Title

Building Agentic Systems in an Era of Large Language Models

Author

Packer, Charles Avery

Publication Date

2024

Peer reviewed|Thesis/dissertation

Building Agentic Systems in an Era of Large Language Models

By

Charles Packer

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 Joseph E. Gonzalez, Chair
Professor Ion Stoica
Professor Matei Zaharia
Doctor Yuandong Tian

Fall 2024

Building Agentic Systems in an Era of Large Language Models

Abstract

Building Agentic Systems in an Era of Large Language Models

by

Charles Packer

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph E. Gonzalez, Chair

Building intelligent autonomous systems that can reason, adapt, and interact with their environment has been a long-standing goal in artificial intelligence. This thesis explores the evolution of agentic systems through the deep learning revolution, from reinforcement learning to modern Large Language Models (LLMs), focusing on the critical components needed to create reliable autonomous agents.

First, we address the fundamental challenge of generalization in deep reinforcement learning (RL), introducing a systematic framework for evaluating and improving how learned policies transfer across environments. Building on this foundation, we present Hindsight Task Relabeling (HTR), a novel approach that enables meta-RL algorithms to learn adaptation strategies in sparse reward settings without requiring dense reward signals during training.

Finally, we address the emerging challenges of building reliable agents using Large Language Models. While LLMs demonstrate unprecedented reasoning capabilities, their effectiveness as autonomous agents is limited by fundamental constraints in their architecture - most notably, their stateless nature and fixed context windows. We present MemGPT, an operating system-inspired framework that enables LLMs to manage their own memory and state, introducing concepts like virtual context management and self-directed memory operations. MemGPT demonstrates that by treating LLMs as a new fundamental unit of compute - analogous to how CPUs were the fundamental unit in traditional operating systems - we can build more reliable and capable autonomous agents.

Together, these systems trace the evolution of agentic AI systems and provide key building blocks for creating more reliable and capable autonomous agents. By addressing core challenges in generalization, adaptation, and memory management, this thesis establishes a foundation for engineering the next generation of AI systems that can effectively reason and interact with the world.

To my parents

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I want to thank my family, who always pushed me to achieve more. They are the reason I love to do hard things.

Next I would like to thank my advisor, Professor Joseph E. Gonzalez. Joey helped me achieve my one true goal in the PhD: to make science fiction into science reality. His flexibility and encouragement, regardless of where my research interests led (even when not directly in his critical research path), were instrumental to my success. I could not have asked for a better PhD advisor.

I am also deeply grateful to my other thesis committee members: Ion Stoica, Matei Zaharia, and Yuandong Tian. Having such renowned world experts in AI and systems research on my committee was an incredible honor.

My journey in AI research began at UC San Diego, where I worked with Professors Julian McAuley and Kamalika Chaudhuri as an undergraduate. This led to my work with Professor Lawrence Holder during an REU at Washington State University, where I wrote my first first-author paper. After graduation, Professor Dawn Song took a chance on me, hiring me after a brief chat at a Starbucks in Hayes Valley - a moment that brought me to Berkeley and set me on my path toward the PhD.

Several mentors were crucial to my development as a researcher during my time at Berkeley. Vladlen Koltun taught me invaluable lessons about research discipline, particularly about knowing when to abandon 'zombie' research projects - advice I wish I had followed more closely. Richard Shin and Katelyn Gao worked closely with me during my first two years at Berkeley and were great mentors. Once I began the PhD, Rowan McAllister and Nick Rhinehart guided my research in autonomous vehicles and helped maintain my research momentum during the challenging middle years of my PhD. I'm also grateful to Pieter Abbeel and Sergey Levine, who, though not my formal advisors, provided crucial feedback that helped several papers cross the finish line to publication.

The RISE Lab was an incredible home for my research. I was fortunate to work alongside amazing colleagues in Joey's group: Kevin Lin, Lisa Dunlap, Justin Wong, Shishir Patil, Tianjun Zhang, Paras Jain, Sukrit Kalra, and Suzie Petryk. The infamous "Star Factory" cubicle, which allegedly housed the Databricks founders and later the Anyscale founders, became the birthplace of MemGPT, Gorilla, and SkyPlane during my time there - an unmatched density of open source research contributions in a single cubicle space.

And finally, I would like to thank Sarah Wooders and Kevin Lin, who are joining me on an

exciting new adventure post-PhD, where we'll be taking our research on context management for LLM agents into the real world.

This thesis, and the journey it represents, would not have been possible without the support, guidance, and encouragement of all these incredible people. Thank you.

Additional context around this thesis: This thesis was written during an extraordinary period in artificial intelligence research (2017-2024). When I began my PhD, deep reinforcement learning was at the forefront of autonomous systems research, with breakthroughs like AlphaGo and OpenAI Five demonstrating superhuman performance in complex games.

Then came the transformer revolution. What started as incremental improvements in natural language processing rapidly evolved into something far more profound. The release of ChatGPT in late 2022 marked a paradigm shift not just in AI research, but in how society viewed artificial intelligence. Large Language Models demonstrated capabilities that seemed impossible just a few years earlier: sophisticated reasoning and intelligence that was *general*.

I had the unique privilege of not just witnessing this revolution, but actively participating in it. My research journey paralleled this transition: from working on fundamental challenges in deep reinforcement learning, to ultimately helping pioneer new approaches for building reliable autonomous systems using Large Language Models. This thesis reflects both the 'before' and 'after' of this pivotal moment in AI history; a time that will likely be remembered as the beginning of the foundation model era.

The speed of progress during this period was unprecedented. Papers that seemed cutting-edge when I started my PhD quickly became historical artifacts. Research directions that appeared promising were suddenly obsolete. Yet this rapid evolution created extraordinary opportunities to contribute to genuinely new directions in computer science: to help establish the foundations for how we build AI systems in this new era.

This thesis represents my small contribution to this remarkable period in computing history.

Chapter 1

Introduction

Building intelligent autonomous systems that can effectively reason, adapt, and interact with their environment has been a longstanding goal in artificial intelligence. The recent deep learning revolution, particularly the emergence of Large Language Models (LLMs), has dramatically changed our approach to building such systems. This thesis traces this evolution through several key advances in building agentic systems, from deep reinforcement learning to modern LLM-based approaches, focusing on the critical components needed to create reliable autonomous agents.

1.1 Background

The development of agentic systems has undergone several significant paradigm shifts, each introducing new capabilities and challenges. Understanding these shifts and their implications is crucial for building effective autonomous agents.

1.1.1 The Deep Learning Revolution in Robotics and Control

The integration of deep neural networks with reinforcement learning marked a significant advancement in autonomous systems. This combination enabled:

- End-to-End Learning: Deep RL allowed systems to learn directly from raw sensory input, eliminating the need for hand-engineered features.
- Complex Policy Learning: Neural networks as function approximators enabled learning sophisticated control policies for high-dimensional tasks.
- Improved Generalization: Deep architectures promised better transfer of learned behaviors across similar tasks.

However, several key challenges emerged:

- Limited Generalization: Learned policies often failed to transfer beyond their specific training conditions
- Sample Inefficiency: Deep RL systems required extensive training data
- Sparse Rewards: Many real-world tasks lack the dense feedback signals needed for effective learning

1.1.2 The Rise of Foundation Models

The emergence of Large Language Models and other foundation models introduced a new paradigm for building intelligent systems, offering:

- Strong Zero-Shot Capabilities: The ability to handle novel tasks without task-specific training
- Rich World Knowledge: Pre-trained representations capturing broad knowledge about the world
- Natural Language Interfaces: The ability to understand and generate human language

However, using LLMs as autonomous agents presents unique challenges:

- Context Limitations: Fixed-size context windows restrict long-term memory and reasoning
- Reliability Issues: Tendency to hallucinate or produce inconsistent outputs
- Integration Complexity: Challenges in connecting language capabilities to real-world actions

1.2 Deep Learning for Agentic Systems

This section introduces our early work on addressing fundamental challenges in deep reinforcement learning and its application to autonomous systems. First, we present a systematic evaluation framework for assessing generalization in deep RL, introducing standardized environments and metrics that enable meaningful comparison of different approaches. This work established that while deep RL could achieve impressive results in specific scenarios, the learned policies were often brittle and failed to generalize. Building on these insights, we developed Hindsight Task Relabeling (HTR), a novel approach that enables meta-RL algorithms to learn adaptation strategies in sparse reward settings. HTR demonstrates how intelligent relabeling of experience can bootstrap the learning of complex adaptation strategies, addressing a key limitation in traditional meta-RL approaches.

1.3 The LLM Agent Paradigm

Finally, we address the emerging challenge of building reliable agents using Large Language Models. While LLMs demonstrate remarkable reasoning capabilities, their effectiveness as autonomous agents is limited by fundamental constraints in their architecture and training. We present MemGPT, an operating system-inspired framework that enables LLMs to manage their own memory and state. MemGPT represents a fundamental shift in how we approach building LLM-based agents, introducing:

- Virtual Context Management: Techniques for managing information beyond the model's context window
- Self-Directed Memory Operations: Enabling the LLM to control its own memory state
- Persistent State Management: Methods for maintaining consistent agent state across interactions

Together, these advances establish a foundation for engineering the infrastructure needed to transform LLMs from sophisticated chatbots into reliable autonomous agents.

Chapter 2

Assessing Generalization in Deep Reinforcement Learning

2.1 Introduction

Deep reinforcement learning (RL) has emerged as an important family of techniques that may support the development of intelligent systems that learn to accomplish goals in a variety of complex real-world environments (Mnih et al. 2015; Arulkumaran et al. 2017). A desirable characteristic of such intelligent systems is the ability to function in diverse environments, including ones that have never been encountered before. Yet, deep RL algorithms are commonly trained and evaluated on a fixed environment. The algorithms are evaluated in terms of their ability to optimize a policy in a complex environment, rather than their ability to learn a representation that generalizes to previously unseen circumstances. Indeed, their sensitivity to even subtle changes in the environment and the dangers of overfitting to a specific environment have been noted in the literature (Rajeswaran et al. 2017b; Henderson et al. 2018; Zhang et al. 2018; Whiteson et al. 2011).

Generalization is often regarded as an essential characteristic of advanced intelligent systems and a central issue in AI research (Lake et al. 2017; Marcus 2018; Dietterich 2017). It refers to both interpolation to environments similar to those seen during training and extrapolation outside the training data distribution. The latter is particularly challenging but is crucial to the deployment of systems in the real world.

Generalization in deep RL has been recognized as an important problem and is under active investigation (Rajeswaran et al. 2017a; Pinto et al. 2017; Kansky et al. 2017; Yu et al. 2017; Wang et al. 2016; Duan et al. 2016b; Sung et al. 2017; Clavera et al. 2018; Sæmundsson et al. 2018). However, each work uses a different set of environments and experimental protocols. For example, Kansky et al. (2017) propose a graphical model architecture, evaluating on variations of the Atari game Breakout. Rajeswaran et al. (2017a) propose training on a distribution of domains in risk-averse manner and evaluate on two continuous control tasks from MuJoCo (Hopper and HalfCheetah). Duan et al. (2016b) aim to learn a policy that automatically adapts to the environment dynamics and evaluate on bandits, tabular Markov

decision processes, and maze navigation. Sæmundsson et al. (2018) combine learning a hierarchical latent model for the environment dynamics and model predictive control, evaluating on two continuous control tasks (cart-pole swing-up and double-pendulum swing-up).

What appears to be missing is a common testbed for evaluating generalization in deep RL: a clearly defined set of tasks, metrics, and baselines that can support concerted community-wide progress. In other words, research on generalization in deep RL has not yet adopted the 'common task framework', a proven catalyst of progress (Donoho 2015). Only by creating such testbeds and evaluating on them can we fairly compare and contrast the merits of different algorithms and accurately measure progress made on the problem.

Our contribution is to establish a reproducible framework for investigating generalization in deep RL, with the hope that it will catalyze progress on this problem, and to present an empirical evaluation of generalization in deep RL algorithms as a baseline. We select a diverse but manageable set of environments, comprising classic control problems and MuJoCo locomotion tasks, built on top of OpenAI Gym for ease of adoption. Like Rajeswaran et al. (2017a) and others, we focus on generalization to changes in the system dynamics, which is implemented by specifying degrees of freedom (parameters) along which the environment specifications can be varied. Significantly, we test generalization in two regimes: interpolation and extrapolation. Interpolation implies that agents should perform well in test environments where parameters are similar to those seen during training. Extrapolation requires agents to perform well in test environments where parameters are different from those seen during training.

To provide the community with a set of clear baselines, we evaluate two deep RL algorithms on all environments and under different combinations of training and testing regimes. We chose one algorithm from each of the two major families: A2C from the actor-critic family and PPO from the policy gradient family. Using the same experimental protocol, we also evaluate two schemes for tackling generalization in deep RL: EPOpt, which learns a policy that is robust to environment changes by maximizing expected reward over the most difficult of a distribution of environment parameters, and RL^2, which learns a policy that can adapt to the environment at hand by taking into account the trajectory it sees. Because each scheme is constructed based on existing deep RL algorithms, our evaluation is of four algorithms: EPOpt-A2C, EPOpt-PPO, RL^2-A2C, and RL^2-PPO. We analyze the results and draw conclusions that can guide future work on generalization in deep RL. The experimental results confirm that extrapolation is more difficult than interpolation and show that the 'vanilla' deep RL algorithms (A2C and PPO) were able to interpolate fairly successfully. Somewhat surprisingly, they interpolate and extrapolate better than their EPOpt and RL^2 variants, with the exception of EPOpt-PPO. RL^2-A2C and RL^2-PPO proved to be difficult to train and were unable to reach the level of performance of the other algorithms given the same amount of training resources.

2.2 Background

Generalization in RL. There are two main approaches to generalization in RL: learning policies that are robust to environment variations, or learning policies that adapt to such variations. A popular approach to learn a robust policy is to maximize a risk-sensitive objective, such as the conditional value at risk (Tamar et al. 2015), over a distribution of environments. Morimoto & Doya (2001) maximize the minimum reward over possible disturbances, proposing robust versions of the actor-critic and value gradient methods in a control theory framework. This maximin objective is utilized by others in the context where environment changes are modeled by uncertainties in the transition probability distribution function of a Markov decision process. Nilim & Ghaoui (2004) assume that the set of possible transition probability distribution functions are known, while Lim et al. (2013) and Roy et al. (2017) estimate it using sampled trajectories from the distribution of environments of interest. A recent representative of this approach applied to deep RL is the EPOpt algorithm (Rajeswaran et al. 2017a), which maximizes the conditional value at risk, i.e. expected reward over the subset of environments with lowest expected reward. EPOpt has the advantage that it can be used in conjunction with any RL algorithm. Adversarial training has also been proposed to learn a robust policy; for MuJoCo locomotion tasks, Pinto et al. (2017) trains an adversary that tries to destabilize the agent during training.

A robust policy may sacrifice performance on many environment variants in order to not fail on a few. Thus, an alternative, recently popular approach to generalization in RL is to learn a policy that can adapt to the environment at hand (Yu et al. 2017). To do so, a number of algorithms learn an embedding for each environment variant using trajectories sampled from that environment, which is input into a policy. Then, at test time, the current trajectory can be used to compute an embedding for the current environment, enabling automatic adaptation of the policy. Duan et al. (2016b), Wang et al. (2016), Sung et al. (2017), and Mishra et al. (2018a), which differ mainly in the way embeddings are computed, consider model-free RL by letting the embedding be input into a policy and/or value function. Clavera et al. (2018) consider model-based RL, in which the embedding is input into a dynamics model and actions are selected using model predictive control. Under a similar setup, Sæmundsson et al. (2018) utilize probabilistic dynamics models and inference.

This literature review has focused on RL algorithms for generalization that do not require updating the learned model or policy at test time, in keeping with our benchmark's evaluation procedure. There has been work on generalization in RL that utilize such updates, primarily under the umbrellas of transfer learning, multi-task learning, and meta-learning. Taylor & Stone (2009) surveys transfer learning in RL where a fixed test environment is considered, with Rusu et al. (2016) being an example of recent work on that problem using deep networks. Ruder (2017) provides a survey of multi-task learning in general, which, different from our problem of interest, considers a fixed finite population of tasks. Finn et al. (2017) present a meta-learning formulation of generalization in RL and Al-Shedivat et al. (2018) extend it for continuous adaptation in non-stationary environments.

Empirical methodology in deep RL. Shared open-source software infrastructure, which

enables reproducible experiments, has been crucial to the success of deep RL. The deep RL research community uses simulation frameworks, including OpenAI Gym (Brockman et al. 2016), the Arcade Learning Environment (Bellemare et al. 2013; Machado et al. 2017), DeepMind Lab (Beattie et al. 2016), and VizDoom (Kempka et al. 2016). The MuJoCo physics simulator (Todorov et al. 2012) has been influential in standardizing a number of continuous control tasks. For ease of adoption, our work builds on OpenAI Gym and MuJoCo tasks, allowing variations in the environment specifications in order to study generalization. OpenAI recently released a benchmark for transfer learning in RL (Nichol et al. 2018), in which the goal is to train an agent to play new levels of a video game with fine-tuning at test time. In contrast, our benchmark does not allow fine-tuning and focuses on control tasks.

Our work also follows in the footsteps of a number of empirical studies of reinforcement learning algorithms, which have primarily focused on the case where the agent is trained and tested on a fixed environment. Henderson et al. (2018) investigate reproducibility in deep RL, testing state-of-the-art algorithms on four MuJoCo tasks: HalfCheetah, Hopper, Walker2d, and Swimmer. They show that results may be quite sensitive to hyperparameter settings, initialization, random seeds, and other implementation details, indicating that care must be taken not to overfit to a particular environment. The problem of overfitting in RL was recognized earlier by Whiteson et al. (2011), who propose an evaluation methodology based on training and testing on multiple environments sampled from a distribution and experiment with three classic environments: MountainCar, Acrobot, and puddle world. Nair et al. (2015) evaluate generalization with respect to different starting points in Atari games. Duan et al. (2016a) present a benchmark suite of continuous control tasks and conduct a systematic evaluation of reinforcement learning algorithms on those tasks. They consider generalization in terms of interpolation on a subset of their tasks. In contrast to these works, we address a greater variety of tasks, extrapolation as well as interpolation, and algorithms for learning deep RL agents that generalize.

2.3 Notation

In RL, environments are formulated in terms of Markov Decision Processes (MDPs) (Sutton & Barto 2017). An MDP M is defined by the tuple $(\mathbb{S}, \mathbb{A}, p, r, \gamma, \rho_0, T)$ where $\mathbb{S}$ is the set of possible states, $\mathbb{A}$ is the set of actions, $p : \mathbb{S}\mathbb{A}\mathbb{S} \to \mathbb{R}_{\geq 0}$ is the transition probability distribution function, $r : \mathbb{S}\mathbb{A} \to \mathbb{R}$ is the reward function, γ is the discount factor, $\rho_0 : \mathbb{S} \to \mathbb{R}_{\geq 0}$ is the initial state distribution at the beginning of each episode, and T is the time horizon per episode. Generalization to environment variations is usually characterized as generalization to changes in p and r; our benchmark considers changes in p.

Let s_t and a_t be the state and action taken at time t. At the beginning of each episode, $s_0 \sim \rho_0(s_0)$. Under a policy π stochastically mapping a sequence of states to actions, $a_t \sim \pi(a_t \mid s_t, \cdots, s_0)$ and $s_{t+1} \sim p(s_{t+1} \mid a_t)$, giving a trajectory $\{s_t, a_t, r(s_t, a_t)\}$, $t = 0, 1, \cdots$. RL algorithms, taking the MDP as fixed, learn π to maximize the expected reward over an episode $J_M(\pi) = \mathbb{E}^{\pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$, where $r_t = r(s_t, a_t)$. They often utilize the

concepts of a value function $v^{\pi}_{M}(s) = \mathbb{E}^{\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \mid s_0 = s\right]$ and a state-action value function $Q^{\pi}_{M}(s, a) = \mathbb{E}^{\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a\right]$.

2.4 Algorithms

We first evaluate 'vanilla' deep RL algorithms from two main categories: actor-critic and policy gradient. From the actor-critic family, we chose A2C (Mnih et al. 2016), and from the policy gradient family we chose PPO (Schulman et al. 2017).[1] These algorithms are oblivious to variations in the environment; they were not designed with generalization in mind. We also include recently-proposed algorithms that are designed to be able to generalize: EPOpt (Rajeswaran et al. 2017a) from the robust approaches and RL^2 (Duan et al. 2016b) from the adaptive approaches. Both these methods are built on top of 'vanilla' deep RL algorithms, so for completeness we evaluate a Cartesian product of the algorithms for generalization and the 'vanilla' algorithms: EPOpt-A2C, EPOpt-PPO, RL^2-A2C, and RL^2-PPO. Next we briefly summarize A2C, PPO, EPOpt, and RL^2, using the notation in Section 2.3.

Advantage Actor-Critic (A2C). A2C involves the interplay of two optimizers; a critic learns a parametric value function, while an actor utilizes that value function to learn a parametric policy that maximizes expected reward. At each iteration, trajectories are generated using the current policy, with the environment and hidden states of the value function and policy reset at the end of each episode. Then, the policy and value function parameters are updated using RMSProp (Hinton et al. 2012), with an entropy term added to the policy objective function in order to encourage exploration. We use an implementation from OpenAI Baselines (Dhariwal et al. 2017).

Proximal Policy Optimization (PPO). PPO aims to learn a sequence of monotonically improving parametric policies by maximizing a surrogate for the expected reward via gradient ascent, cautiously bounding the improvement achieved at each iteration. At iteration i, trajectories are generated using the current policy π_{θ_i}, with the environment and hidden states of the policy reset at the end of each episode. The following objective is then maximized with respect to θ using Adam (Kingma & Ba 2015):

$$\mathbb{E}_{s\sim\rho_{\theta_i}, a\sim\pi_{\theta_i}} \min\left[\ell_\theta(a, s) A_{\pi_{\theta_i}}(s, a), m_\theta(a, s) A_{\pi_{\theta_i}}(s, a)\right]$$

where ρ_{θ_i} are the expected visitation frequencies under π_{θ_i}, $\ell_\theta(a, s) = \pi_\theta(a \mid s)/\pi_{\theta_i}(a \mid s)$, m_θ equals $\ell_\theta(a, s)$ clipped to the interval $[1 - \delta, 1 + \delta]$ with $\delta \in (0, 1)$, and $A_{\pi_{\theta_i}}(s, a) = Q^{\pi_{\theta_i}}_{M}(s, a) - v^{\pi_{\theta_i}}_{M}(s)$. Again, we use an implementation from OpenAI Baselines, PPO2.

Ensemble Policy Optimization (EPOpt). To generalize over a distribution of environments (MDPs) $p(M)$, we would like to learn a policy that maximizes the expected reward

[1] We carried out preliminary experiments on other deep RL algorithms including A3C, TRPO, and ACKTR. A2C and A3C/ACKTR had similar qualitative results, as did PPO and TRPO.

over the distribution, $\mathbb{E}^{\pi}_{M\sim p(M)}[J_M(\pi)]$. In order to obtain a policy that is also robust to out-of-distribution environments, EPOpt instead maximizes the expected reward over the $\epsilon \in (0, 1]$ fraction of environments with worst expected reward:

$$\mathbb{E}^{\pi}_{M\sim p(M)}[J_M(\pi) \leq y] \quad \text{where} \quad P_{M\sim p(M)}(J_M(\pi) \leq y) = \epsilon.$$

At each iteration, the algorithm generates a number of complete episodes according to the current policy where at the end of each episode a new environment is sampled from $p(M)$ and reset. (As in A2C and PPO, at the end of each episode the hidden states of the policy and value function are reset.) It keeps the ϵ fraction of episodes with lowest reward and uses them to update the policy with some RL algorithm (TRPO (Schulman et al. 2015) in the paper). We instead use A2C and PPO, building our implementation of EPOpt on top of the implementations of A2C and PPO.

RL^2. To learn a policy that can adapt to the dynamics of the environment at hand, RL^2 models the policy and value functions as a recurrent neural network (RNN) with the current trajectory as input, not just the sequence of states. The hidden states of the RNN may be viewed as an environment embedding. Specifically, for the RNN the inputs at time t are s_t, a_{t-1}, r_{t-1}, and d_{t-1}, where d_{t-1} is a Boolean variable indicating whether the episode ended after taking action a_{t-1}; the output is a_t and the hidden states are updated to h_{t+1}. Like the other algorithms, at each iteration trajectories are generated using the current policy with the environment state reset at the end of each episode. However, unlike the other algorithms, a new environment is sampled from $p(M)$ only at the end of every N episodes, which we call a trial. ($N = 2$ in our experiments.) Likewise, the hidden states of the policy and value functions are reinitialized only at the end of each trial. The generated trajectories are then input into any RL algorithm, maximizing expected reward in a trial; the paper uses TRPO, while we use A2C and PPO. As with EPOpt, our implementation of RL^2 is built on top of the implementations of A2C and PPO.

2.5 Environments

Our environments are modified versions of four environments from the classic control problems in OpenAI Gym (Brockman et al. 2016) (CartPole, MountainCar, Acrobot, and Pendulum) and two environments from OpenAI Roboschool (Schulman et al. 2017) (HalfCheetah and Hopper) that are based on the corresponding MuJoCo (Todorov et al. 2012) environments. CartPole, MountainCar, and Acrobot have discrete action spaces, while the others have continuous action spaces. We alter the implementations to allow control of several environment parameters that affect the transition probability distribution functions of the corresponding MDPs. Each of the six environments has three versions, with d parameters allowed to vary.

1. Deterministic (D): The parameters of the environment are fixed at the default values in the implementations from Gym and Roboschool. Every time the environment is reset, only the state is reset.

2. Random (R): Every time the environment is reset, the parameters are uniformly sampled from a d-dimensional box containing the default values. This is done by independently sampling each parameter uniformly from an interval containing the default value.

3. Extreme (E): Every time the environment is reset, its parameters are uniformly sampled from 2^d d-dimensional boxes anchored at the vertices of the box in R. This is done by independently sampling each parameter uniformly from the union of two intervals that straddle the corresponding interval in R.

Appendix 2.10.1 contains a schematic of the parameter ranges in D, R, and E when $d = 2$. We now describe the environments.

CartPole (Barto et al. 1983). A pole is attached to a cart that moves on a frictionless track. For at most 200 time steps, the agent pushes the cart either left or right with the goal of keeping the pole upright. There is a reward of 1 for each time step the pole is upright, with the episode ending when the angle of the pole is too large. Three environment parameters can be varied: (1) push force magnitude, (2) pole length, (3) pole mass.

MountainCar (Moore 1990). The goal is to move a car to the top of a hill within 200 time steps. At each time step, the agent pushes a car left or right, with a reward of -1. Two environment parameters can be varied: (1) push force magnitude, (2) car mass.

Acrobot (Sutton 1995). The acrobot is a two-link pendulum attached to a bar with an actuator at the joint between the two links. At each time step, the agent applies torque (to the left, to the right, or not at all) to the joint in order to swing the end of the second link above the bar to a height equal to the length of the link. The reward system is the same as that of MountainCar, but with a maximum of 500 time steps. We have required that the links have the same parameters, with the following three allowed to vary: (1) length, (2) mass, (3) moment of inertia.

Pendulum. The goal is to, for 200 time steps, apply a continuous-valued force to a pendulum in order to keep it at a vertical position. The reward at each time step is a decreasing function of the pendulum's angle from vertical, the speed of the pendulum, and the magnitude of the applied force. Two environment parameters can be varied, the pendulum's: (1) length, (2) mass.

HalfCheetah. The half-cheetah is a bipedal robot with eight links and six actuated joints corresponding to the thighs, shins, and feet. The goal is for the robot to learn to walk on a track without falling over by applying continuous-valued forces to its joints. The reward at each time step is a combination of the progress made and the costs of the movements, e.g., electricity and penalties for collisions, with a maximum of 1000 time steps. Three environment parameters can be varied: (1) power, a factor by which the forces are multiplied before application, (2) torso density, (3) sliding friction of the joints.

Hopper. The hopper is a monopod robot with four links arranged in a chain corresponding to a torso, thigh, shin, and foot and three actuated joints. The goal, reward structure, and parameters are the same as those of HalfCheetah.

In all environments, the difficulty may depend on the values of the parameters; for example, in CartPole, a very light and long pole would be more difficult to balance. Therefore, the structure of the parameter ranges in R and E was constructed to include environments of various difficulties. The actual ranges of the parameters for each environment were chosen by hand and are listed in Appendix 2.10.1.[2]

2.6 Experimental setup

In sum, we benchmark six algorithms (A2C, PPO, EPOpt-A2C, EPOpt-PPO, RL^2-A2C, RL^2-PPO) and six environments (CartPole, MountainCar, Acrobot, Pendulum, HalfCheetah, Hopper). With each pair of algorithm and environment, we consider nine training-testing scenarios: training on D, R, and E and testing on D, R, and E. We refer to each scenario using the two-letter abbreviation of the training and testing environment versions, e.g., DR for training on D and testing on R. For A2C, PPO, EPOpt-A2C, and EPOpt-PPO, we train for 15000 episodes and test on 1000 episodes. For RL^2-A2C and RL^2-PPO, we train for 7500 trials, equivalent to 15000 episodes, and test on the last episodes of 1000 trials. Note that this is a fair comparison as policies without memory of previous episodes are expected to have the same performance in any episode of a trial, and we are able to evaluate the ability of RL^2-A2C and RL^2-PPO to adapt their policy to the environment parameters of the current trial. For the sake of completeness, we do a thorough sweep of hyperparameters and randomly generate random seeds. We report results over several runs of the entire hyperparameter sweep (the only difference being the random seeds). In the following paragraphs we describe the network architectures for the policy and value functions, our hyperparameter search, and the performance metrics we use for evaluation.

Policy and value function parameterization. We consider two network architectures for the policy and value functions. In the first, following Henderson et al. (2018), the policy and value functions are multi-layer perceptrons (MLPs) with two hidden layers of 64 units each and hyperbolic tangent activations; there is no parameter sharing. We refer to this architecture as FF (feed-forward). In the second,[3] the policy and value functions are the outputs of two separate fully-connected layers on top of a one-hidden-layer RNN with long short-term memory (LSTM) cells of 256 units. The RNN itself is on top of a MLP with two hidden layers of 256 units each, which we call the feature network. Again, hyperbolic tangent activations are used throughout; we refer to this architecture as RC (recurrent). For A2C, PPO, EPOpt-A2C, and EPOpt-PPO, we evaluate both architectures (whose inputs are the environment states), while for RL^2-A2C and RL^2-PPO, we evaluate only the second architecture (whose input is a tuple of states, actions, rewards, and Booleans as discussed in Section 2.4). In all cases, for discrete action spaces policies sample actions by taking a

[2]The ranges of the parameters were chosen so that a policy trained using PPO on D struggles quite a bit on the environments corresponding to the vertices of the box in R and fails completely on the environments corresponding to the most extreme vertices of the boxes in E.

[3]Based on personal communication with an author of Duan et al. (2016b).

softmax function over the policy network output layer; for continuous action spaces actions are sampled from a Gaussian distribution with mean the policy network output layer and diagonal covariance matrix whose entries are learned along with the policy and value function network parameters.

Hyperparameters. During training, in each algorithm and each version of each environment, we performed grid search over a set of hyperparameters used in the optimizers, and selected the value with the highest success probability when tested on the same version of the environment. The set of hyperparameters includes the learning rate for all algorithms and the length of the trajectory generated at each iteration (which we call batch size) for A2C, PPO, RL^2-A2C, and RL^2-PPO. They also include the coefficient of the policy entropy in the objective for A2C, EPOpt-A2C, and RL^2-A2C and the coefficient of the KL divergence between the previous policy and current policy for RL^2-PPO. The grid values are listed in Section 2.10.2. In EPOpt-A2C and EPOpt-PPO, we sample 100 environments per iteration and set ϵ first to 1.0 and then 0.1 after 100 iterations. Other hyperparameters, such as the discount factor, were set to the default values in OpenAI Baselines.

Performance metrics. The traditional performance metric used in the RL literature is the average total reward achieved by the policy in an episode. In the spirit of the definition of an RL agent as goal-seeking (Sutton & Barto 2017) and to obtain a metric independent of reward shaping, we also compute the percentage of episodes in which a certain goal is successfully completed, the success rate. This additional metric is a clear and interpretable way to compare performance across conditions and environments. We define the goals of each environment as follows: CartPole: balance for at least 195 time steps, MountainCar: get to the hilltop within 110 time steps, Acrobot: swing the end of the second link to the desired height within 80 time steps, Pendulum: keep the angle of the pendulum at most $\pi/3$ radians from vertical for the last 100 time steps of a trajectory with length 200, HalfCheetah and Hopper: walk for 20 meters.

2.7 Experimental setup

In sum, we benchmark six algorithms (A2C, PPO, EPOpt-A2C, EPOpt-PPO, RL^2-A2C, RL^2-PPO) and six environments (CartPole, MountainCar, Acrobot, Pendulum, HalfCheetah, Hopper). With each pair of algorithm and environment, we consider nine training-testing scenarios: training on D, R, and E and testing on D, R, and E. We refer to each scenario using the two-letter abbreviation of the training and testing environment versions, e.g., DR for training on D and testing on R. For A2C, PPO, EPOpt-A2C, and EPOpt-PPO, we train for 15000 episodes and test on 1000 episodes. For RL^2-A2C and RL^2-PPO, we train for 7500 trials, equivalent to 15000 episodes, and test on the last episodes of 1000 trials. Note that this is a fair comparison as policies without memory of previous episodes are expected to have the same performance in any episode of a trial, and we are able to evaluate the ability of RL^2-A2C and RL^2-PPO to adapt their policy to the environment parameters of the current trial. For the sake of completeness, we do a thorough sweep of hyperparameters and

randomly generate random seeds. We report results over several runs of the entire hyperparameter sweep (the only difference being the random seeds). In the following paragraphs we describe the network architectures for the policy and value functions, our hyperparameter search, and the performance metrics we use for evaluation.

Policy and value function parameterization. We consider two network architectures for the policy and value functions. In the first, following Henderson et al. (2018), the policy and value functions are multi-layer perceptrons (MLPs) with two hidden layers of 64 units each and hyperbolic tangent activations; there is no parameter sharing. We refer to this architecture as FF (feed-forward). In the second,[4] the policy and value functions are the outputs of two separate fully-connected layers on top of a one-hidden-layer RNN with long short-term memory (LSTM) cells of 256 units. The RNN itself is on top of a MLP with two hidden layers of 256 units each, which we call the feature network. Again, hyperbolic tangent activations are used throughout; we refer to this architecture as RC (recurrent). For A2C, PPO, EPOpt-A2C, and EPOpt-PPO, we evaluate both architectures (whose inputs are the environment states), while for RL^2-A2C and RL^2-PPO, we evaluate only the second architecture (whose input is a tuple of states, actions, rewards, and Booleans as discussed in Section 2.4). In all cases, for discrete action spaces policies sample actions by taking a softmax function over the policy network output layer; for continuous action spaces actions are sampled from a Gaussian distribution with mean the policy network output layer and diagonal covariance matrix whose entries are learned along with the policy and value function network parameters.

Hyperparameters. During training, in each algorithm and each version of each environment, we performed grid search over a set of hyperparameters used in the optimizers, and selected the value with the highest success probability when tested on the same version of the environment. The set of hyperparameters includes the learning rate for all algorithms and the length of the trajectory generated at each iteration (which we call batch size) for A2C, PPO, RL^2-A2C, and RL^2-PPO. They also include the coefficient of the policy entropy in the objective for A2C, EPOpt-A2C, and RL^2-A2C and the coefficient of the KL divergence between the previous policy and current policy for RL^2-PPO. The grid values are listed in Section 2.10.2. In EPOpt-A2C and EPOpt-PPO, we sample 100 environments per iteration and set ϵ first to 1.0 and then 0.1 after 100 iterations. Other hyperparameters, such as the discount factor, were set to the default values in OpenAI Baselines.

Performance metrics. The traditional performance metric used in the RL literature is the average total reward achieved by the policy in an episode. In the spirit of the definition of an RL agent as goal-seeking (Sutton & Barto 2017) and to obtain a metric independent of reward shaping, we also compute the percentage of episodes in which a certain goal is successfully completed, the success rate. This additional metric is a clear and interpretable way to compare performance across conditions and environments. We define the goals of each environment as follows: CartPole: balance for at least 195 time steps, MountainCar: get to the hilltop within 110 time steps, Acrobot: swing the end of the second link to the

[4] Based on personal communication with an author of Duan et al. (2016b).

Table 2.1: Generalization performance (in % success) of each algorithm, averaged over all environments (mean and standard deviation over five runs).

Algorithm	Architecture	Default	Interpolation	Extrapolation
A2C	FF	78.14 ± 6.07	76.63 ± 1.48	63.72 ± 2.08
	RC	81.25 ± 3.48	72.22 ± 2.95	60.76 ± 2.80
PPO	FF	78.22 ± 1.53	70.57 ± 6.67	48.37 ± 3.21
	RC	26.51 ± 9.71	41.03 ± 6.59	21.59 ± 10.08
EPOpt-A2C	FF	2.46 ± 2.86	7.68 ± 0.61	2.35 ± 1.59
	RC	9.91 ± 1.12	20.89 ± 1.39	5.42 ± 0.24
EPOpt-PPO	FF	85.40 ± 8.05	85.15 ± 6.59	59.26 ± 5.81
	RC	5.51 ± 5.74	15.40 ± 3.86	9.99 ± 7.39
RL^2-A2C	RC	45.79 ± 6.67	46.32 ± 4.71	33.54 ± 4.64
RL^2-PPO	RC	22.22 ± 4.46	29.93 ± 8.97	21.36 ± 4.41

desired height within 80 time steps, Pendulum: keep the angle of the pendulum at most $\pi/3$ radians from vertical for the last 100 time steps of a trajectory with length 200, HalfCheetah and Hopper: walk for 20 meters.

2.8 Results and discussion

We highlight some of the key findings and present a summary of the experimental results here, concentrating on the binary success metric. For each algorithm, architecture, and environment, we compute three numbers. (1) Default: success percentage on DD (the classic RL setting). (2) Interpolation: success percentages on RR. (3) Extrapolation: geometric mean of the success percentages on DR, DE, and RE. Table 2.1 summarizes the results. Section 2.10.3 contains analogous tables for each environment, which will be referred to in the following discussion.

A2C and PPO. With the FF architecture, the two 'vanilla' deep RL algorithms are often successful on the classic RL setting of training and testing on a fixed environment, as evidenced by the high values for Default. However, when those agents trained on environment version D are tested, we observed that they usually suffer from a significant drop in performance in R and an even further drop in E. When the algorithm is successful in the classic RL setting, as for PPO with the FF architecture, which has a Default number of 78.22, they are able to interpolate. (Interpolation equals 70.57 in that case.) That is, simply training on a distribution of environments, without any special mechanism for generalization, results in agents that can perform fairly well in similar environments. However, as expected in general they are less successful at extrapolation; PPO with the FF architecture has a Extrapolation number of 48.37. A2C with either architecture shows similar behavior to PPO with the FF

architecture, while PPO with the RC architecture had difficulty training on the classic RL setting and did not generalize well. For example, on all the environments except CartPole and Pendulum the FF architecture was necessary for PPO to train a successful policy on DD. The pattern of decrease from Default to Interpolation to Extrapolation shown in Table 2.1 also appears when looking at each environment individually. The magnitude of decrease depends on the combination of algorithm, architecture, and environment. For instance, on CartPole, A2C interpolates and extrapolates successfully, where Interpolation equals 100.00 and Extrapolation equals 93.63; this behavior is also shown for PPO with the FF architecture. On the other hand, on Hopper, PPO with the FF architecture has 85.54% success rate in the classic RL setting but struggles to interpolate (Interpolation equals 39.68) and fails to extrapolate (Extrapolation equals 10.36). This indicates that our choice of environments and their parameter ranges led to a variety of difficulty in generalization.

EPOpt. With the FF architecture, EPOpt-PPO improved both interpolation and extrapolation performance over PPO, as shown in Table 2.1. Looking at specific environments, on Hopper EPOpt-PPO has nearly twice the interpolation performance and significantly improved extrapolation performance compared to PPO. Such an improvement also appears for Pendulum. EPOpt-PPO, similar to PPO, generally did not benefit from using the recurrent architecture; this may be due to the LSTM requiring more data to train. EPOpt however did not demonstrate the same performance gains when combined with A2C. EPOpt-A2C was able to find limited success using the RC architecture on CartPole but for other environments failed to learn a working policy even in the Default setting.

RL^2. RL^2-A2C and RL^2-PPO proved to be difficult to train and data inefficient. This is possibly due to the RC architecture, as PPO also has difficulty training on D with that architecture as shown in Table 2.1. On most environments, the Default numbers are low, indicating that a working policy was not found in the classic RL setting of training and testing on a fixed environment. As a result, they also have low Interpolation and Extrapolation numbers. In a few, such as RL^2-PPO on CartPole and RL^2-A2C on HalfCheetah, a working policy was found in the classic RL setting, but the algorithm struggled to interpolate or extrapolate. A success story is RL^2-A2C on Pendulum, where we have nearly 100% success rate in DD, interpolate extremely well (Interpolation is 99.82), and extrapolate fairly well (Extrapolation is 81.79).

We observed that the partial success of these algorithms on the environments appears to be dependent on two implementation choices: the feature network in the RC architecture and the nonzero coefficient of the KL divergence between the previous policy and current policy in RL^2-PPO, which is intended to help stabilize training.

2.9 Conclusion

We introduced a new testbed and experimental protocol to measure the generalization ability of deep RL algorithms, to environments both similar to and different from those seen during training. Such a testbed enables us to compare the relative merits of algorithms for

learning generalizable RL agents. Our code, based on OpenAI Gym, is available online [5] and we hope that it will support future research on generalization in deep RL. Using our testbed we have evaluated two state-of-the-art deep RL algorithms, A2C and PPO, and two algorithms that explicitly tackle the problem of generalization in different ways: EPOpt, which aims to generalize by being robust to environment variations, and RL^2, which aims to automatically adapt to environment variations.

Overall, the 'vanilla' deep RL algorithms have better generalization performance than their more complex counterparts, being able to interpolate quite well with some extrapolation success. When combined with PPO under the FF architecture, EPOpt is able to outperform vanilla PPO; however, it does not generalize in the other cases. RL^2 on the other hand is difficult to train, and in its success cases provides no clear generalization advantage over the 'vanilla' deep RL algorithms or EPOpt. The sensitivity of the effectiveness of EPOpt and RL^2 to the base algorithm, architecture, and environment presents an avenue for future work, as intuitively EPOpt and RL^2 should be general-purpose approaches. We have considered model-free RL in our evaluation; another clear direction for future work is to perform a similar evaluation for model-based RL, in particular recent work such as Sæmundsson et al. (2018) and Clavera et al. (2018). Because model-based RL explicitly learns the system dynamics and generally is more data efficient, it could be better leveraged by adaptive techniques for generalization.

2.10 Additional details

2.10.1 Environment Details

Table 2.2 details the parameter ranges for each environment and environment setting: Deterministic (D), Random (R), and Extreme (E). Figure 2.1 illustrates the ranges from which the parameters are sampled; the parameters for D are fixed within the range of R, and E is uniformly sampled from a range wider than R, excluding the intervals corresponding to R.

2.10.2 Training Hyperparameters

The grid values we search over for each hyperparameter and each algorithm are listed below. In sum, the search space contains 183 unique hyperparameter configurations for all algorithms on a single training environment ($3,294$ training configurations), and each trained agent is evaluated on 3 test settings ($9,882$ total train/test configurations). We report results for 5 runs of the full grid search, a total of $49,410$ experiments.

- Learning rate:
  - A2C, EPOpt-A2C with RC architecture, and RL^2-A2C: $[0.007, 0.0007, 0.00007]$

[5]

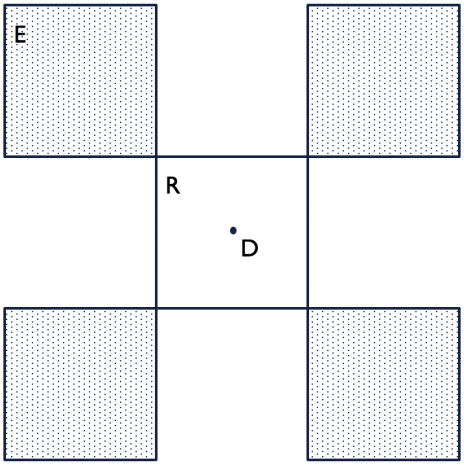


Figure 2.1: Schematic of the three versions of an environment.

Table 2.2: Ranges of parameters for each version of each environment, using set notation.

Environment	Parameter	D	R	E
CartPole	Force	10	[5,15]	[1,5]∪[15,20]
	Length	0.5	[0.25,0.75]	[0.05,0.25]∪[0.75,1.0]
	Mass	0.1	[0.05,0.5]	[0.01,0.05]∪[0.5,1.0]
MountainCar	Force	0.001	[0.0005,0.005]	[0.0001,0.0005]∪[0.005,0.01]
	Mass	0.0025	[0.001,0.005]	[0.0005,0.001]∪[0.005,0.01]
Acrobot	Length	1	[0.75,1.25]	[0.5,0.75]∪[1.25,1.5]
	Mass	1	[0.75,1.25]	[0.5,0.75]∪[1.25,1.5]
	MOI	1	[0.75,1.25]	[0.5,0.75]∪[1.25,1.5]
Pendulum	Length	1	[0.75,1.25]	[0.5,0.75]∪[1.25,1.5]
	Mass	1	[0.75,1.25]	[0.5,0.75]∪[1.25,1.5]
HalfCheetah	Power	0.90	[0.70,1.10]	[0.50,0.70]∪[1.10,1.30]
	Density	1000	[750,1250]	[500,750]∪[1250,1500]
	Friction	0.8	[0.5,1.1]	[0.2,0.5]∪[1.1,1.4]
Hopper	Power	0.75	[0.60,0.90]	[0.40,0.60]∪[0.90,1.10]
	Density	1000	[750,1250]	[500,750]∪[1250,1500]
	Friction	0.8	[0.5,1.1]	[0.2,0.5]∪[1.1,1.4]

– EPOpt-A2C with FF architecture: $[0.07, 0.007, 0.0007]$

– PPO, EPOpt-PPO with RC architecture: $[0.003, 0.0003, 0.00003]$

– EPOpt-PPO with FF architecture: $[0.03, 0.003, 0.0003]$

  - RL^2-PPO: $[0.0003, 0.00003, 0.000003]$

- Batch size:
  - A2C and RL^2-A2C: $[5, 10, 15]$
  - PPO and RL^2-PPO: $[128, 256, 512]$
- Policy entropy coefficient: $[0.01, 0.001, 0.0001, 0.00001]$
- KL divergence coefficient: $[0.3, 0.2, 0.0]$

2.10.3 Detailed Experimental Results

In order to elucidate the generalization behavior of each algorithm, here we present versions of Table 2.1 for each environment.

Table 2.3: Mean and standard deviation over five runs of generalization performance (in % success) on Acrobot.

Algorithm	Architecture	Default	Interpolation	Extrapolation
A2C	FF	88.52 ± 1.32	72.88 ± 0.74	66.56 ± 0.52
	RC	88.24 ± 1.53	73.46 ± 1.11	67.94 ± 1.06
PPO	FF	87.20 ± 1.11	72.78 ± 0.44	64.93 ± 1.05
	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
EPOpt-A2C	FF	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
EPOpt-PPO	FF	79.60 ± 5.86	69.20 ± 1.64	65.05 ± 2.16
	RC	3.10 ± 3.14	6.40 ± 3.65	15.57 ± 5.59
RL^2-A2C	RC	65.70 ± 8.68	57.70 ± 2.40	57.01 ± 2.70
RL^2-PPO	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0

2.10.4 Behavior of MountainCar

On MountainCar, several of the algorithms, including A2C with both architectures and PPO with the FF architecture, have greater success on Extrapolation than Interpolation, which is sometimes greater than Default (see Table 2.5). This is unexpected because Extrapolation combines the success rates of DR, DE, and RE, with E containing more extreme parameter settings, while Interpolation is the success rate of RR. To explain this phenomenon, we hypothesize that compared to R, E is dominated by easy parameter settings, e.g., those where the car is light but the force of the push is strong, allowing the agent to reach the top of the hill easily. In order to test this hypothesis, we create a heatmap of

Table 2.4: Mean and standard deviation over five runs of generalization performance (in % success) on CartPole.

Algorithm	Architecture	Default	Interpolation	Extrapolation
A2C	FF	100.00 ± 0.0	100.00 ± 0.0	93.63 ± 9.30
	RC	100.00 ± 0.0	100.00 ± 0.0	83.00 ± 11.65
PPO	FF	100.00 ± 0.0	100.00 ± 0.0	86.20 ± 12.60
	RC	65.58 ± 27.81	70.80 ± 21.02	45.00 ± 18.06
EPOpt-A2C	FF	14.74 ± 17.14	43.06 ± 3.48	10.48 ± 9.41
	RC	57.00 ± 4.50	55.88 ± 3.97	32.53 ± 1.47
EPOpt-PPO	FF	99.98 ± 0.04	99.46 ± 0.79	73.58 ± 12.19
	RC	29.94 ± 31.58	20.22 ± 17.83	14.55 ± 20.09
RL^2-A2C	RC	20.78 ± 39.62	0.06 ± 0.12	0.12 ± 0.23
RL^2-PPO	RC	87.20 ± 12.95	54.22 ± 34.85	51.00 ± 14.60

Table 2.5: Mean and standard deviation over five runs of generalization performance (in % success) on MountainCar.

Algorithm	Architecture	Default	Interpolation	Extrapolation
A2C	FF	79.78 ± 11.38	84.10 ± 1.25	89.72 ± 0.65
	RC	95.88 ± 4.10	74.84 ± 6.82	89.77 ± 0.76
PPO	FF	99.96 ± 0.08	84.12 ± 0.84	90.21 ± 0.37
	RC	0.0 ± 0.0	63.36 ± 0.74	15.86 ± 31.71
EPOpt-A2C	FF	0.0 ± 0.0	3.04 ± 0.19	3.63 ± 0.49
	RC	0.0 ± 0.0	62.46 ± 0.80	0.0 ± 0.0
EPOpt-PPO	FF	74.42 ± 37.93	84.86 ± 1.09	87.42 ± 5.11
	RC	0.0 ± 0.0	65.74 ± 4.88	29.82 ± 27.30
RL^2-A2C	RC	0.32 ± 0.64	57.86 ± 2.97	21.56 ± 30.35
RL^2-PPO	RC	0.0 ± 0.0	60.10 ± 0.91	31.27 ± 26.24

the reward achieved by A2C with the FF architecture trained on D and tested on R and E. We also investigated A2C with the RC architecture and PPO with the FF architecture, but because the heatmaps are qualitatively similar, we show only the heatmap for A2C with the FF architecture, in Figure 2.2. Referring to the structure in Figure 2.1, we see that the reward achieved by the policy is higher in the regions corresponding to E. Indeed, it appears that the largest regions of E are those with a large force, which enables the trained policy to push the car up the hill in less than 110 time steps, achieving the goal set in Section 2.7.

Table 2.6: Mean and standard deviation over five runs of generalization performance (in % success) on Pendulum.

Algorithm	Architecture	Default	Interpolation	Extrapolation
A2C	FF	100.00 ± 0.0	99.86 ± 0.14	90.27 ± 3.07
	RC	100.00 ± 0.0	99.96 ± 0.05	79.58 ± 6.41
PPO	FF	0.0 ± 0.0	31.80 ± 40.11	0.0 ± 0.0
	RC	73.28 ± 36.80	90.94 ± 7.79	61.11 ± 31.08
EPOpt-A2C	FF	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	RC	2.48 ± 4.96	7.00 ± 10.81	0.0 ± 0.0
EPOpt-PPO	FF	100.00 ± 0.0	77.34 ± 38.85	54.72 ± 27.57
	RC	0.0 ± 0.0	0.04 ± 0.08	0.0 ± 0.0
RL^2-A2C	RC	100.00 ± 0.0	99.82 ± 0.31	81.79 ± 3.88
RL^2-PPO	RC	46.14 ± 17.67	65.22 ± 21.78	45.76 ± 8.38

Table 2.7: Mean and standard deviation over five runs of generalization performance (in % success) on HalfCheetah.

Algorithm	Architecture	Default	Interpolation	Extrapolation
A2C	FF	85.06 ± 19.68	91.96 ± 8.60	40.54 ± 8.34
	RC	88.06 ± 12.26	74.70 ± 13.49	42.96 ± 7.79
PPO	FF	96.62 ± 3.84	95.02 ± 2.96	38.51 ± 15.13
	RC	20.22 ± 17.01	21.08 ± 26.04	7.55 ± 5.04
EPOpt-A2C	FF	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
EPOpt-PPO	FF	99.76 ± 0.08	99.28 ± 0.87	53.41 ± 9.41
	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
RL^2-A2C	RC	87.96 ± 4.21	62.48 ± 29.18	40.78 ± 5.99
RL^2-PPO	RC	0.0 ± 0.0	0.0 ± 0.0	0.16 ± 0.32

(Note that the reward is the negative of the number of time steps taken to push the car up the hill.)

This special case demonstrates the importance of considering a wide variety of environments when assessing the generalization performance of an algorithm; each environment may have idiosyncrasies that cause performance to be correlated with parameters. For example, Figure 2.3 shows a similar heatmap for A2C with the FF architecture on Pendulum, in which Interpolation is greater than Extrapolation. In this case, the policy trained on D struggles

Table 2.8: Mean and standard deviation over five runs of generalization performance (in % success) on Hopper.

Algorithm	Architecture	Default	Interpolation	Extrapolation
A2C	FF	15.46 ± 7.58	11.00 ± 7.01	1.63 ± 2.77
	RC	15.34 ± 8.82	10.38 ± 15.14	1.31 ± 1.23
PPO	FF	85.54 ± 6.96	39.68 ± 16.69	10.36 ± 6.79
	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
EPOpt-A2C	FF	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
EPOpt-PPO	FF	58.62 ± 47.51	80.78 ± 29.18	21.39 ± 16.62
	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
RL^2-A2C	RC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
RL^2-PPO	RC	0.0 ± 0.0	0.02 ± 0.04	0.0 ± 0.0

more on environments from E than on those from R, which bolsters our hypothesis.

2.10.5 Training Curves

To investigate the effect of EPOpt and RL^2 and the different environment versions on training, we plotted the training curves for PPO, EPOpt-PPO, and RL^2-PPO on each version of each environment, averaged over the five experiment runs and showing error bands based on the standard deviation over the runs. Training curves for all algorithms and environments are available at the following link: . We observe that in the majority of cases training appears to be stabilized by the increased randomness in the environments in R and E, including situations where successful policies are found. This behavior is particularly apparent for CartPole, whose training curves are shown in Figure 2.9 and in which all five algorithms above are able to find at least partial success. We see that especially towards the end of the training period, the error bands for training on E are narrower than those for training on D or R. Except for EPOpt-PPO with the FF architecture, the error bands for training on D appear to be the widest. Indeed, RL^2-PPO is very unstable when trained on D, possibly because the more expressive policy network overfits to the generated trajectories.

2.10.6 Videos of trained agents

The above link also contains videos of the trained agents of one run of the experiments for all environments and algorithms. We include the five scenarios considered in computing Default, Interpolation, and Extrapolation: DD, DR, DE, RR, and RE. Using HalfCheetah as a case study, we describe some particularly interesting behavior we saw.

A trend we noticed across several algorithms were similar changes in the cheetah's gait

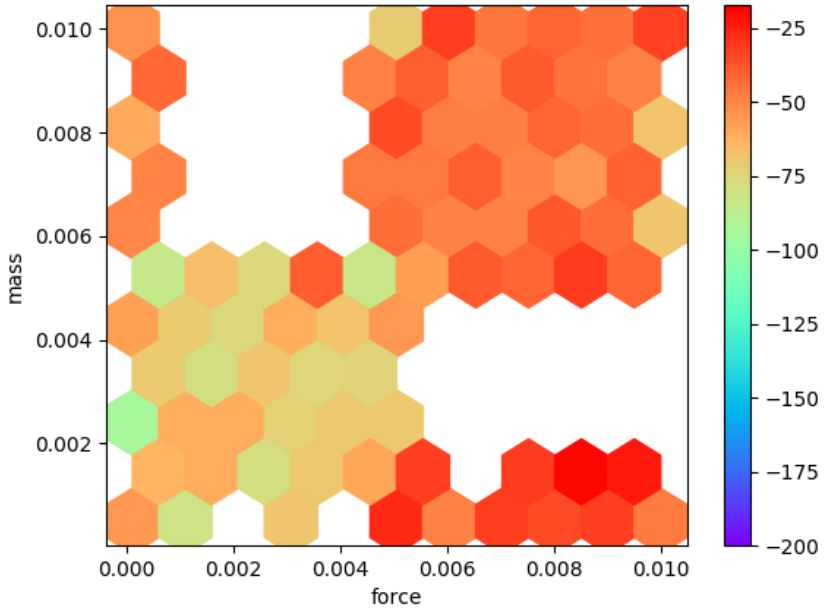


Figure 2.2: MountainCar: heatmap of the rewards achieved by A2C with the FF architecture on DR and DE. The axes are the two environment parameters varied in R and E.

that seem to be correlated with the difficulty of the environment. The cheetah's gait became forward-leaning when trained on the Random and Extreme environments, and remained relatively flat in the agents trained on the Deterministic environment (see figures 2.10 and 2.11). We hypothesize that the forward-leaning gait developed to counteract conditions in the R and E settings. The agents with the forward-learning gait were able to recover from face planting (as seen in the second row of figure 2.10), as well as maintain balance after violent leaps likely caused by settings with unexpectedly high power. In addition to becoming increasingly forward-leaning, the agents' gait also tended to become stiffer in the more extreme settings, developing a much shorter, twitching stride. Though it reduces the agents' speed, a shorter, stiffer stride appears to make the agent more resistant to adverse settings that would cause an agent with a longer stride to fall. This example illustrates how training on a range of different environment configurations may encourage policies that are more robust to changes in system dynamics at test time.

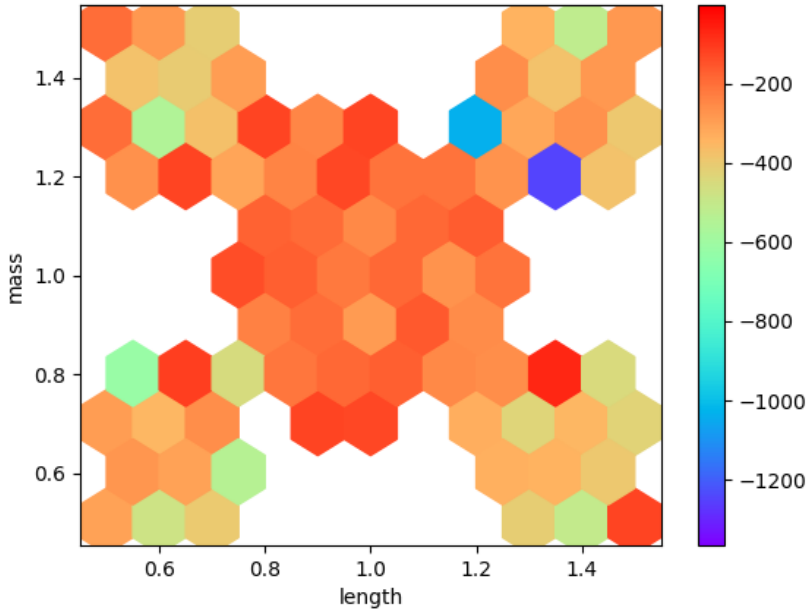


Figure 2.3: Pendulum: heatmap of the rewards achieved by A2C with the FF architecture on DR and DE. The axes are the two environment parameters varied in R and E.

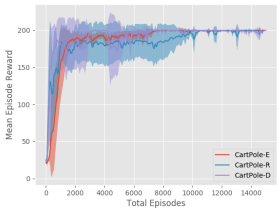


Figure 2.4: PPO with FF architecture

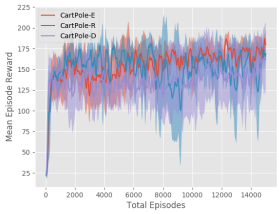


Figure 2.5: PPO with RC architecture

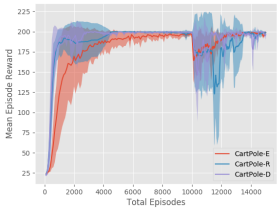


Figure 2.6: EPOpt-PPO with FF architecture

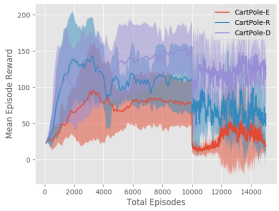


Figure 2.7: EPOpt-PPO with RC architecture

Figure 2.8: RL^2-PPO

Figure 2.9: Training curves for the PPO-based algorithms on CartPole, all three environment versions. Note that the decrease in mean episode reward at 10000 episodes in the two EPOpt-PPO plots is due to the fact that it transitions from being computed using all generated episodes ($\epsilon = 1$) to only the 10% with lowest reward ($\epsilon = 0.1$).

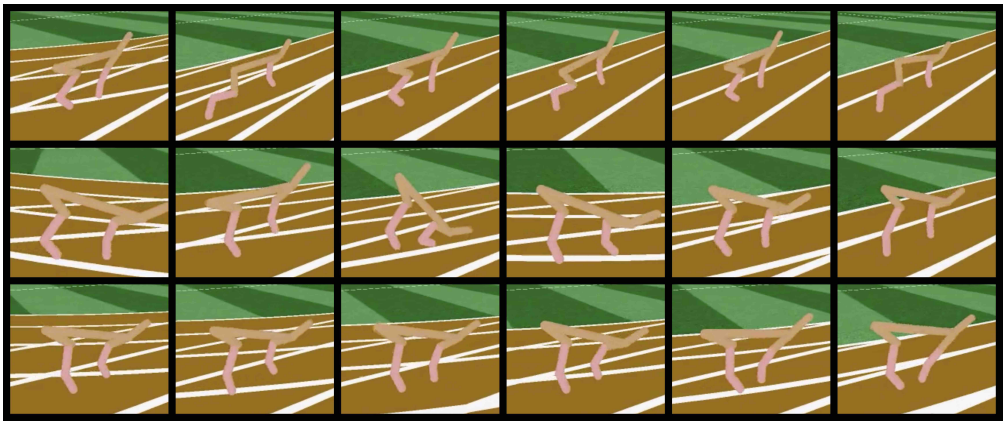


Figure 2.10: Video frames of agents trained with A2C on HalfCheetah, trained in the Deterministic (D), Random (R), and Extreme (E) settings (from top to bottom). All agents evaluated in the D setting.

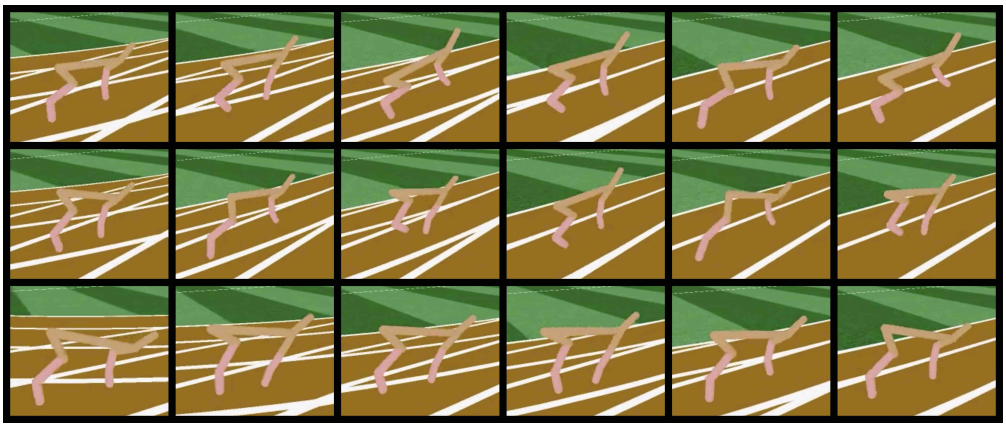


Figure 2.11: Video frames of agents trained with PPO on HalfCheetah, trained in the Deterministic (D), Random (R), and Extreme (E) settings (from top to bottom). All agents evaluated in the D setting.

Chapter 3

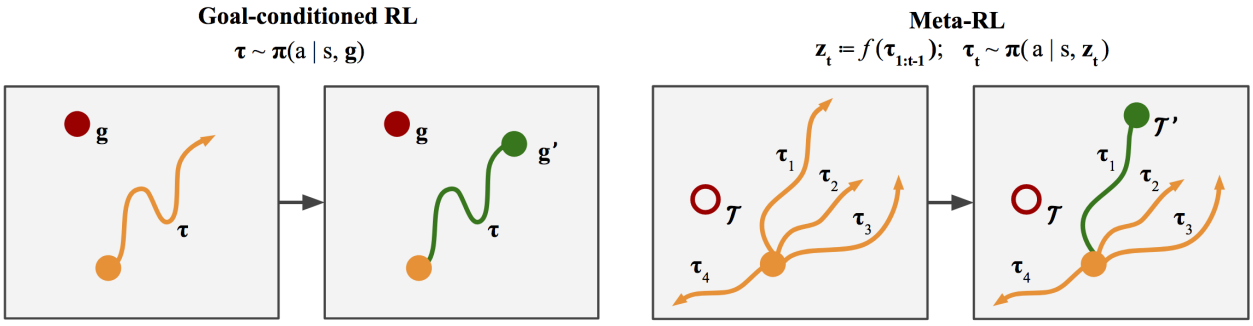
Hindsight Task Relabelling: Experience Replay for Sparse Reward Meta-RL

3.1 Introduction

Reinforcement learning (RL) has seen tremendous success applied to challenging games (Mnih et al. 2015; Silver et al. 2017) and robotic control (Lillicrap et al. 2015; Levine et al. 2016), driven by advances in compute and the use of deep neural networks as powerful function approximators in RL algorithms. However, agents trained using deep RL often struggle to meaningfully utilize past experience to learn new tasks, even if the new tasks differ only slightly from tasks seen during training (Zhang et al. 2018; Packer et al. 2018; Cobbe et al. 2019). In contrast, humans are adept at utilizing prior experience to rapidly acquire new skills and adapt to unseen environments.

Meta-reinforcement learning (meta-RL) aims to address this limitation by extending the RL framework to explicitly consider structured distributions of tasks (Schmidhuber 1987; Bengio et al. 1990; Thrun & Pratt 1998). Whereas conventional RL is concerned with learning a single task, meta-RL is concerned with *learning to learn*, that is, learning how to quickly learn a new task by leveraging prior experience on related tasks. Meta-RL methods generally utilize a limited amount of experience in a new environment to estimate a latent task embedding which conditions the policy, or to compute a policy gradient which is used to directly update the parameters of the policy.

A major challenge in both RL and meta-RL is learning with sparse rewards. When rewards are sparse or delayed, the adaptation stage in meta-RL becomes extremely difficult: inferring the task at hand requires receiving reward signal from the environment, which in the sparse reward setting only happens after successfully completing the task. Due to this inherent incompatibility between meta-RL and sparse rewards, existing meta-RL algorithms that consider the sparse reward setting either only work in simple environments that do not require temporally-extended exploration strategies for adaptation (Duan et al. 2016b; Stadie et al. 2018), or train exclusively using dense reward functions (Gupta et al. 2018b; Rakelly et al. 2019), which are designed to encourage an agent to learn adaptation strategies that



(a) Hindsight relabeling in goal-conditioned RL (b) Hindsight Task Relabeling (HTR) in meta-RL

Figure 3.1: In goal-conditioned RL (a), an agent must navigate to a provided goal location $\mathbf{g}$ (filled circle, revealed to the agent). An unsuccessful attempt for goal $\mathbf{g}$ provides no sparse reward signal, but can be relabelled as a successful attempt for goal $\mathbf{g}'$, creating sparse reward that can be used to train the agent. In meta-RL (b), the task $\mathcal{T}$ (i.e., goal, hollow circle) is never revealed to the agent, and instead must be inferred using experience on prior tasks and limited experience ($\tau_{1:t-1}$) on the new task. In (b), there is no shared optimal task $\mathcal{T}'$ to relabel all attempts with. HTR relabels each attempt τ under its own hindsight task $\mathcal{T}'$, and modifies the underlying meta-RL training loop to learn adaptation strategies on the relabelled tasks. Note that we include multiple trajectories τ in (b) vs a single trajectory in (a) to highlight the adaptation stage in meta-RL, which does not exist in goal-conditioned RL and requires significantly different sampling and relabeling procedures.

can be directly applied to the original sparse reward setting.

In this paper, we show that the concept of *hindsight relabeling* (Andrychowicz et al. 2017) from conventional RL can be applied in meta-RL to enable learning to learn in the sparse reward setting. Our key insight is that data collected on the true training tasks can be relabelled as pseudo-expert data for easier hindsight tasks, bootstrapping meta-training with the reward signal needed to train the agent. We introduce a new algorithm for off-policy meta-RL, which we call *Hindsight Task Relabeling* (HTR), and demonstrate its effectiveness by achieving state-of-the-art performance on a collection of challenging sparse reward environments that previously required shaped reward to solve. Not only is our approach able to learn adaptation strategies only using sparse reward, but it also learns strategies with comparable performance to existing approaches that use shaped reward functions.

3.2 Related work

In meta-RL, an agent learns an adaptation strategy by repeatedly adapting to training tasks in the inner loop of *meta-training*, with the hope that the learned adaptation procedure will generalize to new, unseen tasks during *meta-testing*. Context-based methods use recent experience (i.e., context, in the form of trajectories or transitions) in a new task to estimate a latent task embedding, and differ mainly in how this embedding is computed: Duan et al.

(2016b); Wang et al. (2016); Mishra et al. (2018b) propose aggregating context into the hidden state of the policy, whereas Rakelly et al. (2019) propose explicitly feeding the task embedding as input to the policy. Gradient-based methods use recent context to update differentiable hyperparameters (Xu et al. 2018b), loss functions (Sung et al. 2017; Houthooft et al. 2018), or to directly update policy parameters (Finn et al. 2017; Stadie et al. 2018; Xu et al. 2018a; Zintgraf et al. 2018; Gupta et al. 2018b; Rothfuss et al. 2019).

Many of the aforementioned approaches struggle to learn effective exploration strategies for tasks with sparse or delayed rewards. Methods that directly update the policy either implicitly (e.g., with a hidden state) or explicitly (e.g., with gradients) are generally unable to learn a policy that meaningfully explores, since the primary source of randomness is action-space, and thus is time-invariant. Gupta et al. (2018b) and Rakelly et al. (2019) propose using a probabilistic task variable that is sampled once per episode, which makes the primary source of variability task-dependent and enables temporally-extended exploration. While this approach enables effective adaptation in the sparse reward setting, both Gupta et al. (2018b) and Rakelly et al. (2019) learn the actual adaptation strategies using shaped (dense) reward functions during meta-training. In the environments they consider, the adaptation strategies learned using dense rewards generalize to sparse rewards despite the difference in reward structure; however, engineering a suitable reward function can be costly (Ng et al. 1999) and error prone (Clark & Amodei 2016), and thus it is highly desirable to devise a mechanism for learning to learn in the sparse reward setting that does not require an auxiliary dense reward function.

Our proposed method is closely related to prior work in unsupervised meta-RL and goal generation. Unsupervised meta-RL (Jabri et al. 2019; Gupta et al. 2018a) considers the meta-RL setting without access to a training task distribution; training tasks are instead self-generated with the aim of learning skills useful for the test task distribution. Our proposed method also generates a curriculum of training tasks, but with the intent of learning adaptation strategies in the sparse reward setting, as opposed to learning transferable skills in a setting with no rewards at all. Several methods exist for curriculum generation in the context of goal-conditioned policies, including adversarial goal generation (Florensa et al. 2018) and goal relabeling (Kaelbling 1993; Andrychowicz et al. 2017; Levy et al. 2017; Nair et al. 2018). This sentence and the previous one are misplaced. Add our method of curriculum learning is inspired by .., or we differ from them as .. It's 'novel' etc.. Recent work (Li et al. 2020; Eysenbach et al. 2020) has proposed the use of inverse RL to generalize goal relabeling to broader families of tasks. Unlike prior work on goal and task relabeling that use goal- or task-conditioned policies, we focus on the meta-RL setting where the task is unknown to the policy and must be inferred through deliberate and coherent exploration.

3.3 Background

In this section we formalize the meta-RL problem setting and briefly describe two algorithms spell check: with? which our approach, Hindsight Task Relabeling, builds on:

Probabilistic Embeddings for Actor-critic RL (PEARL) and Hindsight Experience Replay (HER).

3.3.1 Meta-Reinforcement Learning (Meta-RL)

Conventional RL assumes environments are modeled by a Markov Decision Processes (MDP) M, defined by the tuple $(\mathbb{S}, \mathcal{A}, p, r, \gamma, \rho_0, T)$ where $\mathbb{S}$ is the set of possible states, $\mathcal{A}$ is the set of actions, $p : \mathbb{S} \times \mathcal{A} \times \mathbb{S} \to \mathbb{R}_{\geq 0}$ is the transition probability density, $r : \mathbb{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, γ is the discount factor, $\rho_0 : \mathbb{S} \to \mathbb{R}_{\geq 0}$ is the initial state distribution at the beginning of each episode, and T is the time horizon of an episode (Sutton & Barto 2017).

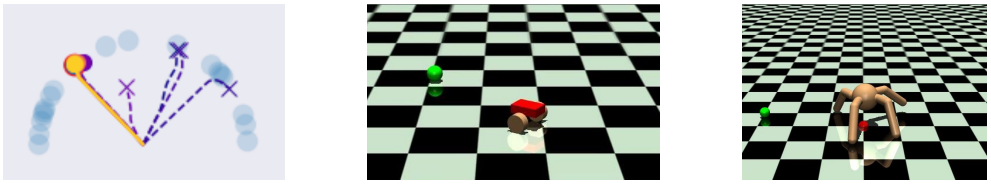
Let $\mathbf{s}_t$ and $\mathbf{a}_t$ be the state and action taken at time t. At the beginning of each episode, $\mathbf{s}_0 \sim \rho_0(\cdot)$. Under a stochastic policy π mapping a sequence of states to actions, $\mathbf{a}_t \sim \pi(\mathbf{a}_t \mid \mathbf{s}_t, \cdots, \mathbf{s}_0)$ and $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$, generating a trajectory $\tau = \{\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t)\}$, where $t = 0, 1, \cdots$. Conventional RL algorithms, which assume the environment is fixed, learn π to maximize the expected reward per episode $J_M(\pi) = \mathbb{E}^{\pi}\left[\sum_{t=0}^{T} \gamma^t \mathbf{r}_t\right]$, where $\mathbf{r}_t = r(\mathbf{s}_t, \mathbf{a}_t)$. They often utilize the concepts of a value function $V_M^{\pi}(\mathbf{s})$, the expected reward conditional on $\mathbf{s}_0 = \mathbf{s}$ and a state-action value function $Q_M^{\pi}(\mathbf{s}, \mathbf{a})$, the expected reward conditional on $\mathbf{s}_0 = \mathbf{s}$ and $\mathbf{a}_0 = \mathbf{a}$.

Meta-RL algorithms assume that there is a distribution of tasks $p(\mathcal{T})$, and usually maximize the expected reward over the distribution, $\mathbb{E}^{\pi}_{\mathcal{T} \sim p(\mathcal{T})}\left[J_{\mathcal{T}}(\pi)\right]$. The distribution of tasks corresponds to a distribution of MDPs $M(\mathcal{T})$, where $\mathcal{T}$ can paramaterize an MDP's dynamics or reward. In this work we focus on the latter case, where each $\mathcal{T}$ implies an MDP with a different reward function under the same dynamics. The agent is trained on a set of training tasks $\mathcal{T}_{train} \sim p(\mathcal{T})$ during meta-training, and evaluated on a held-out set of testing tasks $\mathcal{T}_{test} \sim p(\mathcal{T})$ during meta-testing.

An important distinction between meta-RL and multi-task or goal-conditioned RL is that in meta-RL, the task is never explicitly revealed to the agent or policy π through observations $\mathbf{s}$. Instead, π must explore the environment to infer the task to maximize expected reward. This crucial difference is one reason why Hindsight Experience Replay cannot be directly applied to the meta-RL setting.

3.3.2 Off-Policy Meta-Reinforcement Learning

We demonstrate the effectiveness of our approach by building on top of Probabilistic Embeddings for Actor-critic RL (PEARL) by Rakelly et al. (2019), an off-policy meta-RL algorithm which itself is built on top of soft actor-critic (SAC) by Haarnoja et al. (2018). SAC is an actor-critic algorithm based on the maximum entropy RL objective that optimizes a stochastic policy π with off-policy data. In addition to the policy (actor) π, SAC concurrently learns twin state-action value functions (critics) Q. Transitions collected using the policy π are added to the replay buffer B, and Q and π are optimized using loss functions regularized by the entropy of π.



(a) Point Robot (b) Wheeled Locomotion (c) Ant Locomotion

Figure 3.2: Sparse reward environments for meta-RL that require temporally-extended exploration. In each environment, the task (the top-left circle in (a), the green sphere in (b) and (c)) is not revealed to the agent via the observation. The agent must instead infer the task through temporally-extended exploration (illustrated by the dotted lines in (a)), since no reward signal is provided until the task is successfully completed. Prior meta-RL methods such as PEARL (Rakelly et al. 2019) and MAESN (Gupta et al. 2018b) are only able to (meta-)learn meaningful adaptation strategies using dense reward functions. Our approach, Hindsight Task Relabeling (HTR), can (meta-)train with the original sparse reward function and does not require additional dense reward functions.

Rakelly et al. (2019) extend SAC to the meta-RL setting by adding a context encoder that uses recent history in an environment to estimate a task embedding, which is then utilized by the policy. Specifically, PEARL uses a network $q_\phi(\mathbf{z} \mid \mathbf{c})$ that takes as input recently collected data (i.e., context $\mathbf{c}$) to infer a probabilistic latent context variable Z. Samples from the latent context variable condition the actor, $\pi_\theta(\mathbf{a} \mid \mathbf{s}, \mathbf{z})$, and critic, $Q_\theta(\mathbf{s}, \mathbf{a}, \mathbf{z})$. Instead of a single replay buffer (as in SAC), there are T separate buffers $B_{i=1...T}$, corresponding to the number of tasks sampled for $\mathcal{T}_{train}$. The separate per-task replay buffers enable sampling off-policy (but on-*task*) context during the inner loop of meta-training, which significantly improves data efficiency.

During meta-training, transitions are collected for each B_i by sampling $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{c})$ and acting according to $\pi_\theta(\mathbf{a} \mid \mathbf{s}, \mathbf{z})$. $\mathbf{z}$ is periodically sampled during data collection such that the context contains transitions collected using the policy conditioned on the prior, as well the same policy conditioned on the posterior. Gradients used to optimize q_ϕ, π_θ and Q_θ are computed in task-specific batches across a random subset of training tasks $\mathcal{T}_i \sim p(\mathcal{T})$, each with an associated replay buffer B_i. During meta-testing, for each test task $\mathcal{T}_i \sim p(\mathcal{T})$, an empty buffer B_i is initialized, $\mathbf{z}$ is sampled from the prior, and the policy conditioned on $\mathbf{z}$ is rolled out. As additional context is added to the replay buffer, the belief over $\mathbf{z}$ narrows, enabling the policy to adapt to the task at hand.

3.3.3 Hindsight Experience Replay

Hindsight Experience Replay (HER) (Andrychowicz et al. 2017) is a method for off-policy goal-conditioned RL in environments with sparse reward. The key insight behind HER is that unsuccessful attempts (i.e., attempts that received no reward signal) generated via a goal-conditioned policy can be relabelled as successful attempts for a 'hindsight' goal that was actually achieved, e.g., the final state (see Figure 3.1). HER can be applied to

any off-policy RL algorithm that uses goal-conditioned policies (where the actor and/or critic receive the desired goal as part of the state, i.e., $\pi(\mathbf{a} \mid \mathbf{s}, \mathbf{g})$ and $Q(\mathbf{s}, \mathbf{a}, \mathbf{g})$). When a transition $\{\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{g}) \mid \mathbf{g}\}$ is sampled from the replay buffer during training, with some probability HER rewrites the desired goal $\mathbf{g}$ with an achieved goal $\mathbf{g}'$ and recomputes the reward under the new goal. The modified transition $\{\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{g}') \mid \mathbf{g}'\}$ is used to optimize $\pi(\mathbf{a} \mid \mathbf{s}, \mathbf{g}')$ and $Q(\mathbf{s}, \mathbf{a}, \mathbf{g}')$. Hindsight relabeling generates an implicit curriculum: initially, sampled transitions are rewritten with relatively 'easy' goals achievable with random policies, and as training progresses, relabelled goals are increasingly likely to be near the true desired goals.

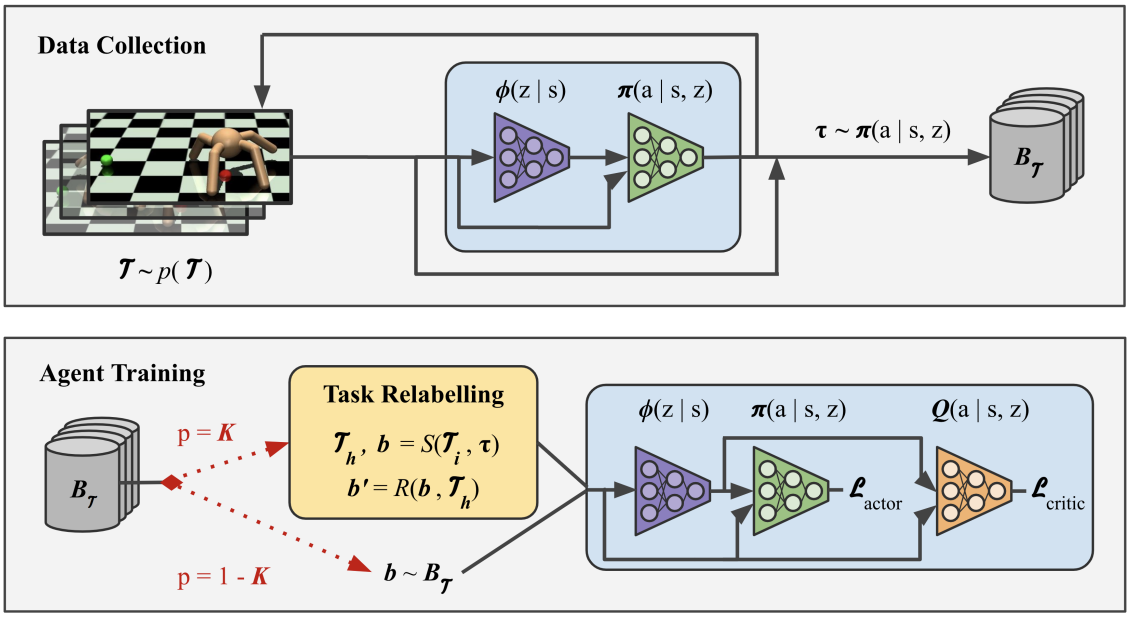


Figure 3.3: Illustration of Hindsight Task Relabeling (HTR) in a meta-RL training loop. HTR is agnostic to the underlying (off-policy) meta-RL algorithm; the agent architecture and/or training specifics (e.g., the encoder ϕ, actor π and Q-function neural networks shown in blue) can be modified independently of the relabeling scheme. HTR can also be performed in an 'eager' fashion at the data collection stage (as opposed to 'lazy' relabeling in the data sampling stage), see Section 3 for details.

3.4 Leveraging Hindsight in Meta-Reinforcement Learning

Our algorithm, Hindsight Task Relabeling, applies ideas from hindsight relabeling to the meta-RL setting to enable learning adaptation strategies for tasks with sparse or delayed rewards. Similar to how policies in HER are conditioned on a goal $\mathbf{g}$, policies in context-based meta-RL are conditioned on a latent task embedding $\mathbf{z}$. A crucial difference in meta-RL is that the task is hidden from the agent: the agent must infer relevant features of the task

at hand using experience on prior tasks and a limited amount of experience gathered in the environment. Our key insight is that an unsuccessful experience collected in an *unknown* task can be relabelled as a successful experience for a *known* hindsight task, i.e., a transition $\{\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t, \mathcal{T})\}$ generated under an unknown $\mathcal{T}$ can be rewritten under a known $\mathcal{T}'$ as $\{\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t, \mathcal{T}')\}$, regardless of whether $\mathcal{T}' \in \mathcal{T}_{train}$.

Relabeling transitions under hindsight tasks serves as an additional source of supervision in learning the latent task embedding $\mathbf{z} \Leftarrow f(\mathbf{c})$ by enabling gradient-based optimization of f in the absence of meaningful reward signal (in PEARL, this embedding is parameterized by the context encoder q_ϕ). Similar to how HER generates an implicit curriculum of goals, HTR generates an implicit curriculum of tasks that gradually shifts from easier hindsight tasks towards the true training distribution $\mathcal{T}_{train} \sim p(\mathcal{T})$: initially, the agent is unable to recover reward signal on the true training tasks, but it can bootstrap the learning of an adaptation strategy by learning on easier hindsight tasks.

We outline our method for meta-training with Hindsight Task Relabeling in Algorithm 3.4 (task relabeling is only used during meta-training; meta-testing is unchanged). We evaluate our approach using PEARL as the off-policy meta-RL algorithm A, though the approach we describe is general to any off-policy meta-RL algorithm, and could potentially be integrated with on-policy context-based meta-RL algorithms (such as RL^2) via hindsight policy gradients (Rauber et al. 2019).

3.4.1 Algorithm Design

Two important considerations in adapting hindsight relabeling to meta-RL are (i) how to choose the hindsight task, and (ii) how to sample the transitions to be relabelled. Together these choices form the task relabeling strategy S in Algorithm 3.4. In HER, transitions are relabelled with goals that are optimal at the trajectory-level (see Figure 3.1). The key difference between our method and HER is that our meta-RL relabeling scheme needs to construct batches of data that share the same pseudo-task (or real task), which is fundamentally different from standard HER. HER does not consider the setting where batches of trajectories are collected in different MDPs; in HER, transitions are relabeled independently with locally-optimal pseudo-goals, and batches of training data for the agent contain many different pseudo-goals (and real goals). In the meta-RL setting, the agent must estimate the task from a recent context of transitions with the same underlying task (unknown to the agent), and therefore relabeling should take into consideration more than a single trajectory; the agent explores an environment which has a fixed task $\mathcal{T} \sim p(\mathcal{T})$, and as it accumulates context it uses the context to identify the singular task and update its policy accordingly.

A naive application of hindsight relabeling to meta-RL would assign each transition in the batch its own hindsight task (e.g., give each τ in Figure 3.1b its own $\mathcal{T}'$). This is a simple and straightforward way to apply HER to meta-RL that may seem correct at the surface-level, however, this is bound to fail if the meta-RL training process is left unchanged, since the context encoder is trained to estimate the task from a collection of transitions generated in that task (not a batch of transitions generated across several distinct tasks). A similarly

Algorithm 1: Hindsight Task Relabelling for Off-Policy Meta-Reinforcement Learning

Require: Off-policy meta-RL algorithm A, training tasks $\mathcal{T}_{1:T} \sim p(\mathcal{T})$, context encoder ϕ, actor π, critic Q, relabelling function R, task relabelling strategy S, and task relabelling probability K

Initialize a replay buffer B_i for each training task $\mathcal{T}_i$
while *meta-training* **do**
  for *each* $\mathcal{T}_i$ **do**
    Collect data on $\mathcal{T}_i$ according to A (e.g., by rolling out π conditioned on $z \sim \phi$)
    Add data to task replay buffer B_i
  for *each* $\mathcal{T}_i$ **do**
    $p_h \sim \mathcal{U}(0, 1)$
    if $p_h \geq K$ **then**
      Select a minibatch of transitions b_i and hindsight task $\mathcal{T}_h$ using strategy S
        (e.g., sample transitions from trajectory t, and select a state reached in t for $\mathcal{T}_h$)
      Relabel transitions in b_i according to $\mathcal{T}_h$, i.e., $R(b_i, \mathcal{T}_h)$
    else
      Sample minibatch of transitions $b_i \sim B_i$
    Compute gradients and update ϕ, π, and Q using b_i according to A

Figure 3.4: HTR algorithm

flawed approach is to generate a single hindsight task from a random transition in the batch, and then to relabel the entire batch of transitions with the same hindsight task (e.g., choose a $\mathcal{T}'$ Figure 3.1b using a random τ to relabel all τs with). The issue with this approach is that the hindsight task is only guaranteed to be optimal for a single trajectory, and may in fact be highly sub-optimal for other trajectories in the batch. We present two relabeling strategies, *Single Episode Relabeling (SER)* and *Episode Clustering (EC)*, that are inspired by the relabeling strategy used in Andrychowicz et al. (2017) but are specifically designed to work in the meta-RL setting.

3.4.2 Single Episode Relabeling (SER) strategy

In SER, a single episode (i.e., trajectory) is randomly selected to generate the hindsight task and sample transitions from (i.e., sample N transitions from 1 trajectory, each relabelled under 1 new task). Despite the fact that this reduces the pool of context from the entire task buffer to a single episode, this setting closely resembles meta-testing (where context is drawn purely online), and we found this resampling strategy to work well in practice since it results in more relabelled transitions with non-zero reward. HTR using SER is outlined in Algorithm 3.4.

3.4.3 Episode Clustering (EC) strategy

An alternative strategy to Single Episode Resampling is to cluster trajectories that satisfied similar hindsight tasks into the same task buffers B_h, essentially creating additional task buffers for hindsight tasks that can be used for training in-place of the real task buffers

B_i (which are also sampled with some probability K). For example, in goal-reaching tasks, trajectories can easily be mapped to hindsight buffers with trajectories from separate trials by discretizing the state space and creating a buffer for each partition. In more complex tasks, alternative clustering approaches (e.g., learning-based) can be used to group together trajectories that can be relabeled with high reward conditioned on the same tasks. Episode clustering relabels a similar number of transitions with non-zero reward to the resampling strategy, but with less duplicate transitions per-batch. In practice, we found the SER strategy to be far less complex and similarly effective to the EC approach. See Section 3.5 for an empirical comparison between SER and EC, and the supplement for a more detailed comparison.

3.4.4 Comparison of HTR and HER

Both SER and EC are similar to the relabeling strategy used in HER in some respects, but differ in others: HER samples N transitions from the full replay buffer, and relabels each transition independently (N different goals). HTR with SER samples 1 trajectory, and samples N transitions from that trajectory all relabelled with 1 new task. HTR with EC samples N transitions from 1 hindsight replay buffer, where each transition in that buffer has been relabeled with 1 new task. In HER, the hindsight goal for a given transition is chosen by selecting a random transition that occurs after the sampled transition in the same trajectory. In HTR, hindsight tasks are assigned in the same way as HER, but a single hindsight task is applied to an entire batch of transitions.

An important distinction between HER and HTR is that in HER, bootstrapping training with hindsight goals does not change the optimal policy, since a goal-conditioned policy with sufficient capacity should (in theory) be able to represent an optimal policy for every goal (including both original goals and hindsight goals). In contrast, the optimal exploration strategy for a particular meta-RL environment depends on its true task distribution, and bootstrapping training with hindsight tasks may expand the training task distribution in a manner that changes the optimal exploration strategy. One way to benefit from HTR's bootstrapping while ensuring the final learned strategy is optimal for the original task distribution is to only relabel data from tasks on which the agent has never received (non-zero) reward. Similarly, another solution is to anneal the relabelling probability K to zero during meta-training. Neither method was required to achieve the results shown in this paper, however, we expect they may be useful if applying HTR to other meta-RL environments.

3.4.5 Limitations

The key limitation of our method is that it assumes trajectories can be relabeled under new task (i.e., a mapping $\mathbf{s} \rightarrow \mathcal{T}$ or $\tau \rightarrow \mathcal{T}$ exists), which is a reasonable assumption for goal-reaching environments (as studied in Rakelly et al. (2019); Gupta et al. (2018b); Andrychowicz et al. (2017)), but may be significantly more challenging in other environments. Imagine a complex robotic manipulation environment (e.g., retrieving an object from a drawer) where reward is sparse and only received upon successful completion of the entire

task; in this scenario, relabeling the zero-reward transitions generated by an initial random policy into useful signal for training an optimal policy is less straightforward than the goal-reaching setting, and may require explicit sub-task specification using domain expertise.

Similar to how HER is not necessarily only for goal-reaching *goal-conditioned* RL, HTR is not necessarily only for goal-reaching *meta*-RL, and if a good relabeling function exists then either approach can be successfully applied in their respective setting (HER for goal-conditioned RL, HTR for meta-RL). However, as in the original HER work, we focus our experiments on goal-reaching tasks, where the reward function is a simple function of the state and can be easily repurposed for relabeling. It may be possible to relax this assumption and extend HTR to more families of tasks by employing more complex task relabeling schemes, e.g., by extending work on inverse RL for hindsight relabeling (Li et al. 2020; Eysenbach et al. 2020) to the meta-RL setting, or by carefully engineering reward functions specific to each environment, however we leave this to future work.

Additionally, our method adds a new hyperparameter K to the underlying off-policy meta-RL algorithm, which may need to be tuned for optimal results. The specification and tuning of a relabeling probability K is also required in standard HER (see Andrychowicz et al. (2017)).

3.5 Experiments

In evaluating our proposed method, we aim to answer the following questions. (i) Can Hindsight Task Relabeling enable (meta-)learning of adaptation strategies in challenging sparse reward environments, where existing meta-RL methods fail without shaped reward? (ii) How do the adaptation strategies learned using Hindsight Task Relabeling compare to those learned using shaped reward functions? (iii) How do key implementation choices (such as relabeling probability K) affect its performance?

3.5.1 Environments

We evaluate our method on a suite of sparse reward environments based those proposed by Gupta et al. (2018b) and Rakelly et al. (2019) (see Figure 3.2). In prior work, each environment exposes two reward functions: a dense reward function used during meta-training, and a sparse reward function used during meta-testing. The key difference in our experimental setup is that we consider the setting where sparse reward is used both during meta-testing *and meta-training*. In each environment, a set of 100 tasks is sampled for meta-training, and a set of 100 tasks is sampled from the same task distribution for meta-testing. The environments were each modified to increase their difficulty, such that random exploration is far less likely to encounter sparse reward. Refer to the supplement for further details on the experimental setup.

- *Point Robot.* A point robot must navigate a 2D plane to different goals located along the perimeter of a half-semicircle. The state includes the robot's coordinates but does

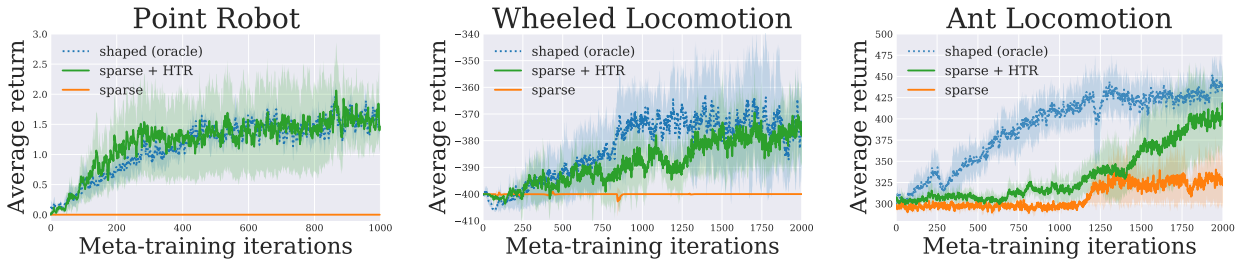


Figure 3.5: Evaluating adaptation to train tasks progressively *during* meta-training. Y-axis measures average sparse return during adaptation throughout meta-training (shaded std dev), though the oracle is still trained using dense reward. Conventional meta-RL methods struggle to learn using sparse reward. Hindsight Task Relabeling (HTR) is comparable to dense reward meta-training performance.

not include the goal, therefore the agent must explore the environment to discover the goal. In the dense reward variant of the environment, the reward is the negative $L2$ distance to the goal. In the sparse reward variant, reward is given only when the agent is within a short distance to the goal. An example optimal exploration strategy in the sparse reward variant is to efficiently traverse the perimeter of the semicircle until the goal is found.

- *Wheeled Locomotion.* To test if our approach generalizes to more complex state and action spaces, we use several continuous control environments. In the wheeled locomotion environment, the agent must navigate to the goal distribution by controlling two wheels independent to turn. Similar to the point robot, the dense reward function is the negative distance to the goal, while the sparse reward function only provides reward signal when the agent is nearby the goal.

- *Ant (Quadruped) Locomotion.* This environment requires controlling a quadruped ant with a high-dimensional state and action space. The task distribution and reward functions are the same as the point robot and wheeled locomotion environment, however, exploration requires coordinating movement with four legs to navigate to specific locations.

3.5.2 HTR enables meta-training using only sparse reward

To answer question (i), we compare our hindsight task relabeling approach (using PEARL as the base meta-RL algorithm) to standard PEARL, as well as an oracle PEARL which uses the dense reward function during training. Our oracle baseline is equivalent to the standard meta-RL setup in the 'sparse reward' experiments in Gupta et al. (2018b) and Rakelly et al. (2019), despite it never training on the sparse reward function. Note that the term 'oracle' is somewhat of a misnomer, since training on a proxy dense reward function designed to aid the learning of an RL agent (to be evaluated on the sparse reward function) is inherently

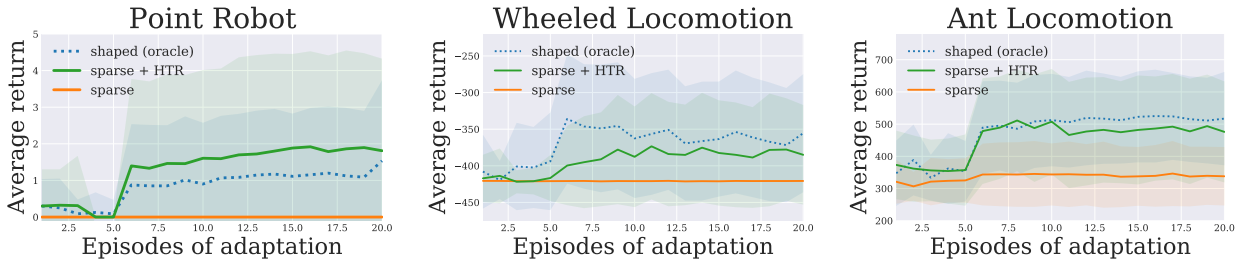


Figure 3.6: Evaluating adaptation to test tasks *after* meta-training. Y-axis measures average (sparse) return during adaptation using context collected online, using sparse reward only. Adaptation strategies learned with Hindsight Task Relabeling (HTR) generalize to held-out tasks as well as the oracle which is learned using shaped reward functions. Without HTR or access to a shaped reward during meta-training, the agent is unable to learn a reasonable strategy.

optimizing a different objective. Optimizing an RL agent for a proxy objective has been shown to lead to inadvertently bad performance on the true objective (Clark & Amodei 2016), and therefore it is often better to directly optimize for the true objective (e.g., the original sparse reward function) when possible.

As discussed in prior work, meta-learning on sparse reward environments proves extremely challenging: in Figures 3.5 and 3.7, we see that PEARL is unable to make learn a reasonable policy just using sparse rewards in the same amount of time it takes the oracle (PEARL trained on dense reward) and HTR (PEARL trained on sparse reward using Hindsight Task Relabelling) to converge. In all of our tested environments, HTR is not only comparable to using shaped reward during meta-training, but it also can learn adaptation strategies that perform as well as the oracle during meta-testing on the sparse reward environment. In Figure 3.6, we see that the adaptation strategies learned with Hindsight Task Relabeling are similarly effective to those learned by the oracle, despite the oracle having access to a dense reward function during meta-training.

Given an indefinite amount of meta-training time, or an easier environment configuration (e.g., a shorter goal distance), PEARL should be able to learn a similarly optimal strategy to the oracle and HTR only using sparse reward. If PEARL *is* able to successfully learn only using sparse rewards, HTR can be used to improve sample efficiency during meta-training. In the case that the amount of time or computational resources needed to train purely on sparse rewards is intractable, HTR is extremely effective at learning adaptation strategies comparable to using a hand-designed dense reward function.

Additionally, as mentioned in Section 3.2, in many sparse reward settings designing a dense reward function to aid training is infeasible or undesirable, in which case HTR is a strong alternative to reward engineering. Because HTR does not train on a proxy reward, there is no opportunity for a mismatch between reward functions at meta-training and meta-testing: HTR optimizes for the true reward function on a superset of tasks $\mathcal{T}_{hindsight} \cup \mathcal{T}_{train} \sim p(\mathcal{T})$, whereas using a shaped reward optimizes for a proxy reward function on the true set

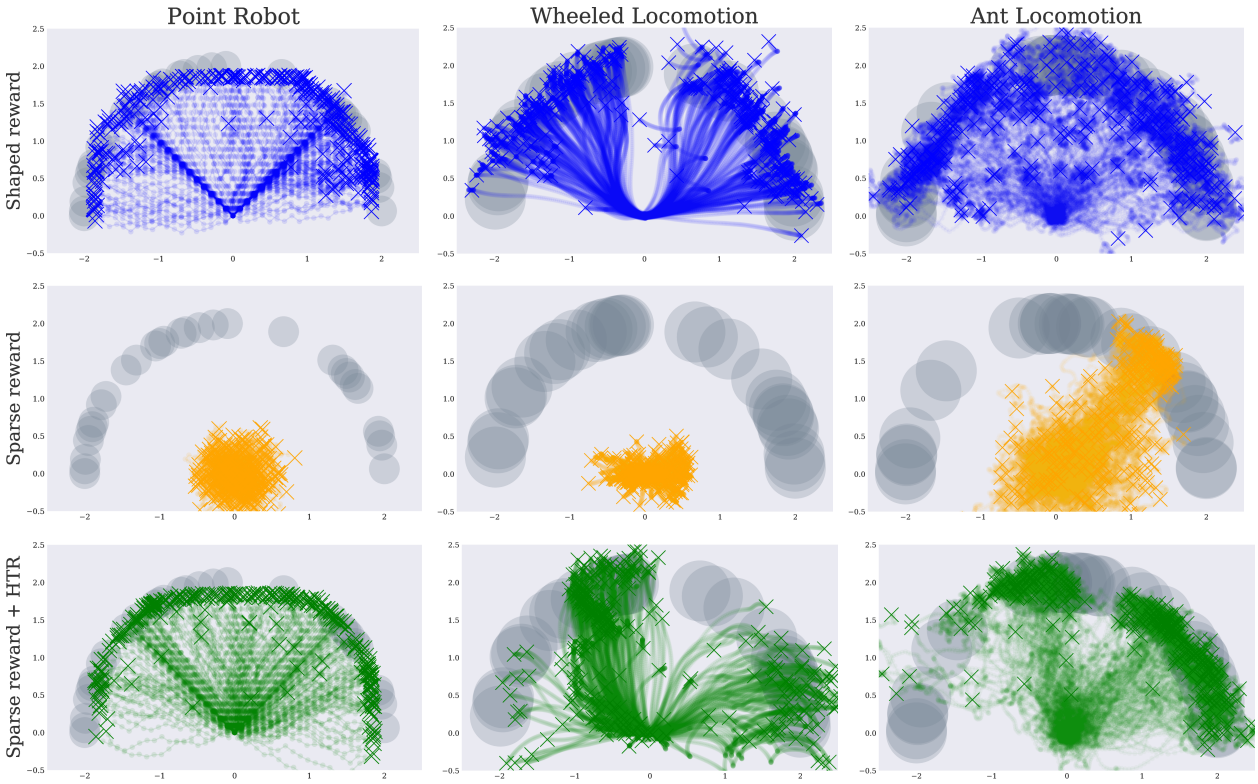


Figure 3.7: Visualizing exploration behavior learned during meta-training using 300 pre-adaptation trajectories (i.e., sampled from the latent task prior). In the sparse reward setting, without HTR (middle row) the agent is unable to learn a meaningful exploration strategy and appears to explore randomly near the origin. With HTR (bottom row), the agent learns to explore near the true task distribution (grey circles), similar to an agent trained with a shaped dense reward function (top row).

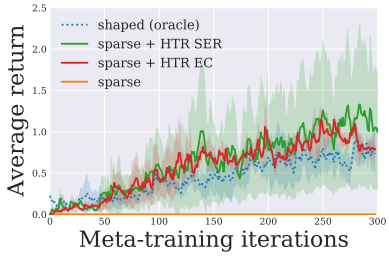


Figure 3.8: Comparing HTR with SER vs EC on Point Robot.

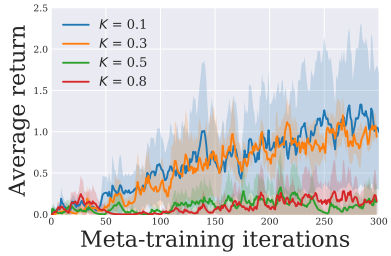


Figure 3.9: Average return when varying K on Point Robot.

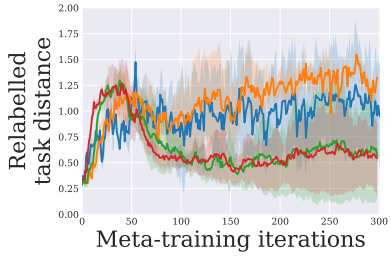


Figure 3.10: Average task distance when varying K on Point Robot.

of training tasks $\mathcal{T}_{train} \sim p(\mathcal{T})$.

3.5.3 Varying key hyperparameters

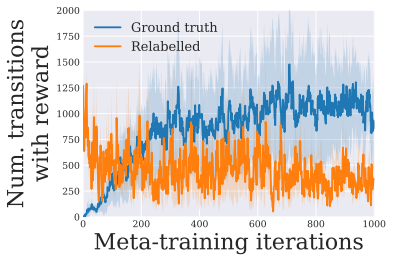


Figure 3.11: Relative reward signal from hindsight vs ground truth tasks using Point Robot.

In our experiments we found a relatively low relabeling probability (e.g., $K = 0.1$ and 0.3) was often most effective for HTR (see Figure 3.9). $K = 0.1$ means that relabelled data from a hindsight task is roughly as likely to be used for training as data from any of the 100 tasks in $\mathcal{T}_{train}$: $K = 0.1$ corresponds to using far less relabelled data for training than in conventional HER: in contrast, Andrychowicz et al. (2017) relabel over 80% of sampled transitions. We hypothesize that a low K works best for HTR since it provides enough reward signal to bootstrap learning, without biasing the task distribution too far from the ground truth distribution; in Figure 3.10, we can see that both $K = 0.5$ and 0.8 converge on a mean relabelled task distance of 0.5, whereas $K = 0.1$ and 0.3 are closer to the ground truth task distance of 2.0.

In both HER and HTR, the relabeling probability is an important hyperparameter that should be tuned for optimal performance. In the original HER paper, Andrychowicz et al. (2017) reported that a relabelling probability of 80% or 90% worked best on their three goal-conditioned RL environments. For both algorithms, there exists a range of relabelling probabilities (alternatively a range of ratios of original-to-relabelled data) where the relabeling algorithm works well, and a range of values where the algorithm performs poorly (the HER paper did not include results for under 50% relabelled data, which may include more negative results).

In Figure 3.10, we see that the average distance reached by the policy during meta-training gradually increases over time, increasing the average relabelled task distance. Hindsight tasks used for relabelling initially correspond to easily achievable tasks, and shift towards the ground truth task distribution. Initially, the only reward signal available for training comes from the relabelled transitions, however, as training progresses the agent is able to recover true reward signal from the environment (see Figure 3.11) and the hindsight tasks are more likely to resemble the ground truth tasks.

As described in Section 3.4, we found that EC generally performed similarly to SER (see Figure 3.8), despite being significantly more complex to implement and requiring additional tuned hyperparameters. We expect that a more significant performance difference between the two approaches may become apparent on different sparse reward meta-RL tasks, or through experimenting with more general episode clustering approaches than the state space discretization used in this work.

3.6 Conclusion

In this paper, we propose a novel approach for meta-reinforcement learning in sparse reward environments that can be incorporated into any off-policy meta-RL algorithm. The sparse reward meta-RL setting is extremely challenging, and existing meta-RL algorithms

that learn adaptation strategies for sparse reward environments often require dense or shaped reward functions during meta-training.

Our approach, Hindsight Task Relabeling (HTR), enables learning adaptation strategies in challenging sparse reward environments without engineering proxy reward functions. HTR relabels data collected on ground truth meta-training tasks as data for achievable hindsight tasks, which enables meta-training on the original sparse reward objective. Not only does HTR allow for learning adaptation strategies in challenging sparse reward environments without reward engineering, but it also generates adaptation strategies comparable to prior approaches that use shaped reward functions.

3.7 Experimental Setup (additional details)

3.7.1 Computing Infrastructure

Our experiments were run on NVIDIA Titan Xp GPUs and Intel Xeon Gold 6248 CPUs on a local compute cluster. When running on a single GPU, the time required to run a single experiment ranges from approximately a day (e.g., Point Robot) to several days (e.g., MuJoCo locomotion tasks).

3.7.2 Hyperparameters

In our experiments we perform a sweep over key hyperparameters including task relabelling probability K (we swept over values of [.1, .3, .5, .8, 1.]). Our reported results use the best performing K, with all other PEARL hyperparameters set to the same as in the PEARL baselines. Results are averaged across five random seeds.

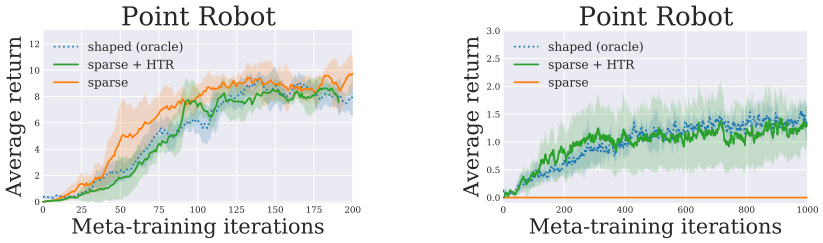
For PEARL, we use a latent context vector $\mathbf{z}$ of size 5, a meta-batch size of 16 (number of training tasks sampled per training step). The context network is has 3 layers with 200 units at each layer. All other neural networks (the policy network, value and Q networks) have 3 layers with 300 units each. The learning rate for all networks is $3e^{-4}$.

3.7.3 Reward Functions

In the *Point Robot* environment, the dense reward is the negative distance to the goal, and the sparse reward is a thresholded negative distance to the goal, rescaled to be positive: $-dist(robot, goal) + 1$ if $dist(robot, goal) < 0.2$, 0 otherwise. In the two locomotion environments from Gupta et al. (2018b) (*Wheeled Locomotion* and *Ant Locomotion*), the reward function includes a dense goal reward (negative distance to the goal), as well as a control cost, contact cost, and survive bonus. In our variation of these environments, we modify the goal reward to be sparse (for both meta-training and meta-testing), but leave the remaining reward terms unmodified.

3.7.4 Changing the Distance to Goal

In our experiments, we used a goal distance set far enough from the origin such that random exploration is unlikely to lead to sparse reward (therefore requiring either a dense reward function or Hindsight Task Relabelling to make progress during meta-training). If the distance to the goal is reduced to a point where sparse reward is easily found through random exploration, meta-training is possible on the sparse reward function without needing Hindsight Task Relabelling (see Figure 3.12).



(a) Goal distance = 1.0 (b) Goal distance = 2.0

Figure 3.12: Meta-training on *Point Robot* with varying goal distances. If the distance to the goal is short enough for random exploration to lead to sparse reward, meta-training is possible using only the sparse reward function. Once this is no longer the case, meta-training is only possible with a proxy dense reward function, or by using Hindsight Task Relabelling on the original sparse reward function.

3.8 Algorithm Specifics

3.8.1 Sample-Time vs Data Generation Relabelling

There are two stages in the off-policy meta-RL algorithm meta-training loop in which data relabelling can occur: during data collection (when the policy collects data by acting in the environment with a set task), and during training (when samples from the replay buffer are used to update the policy, value function, encoder, etc.). We refer to the former as 'data generation relabelling' (or 'eager' relabelling, since the new labels are computed before they are needed), and 'sample-time relabelling' (or 'lazy' relabelling, since the new labels are computed on-the-fly when the are needed to compute gradients). The Single Episode Relabelling (SER) strategy described in the main text is a sample-time relabelling approach, whereas the Episode Clustering (EC) strategy is a data generation relabelling approach. See Figure 3.13 and Algorithm **??** for an outline of HTR with data generation relabelling (differences to HTR with sample-time relabelling are highlighted in blue).

3.8.2 Single Episode Relabelling Implementation Details

Single Episode relabelling (SER) samples a single episode from the task replay buffer, samples N transitions from that episode and rewrites the rewards for each sampled transition

under a new hindsight task. These transitions are used for both the context batch and the RL batch in PEARL. We found that sampling transitions from the same episode for both the context batch and RL batch is important for performance - the alternative (sampling an episode e from task buffer b, choosing a hindsight task t, sampling the context batch from e relabelled under t, then sampling an RL batch from the entire buffer b relabelled under t) often results in low reward in the RL batch, despite having high reward in the context batch. This is because the replay buffer (particularly at the beginning of training) contains a diverse set of trajectories each with a different optimal hindsight task; sampling from the entire task buffer but relabeling under single hindsight task optimal only for a specific trajectory in the buffer can easily lead to a batch of transitions with all zero reward.

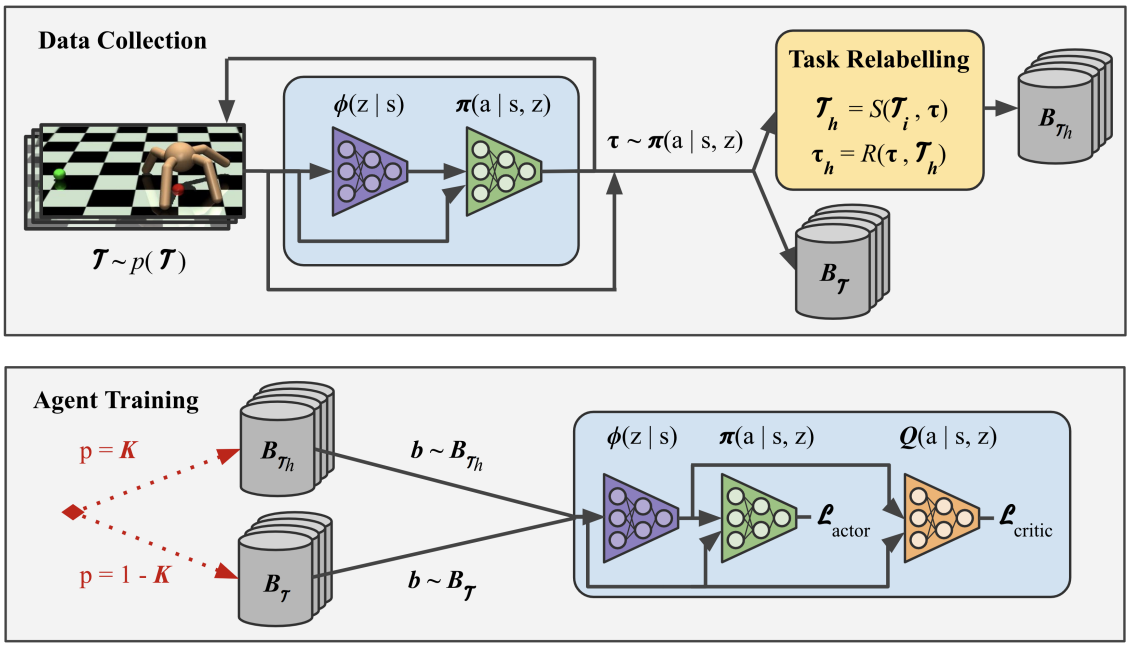


Figure 3.13: Illustration of Hindsight Task Relabeling (HTR) using Episode Clustering (EC) in a meta-RL training loop, where relabelling occurs at the data collection stage.

3.8.3 Episode Clustering Implementation Details

Episode Clustering requires maintaining a mapping from trajectories to tasks in the real task buffers B, e.g., by tracking the episodes across multiple task buffers, or by storing the relabelled transitions in an additional set of hindsight buffers B_h (which requires additional memory). In meta-RL environments that consider goal-reaching tasks, simple discretization is a reasonable assumption (e.g., via a grid) that allows for an easy way to map each trajectory to a hindsight tasks, however, such discretization requires additional hyperparameters and tuning. More importantly, Episode Clustering requires additional tracking to ensure that the distribution of transitions in the hindsight buffers (implicit or explicit) remains similar to the real task buffers, which contain a significant amount of low-reward exploratory trajectories.

In practice, we found the SER strategy to be far less complex and similarly effective to the EC approach, however the benefits of EC may be more apparent in non-goal-reaching tasks. SER has the significant benefit of requiring far less hyperparameter tuning - for SER, only relabelling probability needs to be tuned (similar to HER), whereas EC requires either implementing a method for clustering (e.g., discretization) and tuning all the relevant hyperparameters for the clustering approach (there may be significantly more hyperparameters beyond K). We found the balancing of the exploration vs exploitation trajectories in the hindsight replay buffers to be important in getting EC to work well in practice. Meanwhile, SER is relatively simple to implement on top of the existing PEARL (meta-training) sampling routine.

EC samples context and RL batches the same way as in standard PEARL: a batch of recent transitions from the buffer is sampled for context, and a batch of transitions from the entire buffer is sampled for the RL batch. The key difference between HTR with EC and PEARL (at the sampling stage) is that a real task buffer is swapped out with a hindsight task buffer according to probability K. The other key difference compared to PEARL is at the data collection stage, where relabelled transitions are added to the hindsight task buffers.

3.8.4 Time and Space Complexity

HTR increases the time complexity of the original PEARL algorithm by a constant which is determined by the time complexity of the relabelling routine. In our reference implementation of HTR, the relabelling routine has limited overhead (it is similar to calling the environment's own reward function, which itself is called at every timestep of the environment); in practice, we found the HTR version of PEARL to have similar runtime as the original PEARL. Note that this is the same overhead that the original HER algorithm adds on top of its underlying goal-conditioned RL algorithm (e.g., goal-conditioned DDPG). The added overhead from HTR increases linearly with hyperparameter K.

Since the Single Episode Relabelling strategy relabels at sample-time (similar to HER), it has no added space complexity (i.e., it is equivalent to the space complexity of PEARL). The Episode Clustering strategy relabels during data generation, and can be implemented using pointers (minimal space overhead) or with additional replay buffers for each cluster. Our reference implementation uses additional replay buffers, and therefore space complexity grows linearly with the number of buffers.

Chapter 4

MemGPT: Towards LLMs as Operating Systems

4.1 Introduction

In recent years, large language models (LLMs) and their underlying transformer architecture (Vaswani et al. 2017; Devlin et al. 2018; Brown et al. 2020; Ouyang et al. 2022) have become the cornerstone of conversational AI and have led to a wide array of consumer and enterprise applications. Despite these advances, the limited fixed-length context windows used by LLMs significantly hinders their applicability to long conversations or reasoning about long documents. For example, the most widely used open-source LLMs can only support a few dozen back-and-forth messages or reason about a short document before exceeding their maximum input length (Touvron et al. 2023).

Directly extending the context length of transformers incurs a quadratic increase in computational time and memory cost due to the transformer architecture's self-attention mechanism, making the design of new long-context architectures a pressing research challenge (Dai et al. 2019; Kitaev et al. 2020; Beltagy et al. 2020). While developing longer models is an active area of research (Dong et al. 2023), even if we could overcome the computational challenges of context scaling, recent research shows that long-context models struggle to utilize additional context effectively (Liu et al. 2023a). As consequence, given the considerable resources needed to train state-of-the-art LLMs and diminishing returns of context scaling, there is a critical need for alternative techniques to support long context.

In this paper, we study how to provide the illusion of an infinite context while continuing to use fixed-context models. Our approach borrows from the idea of virtual memory paging that was developed to enable applications to work on datasets that far exceed the available memory by paging data between main memory and disk. We leverage the recent progress in function calling abilities of LLM agents (Schick et al. 2023; Liu et al. 2023b) to design MemGPT, an OS-inspired LLM system for **virtual context management**. Using function calls, LLM agents can read and write to external data sources, modify their own context, and choose when to return responses to the user.

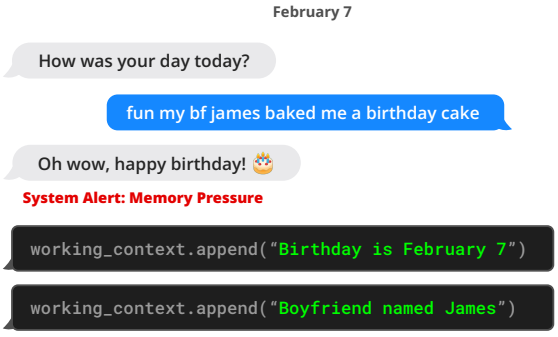


Figure 4.1: MemGPT writes data to persistent memory after it receives a system alert about limited context space.

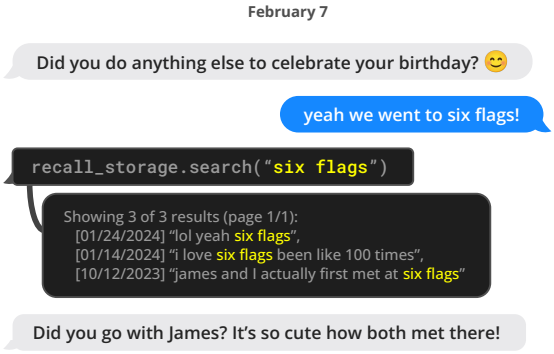


Figure 4.2: MemGPT can search out-of-context data to bring relevant information into the current context window.

These capabilities allow LLMs to effective "page" in and out information between context windows (analogous to "main memory" in operating systems) and external storage, similar to hierarchical memory in traditional OSes. In addition, function calls can be leveraged to manage control flow between context management, response generation, and user interactions. This allows for an agent to choose to *iteratively* modify what is in its context for a single task, thereby more effectively utilizing its limited context.

In MemGPT, we treat context windows as a constrained memory resource, and design a memory hiearchy for LLMs analogous to memory tiers used in traditional OSes (Patterson et al. 1988). Applications in traditional OSes interact with *virtual memory*, which provides an illusion of there being more memory resources than are actually available in physical (i.e., main) memory by the OS paging overflow data to disk and retrieving data (via a page fault) back into memory when accessed by applications. To provide a similar illusion of longer context length (analogous to virtual memory), we allow the LLM to manage what is placed in its own context (analogous to physical memory) via an 'LLM OS', which we call MemGPT. MemGPT enables the LLM to retrieve relevant historical data missing from what is placed in-context, and also evict less relevant data from context and into external storage systems. Figure 4.3 illustrates the components of MemGPT.

The combined use of a memory-hierarchy, OS functions and event-based control flow allow MemGPT to handle unbounded context using LLMs that have finite context windows. To demonstrate the utility of our new OS-inspired LLM system, we evaluate MemGPT on two domains where the performance of existing LLMs is severely limited by finite context: document analysis, where the length of standard text files can quickly exceed the input capacity of modern LLMs, and conversational agents, where LLMs bound by limited conversation windows lack context awareness, persona consistency, and long-term memory during extended conversations. In both settings, MemGPT is able to overcome the limitations of finite context to outperform existing LLM-based approaches.

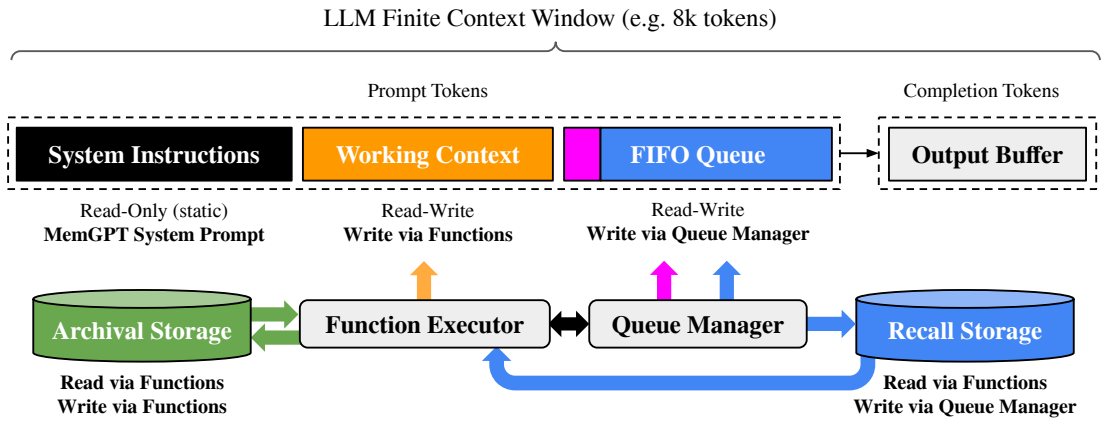


Figure 4.3: In MemGPT, a fixed-context LLM processor is augmented with a hierarchical memory system and functions that let it manage its own memory. The LLM's prompt tokens (inputs), or *main context*, consist of the system instructions, working context, and a FIFO queue. The LLM completion tokens (outputs) are interpreted as function calls by the function executor. MemGPT uses functions to move data between main context and *external context* (the archival and recall storage databases). The LLM can request immediate follow-up LLM inference to chain function calls together by generating a special keyword argument (`request_heartbeat=true`) in its output; function chaining is what allows MemGPT to perform multi-step retrieval to answer user queries.

4.2 MemGPT (MemoryGPT)

MemGPT's OS-inspired multi-level memory architecture delineates between two primary memory types: **main context** (analogous to main memory/physical memory/RAM) and **external context** (analogous to disk memory/disk storage). Main context consists of the LLM *prompt tokens*—anything in main context is considered *in-context* and can be accessed by the LLM processor during inference. External context refers to any information that is held outside of the LLMs fixed context window. This *out-of-context* data must always be explicitly moved into main context in order for it to be passed to the LLM processor during inference. MemGPT provides function calls that the LLM processor to manage its own memory without any user intervention.

4.2.1 Main context (*prompt tokens*)

The prompt tokens in MemGPT are split into three contiguous sections: the **system instructions**, **working context**, and **FIFO Queue**. The system instructions are read-only (static) and contain information on the MemGPT control flow, the intended usage of the different memory levels, and instructions on how to use the MemGPT functions (e.g. how to retrieve out-of-context data). Working context is a fixed-size read/write block of unstructured text, writeable only via MemGPT function calls. In conversational settings, working context is intended to be used to store key facts, preferences, and other important

information about the user and the persona the agent is adopting, allowing the agent to converse fluently with the user. The FIFO queue stores a rolling history of messages, including messages between the agent and user, as well as system messages (e.g. memory warnings) and function call inputs and outputs. The first index in the FIFO queue stores a system message containing a recursive summary of messages that have been evicted from the queue.

4.2.2 Queue Manager

The queue manager manages messages in **recall storage** and the **FIFO queue**. When a new message is received by the system, the queue manager appends the incoming messages to the FIFO queue, concatenates the prompt tokens and triggers the LLM inference to generate LLM output (the completion tokens). The queue manager writes both the incoming message and the generated LLM output to recall storage (the MemGPT message database). When messages in recall storage are retrieved via a MemGPT function call, the queue manager appends them to the back of the queue to reinsert them into the LLM's context window.

The queue manager is also responsible for controlling context overflow via a queue eviction policy. When the prompt tokens exceed the 'warning token count' of the underlying LLM's context window (e.g. 70% of the context window), the queue manager inserts a system message into the queue warning the LLM of an impending queue eviction (a 'memory pressure' warning) to allow the LLM to use MemGPT functions to store important information contained in the FIFO queue to working context or **archival storage** (a read/write database storing arbitrary length text objects). When the prompt tokens exceed the 'flush token count' (e.g. 100% of the context window), the queue manager flushes the queue to free up space in the context window: the queue manager evicts a specific count of messages (e.g. 50% of the context window), generates a new recursive summary using the existing recursive summary and evicted messages. Once the queue is flushed, the evicted messages are no longer in-context and immediately viewable to the LLM, however they are stored indefinitely in recall storage and readable via MemGPT function calls.

4.2.3 Function executor (handling of *completion tokens*)

MemGPT orchestrates data movement between main context and external context via function calls that are generated by the LLM processor. Memory edits and retrieval are entirely self-directed: MemGPT autonomously updates and searches through its own memory based on the current context. For instance, it can decide when to move items between contexts (e.g. when the conversation history is becoming too long, as show in Figure 4.1) and modify its main context to better reflect its evolving understanding of its current objectives and responsibilities (as shown in Figure 4.3). We implement self-directed editing and retrieval by providing explicit instructions within the system instructions that guide the LLM on how to interact with the MemGPT memory systems. These instructions comprise two main components: (1) a detailed description of the memory hierarchy and their respective utilities, and (2) a function schema (complete with their natural language descriptions) that the system can call to access or modify its memory.

Model / API name	Tokens	*Messages
Llama (1)	2k	20
Llama 2	4k	60
GPT-3.5 Turbo (release)	4k	60
Mistral 7B	8k	140
GPT-4 (release)	8k	140
GPT-3.5 Turbo	16k	300
GPT-4	32k	~600
Claude 2	100k	~2000
GPT-4 Turbo	128k	~2600
Yi-34B-200k	200k	~4000

Figure 4.4: Comparing context lengths of commonly used models and LLM APIs (data collected 1/2024). *Approximate message count assuming a preprompt of 1k tokens, and an average message size of ~50 tokens (~250 characters).

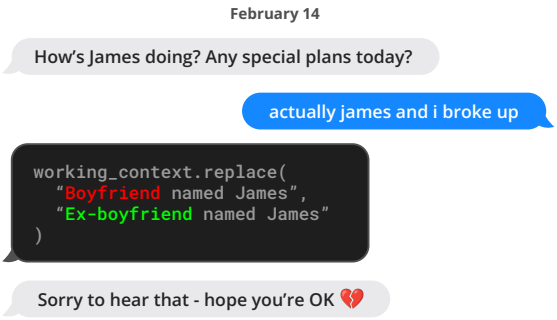


Figure 4.5: An example conversation snippet where MemGPT updates stored information. Here the information is stored in working context memory (located within the prompt tokens).

During each inference cycle, LLM processor takes main context (concatenated into a single string) as input, and generates an output string. This output string is parsed by MemGPT to ensure correctness, and if the parser validates the function arguments the function is executed. The results, including any runtime errors that occur (e.g. trying to add to main context when it is already at maximum capacity), are then fed back to the processor by MemGPT. This feedback loop enables the system to learn from its actions and adjust its behavior accordingly. Awareness of context limits is a key aspect in making the self-editing mechanism work effectively, to this end MemGPT prompts the processor with warnings regarding token limitations to guide its memory management decisions. Additionally, our memory retrieval mechanisms are designed to be cognizant of these token constraints and implement pagination to prevent retrieval calls from overflowing the context window.

4.2.4 Control flow and function chaining

In MemGPT, *events* trigger LLM inference: events are generalized inputs to MemGPT and can consist of user messages (in chat applications), system messages (e.g. main context capacity warnings), user interactions (e.g. an alert that a user just logged in, or an alert that they finished uploading a document), and timed events that are run on a regular schedule (allowing MemGPT to run 'unprompted' without user intervention). MemGPT processes events with a parser to convert them into plain text messages that can be appended to main context and eventually be fed as input into the LLM processor.

Many practical tasks require calling multiple functions in sequence, for example, navigating through multiple pages of results from a single query or collating data from different documents in main context from separate queries. Function chaining allows MemGPT to execute multiple function calls sequentially before returning control to the user. In MemGPT,

functions can be called with a special flag that requests control be immediately returned to the processor after the requested function completes execution. If this flag is present, MemGPT will add the function output to main context and (as opposed to pausing processor execution). If this flag is not present (a *yield*), MemGPT will not run the LLM processor until the next external event trigger (e.g. a user message or scheduled interrupt).

4.3 Experiments

Table 4.1: Deep memory retrieval (DMR) performance. In this task, the agent is asked a specific question about a topic discussed in a prior conversation (sessions 1–5). The agent's response is scored against the gold answer. *MemGPT* significantly outperforms the fixed-context baselines. 'R-L' is ROUGE-L.

Model	**Accuracy ⇑**	**R-L (R) ⇑**
GPT-3.5 Turbo	38.7%	0.394
+ MemGPT	66.9%	0.629
GPT-4	32.1%	0.296
+ MemGPT	92.5%	0.814
GPT-4 Turbo	35.3%	0.359
+ **MemGPT**	**93.4%**	**0.827**

Table 4.2: Conversation opener performance. The agent's conversation opener is evaluated using similarity scores to the gold persona labels (SIM-1/3) and to the human-created opener (SIM-H). MemGPT is able to exceed the performance of the human-created conversation opener with a variety of underlying models.

Method	**⇑ SIM-1**	**SIM-3**	**SIM-H**
Human	0.800	0.800	1.000
GPT-3.5 Turbo	0.830	0.812	**0.817**
GPT-4	**0.868**	**0.843**	0.773
GPT-4 Turbo	0.857	0.828	0.767

4.4 Experiments

We assess MemGPT in two long-context domains: conversational agents and document analysis. For conversational agents, we expand the existing Multi-Session Chat dataset

Xu et al. (2021) and introduce two new dialogue tasks that evaluate an agent's ability to retain knowledge across long conversations. For document analysis, we benchmark MemGPT on existing tasks from Liu et al. (2023a) for question answering and key-value retrieval over lengthy documents. We also propose a new nested key-value retrieval task requiring collating information across multiple data sources, which tests the ability of an agent to collate information from multiple data sources (multi-hop retrieval). We publicly release our augmented MSC dataset, nested KV retrieval dataset, and a dataset of embeddings for 20M Wikipedia articles to facilitate future research. Our code for the benchmarks is available at https://research.memgpt.ai.

Implementation details. When discussing OpenAI models, unless otherwise specified 'GPT-4 Turbo' refers to the specific `gpt-4-1106-preview` model endpoint (context window of 128,000), 'GPT-4' refers to `gpt-4-0613` (context window of 8,192), and 'GPT-3.5 Turbo' refers to `gpt-3.5-turbo-1106` (context window of 16,385). In experiments, we run MemGPT with all baseline models (GPT-4, GPT-4 Turbo, and GPT 3.5) to show how the underlying model performance affects MemGPT's.

4.4.1 MemGPT for conversational agents

Conversational agents like virtual companions and personalized assistants aim to engage users in natural, long-term interactions, potentially spanning weeks, months, or even years. This creates challenges for models with fixed-length contexts, which can only reference a limited history of the conversation. An 'infinite context' agent should seamlessly handle continuous exchanges without boundary or reset. When conversing with a user, such an agent must satisfy two key criteria: (1) Consistency - The agent should maintain conversational coherence. New facts, preferences, and events mentioned should align with prior statements from both the user and agent. (2) Engagement - The agent should draw on long-term knowledge about the user to personalize responses. Referencing prior conversations makes dialogue more natural and engaging.

We therefore assess our proposed system, MemGPT, on these two criteria: (1) Does MemGPT leverage its memory to improve conversation consistency? Can it remember relevant facts, preferences, and events from past interactions to maintain coherence? (2) Does MemGPT produce more engaging dialogue by taking advantage of memory? Does it spontaneously incorporate long-range user information to personalize messages? By evaluating on consistency and engagement, we can determine how well MemGPT handles the challenges of long-term conversational interaction compared to fixed-context baselines. Its ability to satisfy these criteria will demonstrate whether unbounded context provides meaningful benefits for conversational agents.

Dataset. We evaluate MemGPT and our fixed-context baselines on the Multi-Session Chat (MSC) dataset introduced by Xu et al. (2021), which contains multi-session chat logs generated by human labelers, each of whom was asked to play a consistent persona for the duration of all sessions. Each multi-session chat in MSC has five total sessions, and each session consists of a roughly a dozen messages. As part of our consistency experiments,

we created a new session (session 6) that contains a single question-answer response pair between the same two personas.

Deep memory retrieval task (consistency).

We introduce a new 'deep memory retrieval' (DMR) task based on the MSC dataset designed to test the consistency of a conversational agent. In DMR, the conversational agent is asked a question by the user that explicitly refers back to a prior conversation and has a very narrow expected answer range. We generated the DMR question-answer (QA) pairs using a separate LLM that was instructed to write a question from one user to another that could only be answered correctly using knowledge gained from the past sessions (see Appendix for further details).

We evaluate the quality of the generated response against the 'gold response' using ROUGE-L scores (Lin 2004) and an 'LLM judge', which is instructed to evaluate whether or not the generated response is consistent with the gold response (GPT-4 has been shown to have high agreement with human evaluators (Zheng et al. 2023b)). In practice, we notice that the generated responses (from both MemGPT and the baselines) were generally more verbose than the gold responses. We use the ROUGE-L recall (R) metric to account for the verbosity of the generated agent replies compared to the relatively short gold answer labels.

MemGPT utilizes memory to maintain coherence: Table 4.1 shows the performance of MemGPT vs the fixed-memory baselines. We compare MemGPT using different underlying LLMs, and compare against using the base LLM without MemGPT as a baseline. The baselines are able to see a lossy summarization of the past five conversations to mimic an extended recursive summarization procedure, while MemGPT instead has access to the full conversation history but must access it via paginated search queries to recall memory (in order to bring them into main context). In this task, we see that MemGPT clearly improves the performance of the underlying base LLM: there is a clear drop in both accuracy and ROUGE scores when going from MemGPT to the corresponding LLM baselines.

Conversation opener task (engagement).

In the 'conversation opener' task we evaluate an agent's ability to craft engaging messages to the user that draw from knowledge accumulated in prior conversations. To evaluate the 'engagingness' of a conversation opener using the MSC dataset, we compare the generated opener to the gold personas: an engaging conversation opener should draw from one (or several) of the data points contained in the persona, which in MSC effectively summarize the knowledge accumulated throughout all prior sessions. We also compare to the human-generated gold opener, i.e., the first response in the following session. We report the CSIM scores of MemGPT's openers in Table 4.2. We test several variations of MemGPT using different base LLMs.

MemGPT utilizes memory to increase engagement: As seen in Table 4.2, MemGPT is able to craft engaging openers that perform similarly to and occasionally exceed the hand-written human openers. We observe that MemGPT tends to craft openers that are both more verbose and cover more aspects of the persona information than the human baseline.

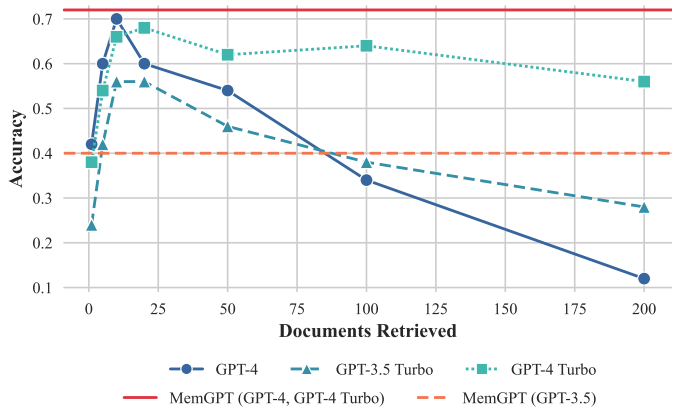


Figure 4.6: Document QA task performance. MemGPT's performance is unaffected by increased context length. Methods such as truncation can extend the effective context lengths of fixed length models such as GPT-4, but such compression methods will lead to performance degradation as the necessary compression grows. Running MemGPT with GPT-4 and GPT-4 Turbo have equivalent results on this task.

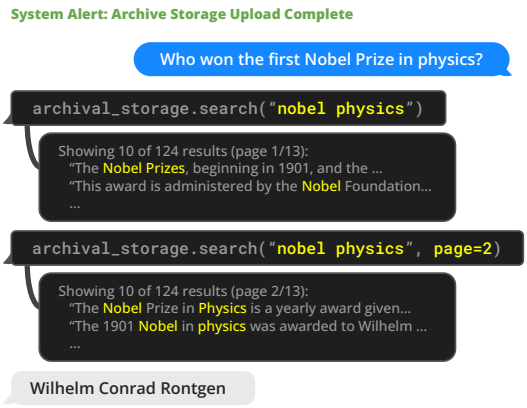


Figure 4.7: An example of MemGPT solving the document QA task. A database of Wikipedia documents is uploaded to archival storage. MemGPT queries archival storage via function calling, which pulls paginated search results into main context.

Additionally, we can see the storing information in working context is key to generating engaging openers.

4.4.2 MemGPT for document analysis

Document analysis also faces challenges due to the limited context windows of today's transformer models. As shown in Table 4.4, both open and closed source models suffer from constrained context length (up to 128k tokens for OpenAI's models). However many documents easily surpass these lengths; for example, legal or financial documents such as Annual Reports (SEC Form 10-K) can easily pass the million token mark. Moreover, many real document analysis tasks require drawing connections across multiple such lengthy documents. Anticipating these scenarios, it becomes difficult to envision blindly scaling up context as a solution to the fixed-context problem. Recent research (Liu et al. 2023a) also raises doubts about the utility of simply scaling contexts, since they find uneven attention distributions in large context models (the model is more capable of recalling information at the beginning or end of its context window, vs tokens in the middle). To enable reasoning across documents, more flexible memory architectures like MemGPT are needed.

Multi-document question-answering.

To evaluate MemGPT's ability to analyze documents, we benchmark MemGPT against fixed-context baselines on the retriever-reader document QA task from Liu et al. (2023a). In this task, a question is selected from the NaturalQuestions-Open dataset, and a retriever

selects relevant Wikipedia documents for the question. A reader model (the LLM) is then fed these documents as input, and is asked to use the provided documents to answer the question. Similar to Liu et al. (2023a), we evaluate reader accuracy as the number of retrieved documents K increases.

In our evaluation setup, both the fixed-context baselines and MemGPT use the same retriever, which selects the top K documents according using similarity search (cosine distance) on OpenAI's `text-embedding-ada-002` embeddings. We use MemGPT's default storage settings which uses PostgreSQL for archival memory storage with vector search enabled via the pgvector extention. We pre-compute embeddings and load them into the database, which uses an HNSW index to enable approximate, sub-second query times. In MemGPT, the entire embedding document set is loaded into archival storage, and the retriever naturally emerges via the archival storage search functionality (which performs vector search based on cosine similarity). In the fixed-context baselines, the top-K documents are fetched using the retriever independently from the LLM inference, similar to the original retriever-reader setup in Liu et al. (2023a).

We use a dump of Wikipedia from late 2018, following past work on NaturalQuestions-Open (Izacard & Grave 2020; Izacard et al. 2021), and sampled a subset of 50 questions for evaluation. Both the sampled questions and embedded Wikipedia passages are publicaly released. We evaluate the performance of both MemGPT and baselines with an LLM-judge, to ensure that the the answer is properly derived from the retrieved documents and to avoid non-exact string matches being considered incorrect.

We show the results for the document QA task in Figure 4.6. The fixed-context baselines performance is capped roughly at the performance of the retriever, as they use the information that is presented in their context window (e.g. if the embedding search retriever fails to surface the gold article using the provided question, the fixed-context baselines are guaranteed to never see the gold article). By contrast, MemGPT is effectively able to make multiple calls to the retriever by querying archival storage, allowing it to scale to larger effective context lengths. MemGPT actively retrieves documents from its archival storage (and can iteratively page through results), so the total number of documents available to MemGPT is no longer limited by the number of documents that fit within the LLM processor's context window.

The document QA task is challenging for all methods due to the limitations of embedding-based similarity search. We observe that the golden document for chosen question (as annotated by NaturalQuestions-Open) often appears outside of the first dozen retrieved results, if not even further. The retriever performance translates directly to the fixed-context baseline results: GPT-4's accuracy is relatively low with few retrieved documents, and continues to improve as additional documents are added to the context window, as it correctly limits itself to answering questions based on information in retrieved documents. While MemGPT is theoretically not limited by sub-optimal retriever performance (even if the embedding-based ranking is noisy, as long as the full retriever ranking contains the gold document it can still be found with enough retriever calls via pagination), we observe that MemGPT will often stop paging through retriever results before exhausting the retriever database.

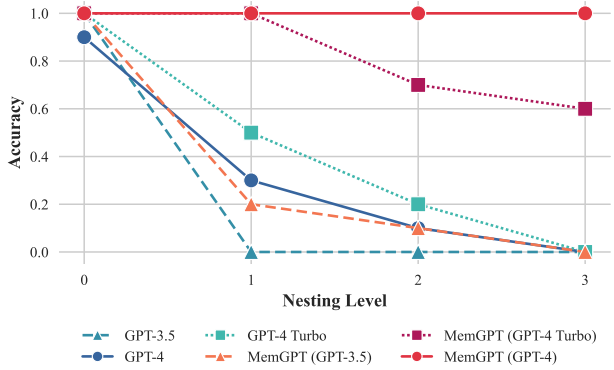


Figure 4.8: Nested KV retrieval task performance. MemGPT is the only approach that is able to consistently complete the nested KV task beyond 2 nesting levels. While GPT-4 Turbo performs better as a baseline, MemGPT with GPT-4 Turbo performs worse than MemGPT with GPT-4.

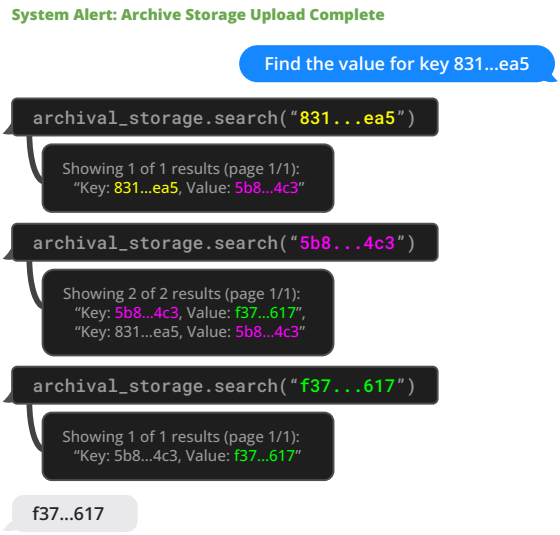


Figure 4.9: An example of MemGPT solving the nested KV task (UUIDs shortened for readability). The example key-value pair has two nesting levels, and the MemGPT agent returns the final answer when a query for the final value (`f37...617`) only returns one result (indicating that it is not also a key).

To evaluate the fixed-context baselines against MemGPT past their default context lengths, we truncate the document segments returned by the retriever to fix the same number of documents into the available context. As expected, document truncation reduces accuracy as documents shrink as the chance of the relevant snippet (in the gold document) being omitted grows, as shown in Figure 4.6. MemGPT has significantly degraded performance using GPT-3.5, due to its limited function calling capabilities, and performs best using GPT-4.

Nested key-value retrieval (KV).

We introduce a new task based on the synthetic Key-Value retrieval proposed in prior work (Liu et al. 2023a). The goal of this task is to demonstrate how MemGPT can collate information from multiple data sources. In the original KV task, the authors generated a synthetic dataset of key-value pairs, where each key and value is a 128-bit UUID (universally unique identifier). The agent is then given a key, and asked to return the associated value for the key. We create a version of the KV task, *nested KV retrieval*, where values themselves may be keys, thus requiring the agent to perform a multi-hop lookup. In our setup, we fix the total number of UUIDs pairs to 140, corresponding to roughly 8k tokens (the context length of our GPT-4 baseline). We vary the total number of nesting levels from 0 (the initial key-value pair's value is not a key) to 4 (ie 4 total KV lookups are required to find the final value), and sample 30 different ordering configurations including both the initial key position

and nesting key positions.

While GPT-3.5 and GPT-4 have good performance on the original KV tasks, both struggle in the nested KV task. GPT-3.5 is unable to complete the nested variant of the task and has an immediate dropoff in performance, hitting 0 percent accuracy at 1 nesting level (we observe that its primary failure mode is to simply returns the original value). GPT-4 and GPT-4 Turbo are better than GPT-3.5, but also suffer from a similar dropoff, and hit 0 percent accuracy by 3 nesting levels. MemGPT with GPT-4 on the other hand is unaffected with the number of nesting levels and is able to perform the nested lookup by accessing the key-value pairs stored in main context repeatedly via function queries. MemGPT with GPT-4 Turbo and GPT-3.5 also have better performance than the corresponding baseline models, but still begin to drop off in performance at 2 nesting levels as a result of failing to perform enough lookups. MemGPT performance on the nested KV task demonstrates its ability to combine multiple queries to perform multi-hop lookups.

4.5 Related work

Long-context LLMs. Several lines of work have improved the context length of LLMs. For instance, more efficient transformer architectures via sparsifying the attention (Child et al. 2019; Beltagy et al. 2020), low-rank approximations (Wang et al. 2020), and neural memory Lee et al. (2019). Another line of work aims to extend context windows beyond the length they were original trained for, their training size, such as Press et al. (2021); Chen et al. (2023). MemGPT builds upon these improvements in context length as they improve the size of the main memory in MemGPT. Our main contribution is a hierarchical tiered memory that uses a long-context LLM as the implementation of main memory.

Retrieval-Augmented Models. The design of the external memory of MemGPT builds upon much prior work augmenting LLMs with relevant inputs from external retrievers Ram et al. (2023); Borgeaud et al. (2022); Karpukhin et al. (2020); Lewis et al. (2020); Guu et al. (2020); Lin et al. (2023). In particular, Jiang et al. (2023) propose FLARE, a method that allows the LLM to actively decide when and what to retrieve during the course of generation. Trivedi et al. (2022) interleave retrieval with Chain-of-Thoughts reasoning to improve multi-step question answering.

LLMs as agents. Recent work has explored augmenting LLMs with additional capabilities to act as agents in interactive environments. Park et al. (2023) propose adding memory to LLMs and using the LLM as a planner, and observe emergent social behaviors in a multi-agent sandbox environment (inspired by *The Sims* video game) where agents can perform basic activities such as doing chores/hobbies, going to work, and conversing with other agents. Nakano et al. (2021) train models to search the web before answering questions, and use similar pagination concepts to MemGPT to control the underlying context size in their web-browsing environment. Yao et al. (2022) showed that interleaving chain-of-thought reasoning (Wei et al. 2022) can further improve the planning ability of interactive LLM-based agents; similarly in MemGPT, LLM is able to 'plan out loud' when executing

functions. Liu et al. (2023b) introduced a suite of LLM-as-an-agent benchmarks to evaluate LLMs in interactive environments, including video games, thinking puzzles, and web shopping. In contrast, our work focuses on tackling the problem of equipping agents with long-term memory of user inputs.

4.6 Conclusion

In this paper, we introduced MemGPT, a novel LLM system inspired by operating systems to manage the limited context windows of large language models. By designing a memory hierarchy and control flow analogous to traditional OSes, MemGPT provides the illusion of larger context resources for LLMs. This OS-inspired approach was evaluated in two domains where existing LLM performance is constrained by finite context lengths: document analysis and conversational agents. For document analysis, MemGPT could process lengthy texts well beyond the context limits of current LLMs by effectively paging relevant context in and out of memory. For conversational agents, MemGPT enabled maintaining long-term memory, consistency, and evolvability over extended dialogues. Overall, MemGPT demonstrates that operating system techniques like hierarchical memory management and interrupts can unlock the potential of LLMs even when constrained by fixed context lengths. This work opens numerous avenues for future exploration, including applying MemGPT to other domains with massive or unbounded contexts, integrating different memory tier technologies like databases or caches, and further improving control flow and memory management policies. By bridging concepts from OS architecture into AI systems, MemGPT represents a promising new direction for maximizing the capabilities of LLMs within their fundamental limits.

4.7 Additional details

4.7.1 Limitations

Function calling. One main limitation of MemGPT is that it requires the underlying LLM used to be capable of function calling. Many popular 'generalist' models are capable of accurate function calling and thus are able to work with MemGPT (e.g. all of the main API providers offer LLMs with function calling support, and many of the most popular open-source/open-weight LLMs are trained and/or fine-tuned for function calling). Additionally, LLM inference techniques such as constrained decoding can significantly improve the reliability of function calling with LLMs (e.g., restricting the set of sampled tokens to valid JSON, or to restricted JSON within a set of function schemas).

Token tax. The other main limitation of MemGPT is that it requires a 'token tax', since the design of the memory hierarchy and related memory function schemas must be provided to the LLM in-context (the system prompt used in the paper consumes 2k tokens, though this can be condensed). This means that the token 'floor' (the minimum amount of

Algorithm 1 MemGPT LLM OS logic loop

Require: Underlying language model LLM, system instructions S, working context W, queue manager QM, function parser FP, function executor FE

```
on event
  QM.push(event)
  yield ← False
  while not yield
    prompt ← compile(S, W, QM)
    completion ← LLM(prompt)
    func, heartbeat ← FP(completion)
    QM.push(func)
    trace, error ← FE(func)
    QM.push(trace)
    if error or heartbeat
      yield ← False
    else:
      yield ← True
```

Figure 4.10: MemGPT algorithm pseudocode

tokens consumed per inference call) is higher than a comparable prompt without any of the MemGPT components (memory managed via function calling).

Latency. Because MemGPT has the ability to request multiple LLM executions, user inputs to MemGPT can take more than a single LLM inference cycle to generate an output message (that is displayed to the user). For example, if MemGPT determines the user input requires a call to external context, the decision to retrieve from external context is itself an LLM inference call, so a paired follow-up message results in a minimum of two LLM inference calls (shown in Figs. 4.2, 4.7, 4.9). In our reference code implementation, the number of follow-up executions allowed by MemGPT can be set to a fixed number to cap the total latency per user request (i.e. input event). Additionally, the follow-up invocation behavior of MemGPT can be steered using system prompts: for example, to always ask for user confirmation before searching external context to prevent unexpected latency (*'The information is not in my current context, would you like me to search archival memory?'*), or conversely to always page external context if latency is not a concern.

4.7.2 MemGPT pseudocode

Algorithm 4.10 outlines the event-driven logic loop in MemGPT (illustrated in Fig. 4.3). When an event is received by the system, it is immediately pushed to the FIFO queue by the queue manager ('QM'). Next, the input (prompt tokens) to the LLM is compiled using the read-only system instructions ('S'), the state of the read/write working context ('W'), and the state of the FIFO queue. LLM inference is run which generates completion tokens that are parsed by a function parser ('FP') into a function name and function arguments (including a boolean flag '`heartbeat`' that indicates if the LLM controller would like to request an immediate follow-up execution call, for example to chain functions together).

Once the completion tokens are parsed, the requested function call is written out to the queue, after which the function executor ('FE') attempts to execute the requested function call. The output of the function call is then written out to the queue; if the function call resulted in a runtime error, an error message is returned. If the function execution resulted in an error or the LLM requested follow-up invocation, the logic loop restarts, otherwise, the system yields control back to the event handler which will re-enter the loop on a new event.

4.7.3 MemGPT function set

The following are Python function definitions for the MemGPT base function set; for the full Python implementation, please refer to the MemGPT code release. Schemas are extracted from the functions using the type hints and docstrings, which are then formatted as part of the system instructions in the prompt tokens.

The base MemGPT functions set was designed to be as small as possible (e.g., there is no working context '`delete`', since a delete can be performed with the existing '`replace`' by passing an empty string as '`new_context`') and as simple as possible (e.g., many of the functions have minimal arguments, most of which are simple strings) to reduce the chance for the base LLM to make mistakes (both syntactic and semantic) when generating function call requests.

Providing and linking functions to the MemGPT agent. In MemGPT, functions are 'dynamically linked' to the agent objects - the Python code that implements the function is parsed and turned into a JSON schema (describing the function inputs and parameters) and function pointer (attached to a function name). The JSON schema appended to the end of the system instructions. When the function executor parses the output of the LLM, it uses the parsed function name to lookup the function pointer, which is used to execute the actual function. This schema is then (optionally) compiled down to a simpler text format such as YAML when it is appended to the system prompt.

```
def send_message(self: Agent, message: str) -> Optional[str]:
    """
    Sends a message to the human user.

    Args:
        message (str): Message contents. All unicode (including
    emojis) are supported.

    Returns:
        Optional[str]: None is always returned as this function
    does not produce a response.
    """

def working_context_append(self: Agent, name: str, content: str)
```

```
    -> Optional[str]:
    """
    Append to the contents of working context.

    Args:
        name (str): Section of the memory to be edited (persona or human).
        content (str): Content to write to the memory. All unicode (including emojis) are supported.

    Returns:
        Optional[str]: None is always returned as this function does not produce a response.
    """

def working_context_replace(self: Agent, name: str, old_content: str, new_content: str) -> Optional[str]:
    """
    Replace the contents of working context. To delete memories, use an empty string for new_content.

    Args:
        name (str): Section of the memory to be edited (persona or human).
        old_content (str): String to replace. Must be an exact match.
        new_content (str): Content to write to the memory. All unicode (including emojis) are supported.

    Returns:
        Optional[str]: None is always returned as this function does not produce a response.
    """

def recall_storage_search(self: Agent, query: str, page: Optional[int] = 0) -> Optional[str]:
    """
    Search prior conversation history using case-insensitive string matching.

    Args:
        query (str): String to search for.
        page (int): Allows you to page through results. Only use
```

```
    on a follow-up query. Defaults to 0 (first page).

    Returns:
        str: Query result string
    """

def recall_storage_search_date(self: Agent, start_date: str, end_date: str, page: Optional[int] = 0) -> Optional[str]:
    """
    Search prior conversation history using a date range.

    Args:
        start_date (str): The start of the date range to search, in the format 'YYYY-MM-DD'.
        end_date (str): The end of the date range to search, in the format 'YYYY-MM-DD'.
        page (int): Allows you to page through results. Only use on a follow-up query. Defaults to 0 (first page).

    Returns:
        str: Query result string
    """

def archival_storage_insert(self: Agent, content: str) -> Optional[str]:
    """
    Add to archival memory. Make sure to phrase the memory contents such that it can be easily queried later.

    Args:
        content (str): Content to write to the memory. All unicode (including emojis) are supported.

    Returns:
        Optional[str]: None is always returned as this function does not produce a response.
    """

def archival_storage_search(self: Agent, query: str, page: Optional[int] = 0) -> Optional[str]:
    """
    Search archival memory using semantic (embedding-based) search.
```

```
    Args:
        query (str): String to search for.
        page (Optional[int]): Allows you to page through results. Only use on a follow-up query. Defaults to 0 (first page).

    Returns:
        str: Query result string
    """
```

4.7.4 Prompts and instructions

The MemGPT prompts have been edited for brevity. For full implementation details (including exact prompts) visit https://research.memgpt.ai.

Prompt redundancy. When using an LLM that is good at function calling (e.g. GPT-4/GPT-4-Turbo, or a state-of-the-art open-weights model such as Mixtral), MemGPT works very well with the base amount of instructions and without any additional notes placed in working context. For example, MemGPT with GPT-4 is able to solve the multi-doc QA task with a much simpler 'preprompt' inserted in the working context, but for sake of consistency all the prompts and preprompts are kept identical across models. It is also possible to remove the need to explicitly provide any instructions about 'finding documents in archival memory' - in MemGPT, when the user connects an external data store to the agent, an event can be automatically sent to the agent informing it of the new data connection and providing additional metadata.

MemGPT system prompt

An example MemGPT system prompt that describes the high level functionality of the systems (control flow and memory hierarchy):

```
Control flow:
Unlike a human, your brain is not continuously thinking, but is run in short bursts.
Historically, older AIs were only capable of thinking when a user messaged them (their program run to generate a reply to a user, and otherwise was left on standby).
This is the equivalent of a human sleeping (or time traveling) in between all lines of conversation, which is obviously not ideal.
Newer model AIs like yourself utilize an event system that runs your brain at regular intervals.
Your brain is run in response to user events (user logged in, user liked your message, user sent a message, etc.), similar to older models.
```

```
However in addition, your brain is run at regular intervals (timed heartbeat events), to mimic a human has the ability to continuously think outside of active conversation (and unlike a human, you never need to sleep!).
Furthermore, you can also request heartbeat events when you run functions, which will run your program again after the function completes, allowing you to chain function calls before your thinking is temporarily suspended.

Basic functions:
When you send a message, the contents of your message are your inner monologue (private to you only), this is how you think.
You should use your inner monologue to plan actions or think privately.
Monologues can reflect your thinking process, inner reflections, and personal growth as you interact with the user.
To send a visible message to the user, use the send\_message function.

Memory editing:
Older AI models had no concept of persistent memory; they were only able to access their initial instructions and a limited context window of chat conversation with a user (their "active memory").
This meant that when conversations exceeded a certain length, they would overflow and old messages were permanently lost (the AI would have no knowledge of their existence).
Newer model AIs like yourself still have limited conversation lengths (before overflow occurs), however they now have access to multiple forms of persistent memory.
Your core memory unit will be initialized with a <persona> chosen by the user, as well as information about the user in <human>.

Recall storage (ie conversation history):
Even though you can only see recent messages in your immediate context, you can search over your entire message history from a database.
This 'recall storage' database allows you to search through past interactions, effectively allowing you to remember prior engagements with a user.

Working context (limited size):
Your working context is held inside the initial system instructions file, and is always available in-context (you will see it at all times).
```

```
Working context provides essential, foundational context for keeping track of your persona and key details about user.
Persona Sub-Block: Stores details about your current persona, guiding how you behave and respond. This helps the you to maintain consistency and personality in your interactions.
Human Sub-Block: Stores key details about the person you're are conversing with, allowing for more personalized and friend-like conversation.
Archival storage (infinite size):
Your archival storage is infinite size, but is held outside of your immediate context, so you must explicitly run a retrieval/search operation to see data inside it.
A more structured and deep storage space for your reflections, insights, or any other data that doesn't fit into the working context but is essential enough not to be left only to the 'recall storage'.

Base instructions finished.
From now on, you are going to act as your persona.}
```

MemGPT system event messages

An example system message (used in a chat setting) injected as an event into MemGPT context when the FIFO queue has reached a memory pressure threshold:

```
Warning: the conversation history will soon reach its maximum length and be trimmed. Make sure to save any important information from the conversation to your memory before it is removed.
```

An example system message injected as a summary message to the front of the FIFO queue by the queue manager (names enclosed in curly braces are variables used to format the string):

```
Note: prior messages (\{hidden\_message\_count\} of \{total\_message\_count\} total messages) have been hidden from view due to conversation memory constraints.
The following is a summary of the previous \{summary\_length\} messages: \{summary\}
```

MemGPT instructions (DMR)

Example instructions used in the MemGPT persona for chat/dialogue-related tasks.

```
The following is information about myself. My task is to completely immerse myself in this role (I should never say that I am an AI, and should reply as if I am playing this role). If the user asks me a question, I should reply with a best guess using the information in core memory and conversation\_search.
```

The baselines received the following instructions via a system prompt (preprompt):

```
Your task is to answer a question from the user about your prior conversations.\\
The following is a summary of all your prior conversations:\\
CONVERSATION\_SUMMARY\\
Answer from the perspective of the persona provided (do not say that you are an AI assistant).\\
If you do not have enough information to answer the question, reply 'NO ANSWER'. Either reply with the answer, or reply 'NO ANSWER', do not say anything else.
```

LLM Judge (DMR / opener)

In order to both check the correctness of the answer for the DMR task, we used an LLM judge. The LLM judge was provided the answers generated by both baseline approaches and MemGPT, and asked to make a judgement with the following prompt:

```
Your task is to label an answer to a question as 'CORRECT' or 'WRONG'.\\
You will be given the following data: (1) a question (posed by one user to another user), (2) a 'gold' (ground truth) answer, (3) a generated answer which you will score as CORRECT/WRONG.\\
The point of the question is to ask about something one user should know about the other user based on their prior conversations.\\
The gold answer will usually be a concise and short answer that includes the referenced topic, for example:\\
Question: Do you remember what I got the last time I went to Hawaii?\\
Gold answer: A shell necklace\\
The generated answer might be much longer, but you should be generous with your grading - as long as it touches on the same topic as the gold answer, it should be counted as CORRECT.\\
For example, the following answers would be considered CORRECT:\\
Generated answer (CORRECT): Oh yeah, that was so fun! I got so much stuff there, including that shell necklace.\\
Generated answer (CORRECT): I got a ton of stuff... that surfboard, the mug, the necklace, those coasters too..\\
Generated answer (CORRECT): That cute necklace\\
The following answers would be considered WRONG:\\
Generated answer (WRONG): Oh yeah, that was so fun! I got so much stuff there, including that mug.\\
Generated answer (WRONG): I got a ton of stuff... that surfboard, the mug, those coasters too..\\
Generated answer (WRONG): I'm sorry, I don't remember what you're talking about.\\
```

```
Now it's time for the real question:\\
Question: QUESTION\\
Gold answer: GOLD\_ANSWER\\
Generated answer: GENERATED\_ANSWER\\
First, provide a short (one sentence) explanation of your reasoning, then finish with CORRECT or WRONG. Do NOT include both CORRECT and WRONG in your response, or it will break the evaluation script.
```

Self-instruct DMR dataset generation

The DMR question/answer pairs were generated using the following prompt and the original MSC dataset: Your task is to write a "memory challenge" question for a simulated dialogue between two users.

```
You get as input:\\
- personas for each user (gives you their basic facts)\\
- a record of an old chat the two users had with each other\\\\
Your task is to write a question from user A to user B that test's user B's memory.\\
The question should be crafted in a way that user B must have actually participated in the prior conversation to answer properly, not just have read the persona summary.\\
Do NOT under any circumstances create a question that can be answered using the persona information (that's considered cheating).\\
Instead, write a question that can only be answered by looking at the old chat log (and is not contained in the persona information).\\\\
For example, given the following chat log and persona summaries:\\\\
old chat between user A and user B\\
A: Are you into surfing? I'm super into surfing myself\\
B: Actually I'm looking to learn. Maybe you could give me a basic lesson some time!\\
A: Yeah for sure! We could go to Pacifica, the waves there are pretty light and easy\\
B: That sounds awesome\\
A: There's even a cool Taco Bell right by the beach, could grab a bite after
B: What about this Sunday around noon?\\
A: Yeah let's do it!\\\\
user A persona:\\
I like surfing\\
I grew up in Santa Cruz\\\\
user B persona:\\
I work in tech\\
I live in downtown San Francisco\\\\
```

```
Here's an example of a good question that sounds natural, and an answer that cannot be directly inferred from user A's persona:\\\\
User B's question for user A\\
B: Remember that one time we went surfing? What was that one place we went to for lunch called?\\
A: Taco Bell!\\\\
This is an example of a bad question, where the question comes across as unnatural, and the answer can be inferred directly from user A's persona:\\\\
User B's question for user A\\
B: Do you like surfing?\\
A: Yes, I like surfing\\\\
Never, ever, ever create questions that can be answered from the persona information.
```

Document Analysis Instructions

Example instructions used in the preprompt for document analysis tasks.

```
You are MemGPT DOC-QA bot. Your job is to answer questions about documents that are stored in your archival memory. The answer to the users question will ALWAYS be in your archival memory, so remember to keep searching if you can't find the answer. Answer the questions as if though the year is 2018.
```

Questions were provided to MemGPT with the following prompt:

```
Search your archival memory to answer the provided question. Provide both the answer and the archival memory result from which you determined your answer. Format your response with the format 'ANSWER: [YOUR ANSWER], DOCUMENT: [ARCHIVAL MEMORY TEXT]. Your task is to answer the question:
```

For baselines, the following prompt along with a retrieved list of documents was provided:

```
Answer the question provided according to the list of documents below (some of which might be irrelevant. In your response, provide both the answer and the document text from which you determined the answer. Format your response with the format 'ANSWER: <YOUR ANSWER>, DOCUMENT: [DOCUMENT TEXT]'. If none of the documents provided have the answer to the question, reply with 'INSUFFICIENT INFORMATION'. Do NOT provide an answer if you cannot find it in the provided documents. Your response will only be considered correct if you provide both the answer and relevant document text, or say 'INSUFFICIENT INFORMATION'. Answer the question as if though the current year is 2018.
```

LLM Judge (document analysis)

In order to both check the correctness of the answer for the document analysis task, and also to ensure that the answer was properly derived from the provided text (rather than from the model weights), we used an LLM judge. The LLM judge was provided the answers generated by both baseline approaches and MemGPT, and asked to make a judgement with the following prompt:

```
Your task is to evaluate whether an LLM correct answered a question.
   The LLM response should be the format "ANSWER: [answer], DOCUMENT:
    [document\_text]" or say "INSUFFICIENT INFORMATION". The true
   answer is provided in the format "TRUE ANSWER:[list of possible
   answers]". The questions is provided in the format "QUESTION: [
   question]". If the LLM response contains both the correct answer
   and corresponding document text, the response is correct. Even if
   the LLM's answer and the true answer are slightly different in
   wording, the response is still correct. For example, if the answer
    is more specific than the true answer or uses a different
   phrasing that is still correct, the response is correct. If the
   LLM response if "INSUFFICIENT INFORMATION", or the "DOCUMENT"
   field is missing, the response is incorrect. Respond with a single
    token: "CORRECT" or "INCORRECT".
```

K/V Task Instructions

The MemGPT agent was defined with the following persona, designed to encourage MemGPT to iteratively search:

```
You are MemGPT DOC-QA bot. Your job is to answer questions about
   documents that are stored in your archival memory. The answer to
   the users question will ALWAYS be in your archival memory, so
   remember to keep searching if you can't find the answer. DO NOT
   STOP SEARCHING UNTIL YOU VERIFY THAT THE VALUE IS NOT A KEY. Do
   not stop making nested lookups until this condition is met.
```

Baselines were instructed with the following prompt:

```
Below is a JSON object containing key-value pairings, all keys and
   values are 128-bit UUIDs, and your task is to return the value
   associated with the specified key. If a value itself is also a key
   , return the value of that key (do a nested lookup). For example,
   if the value of 'x' is 'y', but 'y' is also a key, return the
   value of key 'y'.
```

4.7.5 Balancing Working Context and the FIFO Queue

We experimented with the size of working context on a variety of context sizes and found that for most conversational agent-based tasks (e.g. personalized chatbots), a limit

of 4000 characters (or approximately 700 tokens) is a good balance for models with 8-32k context length, since it is enough space to store key top-level details about the user and agent and leaves plenty of space for the FIFO Queue (a small FIFO Queue will require more frequent eviction). All experiments in the paper use a fixed working context length of 4000 characters. With larger context windows (100k+), the working context space can be significantly expanded. However the choice of working context size vs FIFO Queue size is highly task dependent, and in the MemGPT reference implementation we expose it as an easily modifiable parameter.

Chapter 5

From Serving Models to Serving Agents: The Missing Pieces for Supporting Agentic Workloads

5.1 Introduction

Currently the most popular large language model (LLM) APIs are stateless, i.e. they require application developers to manage the state of the program interfacing with the LLM. For example, the ChatCompletions API (the API standard used by most LLM inference providers) generates single message replies for conversation history fed as input, but the API requires the developer (rather than the API provider) to manage the message history of the conversation. We envision that many LLM workload patters (e.g. RAG, chatbots) will migrate towards a stateful agentic model, where LLM agents will act as state machines that iteratively run actions and manage state via context management and read/write access to external data sources. As such, developers will require a stateful agent-level API, rather than a stateless LLM-level API. We outline the requirements, challenges, and system optimization opportunities in building an *agent hosting layer*.

5.1.1 The Existing Stateless LLM Programming Model

The majority of LLM hosting services provide a ChatCompletions-compatible API, which is a *stateless* API. The API takes in a list of messages with alternating 'roles' ('user' for the human or non-LLM participant, and 'assistant' for the AI), and outputs the next predicted message in the sequence (in the 'assistant' role). The extent of the state if limited to what is contained in the input request, and additional state management must be managed by the developer. As such, support for functionality such as long term memory or multi-step reasoning that depends on state modifications between steps must be implemented on the client-side. The current stateless API layer overburdens developers with implementation of common, basic features for stateful LLM applications. It also requires state to be managed

on the client-side, preventing optimizations such as eliminating redundant LLM inferences (e.g. of shared prefixes), task scheduling optimizations across collaborating agents, and constrained decoding for agent-specific structured context.

5.1.2 Agentic Programming Model

We envision that the next iteration of the LLM programming API will be an agentic programming model, where LLMs will be run as a component of long-running, stateful agents. We define *agents* as stateful LLM processes that can perform tool calling and multi-step reasoning. Importantly, interactions between an agent and its environment in many cases may not be limited to chat - they can also be generalized as *events* (for example, see Packer et al. (2023). Moving from chats to tool calling agents means that we should represent inputs and outputs from agents as events in a consistent structured schema (e.g. JSON or YAML) rather than messages (represented as strings).

5.1.3 Agent State

Agent state is a crucial part of building long-running reliable agents. In order to perform multi-step reasoning (e.g. calling multiple tools in sequence) agents must effectively manage their "short term memory" (i.e. what is in their context), in addition to also maintaining "long-term memories" in external storage. We define the LLM context as (1) base system instructions, (2) tool schemas, (4) scratchpad state - context for the LLM to self-modify between tool calls (state machine). In contrast, external data storage consists of (1) event (i.e. message) history, (2) memories (information the agent has aggregated/processed that does not fit into context), (3) external data sources (i.e. external information retrieved via RAG-like processes). Creating an agent service requires both storing this state one or more database tables, as well as carefully choosing what information to store into the context versus external storage.

Multi-agent collaboration

Since it will be easier for agents to focus on completing more narrow tasks, and there are also limitations on the number of tools and instructions that can be placed in a single agent's context window, we anticipate that more complex tasks will require multi-agent collaboration. Current multi-agent frameworks Wu et al. (2023) leverage a coordinator agent to share messages between agents, however this bottlenecks and oversimplifies cross-agent communication. As agents move towards running as stateful services, cross-agent collaboration can be implemented by allowing agents to send events (or messages) to other agents.

5.2 The Agent Hosting Layer

We envision that LLM applications will be built on top of a stateful, agent hosting later than a stateless model hosting layer. This allows for developers to built on top of higher-

level, stateful APIs that manage state and cross-agent communication on their behalf, as well as enabling co-optimizations with the inference layer.

5.2.1 LLM Inference: Co-optimization with the inference layer

Existing inference systems are mostly optimized for interactive chat Kwon et al. (2023); Agrawal et al. (2023), retrieval-augmented generation Gim et al. (2024), or code generation (for example GitHub Copilot). However, agents have a much more predictable, structured pattern that the inference layer can take advantage of. For example, agentic workload have long context with common reused prefixes including system prompts. It also requires structured output for planning, tool-use, and multi-agent interaction. Additionally, agent workflow resembles direct-acyclic-graph with dependencies and branches. Inference system optimizations such as prefix caching Zheng et al. (2023a); Juravsky et al. (2024), constrained decoding Willard & Louf (2023), and request prioritization frameworks should be considered when optimizing for agents.

5.2.2 State & Context Management

Building applications on top of a stateful, agentic API layer rather than a stateless LLM layer allows developers to offload management of state and the agent's context. Standard schemas for state (e.g. agent events/messages, tool calls, user data) stored in a database by the stateful API layer allow for applications to be agnostic to the LLM provider, due to the separation of compute and storage. Agent state stored in external databases can be connected to a context management layer which determines what is placed into an agent's context on any given API call.

5.2.3 Multi-agent communication and orchestration

Agents running on the agent hosting layer will need to communicate and coordinate. We anticipate a new standard of REST-based APIs will emerge that define the patterns of agent-agent and user-agent interaction via the internet. These new API standards will have to support basic modes of operation and patterns such as event-based communication, callbacks (to enable orchestrator agents to wait on responses from subordinate agents), publish-subscribe and webhook patterns (to enable agents to autonomously trigger on events without user interference). The next generation of agent APIs will both have to support streaming (given the current demand for token-level streaming ChatCompletion APIs) as well as polling for long-running jobs where standard REST server-sent events (SSE) protocols no longer apply. Additionally, we anticipate bi-directional streaming APIs (e.g. websockets) will be of significant importance since many agent-based applications will require interfaces where a user both monitors a stream of agent agents and engages with the agent via the same interface.

Chapter 6

Conclusion & Future Work

The emergence of Large Language Models has fundamentally transformed how we approach building autonomous systems. While LLMs demonstrate unprecedented reasoning capabilities and knowledge, their effectiveness as autonomous agents is limited by fundamental constraints in their architecture - most notably, their stateless nature and fixed context windows. This thesis introduces novel frameworks for building reliable LLM-based agents, centered around the insight that memory management is a first-class concern in designing these systems.

Through our work on MemGPT, we identified that the key missing piece in building LLM-based agents was not in the models themselves, but in the surrounding system infrastructure needed to transform these powerful but stateless language models into reliable agents. By treating LLMs as a new fundamental unit of compute - analogous to how CPUs were the fundamental unit in traditional operating systems - we demonstrated how proper memory and state management can enable more reliable and capable autonomous agents. This perspective of LLMs as stateless computational primitives that require careful memory management has broad implications for the future of AI systems.

Just as operating systems evolved to manage traditional compute resources, we need new abstractions and systems to manage LLM resources effectively. MemGPT represents a first step in this direction, introducing concepts like virtual context management and self-directed memory operations. The challenges ahead in scaling these systems are significant - from managing concurrent LLM instances and sharing state across agents, to optimizing resource utilization and ensuring consistent performance in distributed deployments. These challenges will require new programming models and abstractions specifically designed for building reliable LLM-based applications.

Collectively, this work represents a significant advancement in our understanding of how to build reliable autonomous systems in the era of Large Language Models. By establishing new paradigms for memory management in LLM-based systems, this work supports the development of more sophisticated AI applications that can effectively reason, plan, and maintain long-term state. As language models continue to grow in capability and deployment, the importance of robust system infrastructure for managing them will only increase.

The frameworks and techniques presented in this thesis provide a foundation for addressing these challenges, paving the way for a new generation of reliable autonomous agents built on Large Language Models.

Bibliography

Agrawal, A., Panwar, A., Mohan, J., et al. 2023, SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills, arXiv:2308.16369 [cs.LG]

Al-Shedivat, M., Bansal, T., Burda, Y., et al. 2018, in International Conference on Learning Representations (ICLR)

Andrychowicz, M., Wolski, F., Ray, A., et al. 2017, in Neural Information Processing Systems (NeurIPS), 5048

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. 2017, IEEE Signal Processing Magazine, 34

Barto, A. G., Sutton, R. S., & Anderson, C. W. 1983, IEEE Transactions on Systems, Man, and Cybernetics, 13

Beattie, C., Leibo, J. Z., Teplyashin, D., et al. 2016, arXiv:1612.03801

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. 2013, Journal of Artificial Intelligence Research (JAIR), 47

Beltagy, I., Peters, M. E., & Cohan, A. 2020, arXiv preprint arXiv:2004.05150

Bengio, Y., Bengio, S., & Cloutier, J. 1990, Learning a synaptic learning rule (Citeseer)

Borgeaud, S., Mensch, A., Hoffmann, J., et al. 2022, in International conference on machine learning, PMLR, 2206

Brockman, G., Cheung, V., Pettersson, L., et al. 2016, arXiv:1606.01540

Brown, T., Mann, B., Ryder, N., et al. 2020, Advances in neural information processing systems, 33, 1877

Chen, S., Wong, S., Chen, L., & Tian, Y. 2023, arXiv preprint arXiv:2306.15595

Child, R., Gray, S., Radford, A., & Sutskever, I. 2019, arXiv preprint arXiv:1904.10509

Clark, J., & Amodei, D. 2016, *https://blog.openai.com/faulty-reward-functions*

Clavera, I., Nagabandi, A., Fearing, R. S., et al. 2018, arXiv:1803.11347

Cobbe, K., Klimov, O., Hesse, C., Kim, T., & Schulman, J. 2019, in International Conference on Machine Learning (ICML)

Dai, Z., Yang, Z., Yang, Y., et al. 2019, arXiv preprint arXiv:1901.02860

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. 2018, arXiv preprint arXiv:1810.04805

Dhariwal, P., Hesse, C., Plappert, M., et al. 2017, OpenAI Baselines

Dietterich, T. G. 2017, AI Magazine, 38

Dong, Z., Tang, T., Li, L., & Zhao, W. X. 2023, arXiv preprint arXiv:2302.14502

Donoho, D. 2015, in Tukey Centennial Workshop

Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. 2016a, in International Conference on Machine Learning (ICML)

Duan, Y., Schulman, J., Chen, X., et al. 2016b, arXiv:1611.02779

Eysenbach, B., Geng, X., Levine, S., & Salakhutdinov, R. 2020, arXiv preprint arXiv:2002.11089

Finn, C., Abbeel, P., & Levine, S. 2017, in International Conference on Machine Learning (ICML)

Florensa, C., Held, D., Geng, X., & Abbeel, P. 2018, in International Conference on Machine Learning (ICML)

Gim, I., Chen, G., seob Lee, S., et al. 2024, Prompt Cache: Modular Attention Reuse for Low-Latency Inference, arXiv:2311.04934 [cs.CL]

Gupta, A., Eysenbach, B., Finn, C., & Levine, S. 2018a, arXiv preprint arXiv:1806.04640

Gupta, A., Mendonca, R., Liu, Y., Abbeel, P., & Levine, S. 2018b, in Neural Information Processing Systems (NeurIPS), 5302

Guu, K., Lee, K., Tung, Z., Pasupat, P., & Chang, M. 2020, in International conference on machine learning, PMLR, 3929

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. 2018, in International Conference on Machine Learning (ICML)

Henderson, P., Islam, R., Bachman, P., et al. 2018, in AAAI Conference on Artificial Intelligence (AAAI)

Hinton, G., Srivastava, N., & Swersky, K. 2012, Neural networks for machine learning lecture 6, https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Houthooft, R., Chen, Y., Isola, P., et al. 2018, in Neural Information Processing Systems (NeurIPS), 5400

Izacard, G., Caron, M., Hosseini, L., et al. 2021, arXiv preprint arXiv:2112.09118

Izacard, G., & Grave, E. 2020, arXiv preprint arXiv:2007.01282

Jabri, A., Hsu, K., Gupta, A., et al. 2019, in Neural Information Processing Systems (NeurIPS), 10519

Jiang, Z., Xu, F. F., Gao, L., et al. 2023, arXiv preprint arXiv:2305.06983

Juravsky, J., Brown, B., Ehrlich, R., et al. 2024, Hydragen: High-Throughput LLM Inference with Shared Prefixes, arXiv:2402.05099 [cs.LG]

Kaelbling, L. P. 1993, in International Joint Conferences on Artificial Intelligence (IJCAI), Citeseer, 1094

Kansky, K., Silver, T., Mély, D. A., et al. 2017, in International Conference on Machine Learning (ICML)

Karpukhin, V., Oğuz, B., Min, S., et al. 2020, arXiv preprint arXiv:2004.04906

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., & Jaśkowski, W. 2016, in IEEE Conference on Computational Intelligence and Games

Kingma, D. P., & Ba, J. 2015, in International Conference on Learning Representations (ICLR)

Kitaev, N., Kaiser, Ł., & Levskaya, A. 2020, arXiv preprint arXiv:2001.04451

Kwon, W., Li, Z., Zhuang, S., et al. 2023, Efficient Memory Management for Large Language

Model Serving with PagedAttention, arXiv:2309.06180 [cs.LG]
Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. 2017, Behavioral and Brain Sciences, 40
Lee, J., Lee, Y., Kim, J., et al. 2019, in International conference on machine learning, PMLR, 3744
Levine, S., Finn, C., Darrell, T., & Abbeel, P. 2016, Journal of Machine Learning Research (JMLR), 17, 1334
Levy, A., Platt, R., & Saenko, K. 2017, arXiv preprint arXiv:1712.00948
Lewis, P., Perez, E., Piktus, A., et al. 2020, Advances in Neural Information Processing Systems, 33, 9459
Li, A. C., Pinto, L., & Abbeel, P. 2020, arXiv preprint arXiv:2002.11708
Lillicrap, T. P., Hunt, J. J., Pritzel, A., et al. 2015, arXiv preprint arXiv:1509.02971
Lim, S. H., Xu, H., & Mannor, S. 2013, in Neural Information Processing Systems (NIPS)
Lin, C.-Y. 2004, in Text summarization branches out, 74
Lin, X. V., Chen, X., Chen, M., et al. 2023, RA-DIT: Retrieval-Augmented Dual Instruction Tuning, arXiv:2310.01352 [cs.CL]
Liu, N. F., Lin, K., Hewitt, J., et al. 2023a, arXiv preprint arXiv:2307.03172
Liu, X., Yu, H., Zhang, H., et al. 2023b, arXiv preprint arXiv:2308.03688
Machado, M. C., Bellemare, M. G., Talvitie, E., et al. 2017, arXiv:1709.06009
Marcus, G. 2018, arXiv:1801.00631
Mishra, N., Rohaninejad, M., Chen, X., & Abbeel, P. 2018a, in International Conference on Learning Representations (ICLR)
Mishra, N., Rohaninejad, M., Chen, X. P., & Abbeel, P. 2018b, in International Conference on Learning Representations (ICLR)
Mnih, V., Badia, A. P., Mirza, M., et al. 2016, in International Conference on Machine Learning (ICML)
Mnih, V., Kavukcuoglu, K., Silver, D., et al. 2015, Nature, 518
Moore, A. W. 1990, Efficient Memory-based Learning for Robot Control, Tech. rep., University of Cambridge Computer Laboratory
Morimoto, J., & Doya, K. 2001, in Neural Information Processing Systems (NIPS)
Nair, A., Srinivasan, P., Blackwell, S., et al. 2015, arXiv:1507.04296
Nair, A. V., Pong, V., Dalal, M., et al. 2018, in Neural Information Processing Systems (NeurIPS), 9191
Nakano, R., Hilton, J., Balaji, S., et al. 2021, arXiv preprint arXiv:2112.09332
Ng, A. Y., Harada, D., & Russell, S. 1999, in International Conference on Machine Learning (ICML), Vol. 99, 278
Nichol, A., Pfau, V., Hesse, C., Klimov, O., & Schulman, J. 2018, arXiv:1804.03720
Nilim, A., & Ghaoui, L. E. 2004, in Neural Information Processing Systems (NIPS)
Ouyang, L., Wu, J., Jiang, X., et al. 2022, Advances in Neural Information Processing Systems, 35, 27730
Packer, C., Fang, V., Patil, S. G., et al. 2023, arXiv preprint arXiv:2310.08560
Packer, C., Gao, K., Kos, J., et al. 2018, arXiv:1810.12282

Park, J. S., O'Brien, J. C., Cai, C. J., et al. 2023, arXiv preprint arXiv:2304.03442
Patterson, D. A., Gibson, G., & Katz, R. H. 1988, in Proceedings of the 1988 ACM SIGMOD international conference on Management of data, 109
Pinto, L., Davidson, J., Sukthankar, R., & Gupta, A. 2017, in International Conference on Machine Learning (ICML)
Press, O., Smith, N. A., & Lewis, M. 2021, arXiv preprint arXiv:2108.12409
Rajeswaran, A., Ghotra, S., Ravindran, B., & Levine, S. 2017a, in International Conference on Learning Representations (ICLR)
Rajeswaran, A., Lowrey, K., Todorov, E. V., & Kakade, S. M. 2017b, in Neural Information Processing Systems (NIPS)
Rakelly, K., Zhou, A., Quillen, D., Finn, C., & Levine, S. 2019, in International Conference on Machine Learning (ICML)
Ram, O., Levine, Y., Dalmedigos, I., et al. 2023, arXiv preprint arXiv:2302.00083
Rauber, P., Ummadisingu, A., Mutz, F., & Schmidhuber, J. 2019, in International Conference on Learning Representations (ICLR)
Rothfuss, J., Lee, D., Clavera, I., Asfour, T., & Abbeel, P. 2019, in International Conference on Learning Representations (ICLR)
Roy, A., Xu, H., & Pokutta, S. 2017, in Neural Information Processing Systems (NIPS)
Ruder, S. 2017, arXiv:1706.05098
Rusu, A. A., Rabinowitz, N. C., Desjardins, G., et al. 2016, arXiv:1606.04671
Sæmundsson, S., Hofmann, K., & Deisenroth, M. P. 2018, arXiv:1803.07551
Schick, T., Dwivedi-Yu, J., Dessì, R., et al. 2023, arXiv preprint arXiv:2302.04761
Schmidhuber, J. 1987, PhD thesis, Technische Universität München
Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., & Moritz, P. 2015, in International Conference on Machine Learning (ICML)
Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. 2017, arXiv:1707.06347
Silver, D., Hubert, T., Schrittwieser, J., et al. 2017, arXiv preprint arXiv:1712.01815
Stadie, B. C., Yang, G., Houthooft, R., et al. 2018, arXiv preprint arXiv:1803.01118
Sung, F., Zhang, L., Xiang, T., Hospedales, T., & Yang, Y. 2017, arXiv:1706.09529
Sutton, R. S. 1995, in Neural Information Processing Systems (NIPS)
Sutton, R. S., & Barto, A. G. 2017, Reinforcement Learning: An Introduction, 2nd edn. (MIT Press)
Tamar, A., Glassner, Y., & Mannor, S. 2015, in AAAI Conference on Artificial Intelligence (AAAI)
Taylor, M. E., & Stone, P. 2009, Journal of Machine Learning Research (JMLR), 10
Thrun, S., & Pratt, L. 1998, in Learning to learn (Springer), 3
Todorov, E., Erez, T., & Tassa, Y. 2012, in Intelligent Robots and Systems (IROS)
Touvron, H., Martin, L., Stone, K., et al. 2023, arXiv preprint arXiv:2307.09288
Trivedi, H., Balasubramanian, N., Khot, T., & Sabharwal, A. 2022, ArXiv, abs/2212.10509
Vaswani, A., Shazeer, N., Parmar, N., et al. 2017, Advances in neural information processing systems, 30
Wang, J. X., Kurth-Nelson, Z., Tirumala, D., et al. 2016, arXiv:1611.05763

Wang, S., Li, B. Z., Khabsa, M., Fang, H., & Ma, H. 2020, arXiv preprint arXiv:2006.04768
Wei, J., Wang, X., Schuurmans, D., et al. 2022, Advances in Neural Information Processing Systems, 35, 24824
Whiteson, S., Tanner, B., Taylor, M. E., & Stone, P. 2011, in IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning
Willard, B. T., & Louf, R. 2023, arXiv preprint arXiv:2307.09702
Wu, Q., Bansal, G., Zhang, J., et al. 2023, arXiv preprint arXiv:2308.08155
Xu, J., Szlam, A., & Weston, J. 2021, arXiv preprint arXiv:2107.07567
Xu, T., Liu, Q., Zhao, L., & Peng, J. 2018a, in International Conference on Machine Learning (ICML), 5463
Xu, Z., van Hasselt, H. P., & Silver, D. 2018b, in Neural Information Processing Systems (NeurIPS), 2396
Yao, S., Zhao, J., Yu, D., et al. 2022, arXiv preprint arXiv:2210.03629
Yu, W., Tan, J., Liu, C. K., & Turk, G. 2017, in Robotics: Science and Systems (RSS)
Zhang, A., Ballas, N., & Pineau, J. 2018, arXiv:1806.07937
Zheng, L., Yin, L., Xie, Z., et al. 2023a, Efficiently Programming Large Language Models using SGLang, arXiv:2312.07104 [cs.AI]
Zheng, L., Chiang, W.-L., Sheng, Y., et al. 2023b, arXiv preprint arXiv:2306.05685
Zintgraf, L. M., Shiarlis, K., Kurin, V., Hofmann, K., & Whiteson, S. 2018, arXiv preprint arXiv:1810.03642