Emergence of Intelligent Navigation Behavior
in Embodied Agents
from Massive-Scale Simulation

A Dissertation
Presented to
The Academic Faculty

By

Erik Wijmans

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing
School of Interactive Computing

Georgia Institute of Technology

August 2022

© Erik Wijmans 2022
Emergence of Intelligent Navigation Behavior
in Embodied Agents
from Massive-Scale Simulation

Thesis committee:

Dr. Dhruv Batra (Advisor)
School of Interactive Computing
Georgia Institute of Technology

Dr. Irfan Essa (Advisor)
School of Interactive Computing
Georgia Institute of Technology

Dr. Sonia Chernova
School of Interactive Computing
Georgia Institute of Technology

Dr. Vladlen Koltun
Distinguished Scientist
Apple

Dr. Vincent Vanhoucke
Distinguished Scientist
Google

Dr. Gregory Wayne
Research Scientist
DeepMind

Date approved: July 15, 2022
ACKNOWLEDGMENTS

Acknowledgements are perhaps the part of my dissertation that make me the most nervous. Someone will be left out. For those that are listed, I certainly won’t capture all you’ve contributed to both my work and my education. This will be shorter than it should be and certainly not complete.

To my advisors, Dhruv Batra and Irfan Essa, thank you for your guidance, advice, and support. Being co-advised has allowed me to blend together your advice and feedback, enabling me to become a better researcher. There simply isn’t the time nor space to list all that I’ve learned working with you.

To Yasu Furukawa, thank you for your formative guidance and giving my start. I knew pretty much nothing when we started working together and really got my footing in computer science (both research and in general) while working with you.

To Devi Parikh, thank you for your guidance and inspiration.

To my collaborators at FAIR, my labmates in CVMLP and EYE-Team, thank you for your friendship. Working with and learning from each of you has been a highlight.

To my committee, Dhruv Batra, Irfan Essa, Sonia Chernova, Vladlen Koltun, Vincent Vanhouche, and Greg Wayne, thank you for your feedback, input, and questions. Your questions in particular have helped me to focus and clarify my thoughts.

To my parents, Judy Mohr and Hans Wijmans, thank you for your love and support throughout the entirety of my education journey.
# TABLE OF CONTENTS

Acknowledgments ......................................................... iii

List of Figures .......................................................... vii

List of Tables ............................................................ xiii

Summary ................................................................. xv

Chapter 1: Introduction .................................................. 1

1.1 Example Tasks .................................................... 3

1.1.1 Indoor Navigation ............................................. 3

1.1.2 Mobile Manipulation .......................................... 4

1.2 Thesis Statement .................................................. 5

1.2.1 Reinforcement Learning ...................................... 5

1.2.2 Simulation ...................................................... 6

1.2.3 Massive-scale .................................................. 6

1.2.4 Intelligent Behavior .......................................... 6

1.3 Summary of Contributions ....................................... 7

1.4 Contributed Papers ............................................... 10

1.5 Other Papers ...................................................... 10
Chapter 2: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames

2.1 Introduction ........................................ 12
2.2 Preliminaries: RL and PPO .......................... 15
2.3 Decentralized Distributed Proximal Policy Optimization ........... 16
2.4 Experimental Setup: PointGoal navigation, Agents, Simulator ...... 18
2.5 Benchmarking: How does DD-PPO scale? ....................... 20
2.6 Mastering PointGoal navigation with GPS+Compass ............... 22
2.7 Related Work ......................................... 26

Chapter 3: Blind Agents Take Shortcuts and Build Maps ............... 30

3.1 Introduction ......................................... 30
3.2 Efficient Navigation with No Vision .......................... 36
3.3 Emergence of Wall-following Behavior and Collision-detection Neurons . 36
3.4 Memory Length ....................................... 38
3.5 Memory Enables Shortcuts .............................. 39
3.6 Decoding a Metric Map ................................ 43
3.7 Maps are Task-dependent .............................. 44
3.8 Outlook ............................................... 45

Chapter 4: Emergence of Navigation in Mobile Manipulation Agents .... 48

4.1 Introduction ......................................... 48
4.2 VER: Variable Experience Rollout .......................... 51
   4.2.1 Action-Level Straggler Mitigation .................. 52
   4.2.2 Environment-Level Straggler Mitigation .......... 53
LIST OF FIGURES

1.1 In PointGoalNav, an agent must navigate from a random starting location (blue) to a target location (red) specified relative to the agent ("Go 5m north, 10m east of you") in a previously unseen environment without access to a map. ............................... 3

1.2 In GeometricGoal rearrangement a robot is initialized in a novel unseen environment and tasked with rearranging it such that objects are placed at their target locations. This example shows the robot picking up an object, navigating to place it, and then placing it. ............................... 5

1.3 In Chapter 2 we show the emergence of near-perfect navigation from massive-scale. In Chapter 3 we show the emergence of mapping in 'blind' navigation agents from massive-scale. In Chapter 4 we show the emergence of navigation in mobile manipulation agents from massive-scale. ............................... 8

2.1 Left: In PointGoal navigation, an agent must navigate from a random starting location (blue) to a target location (red) specified relative to the agent ("Go 5m north, 10m east of you") in a previously unseen environment without access to a map. Right: Performance (SPL; higher is better) of an agent equipped with RGB-D and GPS+Compass sensors on the Habitat Challenge 2019 [22] train & val sets. Using DD-PPO, we train agents for over 180 days of GPU-time in under 3 days of wall-clock time with 64 GPUs, achieving state-of-art results and ‘solving’ the task. ............................... 13

2.2 Comparison of asynchronous distribution (left) and synchronous distribution via distributed data parallelism (right) for RL. Left: rollout workers collect experience and asynchronously send it to the parameter-server. Right: a worker alternates between collecting experience, synchronizing gradients, and optimization. We find this highly effective in resource-intensive environments. ............................... 15
2.3 Our agent for PointGoalNav. At very time-step, the agent receives an ego-centric Depth or RGB (shown here) observation, utilizes its GPS+Compass sensor to update the target position to be relative to its current position, and outputs the next action and an estimate of the value function. 

2.4 Scaling performance (in steps of experience per second relative to 1 GPU) of DD-PPO for various preemption threshold, $p\%$, values. Shading represents a 95% confidence interval. 

2.5 Performance (higher is better) on Flee (left) and Exploration (right) under five settings. 

3.1 PointGoal navigation. An agent is initialized in a novel environment (bluesquare) and task with navigation to a point specified relative to the start location (red square)—e.g. $(5, 2)$ means go 5 meters forward and 2 meters right. We study agents that must do solely from egocentric GPS+Compass and without a map. 

3.2 (A) ‘Blind’ agent vs. bug. Our learned ‘blind’ agent compared to 2 variants and an oracle equipped variant of the Bug algorithm [105]. The Bug algorithm initially orients itself towards the goal and then proceeds towards the goal. Upon hitting a wall, it follows along the wall until it reaches the other side. The oracle version is told whether wall-following left or right is optimal, providing an upper-bound on Bug algorithm performance. (B) Navigation performance vs. memory length. We find that the performance of agents does not saturate until their memory can contain information from hundreds of time steps in the past. Navigation episodes vary in length from slightly over 100 time steps to the maximum of 2,000, so a memory of $10^3$ steps is nearly half the maximum episode length. 

3.3 t-SNE of the agent’s internal representation for collisions. We find 4 overall clusters corresponding to the previous action taken and whether or not that action led to a collision. The t-SNE manifold is computed using 20,000 samples. This is randomly subsampled to 1,500 for visualization. 

3.4 Primary experimental paradigm. First, an agent navigates (blue path, blue LSTM) from start (green sphere) to target (red sphere). After the agent navigates, we task a probe (purple LSTM) with performing the same navigation episode with the additional information encapsulated in the agent’s internal representation (or memory), $h_A$. The probe is able to navigate more efficiently by taking shortcuts (purple path). As denoted by the dashed line between the probe and agent networks, the probe does not influence what the agent stores in its internal representation. Environment in the image from the Replica Dataset [108].
3.5 **Map prediction.** Accuracy (Intersection over Union) distributions (via kernel density estimation), dashed lines are the mean, TrainedAgentMemory has a higher mean with p-value $\leq 10^{-5}$ (via Wilcoxon signed-rank test [110]). We show example ground truth - predictions pairs for various IoU values using TrainedAgentMemory. Light grey is non-navigable and dark grey is navigable. The agent path is drawn in light blue and navigates from start (green) to target (red). 43

3.6 **(A) Excursion prediction example.** Qualitative example of the previously-visited location decoder making systematic errors when decoding an excursion. Blue represents the confidence of the decoder that the agent was previously at a given location. **(B) Remembrance of excursions.** Performance of decoders when predicting previous agent locations broken down into three categories. ‘Non-excursion’ is all predictions where the current location of the agent and the prediction time step are not part of an excursions. ‘Excursion’ is when the prediction time step is part of an excursion. ‘Exit’ is when the prediction time step is part of the last 10% of the excursion. X-axis is the distance into the past the decoder is predicting. Y-axis is the relative error between the true location and the predicted location. 45

4.1 **(Right) RL Training Systems.** In SyncOnRL, actions are computed for all environments, then all environments are stepped. Experience collection is paused during learning. In AsyncOnRL, computing actions, stepping environments, and learning all occur without synchronization. In VER, a variable amount of experience is collected from each environment, enabling synchronous learning without the straggler effect. **(Left) skill policies** with navigation are more robust to handoff errors. 49

4.2 **VER system architecture.** Environment workers receive actions to simulate and return the result of that environment step (EnvStep). Inference workers receive batches experience from environment workers. They return the new action to take to environment workers and write the experience into GPU shared memory for learning. 52

4.3 **(A) VER** collects a variable amount of experience from each environment. The length of each step represents the time taken to collect it. **(B) VER mini-batch.** The solid bars denote episode boundaries. The steps selected for the first mini-batch have a dashed border. **(C) The PackedSequence data format** represents a set of sequences with variable length in a linear buffer such that all elements from each timestep area next to one-another in memory. 54

4.4 **Validation performance** on ObjectNav and PointNav. VER has similar or
slightly better sample efficiency than DD-PPO, indicating that performance is not negatively impacted by the non-uniform sampling of experience from environments. Shading is a 95% confidence interval over 3 seeds.

4.5 Training sample efficiency on Open Fridge. VER has similar sample efficiency as DD-PPO (SyncOnRL). SampleFactory (AsyncOnRL) has similar sample efficiency with 1 GPU but this reduces as policy lag increases with more GPUs. When considering both sample efficiency and throughput (time-to-performance), VER is always achieves a given performance the fastest (outright or a tie). The shaded region is a 95% bootstrapped confidence interval over 5 seeds. We use interquartile mean (IQM) as our summary statistic [133].

4.6 HAB Performance on the Tidy House, Prepare Groceries, and Set Table scenarios. Skill policies with navigation (TP-SRL+Skill Navigation) outperform skill policies without navigation (TP-SRL) despite not strictly needing this ability. Further, we find that these skill policies have learned emergent navigation. TP-SRL(NoNav)+Skill Navigation achieves 50% success on NavPick (Tidy House) and 20% success on NavPickNavPlace (Tidy House) despite all navigation being performed by skills policies that ostensibly don’t need to navigate.

A.1 Example episodes broken down by geodesic distance between agent’s spawn location and target (on rows) vs SPL achieved by the agent (on cols). Gray represents navigable regions on the map while white is non-navigable. The agent begins at the blue square and navigates to the red square. The green line shows the shortest path on the map (or oracle navigation). The blue line shows the agent’s trajectory. The color of the agent’s trajectory changes from dark to light over time. Navigation dataset from the longer validation episodes proposed in [68].

A.2 Histogram of SPL for non-perfect (SPL<0.99) episodes.

A.3 Scaling of DD-PPO under homogeneous and heterogeneous workloads for various different values of the percentage of rollouts that are fully completed by optimizing the model. Shading represents a bootstrapped 95% confidence interval.

A.4 Implementation of DD-PPO using PyTorch [58] v1.1 and the NCCL backend. We use SLURM to populate the world_rank, world_size, and local_rank fields.
A.5 Illustration of DD-PPO. Processes collecting experience in environments that are more costly to simulate (stragglers) have their experience collection stage preempted such that other processes do not have to wait for them. Note that we implement the monitor with a simple key-value storage and have processes preempt themselves. Note that the order of processes is irrelevant and done solely for aesthetic purposes.

A.6 Examples of Gibson meshes for a given quality rating from [22].

A.7 Training and validation performance (in SPL; higher is better) of different architectures for Depth agents with GPS+Compass on the Habitat Challenge 2019 [22]. Gibson [66]-4+ refers to the subset of Gibson train scenes with a quality rating of 4 or better. Gibson-4+ and MP3D refers to training on both Gibson-4+ and all of Matterport3D. Gibson-2+ refers to training on the subset of Gibson train scenes with a quality rating of 2 or better.

A.8 Performance vs. Geodesic Distance from start to goal for Blind, RGB, and RGB-D (using Depth only) models trained with DD-PPO on the Habitat Challenge 2019 [22] validation split. Bars at the bottom represent the fraction of episodes within each geodesic distance bin.

A.9 Performance vs. Geodesic Distance from start to goal for Blind, RGB, and RGB-D (using Depth only) models trained with DD-PPO on the longer and harder validation episodes proposed in [68]. Bars at the bottom represent the fraction of episodes within each geodesic distance bin.

B.1 True state trajectory comparison. Example trajectories of an agent with true state (trained for a specific environment and using global coordinates), green line, compared to an agent trained for many environments and using episodic coordinates, blue line. The later is what we examine in this work. Notice that the agent with true state take shortest path trajectories while the agent without true state instead exhibits strong wall-following behavior.

B.2 Two categories of probe shortcut. ‘Excursion Removal’ is when the probe removes excursions from the agent’s path. The dashed line shows the distance between the points in the excursion and the closest point in the probe’s path. ‘Free Space Inference’ occurs when the probe travels through previously unvisited locations in the environments. The dashed lines show the distance between any points in the probe’s path and the closest point in the agent’s path.

B.3 Past and future prediction. Performance of decoders trained to predict
where the agent was in the past/will be in the future. On the x-axis is how far into the past or future the decoder is predicting (positive values are future predictions and negative values are past predictions). The y-axis is either absolute or relative L2 error between the predicted location of the agent and the true location.

B.4 Map prediction accuracy (Intersection over Union) for Depth sensor equipped agents.

B.5 Map prediction accuracy (class balanced accuracy) for Depth sensor equipped agents.

B.6 Memory-only probe performance. Performance (in SPL; higher is better) as a function of geodesic distance from start to goal for the TrainedAgentMemory probe without inputs on SecondNav(S→T) and SecondNav(T→S). More information can be found under the ‘Navigation from memory alone’ header.

B.7 Free Space Inference for the TrainedAgentMemory probe on both SecondNav(S→T) and SecondNav(T→S) as a function of agent SPL. We see that as agent SPL decreases, the probe is able to take paths that inference more free space.

B.8 Excursion Removal for the TrainedAgentMemory probe on both SecondNav(S→T) and SecondNav(T→S) as a function of agent SPL. We see that as agent SPL decreases, excursion removal increases since the probe is able to remove additional excursions.

B.9 Performance vs. memory length for agents trained under a given memory length. Note that longer memory lengths are challenging to train for under this methodology as it induces the negative effects of large-batch optimization and is computationally expensive.

C.1 Sample efficiency and time-to-sample for each system on a varying number of GPUs. Even with the reduction of sample efficiency when increasing the number of GPUs, increasing the the number of GPUs always reduces the time to reach convergence.
LIST OF TABLES

2.1 Performance (higher is better) of different architectures for agents with RGB-D and GPS+Compass sensors on the Habitat Challenge 2019 [22] validation and test-std splits (checkpoint selected on val). 10 samples taken for each episode on val. Gibson-4+ (2+) refers to the subset of Gibson train scenes [66] with a quality rating of 4 (2) or higher. See Table A.1 for results of the best DD-PPO agent for Blind, RGB, and RGB-D and other baselines. 23

3.1 Probe performance. Results of our trained probe agent under three configurations – initialized with an empty representation (AllZeroMemory), a representation of a random agent walked along the trained agent’s path (UntrainedAgentMemory), and the final representation of the trained agent (TrainedAgentMemory). 95% confidence interval reported over 5 agent-probe pairs. Success and SPL are in percent. 41

4.1 Navigation Tasks training steps per second on 8 GPUs. VER is 60%-100% faster than DD-PPO. 57

4.2 SyncOnRL, VER, and AsyncOnRL benchmarking. Mean/max system throughput (SPS) over 20 million training steps. Hardware: Tesla V100(s) with 10 CPUs per GPU. 61

A.1 Performance (higher is better) of various sensors and agent methods on the Habitat Challenge 2019 [22] validation and test splits (checkpoint selected on val). Random, Forward-only, and Goal-follower taken from [22]. Best visual encoder reported for DD-PPO. 80

B.1 Excursion removal result of our trained probe agent under three configurations – initialized with an empty representation (AllZeroMemory), a representation of a random agent walked along the trained agent’s path (UntrainedAgentMemory), and the final representation of the trained agent (TrainedAgentMemory). 95% confidence interval reported over 5 agent-probe pairs. 97
C.1 Hyperparameters 107

C.2 HTS-RL comparison. Mean system throughput (SPS) over 1 million training steps. HTS-RL (Provided) does not support training recurrent policies. Hardware: Nvidia 2080 Ti with 16 CPUs. 107
SUMMARY

The goal of Artificial Intelligence is to build ‘thinking machines’ [1] that ‘use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves.’ [2] In this dissertation, we will argue that the intelligence required for this goal emerges from massive-scale simulation. We will show a specific case: that intelligent navigation behavior emerges from massive-scale simulation and deep reinforcement learning.

Towards this end, we introduce Decentralized Distributed PPO (DD-PPO), a method that scales reinforcement learning to multiple GPUs and machines. We use DD-PPO to train agents for PointGoal navigation (e.g. ‘Go 5 meters north and 10 meters east relative to start’) for the equivalent of 80 years of human experience. This massive-scale training results in near-perfect autonomous navigation in an unseen environment without access to a map. We then examine the inner workings of special case of PointGoalNav agents. We find that (1) their memory enables shortcuts, i.e. efficiently travel through previously unexplored parts of the environment; (2) there is emergence of maps in their memory, i.e. a detailed occupancy grid of the environment can be decoded from it.

We then introduce Variable Experience Rollout (VER), a method that efficiently scales reinforcement learning on a single GPU or machine. We use VER to train chained skills for mobile manipulation. We find a surprising emergence of navigation in skills that do not ostensibly require any navigation. Specifically, the pick skill involves a robot picking an object from a table. During training, the robot was always spawned close to the table and never needs to navigate. However, we find that if navigation actions are part of the action space, the robot learns to navigate then pick an object in new environments with 50% success, demonstrating surprisingly high out-of-distribution generalization.
The goal of Artificial Intelligence (AI), as set by McCarthy [2], is to build ‘thinking machines’ [1]. Machines that ‘use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves.’ [2] Such thinking machines, when deployed as robots, would have profound impact on the world, from completing simple daily tasks to caring for older populations and performing search and rescue missions in dangerous environments.

Turing proposed that we build a simpler machine and teach it increasingly complex tasks [1]. This paradigm has been embraced by the recent data-driven approaches that have yielded fantastic progress in computer vision [3, 4], natural language [5], and robotics [6, 7]. However, challenges – such as transitioning from static to active perception and novel environments – have made it difficult to fuse these independent advancements to create a thinking machine.

We believe that systems will exhibit the only level of intelligence that their world requires of them. If the world only consists of static images, you will get increasingly smarter versions of Mask R-CNN [4], but not a system that can act in the world. If the world only consists of tokens, you will get an increasingly smart GPT [5], but not a system that can ground concepts from language to reality. If the world consists of only a table-top, you will get increasingly more accurate grasping/manipulation [6, 7], but not a system capable of moving about the world or interacting with other agents (e.g. humans).

In this dissertation, we will argue for training systems in rich, diverse, and complex worlds. We will operate on 3 hypotheses.

1. Intelligence is emergent. Emergence is ‘complexity at low-level giving rise to simplicity at high-level.’ [8] Chemistry emerges from the laws of physics [9]. Biology emerges
from chemistry [10]. Neuroscience (brains) emerges from biology. Cognitive science and intelligence (AI) emerge from brains.

2. The scaling hypothesis [11]. The idea that once we find a sufficiently scalable neural network architecture (e.g. convolutions or self-attention), we can simply train ever larger networks with ever more data and ever more sophisticated behavior will emerge.

3. The embodiment hypothesis [12]. The idea that ‘intelligence emerges in the interaction of an agent with an environment and as a result of sensorimotor activity.’ [12]

Taken together, these hypotheses say that our goal should be to create increasingly rich habitats for AI, where embodied intelligent systems can learn at scale and increasingly intelligent behavior will emerge. That we should build a simpler machine and teach it increasingly complex tasks from massive-scale interaction with its environment.

Ideally we would train and test such “embodied agents” directly in the real world, therefore exposing them to all its richness diversity, and complexity. However, the real world is:

– **Slow**: the real world runs no faster than real time and cannot be parallelized,

– **Dangerous**: poorly-trained agents can unwittingly injure themselves, the environment, or others,

– **Expensive**: both the agent and the environment(s) in which they execute are expensive,

– **Difficult to control/reproduce**: replicating conditions (particularly corner-cases) or experiments is often difficult.

Due to these limitations we are unable to achieve massive-scale in the real world. We instead use simulation as our environment to train embodied agents. Simulations can run orders of magnitude faster than real-time and can be trivially parallelized over a cluster; training/testing in simulation is safe, cheap, and enables fair systematic benchmarking of progress. Once a promising approach has been developed and tested in simulation, it can be transferred to physical platforms.
Figure 1.1: **In PointGoalNav**, an agent must navigate from a random starting location (blue) to a target location (red) specified relative to the agent (“Go 5m north, 10m east of you”) in a previously *unseen* environment *without* access to a map.

We now describe two example tasks for embodied agents that will serve as testbeds.

### 1.1 Example Tasks

#### 1.1.1 Indoor Navigation

Indoor navigation is a necessary skill for any robot deployed in an indoor environment. We use PointGoal navigation (**PointGoalNav**) [13] as our testbed.

In **PointGoalNav** (Fig. 1.1), an agent is initialized in a novel unseen environment and tasked with navigating to a point specified relative to its initial localization – ‘go 5m north, 10m east relative to start’\(^1\) without access to a map. It must do so by leveraging its ego-centric sensors: an RGB camera, a Depth sensor, and a egomotion sensor, which provides its location and heading relative to the starting location.

The task specification in **PointGoalNav** allows us to focus on an agent’s navigation abilities. There are no ambiguities about where the agent is being directed to and it removes confounding variables like language or semantic understanding. Further, the task

\(^1\)The description in English is purely for explanatory purposes; the agent receives relative goal coordinates.
specification has biological plausibility. It is analogous to the direction and distance of foraging locations communicated by the waggle dance of honey bees [14].

The task specification also makes an agent trained for PointGoalNav an ideal candidate to function as a low-level controller as part of a hierarchical system. Waypoints are a logical output of a high-level policy. Waypoints precisely communicate where the high-level policy wants to go but leave the how up to the low-level controller. This paradigm has been used for agents that navigate by following instructions in natural language [15]. A waypoint is the exactly the goal specification of a PointGoalNav agent.

This is a challenging task. Mapping+Planning [16, 17, 18, 19] approaches have been studied to perform similar tasks; however, they have a number of challenges. Maps are simultaneously over-complete and immediately out of date. You may not need a detailed metric map to navigate down a hallway the world is constantly changing. Further, the planning component must be able to handle a myriad of edge cases due to imperfect sensing, imperfect location, and mapping errors. In practice, these nearly prohibit integrating all the components to create a system that performs the full task [20].

1.1.2 Mobile Manipulation

Next, we turn our focus to mobile manipulation – once we’ve navigated to an object, the next step is to manipulate it (e.g. pick it up). We use GeometricGoal rearrangement (GeoRearrange) [21] as our testbed.

In GeoRearrange (Fig. 1.2), an agent is initialized in a novel unseen environment and tasked with rearranging its environment. The task is specified as the initial location of an object $p_0$ and a final location $p^*$. The agent must bring the object located at $p_0$ to $p^*$. This task specification again has the benefit of allowing us to focus solely on the agents navigation and manipulation abilities.

The agent is equipped with an RGB camera, a Depth sensor, an egomotion sensor, and a proprioception sensor, which provides the agent the location of its limbs (e.g. arm).
Figure 1.2: In GeometricGoal rearrangement a robot is initialized in a novel unseen environment and tasked with rearranging it such that objects are placed at their target locations. This example shows the robot picking up an object, navigating to place it, and then placing it.

This is an even more challenging task. The robot action space is larger (2-dof to, e.g., 10-dof) and configuration space is now 3D instead of on a 2D manifold.

1.2 Thesis Statement

Massive-scale simulation and deep reinforcement learning lead to the emergence of intelligent navigation behavior.

1.2.1 Reinforcement Learning

Reinforcement Learning (RL) is concerned with decision making in Markov Decision Process (MDP). The tasks we study are formulated as a special case, a Partially Observable MDP (POMDP). In a POMDP, the agent receives an observation that does not fully specify the state \( s_t \) of the environment, \( o_t \) (e.g. an egocentric RGB image), takes an action \( a_t \), and is given a reward \( r_t \). The objective is to maximize cumulative reward over an episode. Formally, let \( \tau \) be a sequence of \((o_t, a_t, r_t)\) where \( a_t \sim \pi(\cdot \mid o_t) \), and \( s_{t+1} \sim \mathcal{T}(s_t, a_t) \). For a discount factor \( \gamma \), which balances the trade-off between exploration and exploitation, the optimal policy, \( \pi^* \), is specified by

\[
\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau \sim \pi} [R_T], \quad \text{where} \quad R_T = \sum_{t=1}^{T} \gamma^{t-1} r_t. \tag{1.1}
\]
1.2.2 Simulation

Simulation is a world for embodied agents to perform various tasks. Simulators should be much faster than real time, and support many tasks and datasets. To support these needs, my collaborators and I built AI Habitat [22, 23].

1.2.3 Massive-scale

Amount of training experience. The first aspect of massive-scale is the amount of training steps of experience, or number of training samples – the number of \((o_t, a_t, r_t)\) tuples collected.

Deep neural network size. The next aspect is the representational capacity of the neural network that parametrizes the agent. We use two pseudo-metrics for network capacity: The number of parameters and network depth.

Task diversity is both the number of tasks an agent is trained to perform and the number of scenarios (e.g. a different houses to navigate in or different objects to pick up) the agent is trained to operate under. We consider the latter.

Task complexity entails many aspects of the agents training task(s). This involves the action space of the agent (higher dimensional action spaces are more complex), the length of the task (longer tasks are more complex), the precision (surgery is more complex than picking an object from a bin), the abstraction of agent inputs (an RGB camera is more complex than a Depth sensor for navigation and a goal specified in language is more complex than a coordinate), the reasoning needed to complete the task (go is more complex than chess, which is in turn more complex than checkers), etc. We examine action space of the agent and the task precision.

1.2.4 Intelligent Behavior

We consider two categories of intelligent behavior. High performance on a challenging task and behavior that is similar to agents that we already deem intelligent.
For navigation, examples of intelligent behavior are near-perfect performance (e.g. reaching the goal nearly 100% of the time) or using mechanisms similar to animal (e.g. a mechanism similar to cognitive mapping [24]).

### 1.3 Summary of Contributions

In Chapter 2 we introduce Decentralized Distributed PPO (DD-PPO), a distributed reinforcement learning framework designed for training large neural networks in environments with photo-realistic rendering. DD-PPO is distributed (uses multiple machines), decentralized (lacks a centralized server), and synchronous (no computation is ever ‘stale’). In our experiments on training virtual robots for PointGoal navigation in Habitat-Sim [22], DD-PPO exhibits near-linear scaling – achieving a speedup of 107x on 128 GPUs over a serial implementation.

We leverage this scaling to train an agent for 2.5 Billion steps of experience (the equivalent of 80 years of human experience) – over 6 months of GPU-time training in under 3 days of wall-clock time with 64 GPUs. This massive-scale training not only sets the state of art on PointGoal navigation, but also essentially ‘solves’ the task – near-perfect autonomous navigation in an unseen environment without access to a map, directly from an RGB-D camera and a GPS+Compass sensor. Overall, we demonstrate that PointGoal navigation is learnable with generic components, i.e. a CNN+RNN agent and on-policy RL, when trained with billions of steps of experience.

This demonstrates intelligent navigation behavior via near-perfect task performance. While the agent was trained for navigation, its components and training procedure are generic off-the-shelf building blocks and don’t contain any priors for navigation. Only the reward (which isn’t available during evaluation) encodes anything about navigation. We examine 3 types of massive scale, amount of training experience (more experience leads to higher task performance), neural network size (larger networks perform better), and task diversity (more training environments leads to better generalization).
Figure 1.3: **In Chapter 2** we show the emergence of near-perfect navigation from massive-scale. **In Chapter 3** we show the emergence of mapping in ‘blind’ navigation agents from massive-scale. **In Chapter 4** we show the emergence of navigation in mobile manipulation agents from massive-scale.

In Chapter 3 we examine the inner workings of these PointGoalNav agents. Animal navigation research posits that organisms build and maintain internal spatial representations, or maps, of their environment. We ask if artificial intelligence (AI) navigation agents also build maps. Specifically, we train ‘blind’ AI agents – with sensing limited to only egomotion and no other sensing of any kind – to perform PointGoalNav. We find that ‘blind’ AI agents are (1) surprisingly effective navigators in new environments (∼95% success); (2) they utilize memory over long horizons (remembering ∼1,000 steps of past experience in an episode); (3) this memory enables them to take shortcuts; (4) there is emergence of maps in this memory, i.e. a detailed occupancy grid of the environment can be decoded from it; and (5) the emergent maps are selective and task dependent. Overall, our experiments and analysis show that blind AI agents take shortcuts and build maps purely from learning to navigate.

This demonstrates intelligent navigation behavior via the similar mechanisms ‘blind’ agents use for navigation and animals use for navigation.

In Chapter 4 we present Variable Experience Rollout (VER), a technique for scaling
batched on-policy reinforcement learning in heterogeneous environments (where different environments take vastly different times for generating rollouts). VER combines the strengths of and blurs the line between synchronous (SyncOnRL) and asynchronous (AsyncOnRL) on-policy RL methods – specifically, it learns from on-policy experience but has no synchronization points, enabling high throughput.

We find that VER leads to significant and consistent speed-ups across a broad range of embodied navigation and mobile manipulation tasks in photorealistic 3D simulation environments. Specifically, for PointGoal navigation and ObjectGoal navigation, VER is 60-100% faster (1.6-2x speedup) over DD-PPO. For mobile manipulation tasks (open fridge/cabinet, pick/place objects) in Habitat 2.0 VER is 150% faster (2.5x speedup) on 1 GPU and 200% faster (3x speedup) on 8 GPUs with similar or better sample efficiency. Compared to SampleFactory (AsyncOnRL), VER matches its speed on 1 GPU, and is 70% faster (1.7x speedup) on 8 GPUs with better sample efficiency.

We leverage these speed-ups to train chained skills for GeometricGoal rearrangement tasks in the Home Assistant Benchmark (HAB). We find a surprising emergence of navigation in skills that do not ostensibly require any navigation. Specifically, the pick skill involves a robot picking an object from a table. During training, the robot was always spawned close to the table and never needed to navigate. However, we find that if base movement is part of the action space, the robot learns to navigate then pick an object in new environments with 50% success, demonstrating surprisingly high out-of-distribution generalization.

This demonstrates intelligent navigation behavior along two axes. The skill policies are well-coordinated and use their navigation actions to achieve their goal. For instance, the pick policy uses navigation to move its arm in and out of the fridge as this is less likely to bump into the fridge. We also find the emergence of navigation itself. Specifically, skill polices that ostensibly didn’t need to navigate during training are able to use their emergent navigation to correct for handoff errors or replace the navigation skill itself.
We examine two aspects of massive scale, amount of training experience (more experience again leads to higher performance), and task complexity. GeoRearrange is a more complex task than PointGoalNav and we found both intelligent navigation behavior and the emergence of navigation itself.

The contributions are summarized visually in Fig. 1.3.

1.4 Contributed Papers

As part of this dissertation, we contribute the following papers:


E. Wijmans, M. Savva, S. Lee, I. Essa, A. Morcos, and D. Batra, “Blind artificial navigation agents take shortcuts and build maps,” In Submission, 2022 (Chapter 3)

E. Wijmans, I. Essa, and D. Batra, “Variable experience rollout: Learning robust skills for embodied rearrangement,” In Submission, 2022 (Chapter 4)

1.5 Other Papers

We have used and built upon the ideas presented in this dissertation in the following papers.

Simulators. In [22] we built Habitat 1.0 to enable massive-scale simulation for navigation agents. In [23] we built Habitat 2.0 to enable massive-scale simulation for rearrangement agents.

In [28] we proposed batched rendering. A batched renderer renders many agents simultaneously, enabling a 100x speed-up in training performance. In [29] we integrated this with a physics engine and created a suite of extremely fast to simulate, interactive, and 3D tasks.

Datasets. In [30] we created the Replica Dataset, as set of high quality 3D reconstructions of houses to foster embodied AI research. In [31] we created a dataset of 1,000 3D
reconstructions, the largest of its kind.

**PointGoal navigation.** In [32] we demonstrated that combining multiple auxiliary objectives improves sample efficiency when training PointGoalNav agents. In [33] we studied simple tips-and-tricks to train PointGoalNav agents on a sample- and compute-budget. In [34] we near-perfect navigation without the egomotion sensor.

**Semantic goal.** In [35] we proposed a human-annotation free data augmentation for training ObjectGoal navigation agents. In [36] we demonstrated that auxiliary tasks and exploration rewards are highly effective for ObjectGoalNav. In [37] we used a PointGoalNav agent to perform RoomNav by predicted the room location on a hallucinated map.

In [38] we created a dataset to train agents navigate to a location specified in natural language in continuous environments. In [39] we created a dataset for embodied question answering (EQA) large 3D reconstructions and studied various perception inputs.

**Sim-to-real.** In [40] we transferred PointGoalNav agents to reality and studied the correlation between performance in simulation and performance in reality. We found this is high with simple changes to the simulator.

The sources for these papers can be found online.
2.1 Introduction

Recent advances in deep reinforcement learning (RL) have given rise to systems that can outperform human experts at variety of games \[41, 42, 43\]. These advances, even more-so than those from supervised learning, rely on significant numbers of training samples, making them impractical without large-scale, distributed parallelization. Thus, scaling RL via multi-node distribution is of importance to AI – that is the focus of this work.

Several works have proposed systems for distributed RL \[44, 45, 42, 46, 43, 47\]. These works utilize two core components: 1) workers that collect experience (‘rollout workers’), and 2) a parameter server that optimizes the model. The rollout workers are then distributed across, potentially, thousands of CPUs\(^1\). However, synchronizing thousands of workers introduces significant overhead (the parameter server must wait for the \textit{slowest} worker, which can be costly as the number of workers grows). To combat this, they wait for only a few rollout workers, and then \textit{asynchronously} optimize the model.

However, this paradigm – of a single parameter server and thousands of (typically CPU) workers – appears to be fundamentally incompatible with the needs of modern computer vision and robotics communities. Over the last few years, a large number of works have proposed training virtual robots (or \textit{‘embodied agents’}) in rich 3D simulators before transferring the learned skills to reality \[49, 50, 51, 52, 53, 39, 22\]. Unlike Gym or Atari, 3D simulators require GPU acceleration, and, consequently, the number of workers is greatly limited (\(2^{5 \text{ to } 8}\) vs. \(2^{12 \text{ to } 15}\)). The desired agents operate from high dimensional inputs (pix-

\(^{1}\)Environments in OpenAI Gym \[48\] and Atari games can be simulated on solely CPUs.
Figure 2.1: Left: In PointGoal navigation, an agent must navigate from a random starting location (blue) to a target location (red) specified relative to the agent (“Go 5m north, 10m east of you”) in a previously unseen environment without access to a map. Right: Performance (SPL; higher is better) of an agent equipped with RGB-D and GPS+Compass sensors on the Habitat Challenge 2019 [22] train & val sets. Using DD-PPO, we train agents for over 180 days of GPU-time in under 3 days of wall-clock time with 64 GPUs, achieving state-of-art results and ‘solving’ the task.

Contributions. We propose a simple, synchronous, distributed RL method that scales well. We call this method Decentralized Distributed Proximal Policy Optimization (DD-PPO) as it is decentralized (has no parameter server), distributed (runs across many different machines), and we use it to scale Proximal Policy Optimization [54].

In DD-PPO, each worker alternates between collecting experience in a resource-intensive and GPU accelerated simulated environment and optimizing the model. This distribution is synchronous – there is an explicit communication stage where workers synchronize their updates to the model (the gradients). To avoid delays due to stragglers, we propose a pre-emption threshold where the experience collection of stragglers is forced to end early once a pre-specified percentage of the other workers finish collecting experience. All workers then begin optimizing the model.

We characterize the scaling of DD-PPO by the steps of experience per second with
N workers relative to 1 worker. We consider two different workloads, 1) simulation time is roughly equivalent for all environments, and 2) simulation time can vary dramatically due to large differences in environment complexity. Under both workloads, we find that DD-PPO scales near-linearly. While we only examined our method with PPO, other on-policy RL algorithms can easily be used and we believe the method is general enough to be adapted to off-policy RL algorithms.

We leverage these large-scale engineering contributions to answer a key scientific question arising in embodied navigation. [20] benchmarked classical (mapping + planning) and learning-based methods for agents with RGB-D and GPS+Compass sensors on PointGoal navigation [13] (PointGoalNav), see Fig. 2.1, and showed that classical methods outperform learning-based. However, they trained for ‘only’ 5 million steps of experience. [22] then scaled this training to 75 million steps and found that this trend reverses – learning-based outperforms classical, even in unseen environments! However, even with an order of magnitude more experience (75M vs 5M), they found that learning had not yet saturated. This begs the question – what are the fundamental limits of learnability in PointGoalNav? Is this task entirely learnable? We answer this question affirmatively via an ‘existence proof’.

Utilizing DD-PPO, we find that agents continue to improve for a long time (Fig. 2.1) – not only setting the state of art in Habitat Autonomous Navigation Challenge 2019 [22], but essentially ‘solving’ PointGoalNav (for agents with GPS+Compass). Specifically, these agents 1) almost always reach the goal (failing on 1/1000 val episodes on average), and 2) reach it nearly as efficiently as possible – nearly matching (within 3% of) the performance of a shortest-path oracle! It is worth stressing how uncompromising that comparison is – in a new environment, an agent navigating without a map traverses a path nearly matching the shortest path on the map. This means there is no scope for mistakes of any kind – no wrong turn at a crossroad, no back-tracking from a dead-end, no exploration or deviation of any kind from the shortest-path. Our hypothesis is that the model learns to
Figure 2.2: Comparison of asynchronous distribution (left) and synchronous distribution via distributed data parallelism (right) for RL. Left: rollout workers collect experience and asynchronously send it to the parameter-server. Right: a worker alternates between collecting experience, synchronizing gradients, and optimization. We find this highly effective in resource-intensive environments.

exploit the statistical regularities in the floor-plans of indoor environments (apartments, offices) in our datasets. The more challenging task of navigating purely from an RGB camera without GPS+Compass demonstrates progress but remains an open frontier.

Finally, we show that the scene understanding and navigation policies learned on PointGoalNav can be transferred to other tasks (Flee and Explore [55]) – the analog of ‘ImageNet pre-training + task-specific fine-tuning’ for Embodied AI. Our models are able to rapidly learn these new tasks (outperforming ImageNet pre-trained CNNs) and can be utilized as near-perfect neural PointGoal controllers, a universal resource for other high-level navigation tasks [53, 51]. We make code and trained models publicly available.

2.2 Preliminaries: RL and PPO

Reinforcement Learning (RL) is concerned with decision making in Markov Decision Process (MDP). In a Partially Observable MDP (POMDP), the agent receives an observation that does not fully specify the state \( s_t \) of the environment, \( o_t \) (e.g. an egocentric RGB image), takes an action \( a_t \), and is given a reward \( r_t \). The objective is to maximize cumulative reward over an episode. Formally, let \( \tau \) be a sequence of \((o_t, a_t, r_t)\) where \( a_t \sim \pi(\cdot | o_t) \), and \( s_{t+1} \sim T(s_t, a_t) \). For a discount factor \( \gamma \), which balances the trade-off between explo-
ration and exploitation, the optimal policy, $\pi^*$, is specified by

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau \sim \pi} [R_T], \quad \text{where, } R_T = \sum_{t=1}^{T} \gamma^{t-1} r_t. \quad (2.1)$$

One technique to find $\pi^*$ is Proximal Policy Optimization (PPO) [54], an on-policy algorithm in the policy-gradient family. Given a $\theta$-parameterized policy $\pi_\theta$ and a set of trajectories collected with it (commonly referred to as a ‘rollout’), PPO updates $\pi_\theta$ as follows. Let $\hat{A}_t = R_t - \hat{V}_t$, be the estimate of the advantage, where $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$, and $\hat{V}_t$ is the expected value of $R_t$, and $r_t(\theta) = \frac{\pi_\theta(a_t|o_t)}{\pi_\gamma(a_t|o_t)}$ be the ratio of the probability of the action $a_t$ under the current policy and the policy used to collect the rollout. The parameters are then updated by maximizing

$$J^{PPO}(\theta) = E_t \left[ \min \left( \underbrace{r_t(\theta)\hat{A}_t}_{\text{importance-weighted advantage}}, \underbrace{\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t}_{\text{proximity clipping term}} \right) \right] \quad (2.2)$$

This clipped objective keeps this ratio within $\epsilon$ and functions as a trust-region optimization method; allowing for the multiple gradient updates using the rollout, thereby improving sample efficiency.

### 2.3 Decentralized Distributed Proximal Policy Optimization

In reinforcement learning, the dominant paradigm for distribution is asynchronous (see Fig. 2.2). Asynchronous distribution is notoriously difficult – even minor errors can result in opaque crashes – and the parameter server and rollout workers necessitate separate programs.

In supervised learning, however, synchronous distributed training via data parallelism [56] dominates. As a general abstraction, this method implements the following: at step $k,$
worker $n$ has a copy of the parameters, $\theta_n^k$, calculates the gradient, $\partial \theta_n^k$, and updates $\theta$ via

$$\theta_n^{k+1} = \text{ParamUpdate}(\theta_n^k, \text{AllReduce}(\partial \theta_1^k, \ldots, \partial \theta_N^k)) = \text{ParamUpdate}(\theta_n^k, \frac{1}{N} \sum_{i=1}^N \partial \theta_i^k),$$

(2.3)

where ParamUpdate is any first-order optimization technique (e.g., gradient descent) and AllReduce performs a reduction (e.g., mean) over all copies of a variable and returns the result to all workers. Distributed DataParallel scales very well (near-linear scaling up to 32,000 GPUs [57]), and is reasonably simple to implement (all workers synchronously running identical code).

We adapt this to on-policy RL as follows: At step $k$, a worker $n$ has a copy of the parameters $\theta_n^k$; it gathers experience (rollout) using $\pi_{\theta_n^k}$, calculates the parameter-gradients $\nabla \theta$ via any policy-gradient method (e.g., PPO), synchronizes these gradients with other workers, and updates the model:

$$\theta_n^{k+1} = \text{ParamUpdate}(\theta_n^k, \text{AllReduce}\left(\nabla_{\theta} J^{PPO}(\theta_1^k), \ldots, \nabla_{\theta} J^{PPO}(\theta_N^k)\right)).$$

(2.4)

A key challenge to using this method in RL is variability in experience collection runtime. In supervised learning, all gradient computations take approximately the same time. In RL, some resource-intensive environments can take significantly longer to simulate. This introduces significant synchronization overhead as every worker must wait for the slowest to finish collecting experience. To combat this, we introduce a preemption threshold where the rollout collection stage of these stragglers is preempted (forced to end early) once some percentage, $p\%$, (we find 60% to work well) of the other workers are finished collecting their rollout; thereby dramatically improving scaling. We weigh all worker’s contributions to the loss equally and limit the minimum number of steps before preemption to one-fourth the maximum to ensure all environments contribute to learning.
Figure 2.3: Our agent for PointGoalNav. At very time-step, the agent receives an ego-centric Depth or RGB (shown here) observation, utilizes its GPS+Compass sensor to update the target position to be relative to its current position, and outputs the next action and an estimate of the value function.

While we only examined our method with PPO, other on-policy RL algorithms can easily be used and we believe the method can be adapted to off-policy RL algorithms. Off-policy RL algorithms also alternate between experience collection and optimization, but differ in how experience is collected/used and the parameter update rule. Our adaptations simply add synchronization to the optimization stage and a preemption to the experience collection stage.

**Implementation.** We leverage PyTorch’s DistributedDataParallel to synchronize gradients, and TCPStore – a simple distributed key-value storage – to track how many workers have finished collecting experience. See Apx. A.4 for a detailed description with code.

### 2.4 Experimental Setup: PointGoal navigation, Agents, Simulator

**PointGoal Navigation** (PointGoalNav). An agent is initialized at a random starting position and orientation in a new environment and asked to navigate to target coordinates specified relative to the agent’s position; no map is available and the agent must navigate using only its sensors – in our case RGB-D (or RGB) and GPS+Compass (providing current position and orientation relative to start).

The evaluation criteria for an episode is as follows [13]: Let $S$ indicate ‘success’ (did the agent stop within 0.2 meters of the target?), $l$ be the length of the shortest path be-
tween start and target, and \( p \) be the length of the agent’s path, then Success weighted by (normalized inverse) Path Length SPL = \( S \frac{l}{\max(l,p)} \). It is worth stressing that SPL is a highly punitive metric – to achieve SPL = 1, the agent (navigating without the map) must match the performance of the shortest-path oracle that has access to the map! There is no scope for any mistake – no wrong turn at a crossroad, no back-tracking from a dead-end, no exploration or deviation from the shortest path. In general, this may not even be possible in a new environment (certainly not if an adversary designs the map).

**Agent.** As in [22], the agent has 4 actions, stop, which indicates the agent has reached the goal, move_forward (0.25m), turn_left (10\(^\circ\)), and turn_right (10\(^\circ\)). It receives 256x256 sized images and uses the GPS+Compass to compute target coordinates relative to its current state. The RGB-D agent is limited to only Depth as [22] found this to perform best.

Our agent architecture (Fig. 2.3) has two main components – a visual encoder and a policy network.

The visual encoder is based on either ResNet [59] or SE [60]-ResNeXt [61] with the number of output channels at every layer reduced by half. We use a first layer of 2x2-AvgPool to reduce resolution (essentially performing low-pass filtering + down-sampling) – we find this to have no impact on performance while allowing faster training. From our initial experiments, we found it necessary to replace every BatchNorm layer [62] with GroupNorm [63] to account for highly correlated inputs seen in on-policy RL.

The policy is parameterized by a 2-layer LSTM with a 512-dimensional hidden state. It takes three inputs: the previous action, the target relative to the current state, and the output of the visual encoder. The LSTM’s output is used to produce a softmax distribution over the action space and an estimate of the value function. See Appendix A.2 for full details.

**Training.** We use PPO with Generalized Advantage Estimation [64]. We set the discount factor \( \gamma \) to 0.99 and the GAE parameter \( \tau \) to 0.95. Each worker collects (up to) 128 frames of experience from 4 agents running in parallel (all in different environments) and then performs 2 epochs of PPO with 2 mini-batches per epoch. We use Adam [65] with a
learning rate of $2.5 \times 10^{-4}$. Unlike popular implementations of PPO, we do not normalize advantages as we find this leads to instabilities. We use DD-PPO to train with 64 workers on 64 GPUs.

The agent receives terminal reward $r_T = 2.5$ SPL, and shaped reward $r_t(a_t, s_t) = -\Delta_{\text{geo, dist}} - 0.01$, where $\Delta_{\text{geo, dist}}$ is the change in geodesic distance to the goal by performing action $a_t$ in state $s_t$.

**Simulator+Datasets.** Our experiments are conducted using Habitat, a 3D simulation platform for embodied AI research [22]. Habitat is a modular framework with a highly performant and stable simulator, making it an ideal framework for simulating billions of steps of experience.

We experiment with several different sources of data. First, we utilize the training data released as part of the Habitat Challenge 2019, consisting of 72 scenes from the Gibson dataset [66]. We then augment this with all 90 scenes in the Matterport3D dataset [67] to create a larger training set (note that Matterport3D meshes tend to be larger and of better quality). Furthermore, [22] curated the Gibson dataset by rating every mesh reconstruction on a quality scale of 0 to 5 and then filtered all splits such that each only contains scenes with a rating of 4 or above (Gibson-4+), leaving all scenes with a lower rating previously unexplored. We examine training on the 332 scenes from the original train split with a rating of 2 or above (Gibson-2+).

### 2.5 Benchmarking: How does DD-PPO scale?

In this section, we examine how DD-PPO scales under two different workload regimes – homogeneous (every environment takes approximately the same amount of time to simulate) and heterogeneous (different environments can take orders of magnitude more/less time to simulate). We examine the number of steps of experience per second with $N$ workers relative to 1 worker. We compare different values of the preemption threshold $p\%$.

\[ \text{We use all Matterport3D scenes (including test and val) as we only evaluate on Gibson validation and test.} \]
Figure 2.4: Scaling performance (in steps of experience per second relative to 1 GPU) of DD-PPO for various preemption threshold, $p\%$, values. Shading represents a 95% confidence interval.

We benchmark training our ResNet50 PointGoalNav agent with Depth on a cluster with Nvidia V100 GPUs and NCCL2.4.7 with Infiniband interconnect.

**Homogeneous.** To create a homogeneous workload, we train on scenes from the Gibson dataset, which require very similar times to simulate agent steps. As shown in Fig. 2.4 (left), DD-PPO exhibits near-linear scaling (linear = ideal) for preemption thresholds larger than 50%, achieving a 196x speed up with 256 GPUs relative to 1 GPU and an 7.3x speed up with 8 GPUs relative to 1.

**Heterogeneous.** To create a heterogeneous workload, we train on scenes from both Gibson and Matterport3D. Unlike Gibson, MP3D scenes vary significantly in complexity and time to simulate – the largest contains 8GB of data while the smallest is only 135MB. DD-PPO scales poorly at a preemption threshold of 100% (no preemption) due to the substantial straggler effect (one rollout taking substantially longer than the others); see Fig. 2.4 (right). However, with a preemption threshold of 80% or 60%, we achieve near-identical scaling to the homogeneous workload! We found no degradation in performance of models trained with any of these values for the preemption threshold despite learning in large scenes occurring at a lower frequency.
2.6 Mastering PointGoal navigation with GPS+Compass

In this section, we answer the following questions: 1) What are the fundamental limits of learnability in PointGoalNav navigation? 2) Do more training scenes improve performance? 3) Do better visual encoders improve performance? 4) Is PointGoalNav ‘solvable’ when navigating from RGB instead of Depth? 5) What are the open/unsolved problems – specifically, how does navigation without GPS+Compass perform? 6) Can agents trained for PointGoalNav be transferred to new tasks?

Agents continue to improve for a long time. Using DD-PPO, we train agents for 2.5 Billion steps of experience with 64 Tesla V100 GPUs in 2.75 days – 180 GPU-days of training, the equivalent of 80 years of human experience (assuming 1 human second per step). As a comparison, [22] reached 75 million steps (an order of magnitude more than prior work) in 2.5 days using 2 GPUs – at that rate, it would take them over a month (wall-clock time) to achieve the scale of our study. Fig. 2.1 shows the performance of an agent with RGB-D and GPS+Compass sensors, utilizing an SE-ResNeXt50 visual encoder, trained on Gibson-2+ – it does not saturate before 1 billion steps, suggesting that previous studies were incomplete by 1-2 orders of magnitude. Fortuitously, error vs computation exhibits a power-law-like distribution; 90% of peak performance is obtained relatively early (100M steps) and relatively cheaply (in 0.1 day with 64 GPUs and in 1 day with 8 GPUs). Also noteworthy in Fig. 2.1 is the strong generalization (train to val) and corresponding lack of overfitting.

Increasing training data helps. Table 2.1 presents results with different training datasets and visual encoders for agent with RGB-D and GPS+Compass. Our most basic setting (ResNet50, Gibson-4+ training) already achieves SPL of 0.922 (val), 0.917 (test), which nearly misses (by 0.003) the top of the leaderboard for the Habitat Challenge 2019 RGB-D track. Next, we increase the size of the training data by adding in all Matterport3D scenes.

---

3These trends are consistent across sensors (RGB), training datasets (Gibson-4+), and visual encoders.
4The current on-demand price of an 8-GPU AWS instance (p2.8xlarge) is $7.2/hr, or $172.8 for 1 day.
5https://evalai.cloudcv.org/web/challenges/challenge-page/254/leaderboard/839

22
Table 2.1: Performance (higher is better) of different architectures for agents with RGB-D and GPS+Compass sensors on the Habitat Challenge 2019 [22] validation and test-std splits (checkpoint selected on val). 10 samples taken for each episode on val. Gibson-4+ (2+) refers to the subset of Gibson train scenes [66] with a quality rating of 4 (2) or higher. See Table A.1 for results of the best DD-PPO agent for Blind, RGB, and RGB-D and other baselines.

<table>
<thead>
<tr>
<th>Training Dataset</th>
<th>Agent Visual Encoder</th>
<th>Validation</th>
<th>Test Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SPL</td>
<td>Success</td>
</tr>
<tr>
<td>Gibson-4+</td>
<td>ResNet50</td>
<td>0.922 ± 0.004</td>
<td>0.967 ± 0.003</td>
</tr>
<tr>
<td>Gibson-4+ and MP3D</td>
<td>ResNet50</td>
<td>0.956 ± 0.002</td>
<td>0.996 ± 0.002</td>
</tr>
<tr>
<td>Gibson-2+</td>
<td>ResNet50</td>
<td>0.956 ± 0.003</td>
<td>0.994 ± 0.002</td>
</tr>
<tr>
<td></td>
<td>SE-ResNeXt50</td>
<td>0.950 ± 0.002</td>
<td>0.999 ± 0.001</td>
</tr>
<tr>
<td></td>
<td>SE-ResNeXt101 + 1024-d LSTM</td>
<td>0.969 ± 0.002</td>
<td>0.997 ± 0.001</td>
</tr>
</tbody>
</table>

and see an improvement of \(\sim 0.03\) SPL – to 0.956 (val), 0.941 (test). Next, we compare training on Gibson-4+ and Gibson-2+. Recall that Gibson-{2, 3} corresponds to poorly reconstructed scenes (see Fig. A.6). A priori, it is unclear whether the net effect of this addition would be positive or negative; adding them provides diverse experience to the agent, however, it is poor quality data. We find a potentially counter-intuitive result – adding poor 3D reconstructions to the train set improves performance on good reconstructions in val/test by \(\sim 0.03\) SPL – from 0.922 (val), 0.917 (test) to 0.956 (val), 0.944 (test). Our conjecture is that training on poor (Gibson-{2, 3}) and good (4+) reconstructions leads to robustness in representations learned.

**Better visual encoders and more parameters help.** Using a better visual encoder, SE [60]-ResNeXt50 [61] instead of ResNet50, improves performance by 0.003 SPL (Table 2.1). Adding capacity to the visual encoder (SE-ResNeXt101 vs SE-ResNeXt50) and navigation policy (1024-d vs 512-d LSTM) further improves performance by 0.010 SPL.

**PointGoalNav ‘solved’ with RGB-D and GPS+Compass.** Our best agent – SE-ResNeXt101 + 1024-d LSTM trained on Gibson-2+ – achieves SPL of 0.969 (val), 0.948 (test), which not only sets the state of art on the Habitat Challenge 2019 RGB-D track but is also within 3-5% of the shortest-path oracle\(^6\). Given the challenges with achieving near-perfect SPL in

\(^6\)Videos: https://www.youtube.com/watch?v=x3fk-Ylb_7s&list=UUKkMUbmP7atzznCo0LXynlA
new environments, it is important to dig deeper. Fig. A.8 shows (a) distribution of episode lengths in val and (b) SPL vs episode length. We see that while the dataset is dominated by short episodes (2-12m), the performance of the agent is remarkably stable over long distances and average SPL is not necessarily inflated. Our hypothesis is the agent has learned to exploit the structural regularities in layouts of real indoor environments. One (admittedly imperfect) way to test this is by training a Blind agent with only a GPS+Compass sensor.

Fig. A.8 shows that this agent is able to handle short-range navigation (which primarily involve turning to face the target and walking straight) but performs very poorly on longer trajectories – SPL of 0.3 (Blind) vs 0.95 (RGB-D) at 20-25m navigation. Thus, structural regularities, in part, explain performance for short-range navigation. For long-range navigation, the RGB-D agent is extracting overwhelming signal from its Depth sensor. We repeat this analysis on two additional navigation datasets proposed by [68] – longer episodes and ‘harder’ episodes (more navigation around obstacles) – and find similar trends (Fig. A.9). This discussion continues in Apx. A.1.

**Performance with RGB is also improved.** So far we studied RGB-D as this performed best in [22]. We now study RGB (with SE-ResNeXt50 encoder). We found it crucial to train on Gibson-2+ and all of Matterport3D, ensuring diversity in both layouts (Gibson-2+) and appearance (Matterport3D), and to channel-wise normalize RGB (subtract by mean and divide by standard deviation) as our networks lack BatchNorm. Performance improves dramatically from 0.57 (val), 0.47 (test) SPL in [22] to near-perfect success 0.991 (val), 0.977 (test) and high SPL 0.929 (val), 0.920 (test). While SPL is considerably lower than the Depth agent, (0.929 vs 0.959), interestingly, the RGB agent still reaches the goal a similar percentage of the time (99.1% vs 99.9%). This agent achieves state-of-art on the Habitat Challenge 2019 RGB track (rank 2 entry has 0.89 SPL).5

**No GPS+Compass remains unsolved.** Finally, we examine if we also achieve better performance on the significantly more challenging task of navigation from RGB without GPS+Compass. At 100 million steps (an amount equivalent to [22]), the agent achieves 0
Figure 2.5: Performance (higher is better) on Flee (left) and Exploration (right) under five settings.

SPL. By training to 2.5 billion steps, we make some progress and achieve 0.15 SPL. While this is a substantial improvement, the task continues to remain an open frontier for research in embodied AI.

**Transfer Learning.** We examine transferring our agents to the following tasks [55]

- **Flee** The agent maximizes its geodesic distance from its starting location. Let $s_t$ be the agent’s position at time $t$, and $\text{Max}(s_0)$ denote the maximum distance over all reachable points, then the agent maximizes $D_T = \text{Geo}(s_T, s_0)/\text{Max}(s_0)$. The reward is $r_t = 5(D_t - D_{t-1})$.

- **Exploration** The agent maximizes the number of locations (specified by 1m cubes) visited. Let $|\text{Visited}_t|$ denote the number of location visited at time $t$, then the agent maximizes $|\text{Visited}_T|$. The reward is $r_t = 0.25(|\text{Visited}_t| - |\text{Visited}_{t-1}|)$.

We use a PointGoalNav-trained agent with RGB and GPS+Compass, remove the GPS+Compass, and transfer to these tasks under five different settings:

- **Scratch.** All parameters (visual encoder + policy) are trained from scratch for each new task. Improvements over this baseline demonstrate benefits of transfer learning.

- **ImageNetEncoder-ScratchPolicy.** The visual encoder is initialized with ImageNet pretrained weights and frozen; the navigation policy is trained from scratch.

- **PointGoalNavEncoder-ScratchPolicy.** The visual encoder is initialized from PointGoalNav and frozen; the navigation policy is trained from scratch.

- **PointGoalNavEncoder-FinetunePolicy.** Both visual encoder and policy parameters are initialized from PointGoalNav (critic layers are reinitialized). Encoder is frozen, policy
is fine-tuned.\footnote{Since a \texttt{PointGoalNav} policy expects a goal-coordinate, we input a ‘dummy’ arbitrarily-chosen vector for the transfer tasks, which the agent quickly learns to ignore.}

- \textbf{∇ Neural Controller} We treat our agent as a \textit{differentiable neural controller}, a closed-loop low-level controller than can navigate to a specified coordinate. We utilize this controller in a new task by training a light-weight high-level planner that predicts a goal-coordinate (at each time-step) for the controller to navigate to. Since the controller is fully differentiable, we can backprop through it. We freeze the controller, train the planner+controller system with PPO for the new task. The planner is a 2-layer LSTM and shares the (frozen) visual encoder with the controller.

Fig. 2.5 shows performance vs. experience results (higher is better). Nearly all methods outperform learning from scratch, establishing the value of transfer learning. \texttt{PointGoalNav} pre-trained visual encoders dramatically outperforms ImageNet pre-trained ones, indicating that the agent has learned generally useful scene understanding. For both tasks, fine-tuning an existing policy allows it to rapidly learn the new task, indicating that the agent has learned general navigation skills. ∇\texttt{Neural Controller} outperforms \texttt{PointGoalNavEncoder-ScratchPolicy} on Flee and is competitive on Exploration, indicating that the agent can indeed be ‘controlled’ or directed to target locations by a planner. Overall, these results demonstrate that our trained model is useful for more than just \texttt{PointGoalNav}.

\subsection{2.7 Related Work}

\textbf{Visual Navigation.} Visual navigation in indoor environments has been the subject of many recent works \cite{69, 51, 53, 22, 20}. Our primary contribution is DD-PPO, thus we discuss other distributed works.

In the general case, computation in reinforcement learning (RL) in simulators can be broken down into 4 roles: 1) Simulation: Takes actions performed by the agent as input, simulates the new state, returns observations, reward, \textit{etc}. 2) Inference: Takes observations as input and utilizes the agent policy to return actions, value estimate, \textit{etc}. 3) Learner:
Takes rollouts as input and computes gradients to update the policy’s parameters. 4) Parameter server/master: Holds the source of truth for the policy’s parameters and coordinates workers.

**Synchronous RL.** Synchronous RL systems utilize a single process to perform all four roles; this design is found in RL libraries like OpenAI Baselines [70] and PytorchRL [71]. This method is limited to a single node’s worth of GPUs.

**Synchronous Distributed RL.** The works most closely related to DD-PPO also propose to scale synchronous RL by replicating this simulation/inference/learner process across multiple GPUs and then synchronize gradients with AllReduce. [72] experiment with Atari and find it not effective however. We hypothesize that this is due to a subtle difference – this distribution design relies on a single worker collecting experience from multiple environments, stepping through them in lock step. This introduces significant synchronization and communication costs as every step in the rollout must be synchronized across as many as 64 processes (possible because each environment is resource-light, e.g. Atari). For instance, taking 1 step in 8 parallel pong environments takes approximately the same wall-clock time as 1 pong environment, but it takes 10 times longer to take 64 steps in lock-step; thus gains from parallelization are washed out due to the lock-step synchronization. In contrast, we study resource-intensive environments, where only 2 or 4 environments per worker is possible, and find this technique to be effective. [73] mirror our findings (this distribution method can be effective for resource intensive simulation) in GPU-accelerated physics simulation, specifically MuJoCo [74] with NVIDIA Flex. In contrast to our work, they examine scaling up to only 32 GPUs and only for homogeneous workloads. In contrast to both, we propose an adaption to mitigate the straggler effect – preempting the experience collection (rollout) of stragglers and then beginning optimization. This improves scaling for homogeneous workloads and dramatically improves scaling for heterogeneous workloads.

**Asynchronous Distributed RL.** Existing public frameworks for asynchronous distributed
reinforcement learning [44, 45, 47] use a single (CPU-only) process to perform the simulation and inference roles (and then replicate this process to scale). A separate process asynchronously performs the learner and parameter server roles (note it’s not clear how to use more than one these processes as it holds the source of truth for the parameters). Adapting these methods to the resource-intensive environments studied in this work (e.g. Habitat [22]) encounters the following issues: 1) Limiting the inference/simulation processes to CPU-only is untenable (deep networks and need for GPU-accelerated simulation). While the inference/simulation processes could be moved to the GPU, this would be ineffective for the following: GPUs operate most efficiently with large batch sizes (each inference/simulation process would have a batch size of 1), CUDA runtime requires $\sim$600MB of GPU memory per process, and only one CUDA kernel (function that runs on the GPU) can executed by the GPU at a time. These issue contribute and lead to low GPU utilization. In contrast, DD-PPO utilizes a single process per GPU and batches observations from multiple environments for inference. 2) The single process learner/parameter server is limited to a single node’s worth of GPUs. While this not a limitation for small networks and low dimensional inputs, our agents take high dimensional inputs (e.g. a Depth sensor) and utilize large neural networks (ResNet50), thereby requiring considerable computation to compute gradients. In contrast, DD-PPO has no parameter server and every GPU computes gradients, supporting even very large networks (SE-ResNeXt101).

**Straggler Effect Mitigation.** In supervised learning, the straggler effect is commonly caused by heterogeneous hardware or hardware failures. [75] propose a pool of $b$ “back-up” workers (there are $N + b$ workers total) and perform the parameter update once $N$ workers finish. In comparison, their method a) requires a parameter server, and b) discards all work done by the stragglers. [76] propose to dynamically adjust the batch size of each worker such that all workers perform their forward and backward pass in the same amount of time. Our method aims to reduce variance in experience collection times. DD-PPO dynamically adjusts a worker’s batch size as a necessary side-effect of preempting experience collection.
Distributed Synchronous SGD. Data parallelism is a common paradigm in high performance computing [56]. In this paradigm, parallelism is achieved by workers performing the same work on different data. This paradigm can be naturally adapted to supervised deep learning [75]. Works have used this to achieve state-of-the-art results in tasks ranging from computer vision [77, 4] to natural language processing [78, 79, 80]. Furthermore, multiple deep learning frameworks provide simple-to-use wrappers supporting this parallelism model [58, 81, 82]. We adapt this framework to reinforcement learning.
CHAPTER 3
BLIND AGENTS TAKE SHORTCUTS AND BUILD MAPS

3.1 Introduction

Decades of research into intelligent animal navigation posits that organisms build and maintain internal spatial representations (or maps)\(^1\) of their environment, that enables the organism to determine and follow task-appropriate paths [83, 24, 84]. Hamsters, wolves, chimpanzees, and bats leverage prior exploration to determine and follow shortcuts they may never have taken before [85, 86, 87, 88, 89]. Even blind mole rats and animals rendered situationally-blind in dark environments demonstrate shortcut behaviors [90, 91, 92]. Ants forage for food along meandering paths but take near-optimal return trips [93], though there is some controversy about whether insects like ants and bees are capable of forming maps [94, 95].

Analogously, mapping and localization techniques have long played a central role in enabling non-biological navigation agents (or robots) to exhibit intelligent behavior [16, 17, 18, 19]. More recently, the machine learning community has produced a surprising phenomenon – neural-network models for navigation that curiously do not contain any explicit mapping modules but still achieve remarkably high performance [22, 25, 96, 97, 98, 99]. The mechanisms explaining this ability remain unknown. Understanding them is both of scientific and practical importance due to safety considerations involved with deploying such systems.

In this work, we investigate the following question – is mapping an emergent phenomenon? Specifically, do artificial intelligence (AI) agents learn to build internal spatial

---

\(^1\)Throughout this work, we use ‘maps’ to refer to a spatial representation of the environment that enables intelligent navigation behavior like taking shortcuts. We provide a detailed discussion and contrast w.r.t. a ‘cognitive map’ as defined by O’Keefe and Nadel [24] in the text.
representations (or ‘mental’ maps) of their environment as a natural consequence of learning to navigate?

The specific task we study is PointGoal navigation [13], where an AI agent is introduced into a new (unexplored) environment and tasked with navigating to a relative location – ‘go 5m north, 2m west relative to start’. This is analogous to the direction and distance of foraging locations communicated by the waggle dance of honey bees [14].

Unlike animal navigation studies, experiments with AI agents allow us to precisely isolate mapping from alternative mechanisms proposed for animal navigation – the use of visual landmarks [14], orientation by the arrangement of stars [100], gradients of olfaction or other senses [101]. We achieve this isolation by judiciously designing the agent’s perceptual system and the learning paradigm such that these alternative mechanisms are rendered implausible. Our AI agents are effectively ‘blind’; they possess a minimal perceptual system capable of sensing only egomotion, i.e. change in the agent’s location and orientation as it moves – no vision, no audio, no olfactory, no haptic, no magnetic, or any other sensing of any kind. This perceptual system is deliberately impoverished to isolate the contribution of memory, and is inspired by blind mole rats, who perform localization via path integration and use the Earth’s magnetic field as a compass [91]. Further still, our AI agents are composed of navigation-agnostic, generic, and ubiquitous architectural components (fully-connected layers and LSTM-based recurrent neural networks), and our experimental setup provides no inductive bias towards mapping – no map-like or spatial structural components in the agent, no mapping supervision, no auxiliary tasks, nothing other than the task of navigation to a goal.

Surprisingly, even under these deliberately harsh conditions, we find the emergence of map-like spatial representations in the agent’s non-spatial unstructured memory, enabling it to not only successfully navigate to the goal but also exhibit intelligent behavior (like taking shortcuts, following walls, detecting collisions) similar to aforementioned animal

---

2The description in English is purely for explanatory purposes; the agent receives relative goal coordinates.
studies, and predict free-space in the environment. Essentially, we demonstrate an ‘existence proof’ or an ontogenetic developmental account for mapping without any previous predisposition. Our results also explain the aforementioned surprising finding in recent literature [22, 25, 96, 97, 98, 99] – that ostensibly map-free neural-network achieve strong autonomous navigation performance – by demonstrating that these systems in fact learn to construct and maintain map-like representations of their environment.

Overall, our results suggest that mapping may be a *natural solution* to the problem of navigation by intelligent embodied agents, whether they be biological or artificial.

**Summary of Findings:** Concretely, we ask and answer following questions:

1. *‘Is it possible to effectively navigate with just egomotion sensing?’*

   Yes. We find that our ‘blind’ AI agents are *highly* effective in navigating new environments – reaching the goal with 95.1%±1.3% success rate. And they traverse moderately efficient (though far from optimal) paths, reaching 62.9%±1.6% of optimal path efficiency. We stress that these are unseen testing environments, the agent has not (just) memorized paths within a training environment but has learned efficient navigation strategies (wall-following, detecting collisions, *etc.*.) that generalize to novel environments.

2. *‘What mechanism explains this strong performance by ‘blind’ AI agents?’*

   Memory. We find that memoryless AI agents completely fail at this task, achieving nearly 0% success. More importantly, we find that agents with memory utilize information stored over a long temporal and spatial horizon. Navigation performance as a function of the number of past actions/observations encoded in the agent’s memory does not saturate till one thousand steps (corresponding to the agent traversing 89.1±0.66 meters).

3. *‘What information does the memory encode about the environment?’*

   Implicit maps. We perform an AI rendition of Menzel’s experiments [87], where a
chimpanzee is carried and shown the location of food hidden in the environment. The animal is then returned to the starting location and allowed to collect food. It does not simply retrace the human demonstrator’s steps but takes shortcuts to collect the food faster.

Analogously, we train a blind AI agent to navigate from a source location (S) to a target location (T). After it has finished navigating, we transplant its constructed episodic memory into a second ‘probe’-agent (which is also blind). We find that this implanted-memory probe-agent performs dramatically better in navigating from S to T (and T to S) than it would without the memory transplant. Similar to the chimpanzee, the probe agent takes shortcuts, typically cutting out backtracks or excursions that the memory-creator had undertaken as it tried to work its way around the obstacles. These experiments provide compelling evidence that blind AI agents learn to build and use implicit map-like representations of their environment solely through learning to navigate. Intriguingly further still, we find that surprisingly detailed metric occupancy maps of the environment (indicating free-space) can be explicitly decoded from the agent’s memory.

4. ‘Are maps task-dependent?’

Yes. We find that the emergent maps are a function of the navigation goal. Agents ‘forget’ excursions and detours, i.e. their episodic memory only preserves the features of the environment relevant to navigating to their goal. This, in part, explains why transplanting episodic memory from one agent to another leads it to take shortcuts – because the excursion and detours are simply forgotten.

Overall, our experiments and analyses demonstrate that ‘blind’ AI agents solve PointGoalNav. Further, that they do so by combining information over long time horizons to build detailed, task-dependent maps of their environment, solely through the learning signals imposed by goal-driven navigation.
GPS+Compass

Figure 3.1: **PointGoal navigation.** An agent is initialized in a novel environment (bluesquare) and tasked with navigation to a point specified relative to the start location (red square)—e.g. (5, 2) means go 5 meters forward and 2 meters right. We study agents that must do so solely from egocentric GPS+Compass and without a map.

**Relationship to Cognitive Maps.** Throughout the text, we use the term ‘map’ to mean a spatial representation that supports intelligent behaviors like taking shortcuts. This is the more general sense of the term and distinct from the specific concept of a ‘cognitive map’ [24].

Cognitive maps, as defined by O’Keefe and Nadel [24], imply a set of properties and are generally attached to a specific mechanism. The existence of a cognitive map requires that the agent be able to reach a desired goal in the environment from any starting location without being given that starting location, *i.e.* be able to navigate *against* a map. Further, cognitive maps refer to a specific mechanism—place cells and grid cells being present in the hippocampus.

Our work shows that the spatial information contained within the agent’s hidden state enables map-like properties—a secondary agent to take shortcuts through previously unexplored free space—and supports the decoding of a metric map. However, these do not fully cover the properties of a cognitive map nor do we make a mechanistic claim about how this information is stored in the neural network, though we do find the emergence of
Figure 3.2: (A) ‘Blind’ agent vs. bug. Our learned ‘blind’ agent compared to 2 variants and an oracle equipped variant of the Bug algorithm [105]. The Bug algorithm initially orients itself towards the goal and then proceeds towards the goal. Upon hitting a wall, it follows along the wall until it reaches the other side. The oracle version is told whether wall-following left or right is optimal, providing an upper-bound on Bug algorithm performance.

(B) Navigation performance vs. memory length. We find that the performance of agents does not saturate until their memory can contain information from hundreds of time steps in the past. Navigation episodes vary in length from slightly over 100 time steps to the maximum of 2,000, so a memory of $10^3$ steps is nearly half the maximum episode length.

We note that LSTM neural networks have been shown to be capable of building grid-cell maps that exhibit the properties of cognitive maps. Prior work has shown that LSTMs build grid-cell maps of an environment when trained directly for path integration within that environment [102, 103, 104]. Banino et al. [102] demonstrated that these maps aid in navigation by training a navigation agent that utilizes this cognitive map. We show that LSTMs trained for goal-driven navigation learn to build spatial representations in new/unseen environments and over the course of a single navigation. Whether or not LSTMs trained under this setting also utilize grid-cells is an open question for future work.

We now describe our findings for each question in detail.
3.2 Efficient Navigation with No Vision

Surprisingly, we find that agents trained under this impoverished sensing regime are able to navigate with near-perfect success – reaching the goal with 95.1% ± 1.3% success rate, even in situations where the agent must take hundreds of actions and traverse over 25m. Furthermore, the paths it takes are surprisingly efficient given the impoverished sensor suite (SPL of 62.9% ± 1.6).

To put these results into context, we compare them with ‘Bug algorithms’ [105], which are classical motion planning algorithms, inspired by insect navigation, involving an agent equipped with only a localization sensor. In these algorithms, the agent first orients itself towards the goal and then travels directly towards it until it encounters a wall, in which case it follows along the wall along one of two directions of travel. The primary challenge for Bug algorithms is determining whether to go left or right upon reaching a wall. To provide an upper bound on performance, we implement a ‘clairvoyant’ Bug algorithm agent with an oracle that tells it whether left or right is optimal. Even with the additional ground-truth information, the ‘clairvoyant’ Bug agent achieves an SPL of 46%, which is considerably less efficient than our ‘blind’ agent. Fig. 3.2a shows an example of the path our ‘blind’ agent takes compared to 3 variants of the Bug algorithm. This shows that blind navigation agents trained with reinforcement learning are highly efficient at navigating in previously unseen environments given their sensor suite.

3.3 Emergence of Wall-following Behavior and Collision-detection Neurons

Fig. 3.2a shows the blind agent exhibiting wall-following behavior. This behavior is remarkably consistent; the agent spends the majority of an episode near a wall. This is surprising because it is trained to navigate to the target location as quickly as possible, thus, it would be rewarded for traveling in straighter paths (that avoid walls). We hypothesize that this strategy emerges due to two factors. 1) The agent is blind, it has no way to determine
Figure 3.3: **t-SNE of the agent’s internal representation for collisions.** We find 4 overall clusters corresponding to the previous action taken and whether or not that action led to a collision. The t-SNE manifold is computed using 20,000 samples. This is randomly subsampled to 1,500 for visualization.

where the obstacles are in the environment besides ‘bumping’ into them. 2) The environment is unknown to the agent. While this is clearly true for testing environments it is also **functionally** true for training environments because the coordinate system is **episodic**, every episode uses a randomly-instantiated coordinate system based on how the agent was spawned; and the since the agent is blind, it cannot perform visual localization.

We test both these hypotheses. To test (2), we provide an experiment in the supplement showing that when the agent is trained and evaluated in a single environment with a consistent global coordinate system, it learns to memorize the shortest paths in this environment and wall-following does not emerge. Thus, this agent is simply unable to navigate in new environment.

To test hypothesis (1), we analyze whether the agent is capable of detecting collisions. Note that the agent is not equipped with a collision sensor. In principle, the agent can infer whether it collided with something in the environment – if tries to move forward and the resulting egomotion forward is less than usual, then a collision happened. This leads us
to ask – *does the agent’s memory contain information about collisions?* Specifically, we train a linear classifier that predicts if the action taken at time $t$, $a_t$, resulted in a collision given the next internal representation $(h_{t+1}, c_{t+1})$. The classifier achieves 98% accuracy on new/held-out data. As comparison, random guessing on this 2-class problem would achieve 50%. This shows the agent’s memory not only predicts its collisions, but also that *collisions-vs-not* are linearly separable in this space, which strongly suggests that the agent has learned a collision sensor.

Next we examine how collisions are structured in the agent’s internal representation by reducing the dimensionality of representation to identify the subspace that is used for collisions. Specifically, we re-train the linear classifier with an $\ell_1$ penalty to encourage sparsity. We then select the top 10 neurons (from 3072) with the largest weight magnitude; this reduces dimensionality by 99% while still achieving 96% accuracy. We use t-SNE [106] and the techniques proposed in [107] to create a 2-dimension visualization of the resulting 10-dimension space.

We find 4 distinct clusters (Fig. 3.3). One cluster always fires for collisions, one for forward actions that did not result in a collision, and the other two correspond to turning actions. Notice that these are an exceedingly small number of dimensions and neurons, essentially predicting collisions and movement of the agent. We include videos in the supplementary materials.

### 3.4 Memory Length

Next, we examine how memory is utilized by asking if the agent uses memory solely to remember short-term information (*e.g.* did it collide with the environment in the last step?) or whether it also includes long-range information (*e.g.* did it collide with the environment hundreds of steps ago?). To answer this question, we restrict the memory capacity of our agent via a memory-limited LSTM. Specifically, let $k$ denote the memory budget. At each time $t$, we take the sub-sequence of the previous $k$ observa-
tions, \([o_{t-k+1}, \ldots, o_t]\) and construct the internal representation \((h_t, c_t)\) via the recurrence 
\[(h_t, c_t) = \text{LSTM}(o_t, (h_{i-1}, c_{i-1}))\] for \(t - k < i \leq t\) where \((h_{t-k}, c_{t-k}) = (0, 0)\).

If the agent is only leveraging its memory for short-term storage we would expect performance to saturate at a small value of \(k\). Instead, Fig. 3.2b shows that the agent leverages its memory for significantly long term storage. Memoryless agents \((k = 1)\) completely fail at the task, achieving nearly 0% success. Navigation performance as a function of the memory budget \((k)\) does not saturate till one thousand steps. Recall that the agent can move forward 0.25 meters or turn 10° at each step. The average distance traveled by the agent in 1000 steps is 89.1 ± 0.66 meters, indicating that it remembers information over both very long temporal and spatial horizons.

### 3.5 Memory Enables Shortcuts

To investigate what information is encoded in the memory of our blind agents, we develop an experimental paradigm based on ‘probe’ agents. A probe is a secondary navigation agent\(^3\) that is structurally identical to the original (sensing, architecture, etc.), but parametrically augmented with the primary agent’s constructed episodic memory representation \((h_T, c_T)\). As the name suggests, a probe is simply a device to gain insight into the agent. The probe has no influence on the agent; any information the probe uses was already emergent in the agent’s representation. We use this paradigm to examine whether the agent’s final internal representation contains sufficient information for taking shortcuts in the environment.

As illustrated in Fig. 3.4, the agent first navigates from source \((S)\) to target \((T)\). After the agent reaches \(T\), a probe is initialized\(^4\) at \(S\), its memory initialized with the agent’s final memory representation, \(i.e. \((h_0, c_0)^\text{probe} = (h_T, c_T)^\text{agent}\)\), and tasked with navigating to \(T\). We refer to this probe task as SecondNav(\(S\rightarrow T\)). All evaluations are conducted in environments not used for training. Thus, any environmental information in the agent’s

---

\(^3\)To avoid confusion, we refer to this probe agent as ‘probe’ and the primary agent as ‘agent’.

\(^4\)The probe’s heading at \(S\) is set to the agent’s final heading upon reaching \(T\).
Figure 3.4: **Primary experimental paradigm.** First, an agent navigates (blue path, blue LSTM) from start (green sphere) to target (red sphere). After the agent navigates, we task a probe (purple LSTM) with performing the same navigation episode with the additional information encapsulated in the agent’s internal representation (or memory), $h^A_T$. The probe is able to navigate more efficiently by taking shortcuts (purple path). As denoted by the dashed line between the probe and agent networks, the probe does not influence what the agent stores in its internal representation. Environment in the image from the Replica Dataset [108].
Table 3.1: **Probe performance.** Results of our trained probe agent under three configurations – initialized with an empty representation (AllZeroMemory), a representation of a random agent walked along the trained agent’s path (UntrainedAgentMemory), and the final representation of the trained agent (TrainedAgentMemory). 95% confidence interval reported over 5 agent-probe pairs. Success and SPL are in percent.

<table>
<thead>
<tr>
<th>Probe Type</th>
<th>SecondNav(S→T)</th>
<th></th>
<th>SecondNav(T→S)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Success</td>
<td>SPL</td>
<td>Success</td>
<td>SPL</td>
</tr>
<tr>
<td>1 AllZeroMemory</td>
<td>91.6±0.40</td>
<td>71.1±0.27</td>
<td>91.0±0.40</td>
<td>70.8±0.25</td>
</tr>
<tr>
<td>2 UntrainedAgentMemory</td>
<td>92.4±0.28</td>
<td>72.0±0.19</td>
<td>91.2±0.54</td>
<td>72.2±0.35</td>
</tr>
<tr>
<td>3 TrainedAgentMemory</td>
<td>96.2±0.23</td>
<td>85.0±0.16</td>
<td>96.0±0.16</td>
<td>84.8±0.22</td>
</tr>
</tbody>
</table>

memory must have been gathered during its trajectory (and not during any past exposure during learning). Similarly, all initial knowledge the probe has of the environment must come from the agent’s memory \((h_T, c_T)^\text{agent}\).

Our hypothesis is that the agent’s memory contains a spatial representation of the environment, which the probe can leverage. If the hypothesis is true, we would expect the probe to navigate SecondNav(S→T) more efficiently than the agent (*e.g.* by taking shortcuts and cutting out exploratory excursions). If not, we would expect the probe to perform on-par with the agent. In our experiments, we find that the probe is *significantly* more efficient than the agent – SPL of 62.9%±1.6% (agent) vs. 85.0%±1.6% (probe). It is worth stressing how remarkable the performance of the probe is – in a new environment, a blind probe navigating without a map traverses a path that is within 15% of the the *shortest path* on the map. The best known *sighted* agents (equipped with an RGB camera, Depth sensor, and egomotion sensor) achieve an SPL of 96.7% on this task Chapter 2, thus re-using the memory of blind agents closes 65% of the gap between blind and sighted agents.

Fig. 3.4 shows the difference in path’s between the agent and probe. While the agent exhibits wall-following behavior, the probe instead takes more direct paths and rarely performs wall following. Recall that the only difference in the agent and probe conditions is the contents of the initial hidden state – the reward is identical (and available only during...
training for both), the training environments are identical, and the evaluation episodes are identical – meaning that the spatial representation, map, within the agent’s hidden state is what enables the probe to navigate more efficiently.

We further compare this result (which we denote as TrainedAgentMemory) with two control groups: 1) AllZeroMemory: An empty (all zeros) episodic memory to test for any systematic biases in the probe tasks. This probe contains identical information at the start of an episode as the agent (i.e. no information). 2) UntrainedAgentMemory: Episodic memory generated by an untrained agent (i.e. with a random setting of neural network parameters) as it is walked along the trajectory of the trained agent. This disentangles the agent’s structure from its parameters; and tests whether simply being encoded by an LSTM (even one with random parameters) provides an inductive bias towards building good environmental representations [109].

We find no evidence for this inductive bias – UntrainedAgentMemory performs no better than AllZeroMemory (Table 3.1, row 1 vs. 2). Furthermore, TrainedAgentMemory significantly outperforms both controls by +13 points SPL and +4 points Success (Table 3.1, row 3 vs. 1 and 2). Taken together, these two results indicate that the ability to construct useful spatial representations of the environment from a trajectory is a decidedly learned behavior.

Next, we examine if there is any directional preference in the episodic memory constructed by the agent. Our claim is that even though the agent navigates from S to T, if its memory indeed contains map-like spatial representations, it should also support probes for the reverse task SecondNav(T→S). Indeed, we find that TrainedAgentMemory probe performs the same (within margin of error) on both SecondNav(S→T) and SecondNav(T→S) (Table 3.1 right column) – indicating that the memory is equally useful in both directions. In the supplementary text, we demonstrate that the probe achieves this higher SPL by removing excursions from the agent’s path and taking shortcuts through previously unseen parts of the environment.
Figure 3.5: **Map prediction.** Accuracy (Intersection over Union) distributions (via kernel density estimation), dashed lines are the mean, TrainedAgentMemory has a higher mean with p-value \( \leq 10^{-5} \) (via Wilcoxon signed-rank test [110]). We show example ground truth - predictions pairs for various IoU values using TrainedAgentMemory. Light grey is non-navigable and dark grey is navigable. The agent path is drawn in light blue and navigates from start (green) to target (red).

Overall, these results provide compelling evidence that blind AI agents learn to build and use implicit map-like representations that enable shortcuts and reasoning about previously *untraversed* locations in the environment, solely through learning to navigate between two points.

### 3.6 Decoding a Metric Map

Next, we tackle the question – *‘Does the agent build episodic representations capable of decoding metric maps (occupancy grids) of the environment?’* Formally, given the final representation \((h_T, c_T)^{agent}\), we train a separate decoding network to predict an allocentric top-down occupancy grid (free-space vs not) of the environment. As with the probes, the decoder has no influence on the agent’s internal representation. We constrain the network to only make a prediction for a location if the agent reached within 2.5 meters of it. Note that since the agents are ‘blind’ predictions about *any* unvisited location require reasoning about unseen space and leveraging learned structural priors. As before, we compare the internal representation produced by TrainedAgentMemory to internal representation produced by an agent with random parameters, UntrainedAgentMemory.
Fig. 3.5 shows that we are able to predict top-down occupancy grids considerably more accurately with TrainedAgentMemory than with UntrainedAgentMemory – 32.4% vs. 12.4% intersection over union (IoU) relative to ground truth occupancy grid. In the qualitative examples, we see that the predictor is commonly able to make accurate predictions about unvisited locations. These results show that the internal representation contains the information needed to decode accurate occupancy maps of the environment even for unseen locations.

In the supplementary text, we conduct this analysis on ‘sighted’ navigation agents (equipped with a Depth camera and egomotion sensor). Perhaps counter-intuitively, we do not find conclusive evidence that metric maps can be decoded from the memory of sighted agents (despite their sensing suite being a strict super-set of blind agents). Our conjecture is that for higher-level strategies like map-building to emerge, the learning problem must not admit ‘trivial’ solutions such as the ones deep reinforcement learning is known to latch onto [111, 112, 97]. We believe that the minimal perception system used in our work served to create a challenging learning problem, which in turn limited the possible ‘trivial’ solutions, thus inducing map-building.

3.7 Maps are Task-dependent

Given that the agent is memory-limited, it stands to reason that it might need to choose what information to preserve and what to ‘forget’. To examine this, we attempt to decode the agent’s past observations from its memory. The agent’s observation consists of position and orientation. We consider just the position component as this correlates directly with the map-like properties of the memory. Formally, given internal state at time \( t \), \((h_t, c_t)\), we train a prediction network \( f_k(\cdot) \) to predict the agent’s location \( k \) steps in to the past, i.e.
\[
\hat{s}_{t-k} = f_k(h_t, c_t) + s_t, \quad k \in [1, 256].
\]
Given ground truth location \( s_{t+k} \), we evaluate the decoder via i) absolute L2 error \( ||\hat{s}_{t+k} - s_{t+k}|| \) and ii) relative L2 error \( ||\hat{s}_{t+k} - s_{t+k}||/||s_{t+k} - s_t|| \). To determine baseline (or chance) performance, we train a second set of decoders
where instead of using the correct internal state $(h_t, c_t)$ as the input, we randomly select an internal state from a different trajectory. This will evaluate if there are any inherent biases in the task.

Qualitative analysis of past prediction results shows that the agent forgets excursions\(^5\), i.e. excursions are harder to decode than non-excursions (see Fig. 3.6a). To quantify this, we manually labelled excursions in 216 randomly sampled episodes in evaluation environments. Fig. 3.6b shows that excursions are harder to decode than non-excursions, indicating that the agent does indeed forget excursions. Interestingly, we find that the exit of the excursion is considerably easier to decode, indicating that the end of the excursion performs a similar function to landmarks in animal and human navigation [113].

### 3.8 Outlook

In this work, we have shown that ‘blind’ AI navigation agents – agents with similar perception as blind mole rats – are capable of performing goal-driven navigation to a high

\(^5\)We define an excursion as a sub-path that approximately forms a loop.
degree of performance. We then showed that these AI navigation agents learn to build map-like representations (*i.e.* the ability to take short-cuts, follow walls, and predict free-space and collisions) of their environment solely through learning goal-driven navigation. Our agents and training regime have no added inductive bias towards map-building, be it explicit or implicit, implying that cognitive maps may be a *natural solution* to the inductive biases imposed by navigation by intelligent embodied agents, whether they be biological or artificial. In a similar manner, convergent evolution [114], where two unrelated intelligent systems independently arrive at similar mechanisms, suggests that the mechanism is a natural response of having to adapt to the environment and the task.

Our results also provide insight into the black-boxes neural networks that parameterize current AI navigation agents. We demonstrate that these agents are capable of learning to build map-like representations with no learning signal other than goal driven navigation. This establishes a fundamental link between AI navigation and animal navigation. Further, it links how AI systems navigate with analytic mapping-and-planning techniques [16, 17, 18, 19].

Our results and analyses also point towards future directions in AI navigation research. Specifically, imbuing AI navigation agents with explicit (*e.g.* architectural design) or implicit (*e.g.* training regime or auxiliary objectives) priors that bias agents towards learning an internal representation with the features found here may improve their performance. Further, it may better equip them to learn more challenging tasks that require even longer term memory such as rearrangement of an environment by moving objects [21].

We see several limitations and areas for future work. First, we examined ground-based navigation agents operating in digitizations of real houses. This limits the agent a 2D manifold and induces strong structural priors on environment layout. As such, it is unclear how our results generalize to a drone flying through a large forest. Second, we examined agents with a minimal perceptual system. In the supplementary text, we attempted to decode occupancy grids (metric maps) from Depth sensor equipped agents and did not find convincing
evidence. Our conjecture is that for higher-level strategies like map-building to emerge, the learning problem must not admit ‘trivial’ solutions. We believe that the minimal perception system used in our work also served to create such a challenging learning problem.
CHAPTER 4

EMERGENCE OF NAVIGATION IN MOBILE MANIPULATION AGENTS

4.1 Introduction

Scaling matters. Progress towards building embodied intelligent agents that are capable of performing goal driven tasks has been driven, in part, by training large neural networks in photo-realistic 3D environments with deep reinforcement learning (RL) for (up to) billions of steps of experience (Chapter 2) [35, 115, 31, 116]. To enable this scale, RL systems must be able to efficiently utilize the available resources (e.g. GPUs), and scale to multiple machines all while maintaining sample-efficient learning.

One promising class of techniques to achieve this scale is batched on-policy RL. These methods collect experience from many \( N \) environments simultaneously using the policy and update it with this cumulative experience. These methods are broadly divided into two classes: synchronous (SyncOnRL) and asynchronous (AsyncOnRL). SyncOnRL contains two potential synchronization points: first the policy is executed for the entire batch \((o_t \rightarrow a_t)_{b=1}^B\) (Fig. 4.1 A), then actions are executed in all environments, \((s_t, a_t \rightarrow s_{t+1}, o_{t+1})_{b=1}^B\) (Fig. 4.1 B), until \( T \) steps have been collected from all \( N \) environments. This \((T, N)\)-shaped batch of experience is used to update the policy (Fig. 4.1 C). These synchronization points reduce throughput due to the straggler effect [117, 118], where the system spends significant (sometimes the most) time idling, waiting for the slowest worker to finish.

AsyncOnRL removes these synchronization points, thereby mitigating the straggler effect and improving throughput. Actions are taken as soon as they are computed, \(a_t \rightarrow o_{t+1}\) (Fig. 4.1 D), the next action is computed as soon as the observation is ready, \(o_t \rightarrow a_t\) (Fig. 4.1 E), and the policy is updated as soon as enough experience is collected. How-

\footnote{Following standard notation, \( s_t \) is (PO)MDP state, \( a_t \) is the action taken, and \( o_t \) is the agent observation.}
Figure 4.1: **(Right) RL Training Systems.** In SyncOnRL, actions are computed for all environments, then all environments are stepped. Experience collection is paused during learning. In AsyncOnRL, computing actions, stepping environments, and learning all occur without synchronization. In VER, a variable amount of experience is collected from each environment, enabling synchronous learning without the straggler effect. **(Left) skill policies** with navigation are more robust to handoff errors.

However, AsyncOnRL systems are not able to ensure that all experience has been collected by only the current policy and thus must consume near-policy data. This reduces sample efficiency [119]. Thus, status quo leaves us with an unpleasant tradeoff – high sample-efficiency with low throughput or high throughput with low sample-efficiency.

In this work, we propose Variable Experience Rollout (VER). VER combines the strengths of and blurs the line between SyncOnRL and AsyncOnRL. Like SyncOnRL, VER collects experience with the current policy and then updates it. Like AsyncOnRL, VER does not have synchronization points – it computes next actions, steps environments, and updates the policy as soon as possible. The inspiration for VER comes from two key observations:

1) AsyncOnRL mitigates the straggler effect by implicitly collecting a variable amount of experience from each environment – more from fast-to-simulate environments and less from slow ones.

2) Both SyncOnRL and AsyncOnRL use a fixed rollout length, $T$ steps of experience. Our key insight is that while a fixed rollout length may simplify an implementation, it is not a requirement for RL.

These two key observations naturally lead us to *variable experience rollout* (VER), i.e. collecting rollouts with a variable number of steps. VER adjusts the rollout length for each environment based on its simulation speed. It explicitly collects more experience from
fast-to-simulate environments and less from slow ones (Fig. 4.1). The result is an RL system that overcomes the straggler effect and maintains sample-efficiency by learning from on-policy data.

VER focuses on efficiently utilizing a single GPU. To enable efficient scaling to multiple GPUs, we combine VER with the decentralized distributed method proposed in Chapter 2.

First, we evaluate VER on well-established embodied navigation tasks using Habitat 1.0 [22]. On 8 GPUs, VER trains PointGoal navigation [13] 60% faster (1.6x speedup) than Decentralized Distributed PPO (DD-PPO) (Chapter 2), the current state-of-the-art for distributed on-policy RL, with the same sample efficiency. For ObjectGoal navigation [120], an active area of research, VER trains 100% faster than DD-PPO with better sample efficiency.

Next, we evaluate VER on the recently introduced (and significantly more challenging) GeometricGoal rearrangement tasks [21] in Habitat 2.0 [23]. In GeoRearrange, a virtual robot is spawned in a new environment and asked to rearrange a set of objects from their initial to desired coordinates. These environments have highly variable simulation time (physics simulation time increases if the robot bumps into something) and require GPU-acceleration (for photo-realistic rendering), limiting the number of environments that can be run in parallel.

On 1 GPU, VER is 150% faster (2.5x speedup) than DD-PPO with the same sample efficiency. VER is as fast as SampleFactory [96], the state-of-the-art AsyncOnRL, with the same sample efficiency. VER is as fast as AsyncOnRL in terms of compute efficiency; this is a surprisingly strong result. AsyncOnRL never stops collecting experience (it collects experience during learning) so it should, in theory, be a strict upper bound on performance. However, VER is able to match AsyncOnRL for environments like Habitat that heavily utilize the GPU for rendering. In AsyncOnRL, learning, inference, and rendering all contend the GPU which reduces throughput. VER reduces this contention, and thereby increases throughput, by not overlapping learning and experience collection.
On 8 GPUs, VER achieves better scaling than DD-PPO, achieving a 6.7x speed-up (vs. 6x for DD-PPO) due to lower variance in experience collection time between GPU-workers. Due to this efficient multi-GPU scaling, VER is 70% faster (1.7x speedup) than SampleFactory on 8 GPUs and has better sample efficiency as it learns from on-policy data.

Finally, we leverage these SysML contributions to study open research questions posed in prior work. Specifically, we train RL policies for mobile manipulation skills (Navigate, Pick, Place, etc.) and chain them via a task planner. [23] called this approach TP-SRL and identified a critical ‘handoff problem’ – downstream skills are set up for failure by small errors made by upstream skills (e.g. the Pick skill failing because the navigation skill stopped the robot a bit too far from the object).

We demonstrate a number of surprising findings when TP-SRL is scaled via VER. Most importantly, we find the emergence of navigation when skills that do not ostensibly require navigation (e.g. pick) are trained with navigation actions enabled. In principle, Pick and Place policies do not need to navigate during training since the objects are always in arm’s reach, but in practice they learn to navigate to recover from their mistakes and this results in strong out-of-distribution test-time generalization. Specifically, TP-SRL without a navigation skill achieves 50% success on NavPick and 20% success on a NavPickNavPlace task simply because the Pick and Place skills have learned to navigate (sometimes across the room!). TP-SRL with a Navigate skill performs even stronger: 90% on NavPickNavPlace and 32% on 5 successive NavPickNavPlaces (called Tidy House in [23]), which are +32% and +30% absolute improvements over [23], respectively. Prepare Groceries and Set Table, which both require interaction with articulated receptacles (fridge, drawer), remain as open problems (5% and 0% Success, respectively) and are the next frontiers.

4.2 VER: Variable Experience Rollout

The key challenge that any batched on-policy RL technique needs to address is variability of simulation time for the environments in a batch. There are two primary sources of
this variability: action-level and episode-level. The amount of time needed to simulate an action within an environment varies depending on the specific action, the state of the robot, and the environment (e.g., simulating the robot navigating on a clear floor is much faster than simulating the robot’s arm colliding with objects). The amount of time needed to simulate an entire episode also varies environment to environment irrespective of action-level variability (e.g., rendering images takes longer for visually-complex scenes, simulating physics takes longer for scenes with a large number of objects).

4.2.1 Action-Level Straggler Mitigation

We mitigate the action-level straggler effect by applying the experience collection method of AsyncOnRL to SyncOnRL. We represent this visually in Fig. 4.2 and describe it in text below.

**Environment workers** receive the next action and step the environment, EnvStep, e.g. \( s_t, a_t \rightarrow s_{t+1}, o_{t+1}, r_t \). They write the outputs of the environment (observations, reward, etc.) into pre-allocated CPU shared memory for consumption by inference workers.

**Inference workers** receive batches of steps of experience from environment workers. They perform inference with the current policy to select the next action and send it to the environment worker using pre-allocated CPU shared memory. After inference they store ex-
perience for learning in shared GPU memory. Inference workers use dynamic batching and perform inference on all outstanding inference requests instead of waiting for a fixed number of requests to arrive. This allows us to leverage the benefits of batching without introducing synchronization points between environment workers.

This experience collection technique is similar to that of HTS-RL \([119]\) (SyncOnRL) and SampleFactory \([96]\) (AsyncOnRL). Unlike both, we do not overlap experience collection with learning. This has various system benefits, including reducing GPU memory usage and reducing GPU driver contention. More details are available in Section \(C.1\).

4.2.2 Environment-Level Straggler Mitigation

In both SyncOnRL and AsyncOnRL, the data used for learning consists of \(N\) rollouts of equal-\(T\) steps of experience, an \((T, N)\)-shaped batch. In SyncOnRL these \(N\) sets are all collected with the current policy, this leads to the environment-level straggler effect. AsyncOnRL mitigates this by relaxing the constraint that experience must be strictly on-policy, and thereby implicitly changes the experience collection rate for each environment.

**Variable Experience Rollout (VER).** We instead relax the constraint that we must use \(N\) rollouts of equal-\(T\) steps. Specifically, VER collects \(T \times N\) steps of experience from \(N\) environments without a constraint on how many steps of experience are collected from each environment. This explicitly varies the experience collection rate for each environment – in effect, collecting more experience from environments that are fast to simulate. Consider the 4 environments shows in Fig. 4.3A. The length of the each step representation the wall-clock time taken to collect it, some steps are fast, some are slow. VER collects more experience from environment 0 as it is fastest to step and less from 1 because it is the slowest.

**Learning mini-batch creation.** We focus on training recurrent polices because memory is key in long-range and partially observable tasks like HAB. When training recurrent

\(^2\)In practice we introduce both a minimum and maximum number of requests to prevent under-utilization of compute and over-utilization of memory.
Figure 4.3: (A) VER collects a variable amount of experience from each environment. The length of each step represents the time taken to collect it. (B) VER mini-batch. The solid bars denote episode boundaries. The steps selected for the first mini-batch have a dashed border. (C) The PackedSequence data format represents a set of sequences with variable length in a linear buffer such that all elements from each timestep area next to one-another in memory.

policies, we must create mini-batches of experience with sequences for back-propagation-through-time. Normally $B$ mini-batches are constructed by spitting the $N$ environments’ experience into $B \frac{T \times N}{B}$-sized mini-batches. A similar procedure would result in mini-batches of different sizes for VER. This would harm optimization as the learning rate and optimization mini-batch size are intertwined and automatically adjusting the learning rate is an open question [77, 121].

To understand VER’s mini-batching, first note that there are two reasons for the start of a new sequence of experience: rollout starts (Fig. 4.3B, step 0) and episode starts (Fig. 4.3B, a step after a bar). These two boundary types are independent – episodes can end at any arbitrary step within the rollout and then that environment will reset and start a new episode. Thus when we collect experience from $N$ environments, we will have $K \geq N$ sequences to divide between the mini-batches. We distributed these $K$ sequences between the mini-batches. We randomly order the sequences, then the first $T \times \frac{N}{B}$ steps are the first mini-batch, the next $T \times \frac{N}{B}$ to the second, etc. See Fig. 4.3B for an example.

**Batching computation for learning.** The mini-batches constructed from the algorithm above have sequences with variable length. To batch the computation of these sequences we use cuDNN’s PackedSequence data model. This data model represents a set of variable-length sequences (Fig. 4.3C left) such that all sequence-elements at a time-step are contiguous in memory (Fig. 4.3C right) – this enables efficient batched computation on each
time-step for components with a temporal dependence, \textit{e.g.} the RNN. Further, it uses a contiguous block of memory for all elements across all time-steps – this enables batched computation across \textit{all} time-steps for network components that don’t have a temporal dependence, \textit{e.g.} the visual encoder.

During experience collection we write experience into a linear buffer in GPU memory and then arrange each mini-batch as a \texttt{PackedSequence}. This takes less than 10 milliseconds (per learning phase); orders of magnitude less than experience collection (\textasciitilde3s) or learning (\textasciitilde1.5s) in our experiments.

\textbf{Learning method.} The experience and mini-batches generated from \texttt{VER} are well-suited for use with RL methods that use on-policy data \cite{122} or a mix of off-policy and on-policy data \cite{123,124}. We use Proximal Policy Optimization (PPO) \cite{54}, an on-policy method, as it is known to work well for embodied AI tasks and recurrent policies.

\textbf{Inflight actions.} One subtle design choice is the following – when \texttt{VER} finishes a \((T \times N)\) experience collection, there will be (slow) environments that haven’t completed simulation yet. Instead of discarding that data, we choose to collect this experience in the \textit{next} rollout. This experience is at most 1 policy-update old, contains at most \(N - 1\) steps, and we find this choice leads to speed gains without any sample-efficiency loss.

\textbf{4.2.3 Multiple GPUs}

We leverage the decentralized distributed training architecture from Chapter 2 to scale \texttt{VER} to multiple GPUs (residing on a single or multiple nodes). In this architecture, each GPU both collects experience and learns from that experience. During learning, gradients from each GPU-worker are averaged with an \texttt{AllReduce} operation. This is a synchronous operation and thus introduces a GPU-worker-level straggler effect. Chapter 2 mitigate this effect by preempting stagglers – stopping collecting experience early and proceeding to learning – after a fixed number of GPU-workers have finished experience collection. While this method is effective, it introduces an additional hyperparameter and isn’t able to adapt to
changes in simulation time throughout learning.

We instead approximate the optimal preemption for each experience collection phase. Given the learning time, LT, and a function $\text{Time}(S)$ that returns the time needed to collect $S$ steps of experience, the optimal number of steps $S$ to collect before preempting stragglers is

$$\max_S \frac{S}{\text{Time}(S) + LT}, \quad \text{s.t.} \quad S \leq T \cdot N \cdot \#\text{GPUs}. \quad (4.1)$$

For LT, we record the time from the last iteration as this doesn’t change between iterations. We approximate $\text{Time}(S)$ using the average step time to receive a step of experience from each environment (this includes both inference time and simulation time) from the previous rollout. This will be a poor approximation in some circumstances but we find it to work well in practice.

We improve upon sample efficiency of DD-PPO by filling the preempted rollouts with experience from the previous rollout. We perform extra epochs of PPO on this now ‘stale’ data instead of correcting for the off-policy return as we find this simpler and effective. This comes with no effective computation cost since the result is that all GPUs have the same batch size instead of some having a smaller batch size.

### 4.3 Embodied Navigation: Benchmarking

First, we benchmark VER on the embodied navigation [13] tasks in Habitat 1.0 [22] – PointNav [13] and ObjectNav [120]. Our goal here is simply to show training speed-ups in well-studied tasks (and in the case of PointNav, a well-saturated task with no room left for accuracy improvements). We present accuracy improvements and in-depth analysis on challenging rearrangement tasks in Section 4.6.

For both tasks, we use standard architectures from Habitat Baselines [22, 25] – the ResNet18 encoder and a 2 layer LSTM. Following [32, 36], we add Action Conditional Contrastive Coding [125], using the hyper-parameters from [126].
Table 4.1: **Navigation Tasks** training steps per second on 8 GPUs. VER is 60%-100% faster than DD-PPO.

<table>
<thead>
<tr>
<th>Task</th>
<th>DD-PPO</th>
<th>VER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointGoalNav</td>
<td>3065±17</td>
<td>5325±30</td>
</tr>
<tr>
<td>ObjectNav</td>
<td>1015±19</td>
<td>2019±62</td>
</tr>
</tbody>
</table>

Figure 4.4: **Validation performance** on ObjectNav and PointNav. VER has similar or slightly better sample efficiency than DD-PPO, indicating that performance is not negatively impacted by the non-uniform sampling of experience from environments. Shading is a 95% confidence interval over 3 seeds.

**PointNav.** We train PointNav agents with one RGB camera on the HM3D dataset [31] for 1.85 billion steps of experience on 8 GPUs. We study the RGB setting (and not Depth) because this is the more challenging version of the task and thus we expect it to be more sensitive to possible differences in the training system. We examine VER along two axes: 1) training throughput – the number of samples of experience per second (SPS) the system collects and learns from, 2) sample efficiency. On 8 GPUs, VER trains agents 60% faster than DD-PPO, from 3065 SPS to 5325 SPS (Table 4.1), with similar sample efficiency (Fig. 4.4).

**ObjectNav.** We train ObjectNav agents with one RGB and one Depth camera on the MP3D [67] dataset for 600 million steps of experience on 8 GPUs. VER trains agents 100% faster than DD-PPO, 1021 to 2019 (Table 4.1), with slightly better sample efficiency
There are two effects that enable better sample efficiency with VER. First, we perform additional epochs of PPO on experience from the last rollout when a GPU-worker is preempted. Second, the variable experience rollout mechanism results in a natural curriculum. Environments are often faster to simulate when they are easier (i.e. a smaller home), so more experience will be collected in these easier cases.

### 4.4 Embodied Rearrangement: Task, Agent, and Training

Next, we use VER to study the recently introduced (and more challenging) GeometricGoal rearrangement tasks [21] in Habitat 2.0 [23].

**Task.** In GeoRearrange, an agent is initialized in an unknown environment and tasked with rearranging objects in its environment. The task is specified as a set of coordinate pairs \( \{(\text{Pose}_{\text{Initial}}, \text{Pose}_{\text{Final}})\}_{o=1}^O \). The agent must bring each object at \( \text{Pose}_{\text{Initial}} \) to \( \text{Pose}_{\text{Final}} \) where \( \text{Pose} \) is the initial or desired center-of-mass location for the object(s). We use the Home Assistant Benchmark (HAB) which consists of 3 scenarios of increasing difficulty: Tidy House, Prepare Groceries, and Set Table.

In Tidy House, the agent is tasked with moving 5 objects from their initial locations to their final locations. The objects are rarely in containers (i.e. fridge or cabinet drawer) and when they are, the containers are already opened. In Prepare Groceries, the agent must move 3 objects from the kitchen counter into the open fridge (or open fridge to counter). This stresses picking and placing in the fridge, which is challenging. In Set Table, the agent must move 1 object from the closed kitchen cabinet drawer to the table and 1 object from the closed fridge to the table. This requires opening the cabinet and fridge, and picking from these challenging receptacles.

**Simulation.** We use the Habitat simulator with the ReplicaCAD Dataset [22, 23]. The robot policy operates at 30 Hz and physics is simulated at 120 Hz.

**Agent.** The agent is embodied as a Fetch robot with a 7-DOF arm. The arm is controlled via joint velocities. At every time step the policy predicts a delta in motor position for
each of the 7 joints in the arm. We find joint velocity control equally easy to learn but faster to simulate than the end-effector control used in [23]. The arm is equipped with a suction gripper. The agent must control the arm such that the gripper is in contact with the object to grasp and then activate the gripper. The object is dropped once the gripper is deactivated. This is more realistic than the ‘magic’ grasp action used in [23]. The robot base (navigation) is controlled by the policy commanding a desired linear speed and angular velocity. The robot is equipped with one Depth camera attached to its head, proprioceptive sensors that provide the joint positions of its arm, and a GPS+Compass sensor that provides its heading and location relative to its initial location. The policy models $a_t \sim \pi(\cdot \mid s_{t-1})$ instead of $a_t \sim \pi(\cdot \mid s_t)$. This is both more realistic and enables physics and rendering to be overlapped [23].

We build upon the TaskPlanning-SkillRL (TP-SRL) method proposed in [23]. TP-SRL is a hierarchical method for GeoRearrange that decomposes the task into a series of skills – Navigate, Pick, Place, and \{Open, Close\} $\times$ \{Cabinet, Fridge\}. Skills are controlled via a skill-policy (learned with RL) and chained together via a task planner. One of the key challenges is the ‘handoff problem’ – downstream skills are setup for failure due to slight errors made by the upstream skill. We give all skill policies access to navigation actions to allow them to correct for these errors.

**Architecture.** All skill policies share the same architecture. We use ResNet18 [59] to process the $128 \times 128$ visual input. We reduce with width of the network by half and use GroupNorm [63]. We also apply some of the recent advancements from ConvNeXt [127]. We use a patch-ify stem, dedicated down-sample stages, layer scale [128], and dilated convolutions [129] (this mimics larger kernel convolutions without increasing computation). The visual embedding is then combined with the proprioceptive observations and previous action, and then processed with a 2-layer LSTM [130]. The output of the LSTM is used to predict the action distribution and value function. Actions are sampled from a multivariate Gaussian distribution with a diagonal covariance matrix.
Training. We train agents using VER and PPO [54] with Generalized Advantage Estimation [131]. We use a minimum entropy constraint with a learned coefficient [132] as we find this to be more stable given our diverse set of skills than a fixed coefficient. Formally, let \( \mathcal{H}(\pi) \) be the entropy of the policy, we then minimize \( \alpha (\lambda - [\mathcal{H}(\pi)]_{sg}) - [[\alpha]]_{sg} \mathcal{H}(\pi) \) where \([\cdot]_{sg}\) is the stop gradient operator. We set the target entropy, \( \lambda \), to zero for all tasks. We use the Adam optimizer [65] with an initial learning rate of \( 2.5 \times 10^{-4} \) and decay it to zero with a cosine schedule. To correct for biased sampling in VER, we use truncated importance sampling weighting [47] with a maximum of 1.0.

4.5 Embodied Rearrangement: Benchmarking

In this section, we benchmark VER for training open-fridge policies because this task involves interaction of the robot with an articulated object (the fridge) and represents a challenging case for the training system due to large variability in physics time. In Section 4.6, we analyze the task performance, which requires all skills. For all systems, we set the number of environments, \( N \), to 16 per GPU.

4.5.1 System throughout

VER is 150% faster than DD-PPO (Table 4.2); an even larger difference than in the simpler navigation tasks we studied before. DD-PPO has no mechanism to mitigate the action-level or episode-level straggler effects. In their absence, DD-PPO has similar throughput as VER (Table 4.2, Max). Under these effects, DD-PPO’s throughput reduces 150% compared to 20% for VER (Table 4.2, Mean vs. Max).

Variable experience rollouts are effective. We compare VER with VER minus variable experience rollouts (NoVER). NoVER is a ‘steel-manned’ baseline for VER and benefits from all our micro-optimizations. VER is 30% faster than NoVER (Table 4.2).

VER closes the gap to AsyncOnRL. On 1 GPU, VER is as fast as SampleFactory [96], the fastest single machine AsyncOnRL. Intuitively AsyncOnRL should be a strict upper-bound on
Table 4.2: **SyncOnRL, VER, and AsyncOnRL benchmarking.** Mean/max system throughput (SPS) over 20 million training steps. Hardware: Tesla V100(s) with 10 CPUs per GPU.

<table>
<thead>
<tr>
<th>GPUs</th>
<th>DD-PPO Mean</th>
<th>DD-PPO Max</th>
<th>NoVER Mean</th>
<th>NoVER Max</th>
<th>VER (Ours) Mean</th>
<th>VER (Ours) Max</th>
<th>SampleFactory Mean</th>
<th>SampleFactory Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>174±7</td>
<td>442±49</td>
<td>327±7</td>
<td>428±11</td>
<td>428±5</td>
<td>534±7</td>
<td>427±5</td>
<td>517±3</td>
</tr>
<tr>
<td>2</td>
<td>283±23</td>
<td>696±24</td>
<td>592±5</td>
<td>786±11</td>
<td>716±32</td>
<td>945±29</td>
<td>804±15</td>
<td>1022±0</td>
</tr>
<tr>
<td>4</td>
<td>468±21</td>
<td>1337±34</td>
<td>1097±30</td>
<td>1601±39</td>
<td>1432±10</td>
<td>1915±13</td>
<td>1286±16</td>
<td>1568±26</td>
</tr>
<tr>
<td>8</td>
<td>1066±84</td>
<td>2754±156</td>
<td>2216±60</td>
<td>3438±94</td>
<td>2861±21</td>
<td>3829±23</td>
<td>1662±4</td>
<td>1842±0</td>
</tr>
</tbody>
</table>

Performance – it never stops collecting experience while VER does. However this doesn’t take into account the realities of hardware. Recall that we are training an agent with a large visual encoder. This means that updating the parameters of the agent takes a large amount of time (~150ms per mini-batch of size 1024 on a V100). Further, Habitat uses the GPU for rendering. The use of the GPU for both rendering and learning simultaneously results in GPU driver contention. In SampleFactory, learning time and experience collection time are roughly double that of VER.

**Multi-GPU scaling.** VER has better multi-GPU scaling than DD-PPO, achieving a 6.7x speed-up on 8 GPUs compared to 6x. The rollout collection time in VER is lower variance, which improves scaling. On 2 GPUs, SampleFactory is 12% faster than VER. Here one GPU is used for learning+inference and the other is used for rendering. This creates a nice division of work and doesn’t result in costly GPU contention. On 4 and 8 GPUs however, the single GPU used for learning in SampleFactory is the bottleneck and VER has higher throughput (nearly 100% faster on 8 GPUs). While it is possible to implement multi-GPU learning for AsyncOnRL to overcome this problem, it is left to the user to balance the number of GPUs used for experience collection and learning and this balance is static. VER automatically balances GPU-time used for experience and learning continuously.

### 4.5.2 Sample Efficiency

Next we examine sample efficiency of the training systems. VER has either identical (1, 2, and 4 GPUs), Fig. 4.5) or better sample efficiency (8 GPUs) than DD-PPO. Interestingly,
Figure 4.5: **Training sample efficiency** on Open Fridge. VER has similar sample efficiency as DD-PPO (SyncOnRL). SampleFactory (AsyncOnRL) has similar sample efficiency with 1 GPU but this reduces as policy lag increases with more GPUs. When considering both sample efficiency and throughput (time-to-performance), VER is always achieves a given performance the fastest (outright or a tie). The shaded region is a 95% bootstrapped confidence interval over 5 seeds. We use interquartile mean (IQM) as our summary statistic [133].

SampleFactory has similar sample efficiency with 1 GPU (Fig. 4.5) and is more stable. This is because reducing the number of GPUs also reduces the batch-size and PPO is known to not be batch-size invariant [134]. We believe the stale data serves a similar function to PPO-EWMA [134], which uses an exponential moving average of the policy weights for the trust region. On 2, 4, and 8 GPUs, VER has better sample efficiency than SampleFactory. Combing the findings of compute and sample efficiency, we find that VER always achieves a given performance threshold in the least amount of training time (outright or a tie) (Fig. 4.5 Lower).

### 4.6 Embodied Rearrangement: Analysis of Learned Skills

We examine the performance of TP-SRL on the Home Assistant Benchmark (HAB) [23]. We examine both skill policies trained with the full action space and the limited, per-skill specific action space used in [23]. Each skill is trained with VER for 500 million steps of experience on 8 GPUs. This takes less than 2 days per skill.
4.6.1 Performance on HAB

We find that skills with navigation actions improve performance on the full task but does not change performance on the skill’s train task (e.g. Pick achieves 90% success with and without during training). Fig. 4.6 shows smaller drops in performance between every interaction (there is a navigation between each interaction) demonstrating that skills with navigation effectively correct for handoff errors. This is impactful after place as the navigation policy tends to make more errors when navigating to the next location after place. On Tidy House, full task performance improves from 2% Success to 32%.

On Prepare Groceries (which requires picking/placing from/into the fridge) and Set Table (which requires opening the fridge/cabinet and then picking from it) performance improves slightly. 0% to 5% on Prepare Groceries and 0% to 7.5% on the first pick+place on Set Table. Both these tasks remain as open problems and the next frontier.
4.6.2 Emergent Navigation

Next we examine if the skill policies are able to navigate to correct for highly out-of-distribution initial location. We examine TP-SRL(NoNav), which that omits the navigation skill. In this agent all navigation is done by skill policies that ostensibly never needed to navigate during training.

We find emergent navigation in both the Pick and Place policies. TP-SRL(NoNav) achieves 50% success on NavPick (Fig. 4.6, Tidy House interaction 1), and 20% success on NavPickNavPlace (Fig. 4.6, Tidy House interaction 2). The latter is on-par with the TaskPlanning+SensePlanAct (TP-SPA) classical baseline and significantly better than the MonolithicRL baseline in [23].

The Pick and Place policies were trained on tasks that requires no navigation but both are capable of navigation. We provided examples of both the training task for Pick and Place, and TP-SRL(NoNav) on Tidy House in the supplementary materials.

On Prepare Groceries and Set Table, the navigation performance of these policies is worse (-34% Success and -45% Success on the first interaction, respectively). Prepare Groceries requires picking from the fridge, which is challenging and requires navigation that doesn’t accidentally close the door. Set Table requires opening the cabinet and then picking, which introduces an additional OpenCab skill and requires more precise navigation and picking from Pick. Performance is non-zero (15% and 4% on the first interaction, respectively) in both scenarios; indicting that the skill policies are capable of navigating even in these scenarios, albeit less successfully than in Tidy House.

We hypothesize that the Pick and Place polices learned navigation because this was useful early in training. Early in training the policies have yet to learn that only minimal navigation is needed to complete the task. Therefore the policy will sometimes cause itself to move away from the pick object/place location and will navigate back. Navigation is then not forgotten as the policy converges.

We examined the behavior of a Pick policy early in training and found that it does tend
to move away from the object it needs to pick up and sometimes moves back. Although the magnitude of navigation is small and quite infrequent, so the degree of generalization is high.

This result, and the higher performance on HAB, highlights that it may not always be beneficial to remove ‘unneeded’ actions. [23] removed navigation actions where possible to improve sample efficiency and training throughput\(^3\). Our experiments corroborate this; training without navigation improves both sample efficiency and training speed. However, by enabling navigation and allowing the agent to learn how to (not) use it, we arrived upon emergent navigation and improved full task performance.

### 4.7 Related Work

**AsyncOnRL** methods provide high-throughput on-policy reinforcement learning [47, 96]. However, they have reduced sample efficiency as they must correct for near-policy, or ‘stale’, data. Few support multi-GPU learning and, when they do, the user must manually balance compute between learning and experience collection [135]. **VER** achieves the same throughput on 1-GPU while learning with on-policy data, has better sample efficiency, supports multi-GPU learning, and automatically balances compute between learning and simulation.

**SyncOnRL.** Closely related to our work, HTS-RL [119] also use the same experience collection techniques as AsyncOnRL to mitigate the action-level straggler effect. Unfortunately inefficiencies in the provided implementation prevents a meaningful direct comparison. In Section C.5 we show that our re-implementation is 110% faster (2.1x speedup) and thus instead compare to stronger baseline of NoVER in the main text. We propose a novel mechanism, variable experience rollouts, to mitigate the episode-level straggler effect and thereby close the gap to AsyncOnRL. We use and build upon Decentralized Distributed PPO (DD-PPO) (Chapter 2), which proposed a distributed multi-GPU method based on data\(^3\)Personal correspondence with authors.
parallelism [56].

**Batched simulators** simulate multiple agents (in multiple environments) simultaneously and are responsible for their own parallelization [98, 29, 136, 137]. While these systems offer impressive performance, none currently support a benchmark like HAB (which combines physics and photo-realism) nor the flexibility of Habitat, AI2Thor [138], or ThreeD-World [139]. VER enables researchers to first explore promising directions using existing simulators and then build batched simulators with the knowledge gained from their findings.

### 4.8 Societal Impact, Limitations, and Conclusion

Our main application result is trained using the ReplicaCAD dataset [23], which is limited to only US apartments, and this may have negative societal impacts for deployed assistants. VER was designed and evaluated for tasks with both GPU simulation and large neural networks. For tasks with CPU simulation and smaller networks, we expect it to improve upon SyncOnRL but it may have less throughput than AsyncOnRL and overlapping experience collection and learning would likely be beneficial. Our implementation supports overlapping learning and collecting 1 rollout, but more overlap may be beneficial. The TP-SRL agent we build upon requires oracle knowledge, e.g. that the cabinet must be opened before picking.

We have presented Variable Experience Rollout (VER). VER combines the strengths of and blurs the line between SyncOnRL and AsyncOnRL. Its trains agents for embodied navigation tasks in Habitat 1.0 60%-100% faster (1.6x to 2x speedup) than DD-PPO with similar or better sample efficiency – saving 19.2 GPU-days on PointNav and 28 GPU-day for ObjectNav per seed in our experiments. On the recently introduced (and more challenging) embodied rearrangement tasks in Habitat 2.0, VER trains agents 150% faster than DD-PPO and is fast as SampleFactory (AsyncOnRL) on 1 GPU. On 8 GPUs, VER is 180% faster than DD-PPO and 70% faster than SampleFactory with better sample efficiency – saving 32
GPU-days per skill vs. DD-PPO and 11.2 GPU-days vs. SampleFactory per skill in our experiments. We use VER to study rearrangement. We find the emergence of navigation in policies that ostensibly require no navigation when given access to navigation actions. This results in strong progress on Tidy House (+30% success). This results highlights that it may not always be advantageous to limit a policy’s action space.
CHAPTER 5
DISCUSSION

In this dissertation, we have provided evidence that intelligent navigation behavior emerges as a result of massive-scale simulation and deep reinforcement learning. In Chapter 2, we introduced Decentralized Distributed PPO (DD-PPO), a distributed reinforcement learning framework designed for training large neural networks in environments with photo-realistic rendering. We leverage this scaling to train an agent for 2.5 Billion steps of experience (the equivalent of 80 years of human experience) – over 6 months of GPU-time training in under 3 days of wall-clock time with 64 GPUs. This massive-scale training not only set the state of art on PointGoal navigation, but also essentially ‘solves’ the task – near-perfect autonomous navigation in an unseen environment without access to a map, directly from an RGB-D camera and a GPS+Compass sensor.

In Chapter 3, we examined the inner workings of these PointGoalNav agents. We found that ‘blind’ AI agents are (1) surprisingly effective navigators in new environments (~95% success); (2) they utilize memory over long horizons (remembering ~1,000 steps of past experience in an episode); (3) this memory enables them to take shortcuts; (4) there is emergence of maps in this memory, i.e. a detailed occupancy grid of the environment can be decoded from it; and (5) the emergent maps are selective and task dependent.

In Chapter 4, we presented Variable Experience Rollout (VER), a technique for scaling batched on-policy reinforcement learning in heterogenous environments (where different environments take vastly different times for generating rollouts). We leveraged these speed-ups to train chained skills for GeometricGoal rearrangement tasks in the Home Assistant Benchmark (HAB). We found a surprising emergence of navigation in skills that do not ostensibly require any navigation.

As a direction for future work, we posit a more general hypothesis: ‘Massive-scale
Theoretically, there isn’t a fundamental limit to this more general hypothesis. Homo sapiens, and life on earth for that matter, show that intelligence emerges from massive-scale interaction with the physical world and evolution. We substitute the physical world with simulation and evolution with another optimization algorithm (e.g. deep reinforcement learning), but the principle remains the same: Optimization to maximize some reward and massive-scale interaction. Given this existence proof, what are the failure modes?

One is that while the existence proof shows that intelligence does emerge under these conditions, it says nothing about the probability. The Fermi paradox parallels this in the context of extraterrestrial life. It asks: If the conditions for life are ‘simple’\(^1\), then why are we unaware of any extraterrestrial life? One way to resolve the Fermi paradox is that the conditions for life may not be simple and life may be an extremely low probability event. This probability may be so low such that life will only emerge once. In a similar vein, emergent intelligence may become an increasingly low probability event as the level of intelligence increases.

Another possible failure mode is that of practical limitations. We do not known the amount of compute needed as the level of intelligence increases. It’s reasonable to expect that the amount of compute will increase, but we don’t know the shape of this function. The amount of compute may be so high as to be infeasible.

We also don’t know the complexity of tasks the simulator needs to support. Is single-agent task like rearrangement sufficient? Or does the simulator (and learning system for that matter) need to support large-scale (100,000+) multi-agent dynamics? Clearly these dynamics are needed for certain aspects of intelligence like culture and society, but these may be fundamental to any intelligence emerging.

Finally, we don’t know the fidelity of the simulator needed. Is a relatively coarse physics engine, like we used in mobile manipulation, adequate? Or do we need to ac-

\(^1\)Even 1 in a million would ‘simple’ and lead to an abundance of life given the scale of even just our galaxy let alone the universe.
curately simulate the interactions between subatomic particles?

The answer to all these questions will depend on the level of intelligent behavior we desire. We expect that we can go far without needing a system that could accurately simulate our universe. The question is how far can we go while staying within the bounds of practicality.
Appendices
APPENDIX A

APPENDIX: LEARNING NEAR-PERFECT POINTGOAL NAVIGATORS FROM 2.5 BILLION FRAMES

A.1 DD-PPO Additional analysis and discussion

In this section, we continue the analysis of our agent and examine differences in its behavior from a classical, hand-designed agent – the map-and-plan baseline agent proposed in [69].

Intricacies of SPL. Given an agent that always reaches the goal ($\approx 100\%$ success), SPL can be seen as measuring the efficiency of an agent vs. an oracle – i.e. an SPL of 0.95 means the agent is 5\% less efficient than an oracle. Given the challenges of near-perfect autonomous navigation without a map in novel environments we outlined, being 5\% less efficient than an oracle seems near-impossible. However, this comparison/view is potentially misleading. Percentage errors are potentially misleading for long paths. Over a 10 meter episode, the agent can deviate from the oracle path by up-to a meter and still be within 10\%. As a consequence, significant qualitative errors can result in an insignificant quantitative error (see Fig. A.1).

Error recovery. Given the near-perfect performance of our agent (on average), we explicitly examine if it is able to recover from its own navigation errors. Fig. A.1 column 3 shows several examples of error recovery, including several well executed backtracks (video: https://www.youtube.com/watch?v=a8AugVLSJ50), indicating that the agent is effective at recovering from its own navigation errors. Next, we look at the statistics of non-perfect (SPL<0.99) episodes on the longer validation episodes proposed in [68]. Non-perfect episodes make up the majority of episodes (54\%, see Fig. A.2) with an average SPL of 0.85 (99.0\% success) – compared to 0.92 SPL (99.5\% success) over all episodes. Thus there are many episodes where the agent makes significant deviation from the shortest path
<table>
<thead>
<tr>
<th>Geodesic Distance (rows) / SPL (cols)</th>
<th>0.00-0.20</th>
<th>0.20-0.50</th>
<th>0.50-0.90</th>
<th>0.90-0.95</th>
<th>0.95-1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0-11.7</td>
<td><img src="image1" alt="Image" /></td>
<td><img src="image2" alt="Image" /></td>
<td><img src="image3" alt="Image" /></td>
<td><img src="image4" alt="Image" /></td>
<td><img src="image5" alt="Image" /></td>
</tr>
<tr>
<td>11.7-13.4</td>
<td><img src="image6" alt="Image" /></td>
<td><img src="image7" alt="Image" /></td>
<td><img src="image8" alt="Image" /></td>
<td><img src="image9" alt="Image" /></td>
<td><img src="image10" alt="Image" /></td>
</tr>
<tr>
<td>13.4-15.0</td>
<td><img src="image11" alt="Image" /></td>
<td><img src="image12" alt="Image" /></td>
<td><img src="image13" alt="Image" /></td>
<td><img src="image14" alt="Image" /></td>
<td><img src="image15" alt="Image" /></td>
</tr>
<tr>
<td>15.0-16.7</td>
<td><img src="image16" alt="Image" /></td>
<td><img src="image17" alt="Image" /></td>
<td><img src="image18" alt="Image" /></td>
<td><img src="image19" alt="Image" /></td>
<td><img src="image20" alt="Image" /></td>
</tr>
<tr>
<td>16.7-26.8</td>
<td><img src="image21" alt="Image" /></td>
<td><img src="image22" alt="Image" /></td>
<td><img src="image23" alt="Image" /></td>
<td><img src="image24" alt="Image" /></td>
<td><img src="image25" alt="Image" /></td>
</tr>
</tbody>
</table>

Figure A.1: Example episodes broken down by geodesic distance between agent’s spawn location and target (on rows) vs SPL achieved by the agent (on cols). Gray represents navigable regions on the map while white is non-navigable. The agent begins at the blue square and navigates to the red square. The green line shows the shortest path on the map (or oracle navigation). The blue line shows the agent’s trajectory. The color of the agent’s trajectory changes changes from dark to light over time. Navigation dataset from the longer validation episodes proposed in [68].
and reaches the goal (a 15% deviation on long trajectories (>10m) is significant).

**When does the agent fail?** Column 2 in Fig. A.1 shows that the agent performs poorly when the ratio of the geodesic distance to goal and euclidean distance to goal. However, the agent is able to eventually overcome this failure mode and reach the goal in most cases.

Row 1 column 1 in Fig. A.1 shows that the agent fails or performs poorly when it needs to go slightly up/down stairs. The data-set generation process used in [22] only guarantees a start and goal pair won’t be on different floors, but there remains a possibility that the agent will need to traverse the stairs slightly. However, these situations are rare, and, in general, the stairs should be avoided. Furthermore, the GPS sensor provides location in 2D, not 3D.

The remaining failure cases of column 1 in Fig. A.1 show that a singular location in one environment acts as a sink for the agent (once it enters this location, it is almost never able to leave it). At this location, there is a large hole in the mesh (an entire wall is miss-
ing). Utilizing visual encoders that explicitly handle missing values may allow the agent to overcome this failure mode.

**Differences from a classical agent.** We compare the behavior of our agent with the classical map-and-plan baseline agent proposed in [69]. This agent achieves 0.92 val (0.89 test) SPL with 0.976 success.¹ By comparing and contrasting qualitative behaviors, we can determine what behaviors learning-based methods enable. We make the following observation.

The learned agent is able to recover from unexpected collisions without hurting SPL. The map-and-plan baseline agent incorporates a specific collision recovery behavior where, after repeated collisions, the agent turns around and backs up 1.25m. This behavior brings the obstacle into view, maps it, and then allows the agent to create a plan to avoid it. In contrast, our agent is able to navigate around unseen obstacles without such a large impact on SPL. Determining the set of action sequences and heuristics necessary to do this is what learning enables.

### A.2 DD-PPO Agent Design

In this section, we outline the exact agent design we use. We break the agent into three components: a visual encoder, a goal encoder, and a navigation policy.

**Visual Encoder.** Our visual encoder uses one of three different backbones, ResNet50 [59], Squeeze-Excite(SE) [60]-ResNeXt50 [61], and SE-ResNeXt101. For all backbones, we reduce the number of output channels at each layer by half. We also add a $2\times2$-AvgPool before each backbone so that the effective resolution is $128\times128$. Given these modifications, each backbone produces a $1024\times4\times4$ feature map. We then convert this to a $128\times4\times4$ feature map with a $3\times3$-Conv.

We replace every BatchNorm layer with GroupNorm [63] to account for the highly correlated trajectories seen in on-policy RL and massively distributed training.

¹https://github.com/s-gupta/map-plan-baseline#results
**Goal encoder.** Habitat [22] provides the vector pointing to the goal in ego-centric polar coordinates. We convert this to magnitude and a unit vector, \( [d, \theta] \) to \([d, \cos(\theta), \sin(\theta)]\), to account for the discontinuity at the \(x\)-axis in polar coordinates. We pass the goal vector to a fully connected layer, resulting in a 32-dimensional representation.

**Navigation Policy.** Our navigation policy takes the \(64 \times 4 \times 4\) feature map from the visual encoder, flattens it, and then converts the 2048-d vector to the same size as the hidden size via a fully-connected layer. It then concatenates this vector with output of the goal encoder, and a 32-dimensional embedding of the previous action taken (or the start-token in the case of the first action) and then passes this to a 2-layer LSTM with either a 512-dimensional or 1024-dimensional hidden dimension. The output of the LSTM is used as input to a fully connected layer, resulting in a soft-max distribution of the action space and an estimate of the value function.

**A.3 DD-PPO Additional scaling details**
We use the following procedure for benchmarking the throughput of our proposed DD-PPO: Each optimizer selects 4 scenes at random and then performs the process of collecting experience and optimizing the model based on that experience 10 times. We calculate throughput as the total number of steps of experience collected over the last 5 rollout/optimizing steps divided by the amount of time taken. We repeat this procedure over 10 different random seeds (we use the same random seeds for all variations of number of GPUs and sync-fraction values).

**A.4 DD-PPO Implementation**
Utilizing Distributed Data Parallel in supervised learning is straightforward as frameworks such as PyTorch [58] provide a simple wrapper. The recommended way to use these wrappers is to first write training code that runs on a single GPU and then enable distributed training via the wrapper. We follow a similar approach. Given an implementation of PPO that runs on one GPU we create a decentralized distributed variant by adding gradi-
Figure A.3: Scaling of DD-PPO under homogeneous and heterogeneous workloads for various different values of the percentage of rollouts that are fully completed by optimizing the model. Shading represents a bootstrapped 95% confidence interval.

ent synchronization, leveraging highly performant code written for this purpose in popular deep-learning frameworks, e.g. `tf.distribute.MirroredStrategy` in TensorFlow [81] and `torch.nn.parallel.DistributedDataParallel` in PyTorch. Note that care must be taken to synchronize any training or rollout statistics between workers – in most cases these can also be synchronized via AllReduce.

We track how many workers have finished the experience collection stage with a distributed key-value storage – we use PyTorch’s `torch.distributed.TCPStore`, however almost any distributed key-value storage would be sufficient.

See Fig. A.4 for an example implementation which adds 1) gradient synchronization via `torch.nn.parallel.DistributedDataParallel`, and 2) preempts stragglers by tracking
the number of workers have finished the experience collection stage with a `torch.distributed.TCPStore`.

See Fig. A.5 for a visual depiction of DD-PPO.

A.5 DD-PPO Transfer experiments additional details

For the transfer learning experiments, we utilize the same PPO hyper-parameters as the PointGoalNav experiments. We use DD-PPO to train with 8 workers on 8 GPUs. We train our agents on Gibson-4+ and evaluate on the Habitat Challenge 2019 Validation scene and starting locations (the goal location is simply discarded).

The ImageNet encoder is trained using the same hyper-parameters and training procedure as [61] with no data-augmentation.

A.6 DD-PPO Neural controller additional details

The planner for neural controller used in Section 2.6 shares the same architecture as our agent’s policy, but utilizes a 512-d hidden state. It takes as input the previous action of the controller (or the start token), and the output of the visual encoder (which is shared with the controller). The output of the LSTM is then used to produced an estimate of the value function and a 3-dimensional vector specifying the PointGoal in magnitude and unit direction vector format. The magnitude competent is passed through an ELU activation and offset by 0.75. Each component of the unit direction vector is passed through a tanh activation – note that we do not re-normalize this vector have a length of 1 as we find doing so both unnecessary and harder to optimize.
master_addr = # <hostname of world rank 0's machine>
master_port = # <free TCP port on world rank 0's machine>
world_rank = # <worker's unique ID>
world_size = # <number of workers>
local_rank = # <the ID of the GPU to use>

# Setup the group of workers
store = torch.distributed.TCPStore(
    master_addr,
    master_port,
    world_size,
    world_rank == 0,
)
torch.distributed.init_process_group(
    backend="NCCL",
    world_size=world_size,
    rank=world_rank,
    store=store,
)

# Tracks how many workers have finished their rollout
num_workers_done = torch.distributed.PrefixStore(
    "num_workers_done",
    store
)
device = torch.device("cuda", local_rank)
model = PolicyNetwork(...)
model.to(device)

# Add gradient synchronization to the model
model = torch.nn.parallel.DistributedDataParallel(
    model, [device], device
)

while not_converged():
    num_workers_done.set("done", "0")
    for step in range(max_experience_steps):
        collect_step(model)
        # Preempt stragglers
        if (int(num_workers_done.get("done")) > preemption_threshold * world_size
            and step >> max_experience_steps / 4):
            break
        # Mark that a worker is done collecting experience
        num_workers_done.add("done", 1)

    # Update the model using PPO
    for _ in range(n_ppo_epochs):
        for _ in range(n_ppo_batch):
            batch = get_batch()
            loss = evaluate(model, batch)
            loss.backward()
            # DistributedDataParallel automatically
            # performs an AllReduce on all gradients
            # during the backward call.
            # If this wasn't being used, here is where
            # calls to AllReduce on the gradients would
            # be made.
            step_optimizer(model)

Figure A.4: Implementation of DD-PPO using PyTorch [58] v1.1 and the NCCL backend. We use SLURM to populate the world_rank, world_size, and local_rank fields.
Figure A.5: Illustration of DD-PPO. Processes collecting experience in environments that are more costly to simulate (stragglers) have their experience collection stage preempted such that other processes do not have to wait for them. Note that we implement the monitor with a simple key-value storage and have processes preempt themselves. Note that the order of processes is irrelevant and done solely for aesthetic purposes.

Table A.1: Performance (higher is better) of various sensors and agent methods on the Habitat Challenge 2019 [22] validation and test splits (checkpoint selected on val). Random, Forward-only, and Goal-follower taken from [22]. Best visual encoder reported for DD-PPO.

<table>
<thead>
<tr>
<th>Perception</th>
<th>Method</th>
<th>Validation</th>
<th></th>
<th>Test Standard</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SPL</td>
<td>Success</td>
<td>SPL</td>
<td>Success</td>
</tr>
<tr>
<td>Blind</td>
<td>Random</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Forward-only</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Goal-follower</td>
<td>0.23</td>
<td>0.23</td>
<td>0.23</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>DD-PPO (RL)</td>
<td>0.729 ± 0.005</td>
<td>0.973 ± 0.003</td>
<td>0.676</td>
<td>0.947</td>
</tr>
<tr>
<td>RGB</td>
<td>DD-PPO (RL)</td>
<td>0.929 ± 0.003</td>
<td>0.991 ± 0.002</td>
<td>0.920</td>
<td>0.977</td>
</tr>
<tr>
<td>RGB-D (Depth)</td>
<td>DD-PPO (RL)</td>
<td>0.969 ± 0.002</td>
<td>0.997 ± 0.001</td>
<td>0.948</td>
<td>0.980</td>
</tr>
</tbody>
</table>
Figure A.6: Examples of Gibson meshes for a given quality rating from [22]
Figure A.7: Training and validation performance (in SPL; higher is better) of different architectures for Depth agents with GPS+Compass on the Habitat Challenge 2019 [22]. Gibson [66]-4+ refers to the subset of Gibson train scenes with a quality rating of 4 or better. Gibson-4+ and MP3D refers to training on both Gibson-4+ and all of Matterport3D. Gibson-2+ refers to training on the subset of Gibson train scenes with a quality rating of 2 or better.

Figure A.8: Performance vs. Geodesic Distance from start to goal for Blind, RGB, and RGB-D (using Depth only) models trained with DD-PPO on the Habitat Challenge 2019 [22] validation split. Bars at the bottom represent the fraction of episodes within each geodesic distance bin.
Figure A.9: Performance vs. Geodesic Distance from start to goal for Blind, RGB, and RGB-D (using Depth only) models trained with DD-PPO on the longer and harder validation episodes proposed in [68]. Bars at the bottom represent the fraction of episodes within each geodesic distance bin.
B.1 PointGoal Navigation Training

Task. In PointGoal Navigation, the agent is tasked with navigating to a point specified relative to its initial location, i.e. an input of \((\delta x, \delta y)\) corresponds to going \(\delta x\) meters forward and \(\delta y\) meters to the right. The agent succeeds if it predicts the stop action within 0.2 meters of the specified point. The agent has access to 4 low-level actions – move\_forward (0.25 meters), turn\_left \((10^\circ)\), turn\_right \((10^\circ)\), and stop.

Sensors. The agent has access to solely an idealized GPS+Compass sensor that provides it heading and position relative to the starting orientation and location at each time step.

Architecture. The agent is parameterized by a 3-layer LSTM \([130]\) with a 512-d hidden dimension. At each time-step, the agent receives observations \(g\) (the location of the goal relative to start), GPS (its current position relative to start), and compass (its current heading relative to start). We also explicitly give the agent an indicator of if it is close to goal in the form of \(\min(||g - GPS||, 0.5)\) as we find the agent does not learn robust stopping logic otherwise. All 4 inputs are projected to 32-d using separated fully-connected layers. These are then concatenated with a learned 32-d embedding of the previous action taken to form a 160-d input that is then given to the LSTM. The output of the LSTM is then processed by a fully-connected layer to produce a softmax distribution of the action space and an estimate of the value function.

Training Data. We construct our training data based on the Gibson \([140]\) and Matterport3D dataset \([67]\). We training on 411 scenes from Gibson and 72 from Matterport3D.
Training Procedure. We train our agents using Proximal Policy Optimization (PPO) [141] with Generalized Advantage Estimation (GAE) [131]. We use Decentralized Distributed PPO (DD-PPO)(Chapter 2) to train on 16 GPUs. Each GPU/worker collects 256 steps of experience from 16 agents (each in different scenes) and then performs 2 epochs of PPO with 2 mini-batchs per epoch. We use the Adam optimize [65] with a learning rate of $2.5 \times 10^{-4}$. We set the discount factor $\gamma$ to 0.99, the PPO clip to 0.2, and the GAE hyper-parameter $\tau$ to 0.95. We train until convergence (around 2 billion steps of experience).

At every timestep, $t$, the agent is in state $s_t$ and takes action $a_t$, and transitions to state $s_{t+1}$. It receives shaped reward in the form:

$$
    r_t = \begin{cases}
        2.5 \cdot \text{Success} & \text{if } a_t \text{ is Stop} \\
        -\Delta_{\text{geo-dist}}(s_t, s_{t+1}) - \lambda & \text{Otherwise}
    \end{cases}
$$

(B.1)

where $\Delta_{\text{geo-dist}}(s_t, s_{t+1})$ is the change in geodesic (shortest path) distance to goal between $s_t$ and $s_{t+1}$ and $\lambda=0.001$ is a slack penalty encouraging shorter episodes.

Evaluation Procedure. We evaluate the agent in the 18 scenes from the Matterport3D test set. We use the episodes from Savva et al. [22], which consist of 56 episodes per scene (1008 in total). Episode range in distance from 1.2 to 30 meters. The ratio of geodesic distance to euclidean distance between start and goal is restricted to be greater than or equal to 1.1, ensuring that episodes are not simple straight lines. Note that reward is not available during evaluation.

The agent is evaluated under two metrics, Success, whether or not the agent called the stop action with 0.2 meters of the goal and Success weighted by normalized inverse Path Length (SPL) [13]. SPL is calculated as follows: given the agent’s path $[s_1, \ldots, s_T]$ and the initial geodesic distance to goal $d_i$ for episode $i$, we first compute the length of the agent’s
path

\[ l_i = \sum_{t=2}^{T} ||s_t - s_{t-1}||_2 \]  \hspace{1cm} (B.2)

then SPL for episode \( i \) as

\[ \text{SPL}_i = \text{Success}_i \cdot \frac{d_i}{\min\{d_i, l_i\}} \]  \hspace{1cm} (B.3)

We then report SPL as the average of SPL\(_i\) across all episodes.

**B.2  Probe Training**

**Task.** The probe task is to either navigate from start to goal again (SecondNav(\( S \rightarrow T \)) or navigate from goal to start (SecondNav(\( T \rightarrow S \))). For SecondNav(\( S \rightarrow T \)), the probe is initialized at the starting location but *with* the agent’s final heading. For SecondNav(\( T \rightarrow S \)), the probe is initialized with the agent’s final heading and position. In both cases, the probe and the agent share the same coordinate system – *i.e.* in SecondNav(\( T \rightarrow S \)), the initial GPS and Compass readings for the probe are identical the the final GPS and Compass readings for the agent.

When the agent does not successfully reach the goal, the probe task is necessarily undefined and we do not instantiate a probe. The agent reaches the goal 95% of the time, thus only 50 out of 1008 possible probe episodes are invalidated. The control probe type accounts for this.

**Sensors, Architecture, Training Procedure, Training Data.** The probe uses the same sensor suite, architecture, training procedure, and training data as the agent, described in Section B.1

**Evaluation Procedure.** We evaluate the probe in a similar manner the agent except that any episode which the agent is unable to complete (5%) is removed due to the probe task
being undefined if the agent is unable to complete the task. We ignore the agent’s trajectory when computing SPL for the probe.

### B.3 Occupancy Map Decoding

**Task.** We train a decoding network to predict the top-down occupancy map of the environment from the final internal state of the agent \((h_t, c_t)\). We limit the decoder to only predict within 2.5 meters of any location the agent visited.

**Architecture.** The map-decoder is constructed as follows: First the internal state \((h_t, c_t)\) is concatenated into a \(512 \times 6\)-d vector. The vector is then passed to a 2-layer MLP with a hidden dimension of 512-d that produces a 4608-d vector. This 4608-d vector is then reshaped into a \([128, 6, 6]\) feature-map. The feature map is processed by a series of Coordinate Convolution (CoordConv) \([142]\) Coordinate Up-Convolution (CoordUpConv) layers decrease the channel-depth and increase spatial resolution to \([16, 96, 96]\). Specifically, after an initial CoordConv with an output channel-depth of 128, we use a series of 4 CoordUpConv-CoordConv layers where each CoordUpConv doubles the spatial dimensions (quadruples spatial resolution) and each CoordConv reduces channel-depth by half. We then use a final 1x1-Convolution to create a \([2, 96, 96]\) tensor representing the non-normalized log-probabilities of whether or not an given location is navigable or not.

Each CoordConv has kernel size 3, padding 1, and stride 1. CoordUpConv has kernel size 3, padding 0, and stride 2. Before all CoordConv and CoordUpConv, we use 2D Dropout \([143, 144]\) with a zero-out probability of 0.05. We use Batch Normalization layers \([62]\) and the ReLU activation function \([145]\) after all layers except the terminal layer.

**Training Data.** We construct our training data by having a trained agent perform episodes of PointGoal navigation on the training dataset. Note that while evaluation is done utilizing the final hidden state, we construct our training dataset by taking 30 time steps (evenly spaced) from the trajectory and ensuring the final step is included.
**Training Procedure.** We train on 8 GPUs with a batch size of 128 per GPU (total batch size of 1024). We use the AdamW optimizer [65, 146] with an initial learning rate of $10^{-3}$ and linearly scale the learning rate to $1.6 \times 10^{-2}$ over the first 5 epochs [77] and use a weight-decay of $10^{-5}$. We use the validation dataset to perform early-stopping. We use Focal Loss [147] (a weighted version of Cross Entropy Loss) with $\gamma = 2.0$, $\alpha_{\text{NotNavigable}} = 0.75$, and $\alpha_{\text{Navigable}} = 0.25$ to handle the class imbalance.

**Evaluation Data and Procedure.** We construct our evaluation data using the validation dataset. Note that the scenes in evaluation are novel to both the agent and the decoder. We evaluate the predicted occupancy map from the final hidden state/final time step. We collect a total of 5,000 episodes.

**B.4 Past and Future Position Prediction**

**Task.** We train a decoder to predict the change in agent location given the internal state at time $t$ ($h_t, c_t$). Specifically, let $s_t$ be the agent’s position at time $t$ where the coordinate system is defined by the agent’s starting location (i.e. $s_0 = 0$), and $s_{t+k}$ be its position $k$ steps into the future/past, then the decoder is trained to model $f((h_t, c_t)) = s_{t+k} - s_t$.

**Architecture.** The decoder is a 3-layer MLP that produces a 3 dimensional output with hidden sizes of 256 and 128. We use Batch Normalization [62] and the ReLU activation function [145] after all layers except the last.

**Training Data.** The training data is collected from executing a trained agent on episodes from the training set. For each episode, we collect all possible pairs of $s_t, s_{t+k}$ for a given value of $k$.

**Training Procedure.** We use the AdamW optimizer [65, 146] with a learning rate of $10^{-3}$, a weight decay of $10^{-4}$, and a batch size of 256. We use a Smooth L1 Loss/Hu-
ber Loss \cite{148} between the ground-truth change in position and the predicted change in position. We use the validation set to perform early stopping.

**Evaluation Procedure.** We evaluate the trained decoder on held-out scenes. Note that the held-out scenes are novel both to the agent and the decoder.

**Visualization of Predictions.** For visualization the predictions of past vitiation, we found it easier to instead train a second decoder that predicts all locations the agent visited previously on a 2D top down map given the internal state \((h_t, c_t)\). This decoder shares the exact same architecture and training procedure as the occupancy grid decoder. The decoder removes the temporal aspect from the prediction, so it is ill-suited for any time-dependent analysis, but produces clearer visualizations.

**Excursion Calibrated Analysis.** To perform the excursions forgetting analysis, we use the excursion labeled episodes. We marked the end of the excursion as the last 10\% of the steps that are part of the excursion. For a given point in time \(t\), we classify that point into one of \{Non-Excursion, Excursion, Exit\}. We then examine how well this point is remembered by calculating the error of predicting the point \(t\) from \(t + k\), i.e. how well can \(t\) be predicted when it is \(k\) steps into the past. When \(t\) is part of an excursions (both the excursion and the exit) we limit \(t + k\) to either be part of the same excursion or not part of an excursion. When \(t\) is not part of an excursion, \(t + k\) must also not be part of an excursion nor can there be any excursion in the range \([t, t + k]\).

**B.5 Collision Prediction Linear Probe**

**Task.** The task of this probe is to predict of the previous action taken lead to a collision given the current hidden state. Specifically it seeks to learn a function \(\text{Collided}_t = f((h_t, c_t))\) where \((h_t, c_t)\) is the internal state at time \(t\) and \(\text{Collided}_t\) is whether or not the previous action, \(a_{t-1}\) lead to a collision.
**Architecture.** The architecture is logistic classifier that takes the concatenation of the internal state and produces logprob of Collided.

**Training Data.** We construct our training data by having a trained agent perform episodes of PointGoal navigation on the training set. We collect a total of 10 million samples and then randomly select 1 million for training. We then normalize each dimension independently by computing mean and standard deviation and then subtract mean and divide by standard deviation. This ensures that all dimensions have the same average magnitude.

**Training Procedure.** We training on 1 GPU with a batch size of 256. We use the Adam optimizer [65] with a learning rate of $5 \times 10^{-4}$. We train for 20 epochs.

**Evaluation Data and Procedure.** We construct our evaluation data using the same procedure as the training data, but on the validation dataset and collect 200,00 samples (which is then subsampled to 20,000).

**Important Dimension Selection.** To select which dimensions are important for predicting collisions, we re-train our probe with various L1 penalties. We sweep from 0 to 1000 and then select the penalty that results in the lowest number of significant dimensions without substantially reducing accuracy. We determine the number of significant dimensions by first ordering all dimensions by the L1 norm of the corresponding weight and then finding the smallest number of dimensions we can keep while maintaining 99% of the performance of keeping all dimensions.

**B.6 Blind shortest path navigation with true state**

In the main text, we posited that blind agents learn wall-following as this an effective strategy for blind navigation in unknown environments. We posit that this is because the agent does not have access to true state (it does not know the current environment nor
where it is in global coordinates). In this experiment we show that blind agents learn to take shortest paths, as opposed to wall-following, when trained in a single environment (implicitly informing the agent of the current environment) and uses the global coordinate system.\(^1\)

We use an identical agent architecture and training procedure as outline for PointGoal navigation training in the Materials and Methods with two differences: 1) A single training and test environment and 2) usage of the global coordinates within the environment for both goal specific and the agent’s GPS+Compass sensor. We perform this experiment on 3 scenes, 1 from the Gibson val dataset and 2 from Matterport3D val dataset. The average SPL during training is \(99_{\pm0.1}\) showing that the blind agent learns shortest path navigation not wall-following. Figure B.1 shows examples of an agent trained in a single scene with global coordinates and an agent trained in many scenes with episodic coordinates.

**B.7 Further analysis of the probe’s performance**

In the main text, we showed that the probe is indeed much more efficient than the agent, but how is this gain achieved? Our hypothesis is that the probe improves upon the agent’s path by taking shortcuts and eliminating excursions (representing an ‘out and back’). We define an excursion as a sub-path that approximately forms a loop. To quantify excursions, we manually annotate excursions in 216 randomly sampled episodes in evaluation environments. Of the labeled episodes, 62% have at least 1 excursion. On average, an episode has 0.95 excursions, and excursions have an average length of 101 steps (corresponding to 8.23 meters). Since excursions represent unnecessary portions of the trajectory, this indicates that the probe should be able improve upon the agent’s path by removing these excursions.

We quantify this excursion removal via the normalized Chamfer distance between the

\(^1\)Recall that in the episodic coordinate system the origin is defined by the agent’s starting position and orientation. In the global coordinate system the origin is an arbitrary but consistent location (we simply use the origin for a given scene defined in the dataset). Thus in the global coordinate system the goal is specified as ‘Go to \((x, y)\)’ where \(x\) and \(y\) are specified in the global coordinate system, not with respect to the agent’s current location.
agent’s path and the probe’s path. Formally, given the agent’s path \( \text{Agent}=\left[s^{(agent)}_1,\ldots,s^{(agent)}_T\right] \) and the probe’s path \( \text{Probe}=\left[s^{(probe)}_1,\ldots,s^{(probe)}_N\right] \) where \( s \in \mathbb{R}^3 \) is a point in the environment:

\[
\text{PathDiff}(\text{Agent}, \text{Probe}) = \frac{1}{N} \sum_{i=1}^{N} \min_{1 \leq j \leq T} \text{GeoDist}(s^{(agent)}_i, s^{(probe)}_j),
\]

where GeoDist(\( \cdot, \cdot \)) indicates the geodesic distance (shortest traverseable path-length).

Note that Chamfer distance is not symmetric. \( \text{PathDiff(Probe, Agent)} \) measures the average distance of a point on the probe path \( s^{(probe)}_j \) from the closest point on the agent path. A large \( \text{PathDiff(Probe, Agent)} \) indicates that the probe travels through novel parts of the environments (compared to the agent). Conversely, \( \text{PathDiff(Agent, Probe)} \) measures the average distance of a point on the agent path \( s^{(agent)}_i \) from the closest point on the probe path. A large \( (\text{PathDiff(Agent, Probe)} - \text{PathDiff(Probe, Agent)}) \) gap indicates that agent path contains excursions while the probe does not; thus, we refer to this gap as Excursion Removal. To visually understand why this is the case, consider the example agent and probe paths in Fig. B.2. Point (C) lies on an excursion in the agent path. It contributes a term to \( \text{PathDiff(Agent, Probe)} \) but not to \( \text{PathDiff(Probe, Agent)} \) because (D) is closer to (E) than (C).

On both SecondNav(S\( \rightarrow \)T) and SecondNav(T\( \rightarrow \)S), we find that as the efficiency of a probe increases, Excursion Removal also increases (Table B.1, row 1 vs. 2, 2 vs. 3), confirming that the TrainedAgentMemory probe is more efficient because it removes excursions.

We next consider if the TrainedAgentMemory probe also travels through previously unexplored space in addition to removing excursions. To quantify this, we report \( \text{PathDiff(Probe, Agent)} \) on episodes where agent SPL is less than average (less than 62.9%). If probes take the same path as the agent, we would expect this metric to be zero. If, how-

\(^2\)We restrict to a subset where the agent has relatively low SPL to improve dynamic range. When the agent has high SPL, there won’t be excursions to remove and this metric will naturally be low. In the supplementary text we provide plots of this metric vs. agent SPL.
ever, probes travel through previously unexplored space to minimize travel distance, we would expect this metric to be significantly non-zero. Indeed, on SecondNav(S→T), we find the TrainedAgentMemory probe is 0.32 meters away on average from the closest point on the agent’s path (99% empirical bootstrap of the mean gives a range of (0.299, 0.341)). See Fig. B.2 for a visual example. On SecondNav(T→S), this effect is slightly more pronounced, the TrainedAgentMemory probe is 0.55 meters away on average (99% empirical bootstrap of the mean gives a range of (0.52, 0.588)). Taken holistically, these results show that the probe is both more efficient than the agent and consistently travels through new parts of the environment (that the agent did not travel through). Thus, the spatial representation in the agent’s memory is not simply a ‘literal’ episodic summarization, but also contains anticipatory inferences about previously unexplored spaces being navigable (e.g. traveling along the hypotenuses instead of sides of a room).

B.8 Past and Future Visitation Prediction

In the main text we examined what types of systematic errors are made when decoding past agent locations, here we provide addition analysis and look at predicting future observations as that will reveal if there are any idiosyncrasies in what can be predicted about future vs. what will happen in the future. In Fig. B.3, we find that the decoder is able to accurately predict where the agent has been, even for long time horizons – e.g. at 100 time steps in the past, relative error is 0.55 and absolute error is 1.0m, compared to relative error of 1.0 and absolute error of 3.2m for the chance baseline prediction. For short time horizons the decoder is also able to accurately predict where the agent will be in the future – e.g. at 10 time steps into the future, relative and absolute error are below chance. Interestingly, we see that for longer range future predictions, the decoder is worse than chance in relative error but on-par in absolute error. This apparent contradiction arises due to the decoders making (relatively) large systematic errors when the agent backtracks. In order for the decoder to predict backtracking, the agent would need to already know its future trajectory will be
sub-optimal (i.e. lead to backtracking) but still take that trajectory. This is in contradiction with the objective the agent is trained for, to reach the goal as quickly as possible.

**B.9 Extension to Sighted Navigation Agents**

In the main text we analyzed how ‘blind’ agents, those with limited perceptual systems, utilize their memory and found evidence that they build cognitive maps. Here, we extend our analysis to agents with rich perceptual systems, those equipped with a Depth camera and an egomotion sensor. Our primary experimental paradigm relies on showing that a probe is able to take shortcuts when given the agent’s memory. This experimental paradigm relies on the probe being able to take a shorter path than the agent. Navigation agents with vision can perform PointNav near-perfectly (Chapter 2) and thus there isn’t room for improving, rendering this experiment infeasible. As a supplement to this experiment, we also show that a metric map (top-down occupancy grid) can be decoded from the agents memory. This procedure can also be applied to sighted agents.

We use the ResNet50 [59] Gibson-2plus [140] pre-train model from Wijmans et al. (Chapter 2) and train an occupancy grid decoder using the same procedure as in the main text. Note however we utilize only Gibson for training and the Gibson validation scenes as held-out data instead of Matterport3D as this agent was only trained on Gibson. As before, we compare performance from TrainedAgentMemory with UntrainedAgentMemory.

We find mixed results. When measuring performance with Intersection-over-Union (IoU), UntrainedAgentMemory outperforms TrainedAgentMemory (40.1% vs. 42.9%). However, when measuring performance with average class balanced accuracy, TrainedAgentMemory outperforms UntrainedAgentMemory (61.8% vs. 53.1%). Fig. B.4 and Fig. B.5 show the corresponding distribution plots.

Overall, this experiment does not provide convincing evidence either way to whether vision-equipped agents build metric maps in their memory. However, it does show that vision-equipped agents, if they do maintain a map of their environment, create one that is
considerably more challenging to decode. Further, we note this does not necessarily imply similarly mixed results as to whether or not vision agents maintain a still spatial but sparser representation, such as a topological graph, as their rich perception can fill in the details in the moment.

B.10 Navigation from Memory Alone

In the main text we showed that agents learn to build map-like representations. A map-like representation of the environment, should, to a degree, support navigation with no external information, *i.e.* by dead reckoning. Given that the actions are deterministic, the probe should be able to perform either task without external inputs and only the agent’s internal representation and the previously taken action. The localization performed by the probe in this setting is similar to path integration, however, it must also be able to handle any collisions that occur when navigating.

Fig. B.6 shows performance vs. episode length for SecondNav($S \rightarrow T$) and SecondNav($T \rightarrow S$). There are two primary trends. For short navigation episodes ($\leq 5m$), the agent is able to complete the task often. We also find that under this setting, SecondNav($T \rightarrow S$) is an easier task. This is due to the information conveyed to the probe by its initial heading. In SecondNav($T \rightarrow S$), the probe can make progress by simply turning around and going forward, while in SecondNav($S \rightarrow T$), the final heading of the agent is not informative of which way the probe should navigate initially. Overall, these results show that the representation built by the agent is sufficient to navigate short distances with *no external information*.

**Experiment procedure** This experiment mirrors the probe experiment described in methods and materials with three differences: 1) The input from the GPS+Compass sensor is zero-ed out. 2) The change in distance to goal shaping in the reward is normalized by the distance from initial state to goal. We find that the prediction of the value function suffers considerably otherwise. 3) An additional reward signal as to whether or not the last action
taken decreased the angle between the probe’s current heading and the direction along the shortest path to goal is added. We find the probe has challenges learning to turn around on the SecondNav(T→S) task otherwise (as it almost always starts facing 180° in the wrong direction).

Let $h_{gt}^t$ be the heading along the shortest path to goal from the probe’s current position $s_t$, $h_t$ be the probe’s current heading, then AngularDistance($h_{gt}^t$, $h_t$) is the error in the probe’s heading. The full reward for this probe is then

$$r_t = \begin{cases} 
2.5 \cdot \text{Success} & \text{if } a_t \text{ is Stop} \\
-10.0/\text{GeoDist}(s_0, g) & \\
\Delta_{\text{geo dist}}(s_t, s_{t+1}) & \\
-0.25 \cdot \Delta_{\text{HeadingError}} & \\
-\lambda & \text{Otherwise}
\end{cases}$$  \hspace{1cm} (B.5)

B.11 Excursion Removal and Free Space Inference

In the main text, we reported free space inference only on episodes where the agent gets an SPL below average as. In Fig. B.7 we provide a plot of Free Space Inference vs. Agent SPL to show the impact of other cutoff points. In Fig. B.8 we also provide a similar plot of Excursion Removal vs. Agent SPL.

B.12 Memory Length

The method presented in the main text to examine memory length is post-hoc analysis performed on the ‘blind’ PointGoal Navigation agents. We also examined training agents with a fixed memory length LSTM. Fig. B.9 shows similar trends to those described in the main paper. However, training for a fixed memory length has the consequence of inducing
Table B.1: Excursion removal result of our trained probe agent under three configurations – initialized with an empty representation (AllZeroMemory), a representation of a random agent walked along the trained agent’s path (UntrainedAgentMemory), and the final representation of the trained agent (TrainedAgentMemory). 95% confidence interval reported over 5 agent-probe pairs.

<table>
<thead>
<tr>
<th>Probe Type</th>
<th>SecondNav(S→T) Excursion Removal</th>
<th>SecondNav(T→S) Excursion Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 AllZeroMemory</td>
<td>0.21±0.017</td>
<td>0.21±0.004</td>
</tr>
<tr>
<td>2 UntrainedAgentMemory</td>
<td>0.23±0.009</td>
<td>0.25±0.009</td>
</tr>
<tr>
<td>3 TrainedAgentMemory</td>
<td>0.52±0.014</td>
<td>0.51±0.011</td>
</tr>
</tbody>
</table>

the negative effects of large-batch optimization [149], thus we are unable to recover full performance. We further note the increased compute needed to train the model (training a model with a memory length of 128 is 128× more computationally costly), thus we stopped at 256.

B.13 Videos

Videos B.1-3 Videos showing blind agent navigation with the location of the hidden state in the collision t-SNE space. Notice that the hidden state stays within a cluster throughout a series of actions.
Figure B.1: True state trajectory comparison. Example trajectories of an agent with true state (trained for a specific environment and using global coordinates), green line, compared to an agent trained for many environments and using episodic coordinates, blue line. The later is what we examine in this work. Notice that the agent with true state take shortest path trajectories while the agent without true state instead exhibits strong wall-following behavior.
Figure B.2: Two categories of probe shortcut. ‘Excursion Removal’ is when the probe removes excursions from the agent’s path. The dashed line shows the distance between the points in the excursion and the closest point in the probe’s path. ‘Free Space Inference’ occurs when the probe travels through previously unvisited locations in the environments. The dashed lines show the distance between any points in the probe’s path and the closest point in the agent’s path.
Figure B.3: **Past and future prediction.** Performance of decoders trained to predict where the agent was in the past/will be in the future. On the x-axis is how far into the past or future the decoder is predicting (positive values are future predictions and negative values are past predictions). The y-axis is either absolute or relative L2 error between the predicted location of the agent and the true location.

Figure B.4: Map prediction accuracy (Intersection over Union) for Depth sensor equipped agents.
Figure B.5: Map prediction accuracy (class balanced accuracy) for Depth sensor equipped agents.

Figure B.6: **Memory-only probe performance.** Performance (in SPL; higher is better) as a function of geodesic distance from start to goal for the TrainedAgentMemory probe without inputs on SecondNav(S → T) and SecondNav(T → S). More information can be found under the ‘Navigation from memory alone’ header.
Figure B.7: Free Space Inference for the TrainedAgentMemory probe on both Second-Nav(S→T) and SecondNav(T→S) as a function of agent SPL. We see that as agent SPL decreases, the probe is able to take paths that inference more free space.

Figure B.8: Excursion Removal for the TrainedAgentMemory probe on both Second-Nav(S→T) and SecondNav(T→S) as a function of agent SPL. We see that as agent SPL decreases, excursion removal increases since the probe is able to remove additional excursions.
Figure B.9: Performance vs. memory length for agents trained under a given memory length. Note that longer memory lengths are challenging to train for under this methodology as it induces the negative effects of large-batch optimization and is computationally expensive.
APPENDIX C

APPENDIX: EMERGENCE OF NAVIGATION IN MOBILE MANIPULATION AGENTS

C.1 Systems Considerations

C.1.1 Memory benefits of not overlapping experience collection and learning

VER does not overlap experience collection and learning. This allows the main process to act as an inference worker during experience collection and a learner during learning, saving 1 CUDA context (≈1GB of GPU memory).

Unlike AsyncOnRL, VER is able to use GPU shared memory to transfer experience from inference workers to learner. This is because VER uses (at least) half the GPU memory for storing rollouts. AsyncOnRL must maintain at least 1 set of rollout buffers for storing the rollout currently being used for learning an another set to store the next rollout being collected. In practice, they maintain more than this minimum 2x to enable faster training.

C.1.2 Graphics Processing Units (GPUs)

The application of graphics processing units (GPUs) to training neural networks has led to many advances in artificial intelligence. In reinforcement learning, GPUs are used for policy inference, learning the policy, and rendering visual observations like Depth or RGB. Given the importance of GPUs in reinforcement learning, there are several important attributes of them to understand.

1) GPUs are not optimized for multiple processes using them concurrently. They are unable to efficiently execute multiple processes at once (if at all). While sometimes concurrent use is the only option, a system should whenever possible have a single process that has exclusive use of the GPU and uses it fully. 2) GPU drivers have different compute
modes for graphics (i.e. rendering) and compute (i.e. neural network inference). Current GPU drivers have to perform an expensive context switch whenever alternating between these modes. Thus a system should minimize the number of times the GPU must switch between compute and graphics. 3) The compute mode CUDA context for process and each CUDA context requires a large amount of memory. Thus a system should use the minimal number of CUDA contexts possible.

C.2 Skill Deployment

We use the same skills and deployment procedure as [23] with two changes to the navigation skill: 1) We have a dedicated stop action for navigation instead of using \( a_t = 0 \) as stop. We find this to be more robust. 2) We add a force penalty to training, this makes the navigation skill less likely to bump into things by accident. However, this also makes it stop immediately if it ever starts in contact with the environment. During evaluation, we mask its prediction of stop if the target object is more than 2 meters away. This information is part of its sensor suite, so this does not require any privileged information.

On Set Table, we add an additional stage to the planner that calls navigation again after open cabinet as the open cabinet skill with base movement tends to move away from the cabinet.

C.3 Videos

In the supplement, we include the following videos.

**Video 1 (1-tp-srl-no-nav.mp4)** This is an example of TP-SRL(NoNav)+All Base Movement on Tidy House. The objects to pick are have a white wireframe box drawn around them and at their place locations there is another copy of that object. The grey sphere on the ground denotes where the navigation skill was trained to navigate to. This information is not shown to the policy.

The policy makes errors and picks up the wrong object
**Video 2 (2-tp-srl.mp4)** This is an example of TP-SRL+All Base Movement on Tidy House.

**Video 3 (3-pick-*.mp4)** Examples of the pick policy on its training task. Note the difference in spawn distance between this and Video 1.

**Video 3 (3-place-*.mp4)** Examples of the place policy on its training task.

### C.4 Hyperparamaters

The hyperparameters for our experiments are in Table C.1.

For the benchmark comparisons, we set the rollout length $T$ to 128 for methods that require this, that way all methods collect the same amount of experience per rollout. For SampleFactory, we use a batch size of 1024 on 1 GPU, 2048 on 2, and 4096 on 4 and 8 (this the GPU memory limit). We do no use a learned entropy coefficient for benchmarking as not all frameworks support this. We use a fixed value of $10^{-4}$ as this works well for Open Fridge.

### C.5 HTS-RL Comparison

HTS-RL [119] uses the sampling method of AsyncOnRL to collect experience for SyncOnRL (the same as NoVER and VER). It further uses delayed gradients to enable overlapped experience and learning. Unfortunately the provided implementation\(^1\) has inefficiencies that limit its throughput.

To demonstrate this, we add the same style of overlapped experience collection and learning to NoVER and compare the provided HTS-RL implementation, HTS-RL (Provided), with our re-implementation, HTS-RL (Ours). Our implementation 110% faster than the provided implementation (Table C.2). Key differences in our implementation are fast userspace mutexes (futexes) in shared memory for synchronization (vs. spin locks), pre-allocated pinned memory for CPU to GPU transfers (vs. allocating for each transfer).

\(^{1}\)https://github.com/IouJenLiu/HTS-RL
Table C.1: Hyperparameters

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizer</td>
<td>Adam [65]</td>
</tr>
<tr>
<td>Initial Learning Rate</td>
<td>$2.5 \times 10^{-4}$</td>
</tr>
<tr>
<td>Final Learning Rate</td>
<td>0</td>
</tr>
<tr>
<td>Decay Schedule</td>
<td>Cosine</td>
</tr>
<tr>
<td>PPO Epochs</td>
<td>3</td>
</tr>
<tr>
<td>Mini-batches per Epoch</td>
<td>2</td>
</tr>
<tr>
<td>PPO Clip</td>
<td>0.2</td>
</tr>
<tr>
<td>Initial Entropy Coefficient</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>Entropy Coefficient Bounds</td>
<td>$[10^{-4}, 1.0]$</td>
</tr>
<tr>
<td>Target Entropy</td>
<td>0.0</td>
</tr>
<tr>
<td>Value Loss Coefficient</td>
<td>0.5</td>
</tr>
<tr>
<td>Clipped Value Loss</td>
<td>No</td>
</tr>
<tr>
<td>Normalized Advantage</td>
<td>No</td>
</tr>
<tr>
<td>GAE Parameter [131] ($\lambda$)</td>
<td>0.95</td>
</tr>
<tr>
<td>Discount Factor ($\gamma$)</td>
<td>0.99</td>
</tr>
<tr>
<td>VER Important Sampling</td>
<td>Yes</td>
</tr>
<tr>
<td>Max VER IS</td>
<td>1.0</td>
</tr>
<tr>
<td>Number of Environments ($N$) – per GPU</td>
<td>16</td>
</tr>
<tr>
<td>Experience per rollout ($T \times N$) – per GPU</td>
<td>$128 \times 16$</td>
</tr>
<tr>
<td>Number of GPUs</td>
<td>8</td>
</tr>
</tbody>
</table>

Table C.2: **HTS-RL comparison.** Mean system throughput (SPS) over 1 million training steps. HTS-RL (Provided) does not support training recurrent policies. Hardware: Nvidia 2080 Ti with 16 CPUs.

<table>
<thead>
<tr>
<th>RNN</th>
<th>HTS-RL (Provided)</th>
<th>HTS-RL (Ours)</th>
<th>NoVER</th>
<th>VER</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>242</td>
<td>506</td>
<td>501</td>
<td>620</td>
</tr>
<tr>
<td>✓</td>
<td>-</td>
<td>450</td>
<td>462</td>
<td>590</td>
</tr>
</tbody>
</table>
Figure C.1: Sample efficiency and time-to-sample for each system on a varying number of GPUs. Even with the reduction of sample efficiency when increasing the number of GPUs, increasing the the number of GPUs always reduces the time to reach convergence.

and GPU shared memory to send experience from inference workers to learn and weights from learner to inference workers (vs. CPU shared memory).

We also find that given the efficiency of our implementation, there is no significant change in system SPS if we remove overlapped experience collection and learning (Table C.2, HTS-RL (Ours) vs. NoVER) for Habitat-style tasks. Removing this reduces GPU memory usage (∼2 GB in our experiments, this will be larger for larger networks). We note that overlapped experience collection and learning does have uses, i.e. for CPU simulation or policies with significant CPU components, but it isn’t necessary when both the policy and simulator make heavy use of the GPU.

C.6 Datasets

ReplicaCAD [23] – Creative Commons Attribution 4.0 International (CC BY 4.0) license. This dataset was artist constructed.

YCB dataset [150] – Creative Commons Attribution 4.0 International (CC BY 4.0).
This dataset contains 3D scans of generic objects. There is no PII.

Matterport 3D [67] – http://kaldir.vc.in.tum.de/matterport/MP_TOS.pdf. Consent was given by the owners of the space.

Habitat Matterport Research Dataset [31] – https://matterport.com/matterport-end-user-license-agreement. Consent was given by the owners of the space and any PII was blurred.
REFERENCES


