynchronous methods for deep reinforcement learning." In: *arXiv preprint arXiv:1602.01783* (cit. on pp. 32, 45, 48).
Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller (2013). "Playing atari with deep reinforcement learning." In: *arXiv preprint arXiv:1312.5602* (cit. on p. 2).
Mohamed, S. and D. J. Rezende (2015). "Variational information maximisation for intrinsically motivated reinforcement learning." In: *Advances in neural information processing systems*, pp. 2125–2133 (cit. on p. 7).
Montúfar, G., K. Ghazi-Zahedi, and N. Ay (2016). "Information theoretically aided reinforcement learning for embodied agents." In: *arXiv preprint arXiv:1605.09735* (cit. on p. 7).
Moore, A. (1990). *Efficient Memory-based Learning for Robot Control*. Tech. rep. University of Cambridge, Computer Laboratory (cit. on pp. 13, 24).
Munkhdalai, T. and H. Yu (2017). "Meta Networks." In: *arXiv preprint arXiv:1703.00837* (cit. on p. 5).
Murray, R. M. and J. Hauser (1991). *A Case Study in Approximate Linearization: The Acrobot Example*. Tech. rep. UC Berkeley, EECS Department (cit. on p. 13).
Murthy, S. S. and M. H. Raibert (1984). "3D balance in legged locomotion: modeling and simulation for the one-legged case." In: *ACM SIGGRAPH Computer Graphics* 18.1, pp. 27–27 (cit. on p. 14).
Naik, D. K. and R. Mammone (1992). "Meta-neural networks that learn by learning." In: *International Joint Conference on Neural Netowrks (IJCNN)* (cit. on pp. 4, 6, 79).
Nesterov, Y. (1983). "A method of solving a convex programming problem with convergence rate O (1/k2)." In: *Soviet Mathematics Doklady*. Vol. 27. 2, pp. 372–376 (cit. on p. 6).
Neumann, G. (2006). "A Reinforcement Learning Toolbox and RL Benchmarks for the Control of Dynamical Systems." In: *Dynamical principles for neuroscience and intelligent biomimetic devices*, p. 113 (cit. on p. 24).
Ng, A. Y., D. Harada, and S. Russell (1999). "Policy invariance under reward transformations: Theory and application to reward shaping." In: *ICML*. Vol. 99, pp. 278–287 (cit. on pp. 61, 70).
Ng, A. Y., H. J. Kim, M. I. Jordan, S. Sastry, and S. Ballianda (2003). "Autonomous helicopter flight via reinforcement learning." In: *NIPS*. Vol. 16 (cit. on p. 79).

Ng, A. and S. Russell (2000). "Algorithms for inverse reinforcement learning." In: *International Conference on Machine Learning (ICML)* (cit. on p. 79).
Oh, J., V. Chockalingam, S. Singh, and H. Lee (2016). "Control of Memory, Active Perception, and Action in Minecraft." In: *arXiv preprint arXiv:1605.09128* (cit. on p. 41).
Oh, J., X. Guo, H. Lee, R. L. Lewis, and S. Singh (2015). "Action-conditional video prediction using deep networks in atari games." In: *Advances in Neural Information Processing Systems*, pp. 2863–2871 (cit. on p. 7).
Ong, S. C., S. W. Png, D. Hsu, and W. S. Lee (2010). "Planning under uncertainty for robotic tasks with mixed observability." In: *The International Journal of Robotics Research* 29.8, pp. 1053–1068 (cit. on p. 45).
Oord, A. v. d., S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu (2016a). "Wavenet: A generative model for raw audio." In: *arXiv preprint arXiv:1609.03499* (cit. on p. 6).
Oord, A. v. d., N. Kalchbrenner, and K. Kavukcuoglu (2016b). "Pixel recurrent neural networks." In: *arXiv preprint arXiv:1601.06759* (cit. on p. 6).
OpenAI (2017). *Universe*. URL: https://github.com/openai/universe (visited on 10/16/2017) (cit. on p. 94).
Osband, I. (2016). *TabulaRL*. https://github.com/iosband/TabulaRL (cit. on p. 47).
Osband, I., C. Blundell, A. Pritzel, and B. Van Roy (2016). "Deep exploration via bootstrapped DQN." In: *Advances in Neural Information Processing Systems*, pp. 4026–4034 (cit. on p. 7).
Osband, I., D. Russo, and B. Van Roy (2013). "(More) efficient reinforcement learning via posterior sampling." In: *Advances in Neural Information Processing Systems*, pp. 3003–3011 (cit. on p. 38).
Osband, I. and B. Van Roy (2017). "Why is Posterior Sampling Better than Optimism for Reinforcement Learning." In: *International Conference on Machine Learning (ICML)* (cit. on p. 38).
Osband, I., B. Van Roy, and Z. Wen (2014). "Generalization and exploration via randomized value functions." In: *arXiv preprint arXiv:1402.0635* (cit. on p. 7).
Ostrovski, G., M. G. Bellemare, A. v. d. Oord, and R. Munos (2017). "Count-based exploration with neural density models." In: *arXiv preprint arXiv:1703.01310* (cit. on p. 7).
Oudeyer, P.-Y. and F. Kaplan (2009). "What is intrinsic motivation? A typology of computational approaches." In: *Frontiers in neurorobotics* 1, p. 6 (cit. on p. 7).
Papis, B. and P. Wawrzyński (2013). "dotRL: A platform for rapid Reinforcement Learning methods development and validation." In: *FedCSIS*, pages 129–136. (Cit. on p. 25).

Parisotto, E., J. L. Ba, and R. Salakhutdinov (2015). "Actor-mimic: Deep multitask and transfer reinforcement learning." In: *arXiv preprint arXiv:1511.06342* (cit. on p. 45).
Parker, D. B. (1985). "Learning logic." In: (cit. on p. 2).
Parr, R. and S. Russell (1998). "Reinforcement learning with hierarchies of machines." In: *Advances in neural information processing systems*, pp. 1043–1049 (cit. on p. 16).
Pathak, D., P. Agrawal, A. A. Efros, and T. Darrell (2017). "Curiosity-driven exploration by self-supervised prediction." In: *arXiv preprint arXiv:1705.05363* (cit. on p. 7).
Pazis, J. and R. Parr (2013). "PAC Optimal Exploration in Continuous Space Markov Decision Processes." In: *AAAI* (cit. on p. 7).
Peng, X. B., M. Andrychowicz, W. Zaremba, and P. Abbeel (2017). "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization." In: *arXiv preprint arXiv:1710.06537* (cit. on p. 96).
Perkins, T. J., D. Precup, et al. (1999). "Using options for knowledge transfer in reinforcement learning." In: *University of Massachusetts, Amherst, MA, USA, Tech. Rep* (cit. on p. 44).
Peters, J. (2002). *Policy Gradient Toolbox*. http://www.ausy.tu-darmstadt.de/Research/PolicyGradientToolbox (cit. on p. 24).
Peters, J., K. Mülling, and Y. Altün (2010). "Relative Entropy Policy Search." In: *AAAI*, pp. 1607–1612 (cit. on pp. 18, 23).
Peters, J. and S. Schaal (2007). "Reinforcement learning by reward-weighted regression for operational space control." In: *ICML*, pp. 745–750 (cit. on p. 18).
Peters, J., S. Vijaykumar, and S. Schaal (2003). *Policy gradient methods for robot control*. Tech. rep. (cit. on p. 17).
Peters, J. and S. Schaal (2008). "Reinforcement learning of motor skills with policy gradients." In: *Neural networks* 21.4, pp. 682–697 (cit. on pp. 17, 21, 23, 79).
Plappert, M., R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz (2017). "Parameter Space Noise for Exploration." In: *arXiv preprint arXiv:1706.01905* (cit. on p. 7).
Pomerleau, D. A. (1989). "ALVINN: An Autonomous Land Vehicle in a Neural Network." In: *Advances in Neural Information Processing Systems*, pp. 305–313 (cit. on p. 79).
Purcell, E. M. (1977). "Life at low Reynolds number." In: *Am. J. Phys* 45.1, pp. 3–11 (cit. on p. 13).
Raibert, M. H. and J. K. Hodgins (1991). "Animation of dynamic legged locomotion." In: *ACM SIGGRAPH Computer Graphics*. Vol. 25. 4, pp. 349–358 (cit. on p. 14).
Ravi, S. and H. Larochelle (2017). "Optimization as a Model for Few-Shot Learning." In: *Under Review, ICLR* (cit. on pp. 5, 79).

Rezende, D. J., S. Mohamed, I. Danihelka, K. Gregor, and D. Wierstra (2016). "One-Shot Generalization in Deep Generative Models." In: *International Conference on Machine Learning (ICML)* (cit. on p. 79).

Riedmiller, M., M. Blum, and T. Lampe (2012). *CLS2: Closed loop simulation system*. http://ml.informatik.uni-freiburg.de/research/clsquare (cit. on p. 25).

Riedmiller, M. and H. Braun (1993). "A direct adaptive method for faster backpropagation learning: The RPROP algorithm." In: *Neural Networks, 1993., IEEE International Conference on*. IEEE, pp. 586–591 (cit. on p. 6).

Ross, S., G. J. Gordon, and D. Bagnell (2011). "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." In: *AISTATS*. Vol. 1. 2, p. 6 (cit. on pp. 61, 79).

Rubinstein, R. (1999). "The cross-entropy method for combinatorial and continuous optimization." In: *Methodol. Comput. Appl. Probab.* 1.2, pp. 127–190 (cit. on p. 19).

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1985). *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science (cit. on p. 6).

Runarsson, T. P. and M. T. Jonsson (2000). "Evolution and design of distributed learning rules." In: *Combinations of Evolutionary Computation and Neural Networks, 2000 IEEE Symposium on*. IEEE, pp. 59–63 (cit. on p. 6).

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. (2015). "Imagenet large scale visual recognition challenge." In: *International Journal of Computer Vision* 115.3, pp. 211–252 (cit. on pp. 3, 4).

Russell, S. J. (1987). "Analogical and inductive reasoning." In: (cit. on p. 3).

Rusu, A. A., S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell (2015). "Policy distillation." In: *arXiv preprint arXiv:1511.06295* (cit. on p. 45).

Rusu, A. A., N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell (2016a). "Progressive neural networks." In: *arXiv preprint arXiv:1606.04671* (cit. on pp. 45, 80).

Rusu, A. A., M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell (2016b). "Sim-to-Real Robot Learning from Pixels with Progressive Nets." In: *arXiv preprint arXiv:1610.04286* (cit. on p. 45).

Sadeghi, F. and S. Levine (2016). "$(CAD)^2$ RL: Real Single-Image Flight without a Single Real Image." In: (cit. on pp. 80, 96).

Salimans, T. and D. P. Kingma (2016). "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." In: *Advances in Neural Information Processing Systems*, pp. 901–901 (cit. on pp. 47, 81).

Santoro, A., S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap (2016). "Meta-learning with memory-augmented neural networks." In: *International Conference on Machine Learning (ICML)* (cit. on pp. 5, 45, 79).

Saxe, A. M., J. L. McClelland, and S. Ganguli (2013). "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks." In: *arXiv preprint arXiv:1312.6120* (cit. on p. 47).

Schaal, S. (1999). "Is imitation learning the route to humanoid robots?" In: *Trends in cognitive sciences* 3.6, pp. 233–242 (cit. on p. 79).

Schäfer, A. M. and S. Udluft (2005). "Solving partially observable reinforcement learning problems with recurrent neural networks." In: *ECML Workshops*, pp. 71–81 (cit. on pp. 15, 20).

Schaul, T., J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber (2010). "PyBrain." In: *J. Mach. Learn. Res.* 11, pp. 743–746 (cit. on p. 25).

Schmidhuber, J., J. Zhao, and M. Wiering (1996). "Simple principles of metalearning." In: *Technical report IDSIA* 69, pp. 1–23 (cit. on p. 4).

Schmidhuber, J. (1987). "Evolutionary principles in self-referential learning." In: *On learning how to learn: The meta-meta-... hook.) Diploma thesis, Institut f. Informatik, Tech. Univ. Munich* (cit. on pp. 3, 79).

Schmidhuber, J. (1992). "Learning to control fast-weight memories: An alternative to dynamic recurrent networks." In: *Neural Computation* (cit. on p. 79).

Schmidhuber, J. (2010). "Formal theory of creativity, fun, and intrinsic motivation (1990–2010)." In: *IEEE Transactions on Autonomous Mental Development* 2.3, pp. 230–247 (cit. on p. 7).

Schulman, J., S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz (2015). "Trust Region Policy Optimization." In: *ICML*, pp. 1889–1897 (cit. on pp. 2, 10, 13, 14, 17, 18, 20, 31, 79).

Schulman, J., P. Moritz, S. Levine, M. Jordan, and P. Abbeel (2016). "High-dimensional continuous control using generalized advantage estimation." In: *International Conference on Learning Representations (ICLR)* (cit. on pp. 2, 10, 15, 31, 32, 35).

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov (2017). "Proximal Policy Optimization Algorithms." In: *arXiv preprint arXiv:1707.06347* (cit. on pp. 9, 34).

Schweighofer, N. and K. Doya (2003). "Meta-learning in reinforcement learning." In: *Neural Networks* 16.1, pp. 5–9 (cit. on pp. 44, 95).

ScrapeHero (2017). *Number of Products Sold on Amazon.com - June 2017*. URL: https://www.scrapehero.com/number-of-products-sold-on-amazon-com-june-2017/ (visited on 10/06/2017) (cit. on p. 3).
Shi, T., A. Karpathy, L. Fan, J. Hernandez, and P. Liang (2017). "World of Bits: An Open-Domain Platform for Web-Based Agents." In: *International Conference on Machine Learning*, pp. 3135–3144 (cit. on p. 94).
Shyam, P., S. Gupta, and A. Dukkipati (2017). "Attentive Recurrent Comparators." In: *arXiv preprint arXiv:1703.00767* (cit. on p. 5).
Siegelmann, H. T. and E. D. Sontag (1995). "On the computational power of neural nets." In: *Journal of computer and system sciences* 50.1, pp. 132–150 (cit. on p. 95).
Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587, pp. 484–489 (cit. on pp. 2, 31, 79).
Singh, S. P. (1992). "Transfer of learning by composing solutions of elemental sequential tasks." In: *Machine Learning* 8.3-4, pp. 323–339 (cit. on p. 44).
Snell, J., K. Swersky, and R. S. Zemel (2017). "Prototypical Networks for Few-shot Learning." In: *arXiv preprint arXiv:1703.05175* (cit. on p. 5).
Srivastava, N., G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1, pp. 1929–1958 (cit. on pp. 63, 66).
Stadie, B., P. Abbeel, and I. Sutskever (2017). "Third Person Imitation Learning." In: *Int. Conf. on Learning Representations (ICLR)* (cit. on p. 80).
Stadie, B. C., S. Levine, and P. Abbeel (2015). "Incentivizing exploration in reinforcement learning with deep predictive models." In: *arXiv preprint arXiv:1507.00814* (cit. on p. 7).
Stephenson, A. (1908). "On induced stability." In: *Philos. Mag.* 15.86, pp. 233–236 (cit. on pp. 12, 24).
Stone, P., G. Kuhlmann, M. E. Taylor, and Y. Liu (2005). "Keepaway soccer: From machine learning testbed to benchmark." In: *RoboCup 2005: Robot Soccer World Cup IX*. Springer, pp. 93–105 (cit. on p. 25).
Strens, M. (2000). "A Bayesian framework for reinforcement learning." In: *ICML*, pp. 943–950 (cit. on pp. 31, 38).
Sukhbaatar, S., I. Kostrikov, A. Szlam, and R. Fergus (2017). "Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play." In: *arXiv preprint arXiv:1703.05407* (cit. on p. 96).

Sun, Y., F. J. Gomez, and J. Schmidhuber (2011). "Planning to Be Surprised: Optimal Bayesian Exploration in Dynamic Environments." In: *AGI*. Springer, pp. 41–51 (cit. on p. 7).

Sutskever, I., O. Vinyals, and Q. V. Le (2014). "Sequence to sequence learning with neural networks." In: *Advances in neural information processing systems*, pp. 3104–3112 (cit. on p. 80).

Sutton, R. S. and A. G. Barto (1998). *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge (cit. on pp. 2, 7, 8, 79).

Sutton, R. S., D. Precup, and S. Singh (1999). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning." In: *Artificial intelligence* 112.1, pp. 181–211 (cit. on p. 16).

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015). "Going deeper with convolutions." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9 (cit. on p. 3).

Szita, I. and A. Lőrincz (2006). "Learning Tetris using the noisy cross-entropy method." In: *Neural Comput.* 18.12, pp. 2936–2941 (cit. on p. 19).

Szita, I., B. Takács, and A. Lörincz (2003). "ε-MDPs: Learning in varying environments." In: *J. Mach. Learn. Res.* 3, pp. 145–174 (cit. on p. 15).

Tang, H., P. Abbeel, D. Foote, Y. Duan, O. X. Chen, R. Houthooft, A. Stooke, and F. DeTurck (2017). "# Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning." In: *Advances in Neural Information Processing Systems*, pp. 2750–2759 (cit. on p. 7).

Tassa, Y., T. Erez, and E. Todorov (2012). "Synthesis and stabilization of complex behaviors through online trajectory optimization." In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, pp. 4906–4913 (cit. on p. 15).

Taylor, M. E. and P. Stone (2009). "Transfer learning for reinforcement learning domains: A survey." In: *Journal of Machine Learning Research* 10.Jul, pp. 1633–1685 (cit. on p. 44).

Tesauro, G. (1995). "Temporal difference learning and TD-Gammon." In: *Commun. ACM* 38.3, pp. 58–68 (cit. on pp. 10, 31, 79).

Tesauro, G. (1994). "TD-Gammon, a self-teaching backgammon program, achieves master-level play." In: *Neural computation* 6.2, pp. 215–219 (cit. on p. 2).

Thompson, W. R. (1933). "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples." In: *Biometrika* 25.3/4, pp. 285–294 (cit. on p. 36).

Thrun, S. and L. Pratt (1998). *Learning to learn*. Springer Science & Business Media (cit. on pp. 3, 79).

Tieleman, T. and G. Hinton (2012). "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." In: *COURSERA: Neural networks for machine learning* 4.2, pp. 26–31 (cit. on p. 6).

Tobin, J., R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel (2017a). "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World." In: *arXiv preprint arXiv:1703.06907* (cit. on p. 96).

Tobin, J., W. Zaremba, and P. Abbeel (2017b). "Domain Randomization and Generative Models for Robotic Grasping." In: *arXiv preprint arXiv:1710.06425* (cit. on p. 96).

Todorov, E., T. Erez, and Y. Tassa (2012). "MuJoCo: A physics engine for model-based control." In: *IROS*, pp. 5026–5033 (cit. on p. 12).

Tseng, P. (1998). "An incremental gradient (-projection) method with momentum term and adaptive stepsize rule." In: *SIAM Journal on Optimization* 8.2, pp. 506–531 (cit. on p. 6).

Tzeng, E., C. Devin, J. Hoffman, C. Finn, X. Peng, P. Abbeel, S. Levine, K. Saenko, and T. Darrell (2015). "Towards Adapting Deep Visuomotor Representations from Simulated to Real Environments." In: *arXiv preprint arXiv:1511.07111* (cit. on p. 80).

Tzeng, E., J. Hoffman, N. Zhang, K. Saenko, and T. Darrell (2014). "Deep domain confusion: Maximizing for domain invariance." In: *arXiv preprint arXiv:1412.3474* (cit. on p. 80).

Vapnik, V. and A. Y. Chervonenkis (1971). "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities." In: *Theory of Probability and its Applications* 16.2, p. 264 (cit. on p. 96).

Vezhnevets, A. S., S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu (2017). "Feudal networks for hierarchical reinforcement learning." In: *arXiv preprint arXiv:1703.01161* (cit. on p. 8).

Vezhnevets, A., V. Mnih, S. Osindero, A. Graves, O. Vinyals, J. Agapiou, et al. (2016). "Strategic attentive writer for learning macro-actions." In: *Advances in Neural Information Processing Systems*, pp. 3486–3494 (cit. on p. 8).

Vilalta, R. and Y. Drissi (2002). "A perspective view and survey of meta-learning." In: *Artificial Intelligence Review* 18.2, pp. 77–95 (cit. on p. 45).

Vinyals, O., C. Blundell, T. Lillicrap, D. Wierstra, et al. (2016a). "Matching networks for one shot learning." In: *Neural Information Processing Systems (NIPS)* (cit. on pp. 45, 79).

Vinyals, O., C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra (2016b). "Matching Networks for One Shot Learning." In: *Neural Information Processing Systems (NIPS)* (cit. on pp. 4, 5).

Wahlström, N., T. B. Schön, and M. P. Deisenroth (2015). "From pixels to torques: Policy learning with deep dynamical models." In: *arXiv preprint arXiv:1502.02251* (cit. on p. 32).
Wang, J. X., Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick (2016). "Learning to reinforcement learn." In: *arXiv preprint arXiv:1611.05763* (cit. on pp. 46, 79).
Wang, Z., V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas (2016). "Sample efficient actor-critic with experience replay." In: *arXiv preprint arXiv:1611.01224* (cit. on p. 8).
Watter, M., J. Springenberg, J. Boedecker, and M. Riedmiller (2015a). "Embed to control: A locally linear latent dynamics model for control from raw images." In: *NIPS*, pp. 2728–2736 (cit. on p. 10).
Watter, M., J. Springenberg, J. Boedecker, and M. Riedmiller (2015b). "Embed to control: A locally linear latent dynamics model for control from raw images." In: *Advances in Neural Information Processing Systems*, pp. 2746–2754 (cit. on p. 31).
Wawrzyński, P. (2007). "Learning to control a 6-degree-of-freedom walking robot." In: *IEEE EUROCON*, pp. 698–705 (cit. on p. 14).
Wawrzyński, P. (2009). "Real-time reinforcement learning by sequential actor–critics and experience replay." In: *Neural Networks* 22.10, pp. 1484–1497 (cit. on p. 8).
Werbos, P. J. (1974). "Beyond regression: New tools for prediction and analysis in the behavioral sciences." In: *Doctoral Dissertation, Applied Mathematics, Harvard University, MA* (cit. on p. 2).
Whittle, P. (1982). *Optimization over time*. John Wiley & Sons, Inc. (cit. on p. 36).
Wichrowska, O., N. Maheswaranathan, M. W. Hoffman, S. G. Colmenarejo, M. Denil, N. de Freitas, and J. Sohl-Dickstein (2017). "Learned Optimizers that Scale and Generalize." In: *arXiv preprint arXiv:1703.04813* (cit. on p. 6).
Widrow, B. (1964). "Pattern recognition and adaptive control." In: *IEEE Trans. Ind. Appl.* 83.74, pp. 269–277 (cit. on pp. 12, 24).
Wierstra, D., A. Foerster, J. Peters, and J. Schmidhuber (2007). "Solving deep memory POMDPs with recurrent policy gradients." In: *International Conference on Artificial Neural Networks*. Springer, pp. 697–706 (cit. on pp. 15, 20, 32).
Williams, D. and G. Hinton (1986). "Learning representations by back-propagating errors." In: *Nature* 323.6088, pp. 533–538 (cit. on p. 2).
Williams, R. J. (1992a). "Simple statistical gradient-following algorithms for connectionist reinforcement learning." In: *Mach. Learn.* 8, pp. 229–256 (cit. on p. 17).

Williams, R. J. (1992b). "Simple statistical gradient-following algorithms for connectionist reinforcement learning." In: *Machine learning* 8.3-4, pp. 229–256 (cit. on p. 8).
Wilson, A., A. Fern, S. Ray, and P. Tadepalli (2007). "Multi-task reinforcement learning: a hierarchical Bayesian approach." In: *Proceedings of the 24th international conference on Machine learning*. ACM, pp. 1015–1022 (cit. on p. 44).
Wilson, A. C., R. Roelofs, M. Stern, N. Srebro, and B. Recht (2017). "The Marginal Value of Adaptive Gradient Methods in Machine Learning." In: *arXiv preprint arXiv:1705.08292* (cit. on p. 6).
Woodward, M. and C. Finn (2017). "Active one-shot learning." In: *arXiv preprint arXiv:1702.06559* (cit. on p. 7).
Wu, Y., M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. (2016). "Google's neural machine translation system: Bridging the gap between human and machine translation." In: *arXiv preprint arXiv:1609.08144* (cit. on p. 3).
Wu, Y., E. Mansimov, S. Liao, R. Grosse, and J. Ba (2017). "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation." In: *arXiv preprint arXiv:1708.05144* (cit. on p. 8).
Xu, K., J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio (2015). "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention." In: *ICML*. Vol. 14, pp. 77–81 (cit. on p. 80).
Yamaguchi, A. and T. Ogasawara (2010). "SkyAI: Highly modularized reinforcement learning library." In: *IEEE-RAS Humanoids*, pp. 118–123 (cit. on p. 25).
Yang, J., R. Yan, and A. G. Hauptmann (2007). "Cross-domain video concept detection using adaptive svms." In: *Proceedings of the 15th ACM international conference on Multimedia*. ACM, pp. 188–197 (cit. on p. 80).
Younger, A. S., S. Hochreiter, and P. R. Conwell (2001). "Meta-learning with backpropagation." In: *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*. Vol. 3. IEEE (cit. on pp. 5, 45).
Yu, D., Y.-C. Ju, Y.-Y. Wang, G. Zweig, and A. Acero (2007). "Automated Directory Assistance System - from Theory to Practice." In: *Interspeech*, pp. 2709–2712 (cit. on p. 11).
Yu, F. and V. Koltun (2016). "Multi-scale context aggregation by dilated convolutions." In: *International Conference on Learning Representations (ICLR)* (cit. on p. 63).
Zeiler, M. D. and R. Fergus (2014). "Visualizing and understanding convolutional networks." In: *European conference on computer vision*. Springer, pp. 818–833 (cit. on p. 3).

Zhu, J.-Y., T. Park, P. Isola, and A. A. Efros (2017). "Unpaired image-to-image translation using cycle-consistent adversarial networks." In: *arXiv preprint arXiv:1703.10593* (cit. on p. 96).
Ziebart, B., A. Maas, J. A. Bagnell, and A. K. Dey (2008). "Maximum Entropy Inverse Reinforcement Learning." In: *AAAI Conference on Artificial Intelligence* (cit. on p. 79).
Zoph, B. and Q. V. Le (2016). "Neural architecture search with reinforcement learning." In: *arXiv preprint arXiv:1611.01578* (cit. on p. 7).