Learning to Map Natural Language Instructions to Physical Quadcopter Control Using Simulated Flight

Valts Blukis[1] **Yannick Terme**[2] **Eyvind Niklasson**[3] **Ross A. Knepper**[4] **Yoav Artzi**[5]

[1,4,5]Department of Computer Science, Cornell University, Ithaca, New York, USA
[1,2,3,5]Cornell Tech, Cornell University, New York, New York, USA
{[1]valts, [4]rak, [5]yoav}@cs.cornell.edu [2]yannickterme@gmail.com
[3]een7@cornell.edu

Abstract: We propose a joint simulation and real-world learning framework for mapping navigation instructions and raw first-person observations to continuous control. Our model estimates the need for environment exploration, predicts the likelihood of visiting environment positions during execution, and controls the agent to both explore and visit high-likelihood positions. We introduce Supervised Reinforcement Asynchronous Learning (SuReAL). Learning uses both simulation and real environments without requiring autonomous flight in the physical environment during training, and combines supervised learning for predicting positions to visit and reinforcement learning for continuous control. We evaluate our approach on a natural language instruction-following task with a physical quadcopter, and demonstrate effective execution and exploration behavior.

Keywords: Natural language understanding; quadcopter; uav; reinforcement learning; instruction following; observability; simulation; exploration;

1 Introduction

Controlling robotic agents to execute natural language instructions requires addressing perception, language, planning, and control challenges. The majority of methods addressing this problem follow such a decomposition, where separate components are developed independently and are then combined together [e.g., 1, 2, 3, 4, 5, 6]. This requires a hard-to-scale engineering intensive process of designing and working with intermediate representations, including a formal language to represent natural language meaning. Recent work instead learns intermediate representations, and uses a single model to address all reasoning challenges [e.g., 7, 8, 9, 10]. So far, this line of work has mostly focused on pre-specified low-level tasks. In contrast, executing natural language instructions requires understanding sentence structure, grounding words and phrases to observations, reasoning about previously unseen tasks, and handling ambiguity and uncertainty.

In this paper, we address the problem of mapping natural language navigation instructions to continuous control of a quadcopter drone using representation learning. We present a neural network model to jointly reason about observations, natural language, and robot control, with explicit modeling of the agent's plan and exploration of the environment. For learning, we introduce Supervised and Reinforcement Asynchronous Learning (SuReAL), a method for joint training in simulated and physical environments. Figure 1 illustrates our task and model.

We design our model to reason about partial observability and incomplete knowledge of the environment in instruction following. We explicitly model observed and unobserved areas, and the agent's belief that the goal location implied by the instruction has been observed. During learning, we use an intrinsic reward to encourage behaviors that increase this belief, and penalize for indicating task completion while still believing the goal is unobserved.

SuReAL addresses two key learning challenges. First, flying in a physical environment at the scale needed for our complex learning task is both time-consuming and costly. We mitigate this problem using a simulated environment. However, in contrast to the common approach of domain transfer from simulated to physical environments [11, 12], we simultaneously train in both, while not requiring autonomous flight in the physical environment during training. Second, as each example requires a human instruction, it is prohibitively expensive to collect language data at the scale re-

3rd Conference on Robot Learning (CoRL 2019), Osaka, Japan.

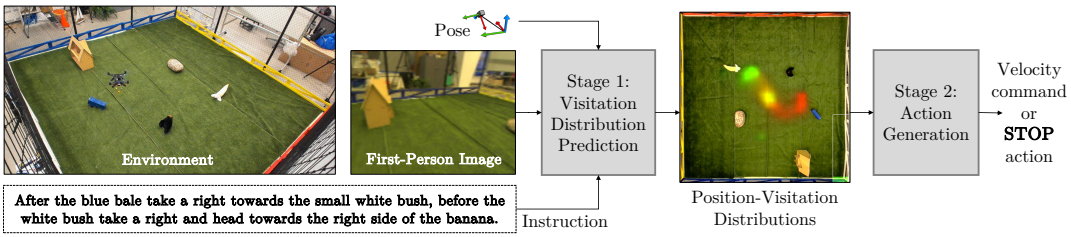


Figure 1: An illustration of our task and model. Correct execution of the instruction requires recognizing objects (e.g., *blue bale*), inferring a path (e.g., to turn to the right *after the blue bale*), and generating the commands to steer the quadcopter and stop at the goal location. The model input at time t is the instruction u, a first-person RGB observation I_t, and a pose estimate P_t. The model has two stages: predicting the probability of visiting positions during execution and generating actions.

Figure 2: Predicted visitation distributions as the instruction execution progresses (left-to-right), trajectory in red and goal in green, for the example from Figure 1. The green bar shows the agent's belief the goal has not been observed yet. A video of the execution and intermediate representations is available at https://youtu.be/PLdsNPE4Gz4.

quired for representation learning [13, 14]. This is unlike tasks where data can be collected without human interaction. We combine supervised and reinforcement learning (RL); the first to best use the limited language data, and the second to effectively leverage experience.

We evaluate our approach with a navigation task, where a quadcopter drone flies between landmarks following natural language instructions. We modify an existing natural language dataset [15] to create a new benchmark with long instructions, complex trajectories, and observability challenges. We evaluate using both automated metrics and human judgements of semantic correctness. To the best of our knowledge, this is the first demonstration of a physical quadcopter system that follows natural language instructions by mapping raw first-person images and pose estimates to continuous control. Our code, data, and demo videos are available at https://github.com/lil-lab/drif.

2 Technical Overview

Task Our goal is to map natural language navigation instructions to continuous control of a quadcopter drone. The agent behavior is determined by a velocity controller setpoint $\rho = (v, \omega)$, where $v \in \mathbb{R}$ is a forward velocity and $\omega \in \mathbb{R}$ is a yaw rate. The model generates actions at fixed intervals. An action is either the task completion action `STOP` or a setpoint update $(v, \omega) \in \mathbb{R}^2$. Given a setpoint update $a_t = (v_t, \omega_t)$ at time t, we fix the controller setpoint $\rho = (v_t, \omega_t)$ that is maintained between actions. Given a start state s_1 and an instruction u, an execution Ξ of length T is a sequence $\langle (s_1, a_1), \ldots, (s_T, a_T) \rangle$, where s_t is the state at time t, $a_{t<T} \in \mathbb{R}^2$ are setpoint updates, and $a_T = \texttt{STOP}$. The state includes the quadcopter pose, internal state, and all landmark locations.

Model We assume access to raw first-person monocular observations and pose estimates. The agent does not have access to the world state. At time t, the agent observes the *agent context* $c_t = (u, I_1, \cdots, I_t, P_1, \cdots P_t)$, where u is the instruction and I_i and P_i are monocular first-person RGB images and 6-DOF agent poses observed at time i. We base our model on the Position Visitation Network [PVN; 16] architecture, and introduce mechanisms for reasoning about observability and exploration and learning across simulated and real environments. The model operates in two stages: casting planning as predicting distributions over world positions indicating the probability of visiting a position during execution, and generating actions to visit high probability positions.

Learning We train jointly in simulated and physical environments. We assume access to a simulator and demonstration sets in both environments, $\mathcal{D}^{\mathtt{R}}$ in the physical environment and $\mathcal{D}^{\mathtt{S}}$ in the simulation. We do not interact with the physical environment during training. Each dataset includes N^D examples $\{(u^{(i)}, \Xi^{(i)})\}_{i=1}^{N^D}$, where $D \in \{\mathtt{R}, \mathtt{S}\}$, $u^{(i)}$ is an instruction, and $\Xi^{(i)}$ is a demonstration execution. We do not require the datasets to be aligned or provide demonstrations for the same set of instructions. We propose SUREAL, a learning approach that concurrently trains the two model

stages in two separate processes. The planning stage is trained with supervised learning, while the action generation stage is trained with RL. The two processes exchange data and parameters. The trajectories collected during RL are added to the dataset used for supervised learning, and the planning stage parameters are periodically transferred to the RL process training the action generation stage. This allows the action generator to learn to execute the plans predicted by the planning stage, which itself is trained using on-policy observations collected from the action generator.

Evaluation We evaluate on a test set of M examples $\{(u^{(i)}, s_1^{(i)}, \Xi^{(i)})\}_{i=1}^{M}$, where $u^{(i)}$ is an instruction, $s_1^{(i)}$ is a start state, and $\Xi^{(i)}$ is a human demonstration. We use human evaluation to verify the generated trajectories are semantically correct with regard to the instruction. We also use automated metrics. We consider the task successful if the agent stops within a predefined Euclidean distance of the final position in $\Xi^{(i)}$. We evaluate the quality of generating the trajectory following the instruction using earth mover's distance between $\Xi^{(i)}$ and executed trajectories.

3 Related Work

Natural language instruction following has been extensively studied using hand-engineered symbolic intermediate representations of world state or instruction semantics with physical robots [1, 2, 3, 4, 17, 5, 18, 6, 19] and simulated agents [20, 21, 22, 23, 24, 25]. In contrast, we study trading off the symbolic representation design with representation learning from demonstrations.

Representation learning has been studied for executing specific tasks such as grasping [7, 8, 10], dexterous manipulation [26, 27], or continuous flight [9]. Our aim is to execute navigation tasks specified in natural language, including new tasks at test time. This problem was addressed with representation learning in discrete simulated environments [28, 29, 30, 15, 31, 32], and more recently with continuous simulations [16]. However, these methods were not demonstrated on physical robots. A host of problems combine to make this challenging, including grounding natural language to constantly changing observations, robustly bridging the gap between relatively high-level instructions to continuous control, and learning with limited language data and the high costs of robot usage.

Our model is based on the Position Visitation Network [16] architecture that incorporates geometric computation to represent language and observations in learned spatial maps. This approach is related to neural network models that construct maps [33, 34, 35, 36] or perform planning [37].

Our approach is aimed at a partial observability scenario and does not assume access to the complete system state. Understanding the instruction often requires identifying mentioned entities that are not initially visible. This requires exploration during task execution. Nyga et al. [38] studied modeling incomplete information in instructions with a modular approach. In contrast, we jointly learn to infer the absence of necessary information and to remedy it via exploration.

4 Model

We model the policy π with a neural network. At time t, given the agent context c_t, the policy outputs a stopping probability $p_t^{\texttt{STOP}}$, a forward velocity v_t, and an angular velocity ω_t. We decompose the architecture to two stages $\pi(c_t) = g(f(c_t))$, where f predicts the probability of visiting positions in the environment and g generates the actions to visit high probability positions. The position visitation probabilities are continuously updated during execution to incorporate the most recent observations, and past actions directly affect the information available for future decisions. Our model is based on the PVN architecture [16]. We introduce several improvements, including explicit modeling of observability in both stages. Appendix B contains further model implementation details, including a detailed list of our improvements. Figure 3 illustrates our model for an example input.

Stage 1: Visitation Distribution Prediction At time t, the first stage $f(\cdot)$ generates two probability distributions: a trajectory-visitation distribution d_t^p and a goal-visitation distribution d_t^g. Both distributions assign probabilities to positions $\mathcal{P}^{\texttt{obs}} \cup \{p^{\texttt{oob}}\}$, where $\mathcal{P}^{\texttt{obs}}$ is the set of positions observed up to time t and $p^{\texttt{oob}}$ represents all yet-unobserved positions. The set $\mathcal{P}^{\texttt{obs}}$ is a discretized approximation of the continuous environment. This approximation enables efficient computation of the visitation distributions [16]. The trajectory-visitation distribution d^p assigns high probability to positions the agent should go through during execution, and the goal-visitation distribution d^g puts high probability on positions where the agent should `STOP` to complete its execution. We add

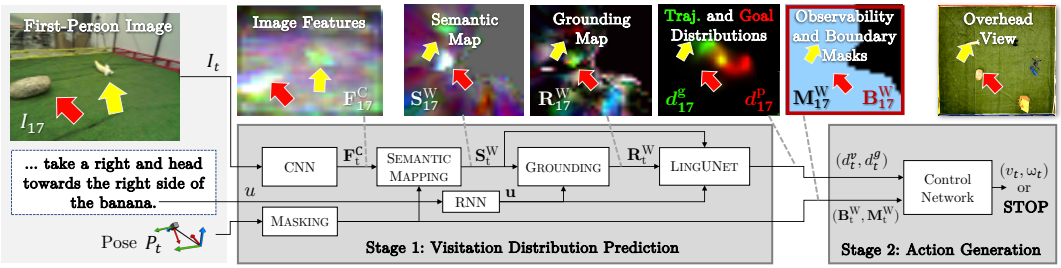


Figure 3: Model architecture illustration. The first stage generates a semantic map $\mathbf{S}^W$, a grounding map $\mathbf{R}^W$, observability masks $\mathbf{M}^W_t$ and $\mathbf{B}^W_t$, and visitation distributions d^p_t and d^g_t. The red and yellow arrows indicate the rock and banana locations. We show all intermediate representations at timestep 17 out of 37, after most of the environment has been observed. Figure 2 shows the visitation distributions for other timesteps, and Figure 8 in the appendix shows all timesteps. An animated version of this figure is available at https://youtu.be/UuZtSl6ckTk.

the special position p^{oob} to PVN to capture probability mass that should otherwise be assigned to positions not yet observed, for example when the goal position has not been observed yet.

The first stage combines a learned neural network and differentiable deterministic computations. The input instruction u is mapped to a vector $\mathbf{u}$ using a recurrent neural network (RNN). The input image I_t is mapped to a feature map $\mathbf{F}^C_t$ that captures spatial and semantic information using a convolutional neural network (CNN). The feature map $\mathbf{F}^C_t$ is processed using a deterministic semantic mapping process [34] using a pinhole camera model and the agent pose P_t to project $\mathbf{F}^C_t$ onto the environment ground at zero elevation. The projected features are deterministically accumulated from previous timesteps to create a semantic map $\mathbf{S}^W_t$. $\mathbf{S}^W_t$ represents each position with a learned feature vector aggregated from all past observations of that position. For example, in Figure 3, the *banana* and the *white bush* can be identified in the raw image features $\mathbf{F}^C_t$ and the projected semantic map $\mathbf{S}^W_t$, where their representations are identical. We generate a language-conditioned grounding map $\mathbf{R}^W_t$ by creating convolutional filters using the text representation $\mathbf{u}$ and filtering $\mathbf{S}^W_t$. The two maps aim to provide different representation: $\mathbf{S}^W_t$ aims for a language-agnostic environment representation and $\mathbf{R}^W_t$ is intended to focus on the objects mentioned in the instruction. We use auxiliary objectives (Appendix C.4) to optimize each map to contain the intended information. We predict the two distributions using LingUNet [15], a language-conditioned variant of the UNet image reconstruction architecture [39], which takes as input the learned maps, $\mathbf{S}^W_t$ and $\mathbf{R}^W_t$, and the instruction representation $\mathbf{u}$. Appendix B.1 provides a detailed description of this architecture.

We add two outputs to the original PVN design: an observability mask $\mathbf{M}^W_t$ and a boundary mask $\mathbf{B}^W_t$. Both are computed deterministically given the agent pose estimate and the camera parameters, and are intended to aid exploration of the environment during instruction execution. $\mathbf{M}^W_t$ assigns 1 to each position $p \in \mathcal{P}$ in the environment if p has been observed by the agent's first-person camera by time t, or 0 otherwise. $\mathbf{B}^W_t$ assigns 1 to environment boundaries and 0 to other positions. Together, the masks provide information about what parts of the environment remain to be explored.

Stage 2: Action Generation The second stage $g(\cdot)$ is a *control network* that receives four inputs: a trajectory-visitation distribution d^p_t, a goal-visitation visitation distribution d^g_t, an observability mask $\mathbf{M}^W_t$, and a boundary mask $\mathbf{B}^W_t$. The four inputs are rotated to the current egocentric agent reference frame, and used to generate the output velocities using a learned neural network. Appendix B.2 describes the network architecture. The velocities are generated to visit high probability positions according to d^p, and the STOP probability is predicted to stop in a likely position according to d^g. In the figure, d^p_t shows the curved flight path, and d^g_t identifies the goal *right of the banana*. Our addition of the two masks and the special position p^{oob} enables generating actions to explore the environment to reduce $d^p(p^{\text{oob}})$ and $d^g(p^{\text{oob}})$. Figure 2 visualizes $d^g(p^{\text{oob}})$ with a green bar, showing how it starts high, and decreases once the goal is observed at step $t = 15$.

5 Learning

We learn two sets of parameters: θ for the first stage $f(\cdot)$ and ϕ for the second stage $g(\cdot)$. We use a simulation for all autonomous flight during learning, and jointly train for both the simulation and the physical environment. This includes training a simulation-specific first stage $f_{\text{S}}(\cdot)$ with additional parameters θ_{S}. The second stage model $g(\cdot)$ is used in both environments. We assume access to

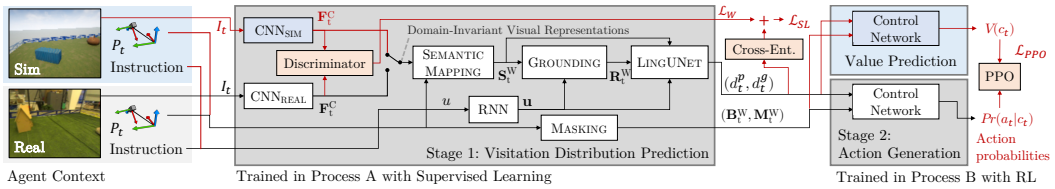


Figure 4: Learning architecture. Stages 1 and 2 of our model are concurrently trained in processes A and B. The blue and orange blocks, and red arrows, represent modules, computation, and loss functions during training only. The white blocks form the final learned policy. We learn from inputs from simulation and real world environments, by switching between the two CNN modules.

sets of training examples $\mathcal{D}^{\text{S}} = \{(u^{(i)}, \Xi^{(i)})\}_{i=1}^{N^{\text{S}}}$ for the simulation and $\mathcal{D}^{\text{R}} = \{(u^{(i)}, \Xi^{(i)})\}_{i=1}^{N^{\text{R}}}$ for the physical environment, where $u^{(i)}$ are instructions and $\Xi^{(i)}$ are demonstration executions. The training examples are not spatially or temporally aligned between domains.

Our learning algorithm, Supervised and Reinforcement Asynchronous Learning (SuReal), uses two concurrent asynchronous processes. Each process only updates the parameters of one stage. Process A uses supervised learning to estimate Stage 1 parameters for both environments: θ for the physical environment model $f(\cdot)$ and θ_{S} for $f_{\text{S}}(\cdot)$. We use both $\mathcal{D}^{\text{R}}$ and $\mathcal{D}^{\text{S}}$ to update the model parameters. We use RL in Process B to learn the parameters ϕ of the second stage $g(\cdot)$ using an intrinsic reward function. We start learning using the provided demonstrations in $\mathcal{D}^{\text{S}}$ and periodically replace execution trajectories with RL rollouts from Process B, keeping a single execution per instruction at any time. We warm-start by running Process A for K^{B}_{iter} iterations before launching Process B to make sure that Process B always receives as input sensible visitation predictions instead of noise. The model parameters are periodically synchronized by copying the simulation parameters of Stage 1 from Process A to B. For learning, we use a learning architecture (Figure 4) that extends our model to process simulation observations and adds a discriminator that encourages learning representations that are invariant to the type of visual input.

Process A: Supervised Learning for Visitation Prediction We train $f(\cdot)$ and $f_{\text{S}}(\cdot)$ to: (a) minimize the KL-divergence between the predicted visitation distributions and reference distributions generated from the demonstrations, and (b) learn domain invariant visual representations that allow sharing of instruction grounding and execution between the two environments. We use demonstration executions in the real environment $\mathcal{D}^{\text{R}}$ and in the simulated environment $\mathcal{D}^{\text{S}}$. The loss for executions from the physical environment Ξ^{R} and the simulation Ξ^{S} is:

$$\mathcal{L}_{\text{SL}}(\Xi^{\text{R}}, \Xi^{\text{S}}) = \frac{1}{|\Xi^{\text{R}}|} \sum_{c \in \mathcal{C}(\Xi^{\text{R}})} D_{\text{KL}}(f(c) \| f^*(c)) + \frac{1}{|\Xi^{\text{S}}|} \sum_{c \in \mathcal{C}(\Xi^{\text{S}})} D_{\text{KL}}(f_{\text{S}}(c) \| f^*(c)) + \mathcal{L}_W(\Xi^{\text{R}}, \Xi^{\text{S}}) \quad , \quad (1)$$

where $\mathcal{C}(\Xi)$ is the sequence of contexts observed by the agent during an execution Ξ and $f^*(c)$ creates the gold-standard visitation distribution examples (i.e., Stage 1 outputs) for a context c from the training data. The term $\mathcal{L}_W(\Xi^{\text{R}}, \Xi^{\text{S}})$ aims to make the feature representation $\mathbf{F}^C$ indistinguishable between real and simulated images. This allows the rest of the model to use either simulated or real observations interchangeably. $\mathcal{L}_W(\Xi^{\text{R}}, \Xi^{\text{S}})$ is the approximated empirical Wasserstein distance between the visual feature distributions extracted from simulated and real agent contexts:

$$\mathcal{L}_W(\Xi^{\text{R}}, \Xi^{\text{S}}) = \frac{1}{|\Xi^{\text{S}}|} \sum_{c_t \in \Xi^{\text{S}}} h(\text{CNN}^{\text{S}}(I_t)) - \frac{1}{|\Xi^{\text{R}}|} \sum_{c_t \in \Xi^{\text{R}}} h(\text{CNN}(I_t)) \quad ,$$

where h is a Lipschitz continuous neural network discriminator with parameters ψ that we train to output high values for simulated features and low values for real features [40, 41]. I_t is the t-th image in the agent context c_t. The discriminator architecture is described in Appendix C.1.

Algorithm 1 shows the supervised optimization procedure. We alternate between updating the discriminator parameters ψ, and the first stage model parameters θ and θ_{S}. At every iteration, we perform $K^{\text{SL}}_{\text{discr}}$ gradient updates of ψ to maximize the Wasserstein loss $\mathcal{L}_W$ (lines 3–7), and then perform a single gradient update to θ and θ_{S} to minimize supervised learning loss $\mathcal{L}_{\text{SL}}$ (lines 8–10). We send the simulation-specific parameters θ_{S} to the RL process every $K^{\text{SL}}_{\text{iter}}$ iterations (line 12).

Process B: Reinforcement Learning for Action Generation We train the action generator $g(\cdot)$ using RL with an intrinsic reward. We use Proximal Policy Optimization [PPO; 42] to maximize the expected return. The learner has no access to an external task reward, but instead computes a reward $r(\cdot)$ from how well the agent follows the visitation distributions generated by the first stage:

$$r(c_t, a_t) = \lambda_v r_v(c_t, a_t) + \lambda_s r_s(c_t, a_t) + \lambda_e r_e(c_t, a_t) - \lambda_a r_a(a_t) - \lambda_{step} \quad ,$$

Algorithm 1 Process A: Supervised Learning

Input: First stage models f and f_{S} with parameters θ and θ_{S}, discriminator h with parameters ψ, datasets of simulated and physical environment trajectories $\mathcal{D}^{\mathrm{S}}$ and $\mathcal{D}^{\mathrm{R}}$.

Definitions: $\mathcal{D}^{\mathrm{S}}$ and f_{S}^{B} are shared with Process B.

1: **while** Process B has not finished **do**
2: **for** $i = 1, \ldots, K^{\mathrm{SL}}_{\mathrm{iter}}$ **do**
3: **for** $j = 1, \ldots, K^{\mathrm{SL}}_{\mathrm{discr}}$ **do**
4: » Sample trajectories
5: $\Xi^{\mathrm{R}} \sim \mathcal{D}^{\mathrm{R}}$ and $\Xi^{\mathrm{S}} \sim \mathcal{D}^{\mathrm{S}}$
6: » Update discriminator to maximize Wasserstein distance
7: $\psi \leftarrow \mathrm{ADAM}(\nabla_{\psi} - \mathcal{L}_{W}(\Xi^{\mathrm{R}}, \Xi^{\mathrm{S}}))$
8: $\Xi^{\mathrm{R}} \sim \mathcal{D}^{\mathrm{R}}$ and $\Xi^{\mathrm{S}} \sim \mathcal{D}^{\mathrm{S}}$
9: » Update first stage parameters
10: $(\theta_{\mathrm{S}}, \theta) \leftarrow \mathrm{ADAM}(\nabla_{\theta_{\mathrm{S}},\theta} \mathcal{L}_{\mathrm{SL}}(\Xi^{\mathrm{R}}, \Xi^{\mathrm{S}}))$
11: » Send f_{S} to Process B if it is running
12: $f_{\mathrm{S}}^{B} \leftarrow f_{\mathrm{S}}$
13: **if** $i = K^{B}_{\mathrm{iter}}$ **then**
14: Launch Process B (Algorithm 2)
15: **return** f

Algorithm 2 Process B: Reinforcement Learning

Input: Simulation dataset $\mathcal{D}^{\mathrm{S}}$, second-stage model g with parameters ϕ, value function V with parameters υ, first-stage simulation model f_{S}.

Definitions: $\textsc{Merge}(\mathcal{D}, E)$ is a set of sentence-execution pairs including all instructions from $\mathcal{D}$, where each instruction is paired with an execution from E, or $\mathcal{D}$ if not in E. $\mathcal{D}^{\mathrm{S}}$ and f_{S}^{B} are shared with Process A.

1: **for** $e = 1, \ldots, K^{\mathrm{RL}}_{\mathrm{epoch}}$ **do**
2: » Get the most recent update from Process A
3: $f_{\mathrm{S}} \leftarrow f_{\mathrm{S}}^{B}$
4: **for** $i = 1, \ldots, K^{\mathrm{RL}}_{\mathrm{iter}}$ **do**
5: » Sample simulator executions of N instructions
6: $\hat{\Xi}^{(1)}, ..., \hat{\Xi}^{(N)} \sim g(f_{\mathrm{S}}(\cdot))$
7: **for** $j = 1, \ldots, K^{\mathrm{RL}}_{\mathrm{steps}}$ **do**
8: » Sample state-action-return tuples and update
9: $X \sim \hat{\Xi}_1, ..., \hat{\Xi}_N$
10: $\phi, \upsilon \leftarrow \mathrm{ADAM}(\nabla_{\phi,\upsilon} \mathcal{L}_{PPO}(X, V))$
11: » Update executions to share with Process A
12: $\mathcal{D}^{\mathrm{S}} \leftarrow \textsc{Merge}(\mathcal{D}^{\mathrm{S}}, \{\hat{\Xi}_1, \ldots, \hat{\Xi}_N\})$
13: **return** g

where c_t is the agent context at time t and a_t is the action. The reward $r(\cdot)$ is a weighted combination of five terms. The visitation reward $r_v(\cdot)$ is the per-timestep reduction in earth mover's distance between the predicted distribution d_t^p and an empirical distribution that assigns equal probability to every position visited until time t. This smooth and dense reward encourages $g(\cdot)$ to follow the visitation distributions predicted by $f(\cdot)$. The stop reward $r_s(\cdot)$ is only non-zero for `STOP`, when it is the earth mover's distance between the predicted goal distribution d_t^g and an empirical stopping distribution that assigns the full probability mass to the stop position in the policy execution. The exploration reward $r_e(\cdot)$ combines a positive reward for reducing the belief that the goal has not been observed yet (i.e., reducing $d_t^g(p^{\texttt{oob}})$) and a negative reward proportional to the probability that the goal position is unobserved according to $d_t^g(p^{\texttt{oob}})$. The action reward $r_a(\cdot)$ penalizes actions outside of the controller range. Finally, λ_{step} is negative per-step reward to encourage efficient execution. We provide the reward implementation details in Appendix C.3.

Algorithm 2 shows the RL procedure. At every iteration, we collect N simulation executions $\hat{\Xi}^{(1)}, ..., \hat{\Xi}^{(N)}$ using the policy $g(f_{\mathrm{S}}(\cdot))$ (line 6). To sample setpoint updates we treat the existing output as the mean of a normal distribution, add variance prediction, and sample the two velocities from the predicted normal distributions. We perform $K^{\mathrm{RL}}_{\mathrm{steps}}$ PPO updates using the return and value estimates (lines 7–10). For every update, we sample state-action-return triplets from the collected trajectories, compute the PPO loss $\mathcal{L}_{PPO}(X, V)$, update parameters ϕ, and update the parameters υ of the value function V. We pass the sampled executions to Process A to allow the model to learn to predict the visitation distributions in a way that is robust to the agent actions (line 12).

6 Experimental Setup

We provide the complete implementation and experimental setup details in Appendix E.

Environment and Data We use an Intel Aero quadcopter with a PX4 flight controller, and a Vicon motion capture system for pose estimates. For simulation, we use the quadcopter simulator from Blukis et al. [34] that is based on Microsoft AirSim [43]. The environment size is 4.7x4.7m. We randomly create environments with 5–8 landmarks, selected randomly out of a set of 15. Figure 1 shows the real environment. We follow the crowdsourcing setup of Misra et al. [15] to collect 997 paragraphs with 4,557 segments. We use 3,245/640/672 for training, development, and testing. We expand this data by concatenating consecutive segments to create more challenging instructions, including with exploration challenges [32]. Figure 1 shows an instruction made of two consecutive segments. The simulator dataset $\mathcal{D}^{\mathrm{S}}$ includes oracle demonstrations of all instructions, while the real-world dataset $\mathcal{D}^{\mathrm{R}}$ includes only 402 single-segment demonstrations. For evaluation in the physical environment, we sample 20 test paragraphs consisting of 93 single-segment and 73 two-

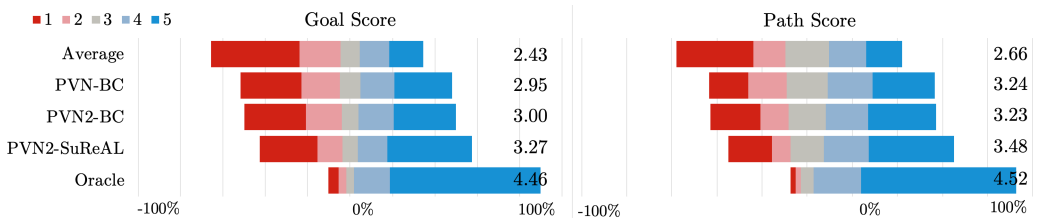


Figure 5: Human evaluation results on the physical quadcopter on two-segment data. We plot the Likert scores using Gantt charts of score frequencies. The black numbers indicate average scores.

segment instructions. We use both single and concatenated instructions for training, and test on each set separately. We also use the original Misra et al. [15] data as additional simulation training data. Appendix D provides data statistics and further details.

Evaluation We use human judgements to evaluate if the agent's final position is correct with regard to the instruction (goal score) and how well the agent followed the path described by the instruction (path score). We present MTurk workers with an instruction and a top-down animation of the agent behavior, and ask for a 5-point Likert-scale score for the final position and trajectory correctness. We obtain five judgements per example per system. We also automatically measure (a) SR: success rate of stopping within 47cm of the correct position; and (b) EMD: earth mover's distance in meters between the agent and demonstration trajectories. Appendix F provides more evaluation details.

Systems We compare our approach, PVN2-SUREAL, with three non-learning and two learning baselines: (1) STOP: output the `STOP` action without movement; (2) AVERAGE: take the average action for the average number of steps; (3) ORACLE: a hand-crafted upper-bound expert policy that has access to the ground truth human demonstration; (4) PVN-BC the Blukis et al. [16] PVN model trained with behavior cloning; and (5) PVN2-BC: our model trained with behavior cloning. The two behavior cloning systems require access to an oracle that provides velocity command output labels, in addition to demonstration data. SUREAL uses demonstrations, but does not require the oracle during learning. None of the learned systems use any oracle data at test time. All learned systems use the same training data $\mathcal{D}^{\text{S}}$ and $\mathcal{D}^{\text{R}}$, and include the domain-adaptation loss (Equation 1).

7 Results

Figure 5 shows human evaluation Likert scores. Our model receives five-point scores 39.72% of the time for getting the goal right, and 37.8% of the time for the path. This is a 34.9% and 24.8% relative improvement compared to PVN2-BC, the next best system. This demonstrates the benefits of modeling observability, using SUREAL for training-time exploration, and using a reward function that trades-off task performance and test-time exploration. The AVERAGE baseline received only 15.8% 5-point ratings in both *path score* and *goal score*, demonstrating the task difficulty.

We study how well our model addresses observability and exploration challenges. Figure 6 shows human path score judgements split to tasks where the goal is visible from the agent's first-person view at start position (34 examples) and those where it is not and exploration is required (38 examples). Our approach outperforms the baselines in cases requiring exploration, but it is slightly outperformed by PVN2-BC in simple examples. This could be explained by our agent attempting to explore the environment in cases where it is not necessary.

Table 1 shows the automated metrics for both environments. We observe that the success rate (SR) measure is sensitive to the threshold selection, and correct executions are often considered as wrong; PVN2-SUREAL gets 30.6% SR compared to a perfect human score 39.72% of the time. This highlights the need for human evaluation, and must be considered when interpreting the SR results. We generally find EMD more reliable, although it also does not account for semantic correctness.

Comparing to PVN2-BC, our approach performs better on the real environment demonstrating the benefit of SUREAL. In simulation, we observe better EMD, but worse SR. Qualitatively, we observe our approach often recovers the correct overall trajectory, with a slightly imprecise stopping location due to instruction ambiguity or control challenges. Such partial correctness is not captured by SR. Comparing PVN2-BC and PVN-BC, we see the benefit of modeling observability. SUREAL further improves upon PVN2-BC, by learning to explore unobserved locations at test-time.

Comparing our approach between simulated and real environments, we see an absolute performance degradation of 2.7% SR and 0.1 EMD from simulation to the real environment. This highlights the

	Method	1-segment		2-segment	
		SR	EMD	SR	EMD
Test Results					
Real	AVERAGE	37.0	0.42	16.7	0.71
	PVN-BC	48.9	0.42	20.8	0.61
	PVN2-BC	52.2	0.37	29.2	0.59
	PVN2-SUREAL	**56.5**	**0.34**	**30.6**	**0.52**
	ORACLE	100.0	0.17	91.7	0.23
Sim	AVERAGE	29.5	0.53	8.7	0.80
	PVN-BC	**64.1**	0.31	**37.5**	0.59
	PVN2-BC	55.4	0.34	34.7	0.58
	PVN2-SUREAL	53.3	**0.30**	33.3	**0.42**
	ORACLE	100.0	0.13	98.6	0.17
Development Results					
Real	PVN2-SUREAL	54.8	0.32	31.0	0.50
	PVN2-SUREAL-NOU	53.8	0.30	14.3	0.56
	PVN2-SUREAL$_{50real}$	**60.6**	**0.29**	**34.5**	**0.44**
	PVN2-SUREAL$_{10real}$	46.2	0.33	17.9	0.56
Sim	PVN2-SUREAL	48.1	0.29	**39.3**	**0.40**
	PVN2-SUREAL-NOU	53.8	**0.28**	27.4	0.50
	PVN2-SUREAL-NOI	**56.2**	**0.28**	25.9	0.45

Table 1: Automated evaluation test and development results. SR: *success rate* (%) and EMD: *earth-mover's distance* in meters between agent and demonstration trajectories.

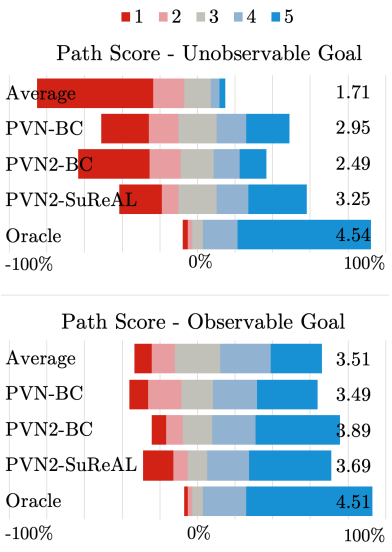


Figure 6: Human evaluation path score frequencies (Figure 5) decomposed by initially unobservable (top) or observable (bottom) goal location.

remaining challenges of visual domain transfer and complex flight dynamics. The flight dynamics challenges are also visible in the ORACLE performance degradation between the two environments.

We study several ablations. First, we quantify the effect of using a smaller number of real-world training demonstrations. We randomly select subsets of demonstrations, with the constraint that all objects are visually represented. We find that using only half (200) of the physical demonstrations does not appear to reduce performance (PVN2-SUREAL$_{50real}$), while using only 10% (40), drastically hurts real-world performance (PVN2-SUREAL$_{10real}$). This shows that the learning method is successfully leveraging real-world data to improve performance, while requiring relatively modest amounts of data. We also study performance without access to the instruction (PVN2-SUREAL-NOU), and with using a blank input image (PVN2-SUREAL-NOI). The relatively high SR of these ablations on 1-segment instructions highlights the inherent bias in simple trajectories. The 2-segment data, which is our main focus, is much more robust to such biases. Appendix G provides more automatic evaluation results, including additional ablations and results on the original data of Misra et al. [15].

8 Discussion

We study the problem of mapping natural language instructions to continuous control of a physical quadcopter drone. Our two-stage model decomposition allows some level of re-use and modularity. For example, a trained Stage 1 can be re-used with different robot types. This decomposition and the interpretability it enables also create limitations, including limited sensory input for deciding about control actions given the visitation distributions. These are both important topics for future study.

Our learning method, SUREAL, uses both annotated demonstration trajectories and a reward function. In this work, we assume demonstration trajectories were generated with an expert policy. However, SUREAL does not necessarily require the initial demonstrations to come from a reasonable policy, as long as we have access to the gold visitation distributions, which are easier to get compared to oracle actions. For example, given an initial policy that immediately stops instead of demonstrations, we will train Stage 1 to predict the given visitation distributions and Stage 2 using the intrinsic reward. Studying this empirically is an important direction for future work.

Finally, our environment provides a strong testbed for our system-building effort and the transition from the simulation to the real world. However, various problems are not well represented, such as reasoning about obstacles, raising important directions for future work. While we do not require the simulation to accurately reflect the real world, studying scenarios with stronger difference between the two is another important future direction. Our work also points towards the need for better automatic evaluation for instruction following, or, alternatively, wide adoption of human evaluation.

Acknowledgments

This research was supported by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, a Google Faculty Award, NSF CAREER-1750499, AFOSR FA9550-17-1-0109, an Amazon Research Award, and cloud computing credits from Amazon. We thank Dipendra Misra, Alane Suhr, and the anonymous reviewers for their helpful comments.

References

[1] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. Gopal Banerjee, S. Teller, and N. Roy. Approaching the Symbol Grounding Problem with Probabilistic Graphical Models. *AI Magazine*, 2011.

[2] C. Matuszek, N. FitzGerald, L. Zettlemoyer, L. Bo, and D. Fox. A Joint Model of Language and Perception for Grounded Attribute Learning. In *ICML*, 2012.

[3] F. Duvallet, T. Kollar, and A. Stentz. Imitation learning for natural language direction following through unknown environments. In *ICRA*, 2013.

[4] M. R. Walter, S. Hemachandra, B. Homberg, S. Tellex, and S. Teller. Learning Semantic Maps from Natural Language Descriptions. In *RSS*, 2013.

[5] S. Hemachandra, F. Duvallet, T. M. Howard, N. Roy, A. Stentz, and M. R. Walter. Learning models for following natural language directions in unknown environments. In *ICRA*, 2015.

[6] N. Gopalan, D. Arumugam, L. L. Wong, and S. Tellex. Sequence-to-sequence language grounding of non-markovian task specifications. In *RSS*, 2018.

[7] I. Lenz, H. Lee, and A. Saxena. Deep learning for detecting robotic grasps. *IJRR*, 2015.

[8] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen. Learning hand-eye coordination for robotic grasping with large-scale data collection. In *ISER*, 2016.

[9] F. Sadeghi and S. Levine. Cad2rl: Real single-image flight without a single real image. In *RSS*, 2017.

[10] D. Quillen, E. Jang, O. Nachum, C. Finn, J. Ibarz, and S. Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. *ICRA*, 2018.

[11] A. A. Rusu, M. Večerík, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell. Sim-to-real robot learning from pixels with progressive nets. In *CoRL*, 2017.

[12] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. *ICRA*, 2018.

[13] K. M. Hermann, F. Hill, S. Green, F. Wang, R. Faulkner, H. Soyer, D. Szepesvari, W. Czarnecki, M. Jaderberg, D. Teplyashin, et al. Grounded language learning in a simulated 3d world. *arXiv preprint arXiv:1706.06551*, 2017.

[14] D. S. Chaplot, K. M. Sathyendra, R. K. Pasumarthi, D. Rajagopal, and R. Salakhutdinov. Gated-attention architectures for task-oriented language grounding. *AAAI*, 2018.

[15] D. Misra, A. Bennett, V. Blukis, E. Niklasson, M. Shatkin, and Y. Artzi. Mapping instructions to actions in 3D environments with visual goal prediction. In *EMNLP*, 2018.

[16] V. Blukis, D. Misra, R. A. Knepper, and Y. Artzi. Mapping navigation instructions to continuous control actions with position-visitation prediction. In *CoRL*, 2018.

[17] D. K. Misra, J. Sung, K. Lee, and A. Saxena. Tell me dave: Context-sensitive grounding of natural language to mobile manipulation instructions. In *RSS*, 2014.

[18] J. Thomason, S. Zhang, R. J. Mooney, and P. Stone. Learning to interpret natural language commands through human-robot dialog. In *International Joint Conferences on Artificial Intelligence*, 2015.

[19] E. C. Williams, N. Gopalan, M. Rhee, and S. Tellex. Learning to parse natural language to grounded reward functions with weak supervision. In *ICRA*, 2018.

[20] M. MacMahon, B. Stankiewicz, and B. Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. In *AAAI*, 2006.

[21] S. R. K. Branavan, L. S. Zettlemoyer, and R. Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *ACL*, 2010.

[22] C. Matuszek, E. Herbst, L. Zettlemoyer, and D. Fox. Learning to parse natural language commands to a robot control system. In *ISER*, 2012.

[23] Y. Artzi and L. Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *TACL*, 2013.

[24] Y. Artzi, D. Das, and S. Petrov. Learning compact lexicons for CCG semantic parsing. In *EMNLP*, 2014.

[25] A. Suhr and Y. Artzi. Situated mapping of sequential instructions to actions with single-step reward observation. In *ACL*, 2018.

[26] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *JMLR*, 2016.

[27] A. Nair, D. Chen, P. Agrawal, P. Isola, P. Abbeel, J. Malik, and S. Levine. Combining self-supervised learning and imitation for vision-based rope manipulation. In *ICRA*, 2017.

[28] D. Misra, J. Langford, and Y. Artzi. Mapping instructions and visual observations to actions with reinforcement learning. In *EMNLP*, 2017.

[29] P. Shah, M. Fiser, A. Faust, J. C. Kew, and D. Hakkani-Tur. Follownet: Robot navigation by following natural language directions with deep reinforcement learning. *arXiv preprint arXiv:1805.06150*, 2018.

[30] P. Anderson, Q. Wu, D. Teney, J. Bruce, M. Johnson, N. Sünderhauf, I. Reid, S. Gould, and A. van den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *CVPR*, 2018.

[31] D. Fried, R. Hu, V. Cirik, A. Rohrbach, J. Andreas, L.-P. Morency, T. Berg-Kirkpatrick, K. Saenko, D. Klein, and T. Darrell. Speaker-follower models for vision-and-language navigation. In *Advances in Neural Information Processing Systems*, 2018.

[32] V. Jain, G. Magalhaes, A. Ku, A. Vaswani, E. Ie, and J. Baldridge. Stay on the path: Instruction fidelity in vision-and-language navigation. In *ACL*, 2019.

[33] S. Gupta, J. Davidson, S. Levine, R. Sukthankar, and J. Malik. Cognitive mapping and planning for visual navigation. In *CVPR*, 2017.

[34] V. Blukis, N. Brukhim, A. Bennet, R. Knepper, and Y. Artzi. Following high-level navigation instructions on a simulated quadcopter with imitation learning. In *RSS*, 2018.

[35] A. Khan, C. Zhang, N. Atanasov, K. Karydis, V. Kumar, and D. D. Lee. Memory augmented control networks. In *ICLR*, 2018.

[36] P. Anderson, A. Shrivastava, D. Parikh, D. Batra, and S. Lee. Chasing ghosts: Instruction following as bayesian state tracking. 2019.

[37] A. Srinivas, A. Jabri, P. Abbeel, S. Levine, and C. Finn. Universal planning networks. *ICML*, 2018.

[38] D. Nyga, S. Roy, R. Paul, D. Park, M. Pomarlan, M. Beetz, and N. Roy. Grounding robot plans from natural language instructions with incomplete world knowledge. In *CoRL*, 2018.

[39] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, 2015.

[40] J. Shen, Y. Qu, W. Zhang, and Y. Yu. Wasserstein distance guided representation learning for domain adaptation. In *AAAI*, 2018.

[41] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In *ICML*, 2017.

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

[43] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[44] A. Suhr, C. Yan, J. Schluger, S. Yu, H. Khader, M. Mouallem, I. Zhang, and Y. Artzi. Executing instructions in situated collaborative interactions. In *EMNLP*, 2019.

Figure 7: Instruction executions from the development set on the physical quadcopter. For each example, the figure shows (from the left) the input instruction, the initial image that the agent observes, the initial visitation distributions overlaid on the top-down view, visitation distributions at the midpoint of the trajectory, and the final visitation distributions when outputting the STOP action. The green bar on the lower-right corner of each distribution plot shows the predicted probability that the goal is not yet observed. The blue arrow indicates the agent pose.

A Execution Examples on Real Quadcopter

Examples of Different Instruction Executions Figure 7 shows a number of instruction-following executions collected on the real drone, showing successes and some typical failures.

Visualization of Intermediate Representations Figure 8 shows the intermediate representations and visitation predictions over time during instruction execution for the examples used in Figures 1-3, illustrating the model reasoning. The model is able to output the STOP action to stop on the right side of the banana, even after the banana has disappeared from the first-person view. This demon-

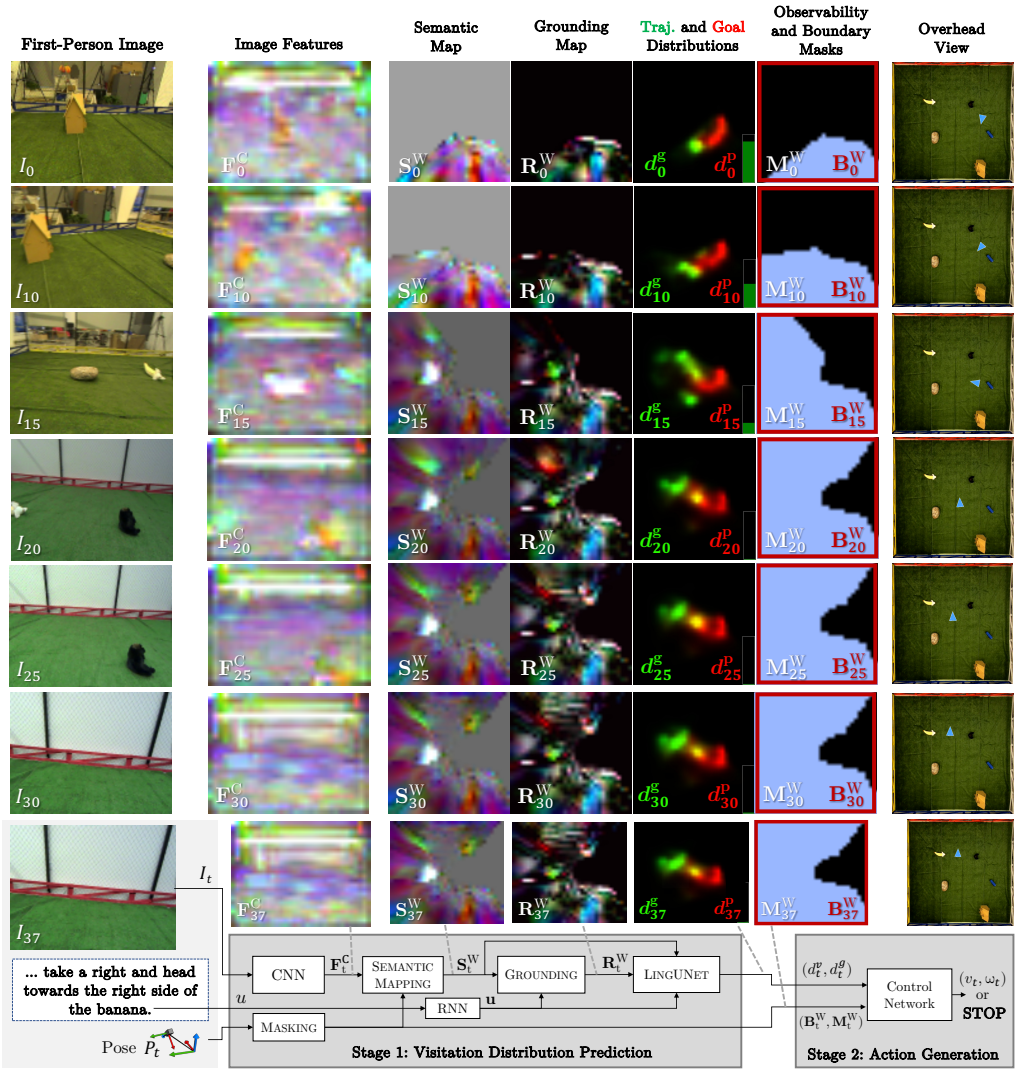


Figure 8: Illustration of changes in the intermediate representations during an instruction execution, showing how information is accumulated in the semantic maps over time, and how that affects the predicted visitation distributions. We show the instruction from Figures 1-3. From top to bottom, representations are shown for timesteps 0, 10, 15, 20, 25, 30, and 37 (the final timestep). From left to right, we show the input image I_t, first-person features $\mathbf{F}_t^C$, semantic map $\mathbf{S}_t^W$, grounding map $\mathbf{R}_t^W$, goal and position visitation distributions d_t^g and d_t^p, observability mask $\mathbf{M}_t^W$ and boundary mask $\mathbf{B}_t^W$, and the overhead view of the environment. The agent position is indicated with a blue arrow in the overhead view. The agent does not have access to the overhead view, which is provided for illustration purposes only.

strates the advantages of using an explicit spatial map aggregated over time instead, for example, a learned hidden state vector representing the agent state.

B Model Details

B.1 Stage 1: Visitation Distribution Prediction

The first-stage model is largely based on the Position Visitation Network, except for several improvements we introduce:

- Computing observability and boundary masks $\mathbf{M}^W$ and $\mathbf{B}^W$ that are used to track unexplored space and environment boundaries.

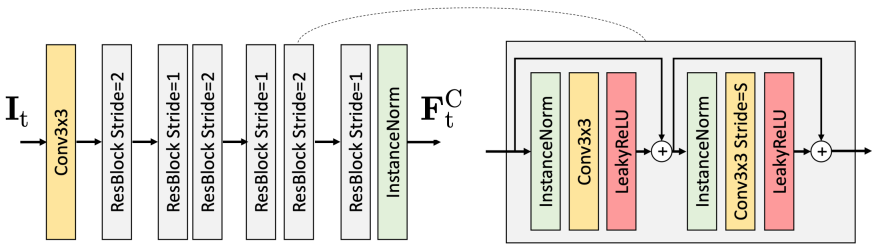


Figure 9: The 13-layer ResNet architecture used in PVN and PVN2 networks (figure adapted from Blukis et al. [34]).

- Introducing a placeholder position p^{oob} that represents all unobserved positions in the environment for use in the visitation distributions.
- Modification to the LINGUNET architecture to support outputting a probability score for the unobserved placeholder position, in addition to the 2D distributions over environment positions.
- Predicting 2D probability distributions only over observed locations in the environment.
- Minor hyperparameter changes to better support the longer instructions.

The description in Section B.1 has been taken from Blukis et al. [16]. We present it here for reference and completeness, with minor modifications to highlight technical differences.

B.1.1 Instruction Representation

We represent the instruction $u = \langle u_1, \cdots u_l \rangle$ as an embedded vector $\mathbf{u}$. We generate a series of hidden states $\mathbf{h}_i = \text{LSTM}(\phi(u_i), \mathbf{h}_{i-1})$, $i = 1 \dots l$, where LSTM is a Long-Short Term Memory recurrent neural network (RNN) and ϕ is a learned word-embedding function. The instruction embedding is the last hidden state $\mathbf{u} = \mathbf{h}_l$. This part is replicated as is from Blukis et al. [16].

B.1.2 Semantic Mapping

We construct the semantic map using the method of Blukis et al. [34]. The full details of the process are specified in the original paper. Roughly speaking, the semantic mapping process includes three steps: feature extraction, projection, and accumulation. At timestep t, we process the currently observed image I_t using a 13-layer residual neural network CNN (Figure 9) to generate a feature map $\mathbf{F}_t^C = \text{CNN}(I_t)$ of size $W_f \times H_f \times C$. We compute a feature map in the world coordinate frame $\mathbf{F}_t^W$ by projecting $\mathbf{F}_t^C$ with a pinhole camera model onto the ground plane at elevation zero.

The semantic map of the environment $\mathbf{S}_t^W$ at time t is an integration of $\mathbf{F}_t^W$ and $\mathbf{S}_{t-1}^W$, the map from the previous timestep. The integration equation is given in Section 4c in Blukis et al. [34]. This process generates a tensor $\mathbf{S}_t^W$ of size $W_w \times H_w \times C$ that represents a map, where each location $[\mathbf{S}_t^W]_{(x,y)}$ is a C-dimensional feature vector computed from all past observations $I_{<t}$, each processed to learned features $\mathbf{F}_{<t}^C$ and projected onto the environment ground in the world frame at coordinates (x, y). This map maintains a learned high-level representation for every world location (x, y) that has been visible in any of the previously observed images. We define the world coordinate frame using the agent starting pose P_1; the agent start position is the coordinates $(0, 0)$, and the positive direction of the x-axis is along the agent heading. This gives consistent meaning to spatial language, such as *turn left* or *pass on the left side of*.

B.1.3 Grounding

We create the *grounding map* $\mathbf{R}_t^W$ with a 1×1 convolution $\mathbf{R}_t^W = \mathbf{S}_t^W \circledast \mathbf{K}_G$. The kernel $\mathbf{K}_G$ is computed using a learned linear transformation $\mathbf{K}_G = \mathbf{W}_G\mathbf{u} + \mathbf{b}_G$, where $\mathbf{u}$ is the instruction embedding. The grounding map $\mathbf{R}_t^W$ has the same height and width as $\mathbf{S}_t^W$, and during training we optimize the parameters so it captures the objects mentioned in the instruction u (Section C.4).

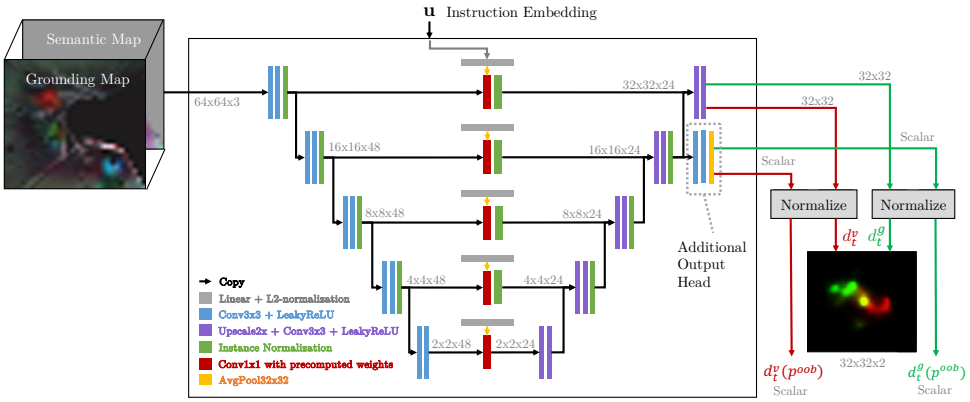


Figure 10: The LINGUNET architecture, showing the additional output head that was added as part of the PVN2 model. LINGUNET outputs raw scores, which we normalize over the domain of each distribution. This figure is adapted from Blukis et al. [16].

B.1.4 LINGUNET and Visitation Distributions

The following paragraphs are adapted from Blukis et al. [16] and formally define the LINGUNET architecture with our modifications. Figure 10 illustrates the architecture.

LINGUNET uses a series of convolution and scaling operations. The input map $\mathbf{F}_0 = [\mathbf{S}_t^W, \mathbf{R}_t^W]$ is processed through L cascaded convolutional layers to generate a sequence of feature maps $\mathbf{F}_k = \text{CNN}_k^D(\mathbf{F}_{k-1})$, $k = 1 \dots L$.[1] Each $\mathbf{F}_k$ is filtered with a 1×1 convolution with weights $\mathbf{K}_k$. The kernels $\mathbf{K}_k$ are computed from the instruction embedding $\mathbf{u}$ using a learned linear transformation $\mathbf{K}_k = \mathbf{W}_k^u \mathbf{u} + \mathbf{b}_k^u$. This generates l language-conditioned feature maps $\mathbf{G}_k = \mathbf{F}_k \circledast \mathbf{K}_k$, $k = 1 \dots L$. A series of L upscale and convolution operations computes L feature maps of increasing size:

$$\mathbf{H}_k = \begin{cases} \text{UPSCALE}(\text{CNN}_k^U([\mathbf{H}_{k+1}, \mathbf{G}_k])), & \text{if } 1 \leq k \leq L-1 \\ \text{UPSCALE}(\text{CNN}_k^U(\mathbf{G}_k)), & \text{if } k = L \end{cases},$$

We modify the original LINGUNET design by adding an output head that outputs a vector $\mathbf{h}$:

$$\mathbf{h} = \text{AVGPOOL}(\text{CNN}^{\mathbf{h}}(\mathbf{H}_2)) \ ,$$

where AVGPOOL takes averages across the dimensions.

The output of LINGUNET is a tuple $(\mathbf{H}_1, \mathbf{h})$, where $\mathbf{H}_1$ is of size $W_w \times H_w \times 2$ and $\mathbf{h}$ is a vector of length 2. This output is used to compute two distributions, and can be increased if more distribution are predicted, such as in Suhr et al. [44]. We use an additional normalization step to produce the position visitation and goal visitation distributions given $(\mathbf{H}_1, \mathbf{h})$.

B.2 Control Network: Action Generation and Value Function

Figure 11 shows the architecture of the control network for the second action generation stage of the model. The value function architecture is identical to the action generator and also uses the control network, except that it has only a single output. The value function does not share the parameters with the action generator.

The control network takes as input the trajectory and stopping visitation distributions d_t^p and d_t^g, as well as the observability and boundary masks $\mathbf{M}_t^W$ and $\mathbf{B}_t^W$. The distributions d_t^p and d_t^g are represented as 2D square-shaped images over environment locations, where unobserved locations have a probability of zero. Additional scalars $d^p(p^{\text{oob}})$ and $d^g(p^{\text{oob}})$ define the probability mass outside of any observed environment location.

The visitation distributions d_t^p and d_t^g are first rotated to the agent's current ego-centric reference frame, concatenated along the channel dimension, and then processed with a convolutional neural network. The output is flattened into a vector. The masks $\mathbf{B}_t^W$ and $\mathbf{M}_t^W$ are processed in an analogous way to the visitation distributions d_t^p and d_t^g, and the output is also flattened into a vector. The scalars $d^p(p^{\text{oob}})$ and $d^g(p^{\text{oob}})$ are embedded into fixed-length vectors:

[1] $[\cdot, \cdot]$ denotes concatenation along the channel dimension.

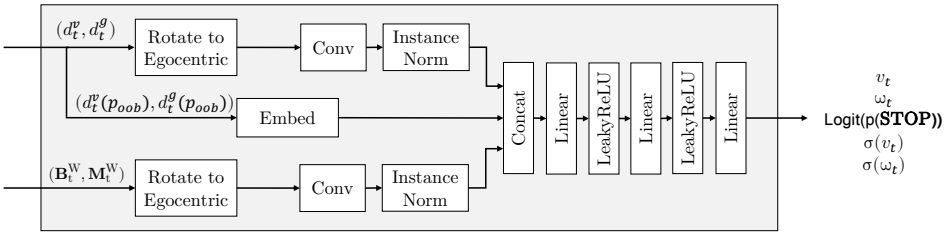


Figure 11: Control network architecture.

$$\begin{aligned} \text{EMBED}_{d^g(p^{\text{oob}})} &= \mathbf{q}_1 \cdot d^g(p^{\text{oob}}) - \mathbf{q}_1 \cdot (1 - d^g(p^{\text{oob}})) \\ \text{EMBED}_{d^p(p^{\text{oob}})} &= \mathbf{q}_2 \cdot d^p(p^{\text{oob}}) - \mathbf{q}_2 \cdot (1 - d^p(p^{\text{oob}})) \ , \end{aligned}$$

where $\mathbf{q}_{(\cdot)}$ are random fixed-length vectors. We do not tune $\mathbf{q}_{(\cdot)}$.

The resulting vector representations for the visitation distributions, out-of-bounds probabilities, and masks are concatenated and processed with a three-layer multi-layer perceptron (MLP). The output are five scalars. Two of the scalars are predicted forward and angular velocities v_t and ω_t, one scalar is the logit of the stopping probability, and two scalars are standard deviations used during PPO training to define a Gaussian probability distribution over actions.

B.3 Coordinate Frames of Reference

At the start of each task, we define the world reference frame according to the agent's starting position, with x and y axis pointing forward and left respectively, according to the agent's position. The maps are represented with the origin at the center. Throughout the instruction execution, this reference frame remains fixed. Within the first model stage, the semantic and grounding maps, observability and boundary masks, and visitation distributions are all represented in the world reference frame. At the input to second stage, we transform the visitation distributions, and observability and boundary masks to the agent's current egocentric frame of reference. This allows the model to focus on generating velocities to follow the high probability regions, without having to reason about coordinate transformations.

C Additional Learning Details

C.1 Discriminator Architecture and Training

Figure 12 shows the neural network architecture of our discriminator h. The Wasserstein distance estimation procedure from Shen et al. [40] requires a discriminator that is K-Lipschitz continuous. We guarantee that our discriminator meets this requirement by clamping the discriminator parameters ψ to a range of $[-T_\psi; T_\psi]$ after every gradient update [40].

C.2 Return Definition

The expected return $R_t(\hat{\Xi})$ is defined as:

$$R_t(\hat{\Xi}) = \sum_{\substack{i \geq t, (s_i, a_i) \in \hat{\Xi}, \\ c_i = \mathcal{C}(s_i)}} \gamma^{i-t} r(c_i, a_i) \ ,$$

where $\hat{\Xi}$ is a policy execution, $\mathcal{C}(s_i)$ is the agent context observed at state s_i, γ is a discount factor, and $r(\cdot)$ is the intrinsic reward. The reward does not depend on any external state information, but only on the agent context and visitation predictions.

C.3 Reward Function

Section 5 provides the high level description and motivation of the intrinsic reward function.

Visitation Reward We design the visitation reward to reward policy executions $\hat{\Xi}$ that closely match the predicted visitation distribution d^p. An obvious choice would be the probability of the trajectory under independence assumptions $P(\hat{\Xi}) \approx \prod_{p \in \hat{\Xi}} d_t^p(p)$. According to this measurement, if $d_t^p(p) = 0$ for any $p \in \hat{\Xi}$, then $P(\hat{\Xi}) = 0$. This would lead to a reward of zero as soon as the pol-

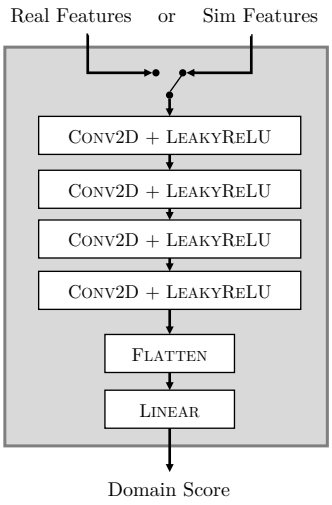


Figure 12: Our discriminator architecture. The discriminator takes as input a 3D feature map with two spatial dimensions and one feature dimension. It processes the feature map with a cascade of four convolutional neural networks with LeakyReLU non-linearities, before processing the output with a linear neural network layer. The discriminator is trained to output a scalar score that assigns high values to feature maps from the simulated domain, and low values from the real domain. The discriminator is used as component in our Wasserstein domain loss $\mathcal{L}_W$.

icy makes a mistake, resulting in sparse rewards and slow learning. Instead, we define the visitation reward in terms of earth mover's distance that provides a smooth and dense reward. The visitation reward r_v is:

$$r_v(c_t, a_t) = \phi_v(c_t, a_t) - \phi_v(c_{t-1}, a_{t-1}) \quad ,$$

where ϕ_v is a reward shaping potential:

$$\phi_v(c_t, a_t) = -\text{EMD}(\mathbb{1}_{p \in \Xi}, d_t^p(p_t \mid p_t \in \mathcal{P}^{\texttt{obs}})) \quad .$$

EMD is the earth mover's distance in Euclidean $\mathbb{R}^2$ space, $\mathbb{1}_{p \in \Xi}$ is a probability distribution that assigns equal probability to all positions visited thus far, $\mathcal{P}^{\texttt{obs}}$ is the set of positions the agent has observed so far,[2] and $d_t^p(p_t \mid p_t \in \mathcal{P}^{\texttt{obs}})$ is the position visitation distribution over all observed positions. Intuitively, r_v rewards per-timestep reduction in earth mover's distance between the predicted visitation distribution and the empirical distribution derived from the agent's trajectory.

Stop Reward Similar to r_v, the stop reward r_s is the negative earth mover's distance between the conditional predicted goal distribution over all observed environment locations, and the empirical stop distribution $\mathbb{1}_{p=\hat{\Xi}_{-1}}$ that assigns unit probability to the agent's final stopping position.

$$r_s(c_t, a_t) = -\mathbb{1}_{a_t=\texttt{STOP}} \cdot \text{EMD}(\mathbb{1}_{p=\hat{\Xi}_{-1}}, d_t^g(p_t \mid p_t \in \mathcal{P}^{\texttt{obs}})) \quad .$$

Exploration Reward The exploration reward r_e is:

$$r_e(c_t, a_t) = (\phi_e(c_t, a_t) - \phi_e(c_{t-1}, a_{t-1})) - \mathbb{1}_{a_t=\texttt{STOP}} \cdot d_t^g(p^{\texttt{oob}}) \quad , \tag{2}$$

where:

$$\phi_e(c_t, a_t) = \max_{t' < t} [1 - d_{t'}^g(p^{\texttt{oob}})] \quad .$$

The term ϕ_e reflects the agent's belief that it has observed the goal location p_g. $1 - d_{t'}^g(p^{\texttt{oob}})$ is the probability that the goal has been observed before time t'. We take the maximum over past timesteps to reduce effects of noise from the model output. The second term in Equation 2 penalizes the agent for stopping while it predicts that the goal is not yet observed.

[2] We restrict the metric to observed locations on the map, because as discussed in Section 4, all unobserved locations are represented by a dummy location $p^{\texttt{oob}} \notin \mathbb{R}^2$.

C.4 Auxiliary Objectives

During training, we add an additional auxiliary loss $\mathcal{L}_{aux}$ to the supervised learning loss $\mathcal{L}_{SL}$ to ensure that the different modules in the PVN model specialize according to their function. The auxiliary loss is:

$$\mathcal{L}_{aux}(c_t) = \mathcal{L}_{\text{percept}}(c_t) + \mathcal{L}_{\text{ground}}(c_t) + \mathcal{L}_{\text{lang}}(c_t) \ . \tag{3}$$

The text in the remainder of Section C.4 has been taken from Blukis et al. [16]. We present it here for reference and completeness.

Object Recognition Loss The object-recognition loss $\mathcal{L}_{\text{percept}}$ ensures the semantic map $\mathbf{S}_t^W$ stores information about locations and identities of objects. At timestep t, for every object o that is visible in the first person image I_t, we classify the feature vector in the position in the semantic map $\mathbf{S}_t^W$ corresponding to the object location in the world. We use a linear softmax classifier to predict the object identity given the feature vector. At a given timestep t the classifier loss is:

$$\mathcal{L}_{\text{percept}}(\theta_1) = \frac{-1}{|O_{\text{FPV}}|} \sum_{o \in O_{\text{FPV}}} [\hat{y}_o log(y_o)] \ ,$$

where $\hat{y}_o$ is the true class label of the object o and y_o is the predicted probability. O_{FPV} is the set of objects visible in the image I_t.

Grounding Loss For every object o visible in the first-person image I_t, we use the feature vector from the grounding map $\mathbf{R}_t^W$ corresponding to the object location in the world with a linear softmax classifier to predict whether the object was mentioned in the instruction u. The objective is:

$$\mathcal{L}_{\text{ground}}(\theta_1) = \frac{-1}{|O_{\text{FPV}}|} \sum_{o \in O_{\text{FPV}}} [\hat{y}_o log(y_o) + (1 - \hat{y}_o) log(1 - y_o)] \ ,$$

where $\hat{y}_o$ is a 0/1-valued label indicating whether the object o was mentioned in the instruction and y_o is the corresponding model prediction. O_{FPV} is the set of objects visible in the image I_t.

Language Loss The instruction-mention auxiliary objective uses a similar classifier to the grounding loss. Given the instruction embedding $\mathbf{u}$, we predict for each of the possible objects whether it was mentioned in the instruction u. The objective is:

$$\mathcal{L}_{\text{lang}}(\theta_1) = \frac{-1}{|O|} \sum_{o \in O_{\text{FPV}}} [\hat{y}_o log(y_o) + (1 - \hat{y}_o) log(1 - y_o)] \ ,$$

where $\hat{y}_o$ is a 0/1-valued label, same as above.

Automatic Word-object Alignment Extraction In order to infer whether an object o was mentioned in the instruction u, we use automatically extracted word-object alignments from the dataset. Let $E(o)$ be the event that an object o occurs within 15 meters of the human-demonstration trajectory Ξ, let $E(\tau)$ be the event that a word type τ occurs in the instruction u, and let $E(o, \tau)$ be the event that both $E(o)$ and $E(\tau)$ occur simultaneously. The pointwise mutual information between events $E(o)$ and $E(\tau)$ over the training set is:

$$\text{PMI}(o, \tau) = P(E(o, \tau)) \log \frac{P(E(o, \tau))}{P(E(o))P(E(\tau))} \ ,$$

where the probabilities are estimated from counts over training examples $\{(u^{(i)}, s_1^{(i)}, \Xi^{(i)})\}_{i=1}^N$. The output set of word-object alignments is:

$$\{(o, \tau) \mid \text{PMI}(o, \tau) > T_{\text{PMI}} \wedge P(\tau) < T_\tau\} \ ,$$

where $T_{PMI} = 0.008$ and $T_\tau = 0.1$ are threshold hyperparameters.

D Dataset Details

Natural Language and Demonstration Data Table 2 provides statistics on the natural language instruction datasets.

LANI Dataset Collection Details The crowdsourcing process includes two stages. First, a Mechanical Turk worker is shown a long, random trajectory in the overhead view and writes an instruction paragraph for a first-person agent. The trajectories were generated with a sampling process biased towards moving around the landmarks. Given the instruction paragraph, a second worker follows the instructions by controlling a discrete simple agent, simultaneously segmenting the instruction and trajectory into shorter segments. The output are pairs of instruction segments and discrete

Dataset	Type	Split	# Paragraphs	# Instr.	Avg. Instr. Len. (tokens)	Avg. Path Len. (m)
LANI	1-segment	Train	4200	19762	11.04	1.53
		Dev	898	4182	10.94	1.54
		Test	902	4260	11.23	1.53
	2-segment	Train	4200	15919	21.84	3.07
		Dev	898	3366	21.65	3.10
		Test	902	3432	22.26	3.07
REAL	1-segment	Train	698	3245	11.10	1.00
		Dev	150	640	11.47	1.06
		Test	149	672	11.31	1.06
	2-segment	Train	698	2582	20.97	1.91
		Dev	150	501	21.42	2.01
		Test	149	531	21.28	1.99

Table 2: Dataset and split sizes. LANI was introduced by Misra et al. [15]. Each layout in LANI consists of 6–13 landmarks out of a total of 64. REAL is the additional data we collected for use on the physical drone. In REAL, each layout has 5–8 landmarks from a set of 15 that is a subset of the landmarks in LANI.

ground-truth trajectories. We restrict to a pool of workers who had previously qualified for our other instruction-writing tasks.

Demonstration Data We collect the demonstration datasets $\mathcal{D}^{\text{R}}$ and $\mathcal{D}^{\text{S}}$ by executing a hand-engineered oracle policy that has access to the ground truth human demonstrations, and collect observation data. The oracle is described in Appendix E.2. $\mathcal{D}^{\text{S}}$ includes all instructions from original LANI data, as well as the additional instructions we collect for our smaller environment. Due to the high cost of data collection on the physical quadcopter, $\mathcal{D}^{\text{R}}$ includes demonstrations on only 100 paragraphs of single-segment instructions, approximately 1.5% of the data available in simulation.

Data Augmentation To improve generalization, we perform two types of data augmentation. First, we train on the combined dataset that includes both single-segment and two-segment instructions. Two-segment data consists of instructions and executions that combine two consecutive segments. This increases the mean instruction length from 11.10 tokens to 20.97, and the mean trajectory length by a factor of 1.91. Second, we randomly rotate the semantic map $\mathbf{S}^W$ and the gold visitation distributions by a random angle $\alpha \sim N(0, 0.5rad)$ to counter the bias of always flying forward, which is especially present in the single-segment data because of how humans segmented the original paragraphs.

E Additional Experimental Setup Details

E.1 Computing hardware and training time

Training took approximately three days on an Intel i9 7980X CPU with three Nvidia 1080Ti GPUs. We used one GPU for the supervised learning process, one GPU for the RL process for both gradient updates and roll-outs, and one GPU for rendering simulator visuals. We ran five simulator processes in parallel, each at 7x real-time speed. We used a total of 400k RL simulation rollouts.

E.2 Oracle implementation

The oracle uses the ground truth trajectory, and follows it with a control rule. It adjusts its angular velocity with a P-control rule to fly towards a dynamic goal on the ground truth trajectory. The dynamic goal is always 0.5m in front of the quadcopter's current position on the trajectory, until it overlaps the goal position. The forward speed is a constant maximum minus a multiple of angular velocity.

E.3 Environment and Quadcopter Parameters

Environment Scaling The original LANI dataset includes a unique 50x50m environment for each paragraph. Each environment includes 6–13 landmarks. Because the original data is for a larger

environment, we scale it down to the same dimension as ours. We use the original data split, where environments are not shared between the splits.

Action Range We clip the forward velocity to the range $[0, 0.7]$m/s and the yaw rate to $[-1.0, 1.0]$rad/s. During training, we give a small negative reward for sampling an action outside the intervals $[-0.5, 1.7]$m/s for forward velocity and $[-2.0, 2.0]$rad/s for yaw-rate. This reduces the chance of action predictions diverging, and empirically ensures they stay mostly within the permitted range.

Quadcopter Safety We prevent the quadcopter from crashing into environment bounds through a safety interface that modifies the controller setpoint ρ. The safety mechanism performs a forward-simulation for one second and checks whether setting ρ as the setpoint would cause a collision. If it would, the forward velocity is reduced, possibly to standstill, until it would no longer pose a threat. Angular velocity is left unmodified. This mechanism is only used when collecting demonstration trajectories and during test-time. No autonomous flight in the physical environment is done during learning.

First-person Camera We use a front-facing camera on the Intel Aero drone, tilted at a 15 degree pitch. The camera has a horizontal field-of-view of 84 degrees, which is less than the 90-degree horizontal FOV of the camera used in simulated experiments of Blukis et al. [16].

F Evaluation Details

F.1 Automated Evaluation Details

We report two automated metrics: success rate and earth mover's distance (EMD). The success rate is the frequency of executions in which the quadcopter stopped within 0.47m of the human demonstration stopping location. To compute EMD, we convert the trajectories executed by the quadcopter and the human demonstration trajectories to probability distributions with uniformly distributed mass across all positions on the trajectory. EMD is then the earth mover's distance between these two distributions, using Euclidean distance as the distance metric. EMD has a number of favorable properties, including: (a) taking into account the entire trajectory and not only the goal location, (b) giving partial credit for trajectories that are very close but do not overlap the human demonstrations, and (c) is smooth in that a slight change in the executed trajectory corresponds to at most a slight change in the metric.

F.2 Human Evaluation Details

Navigation Instruction Quality One out of 73 navigation instructions that the majority of workers identified as unclear is excluded from the human evaluation analysis. The remaining instructions were judged by majority of workers not to contain mistakes, or issues with clarity or feasibility.

Mechanical Turk Evaluation Task Figure 13 shows the instructions given to workers for the human evaluation task. Figure 14 shows an example human evaluation task. We use the simulated environment to visualize the quadcopter behavior to the workers because it is usually simpler to observe. However, we use this interface to evaluate performance on the physical environment, and use trajectories generated in the physical environment. We avoid using language descriptions to describe objects to avoid biasing the workers, and allowing them to judge for themselves how well the instruction matches the objects. We observed that a large number of workers were not able to reliably judge the efficiency of agent behavior, since they generally considered correct behavior efficient and incorrect behavior inefficient. Therefore, we do not report efficiency scores. Figure 15 shows examples of human judgements for different instruction executions.

We need your help to understand how well our drone follows instructions.

Your task: Read the navigation instruction below, and watch the animation of the drone trying to follow the instruction. Then answer three questions. Consider the guidelines below:

- Looking around: The drone can only see what's directly in front of it as indicated by the highlighted region. Depending on the instruction, it may have to look for certain objects to understand where to go, and looking around for them is the right way to go and is efficient.
- Bad instructions: If the instruction is unclear, impossible, or full of confusing mistakes, please indicate it by checking the checkbox. You must still answer all the questions - consider if the drone's behavior was reasonable given the bad instruction.
- Objects: The drone observes the environment from a first-person view. The numbered images illustrate how each object would look like to the drone. Consider the appearance of objects from the first-person perspective in relation to the instructions.
- Note: Try to answer each question in isolation, focusing on the specific behavior. For example, if the drone reached the goal correctly, but took the wrong path, you should answer "Yes, perfectly" for question 2, while giving a worse score for question 1. Similarly, if the drone went straight for the goal, that would be considered "efficient" behavior, even though it may have taken the wrong path to get there
- The drone might sometimes decide not to do anything at all, maybe because it thinks it's already where it was instructed to go. If that happens you won't see any movement in the animation.
- The drone always "stops" at the end, even if the motion appears to end abruptly.
- The field is surrounded by a red fence on top, white fence on the right, blue fence on the bottom, and yellow fence on the left. The colorful borders are there to help you better distinguish between these colors.

Figure 13: The instructions given to the crowdsourcing workers during human evaluation.

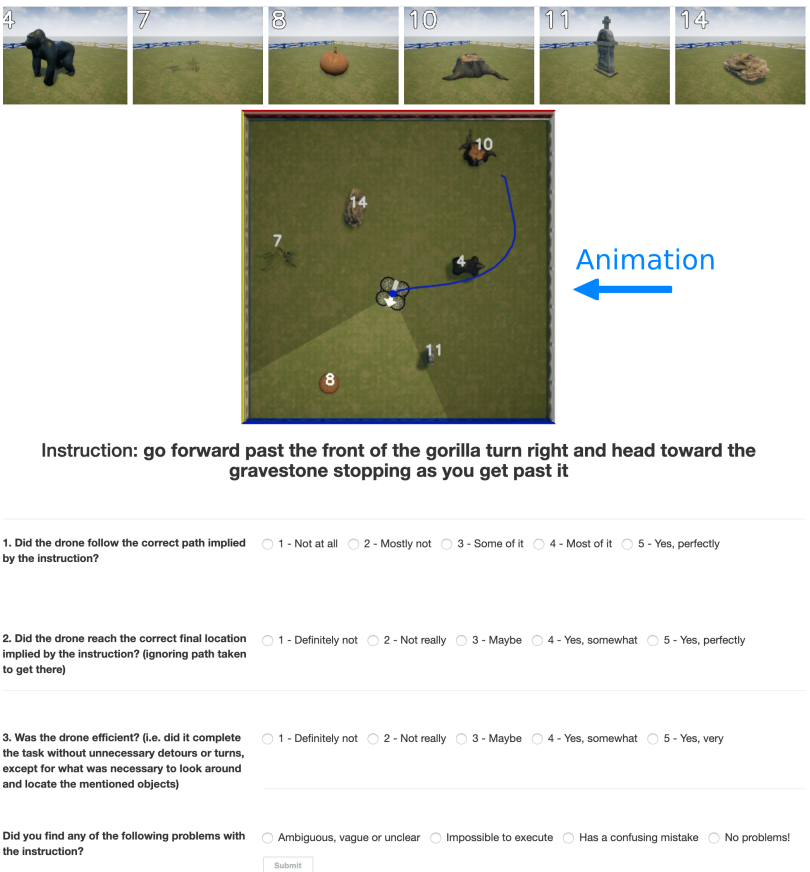


Figure 14: Human evaluation task. The images on top show the various objects in the environment from a reasonable agent point of view, and numbers indicate correspondence to objects in the top-down view animation. The animation shows the trajectory that the agent took to complete the instruction. Because efficiency scores are unreliable, we do not report them.

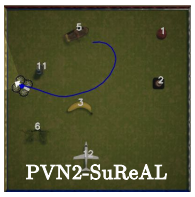
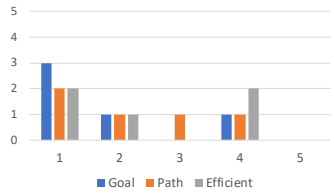
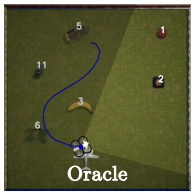
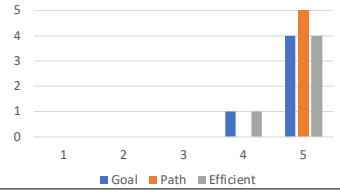
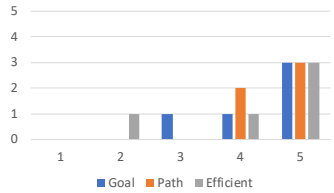
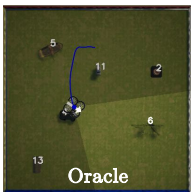
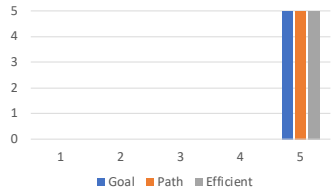
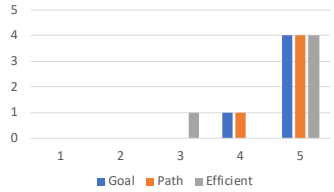
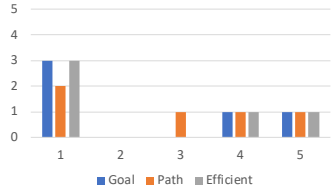
Instruction	Execution	Scores
then take sharp right and fly towards the palm and st op when it is at your 4 o clock then take slight left and fly towards the airplane and stop when its left side is in front of you then pass the airplane on the left side		
		
go forward slightly and then turn left facing the gorilla fly to the head of the gorilla and stop		
		
take sharp right and fly towards the blue fence		
		

Figure 15: Human evaluation scores for three instructions of different difficulty, for our model PVN2-SUREAL and the ORACLE. Horizontal axis represents Likert scores and bar heights represent score frequencies across five MTurk workers. Goal, Path, and Efficiency scores represent answers to the corresponding questions in Figure 14.

Method	1-segment		2-segment	
	SR	EMD	SR	EMD
Test Results On Full LANI Test Set				
STOP	7.7	0.76	0.7	1.29
AVERAGE	13.0	0.62	5.8	0.94
PVN-BC	37.8	0.47	19.6	0.76
PVN2-BC	**39.0**	**0.43**	21.0	0.72
PVN2-SUREAL	37.2	**0.43**	**21.5**	**0.67**
ORACLE	98.6	0.15	93.9	0.20
Dev Results On Full LANI Test Set				
PVN2-SUREAL	35.8	0.44	19.8	0.68
PVN2-SUREAL-NOEXP	38.5	0.43	15.8	0.79
PVN2-SUREAL-NOU	26.1	0.48	7.0	0.90

Table 3: Additional automated evaluation results on the full LANI test and development sets of $50m \times 50m$ environments, including the additional $4.7m \times 4.7m$ examples we added to support experiments on the real quadcopter. The 1-segment numbers in these tables are loosely comparable to prior work that used LANI [16, 15], except in that we used additional data during training, and trained in a joint two-dataset domain-adversarial setting, which may have unpredictable effects on simulation performance. We also have reduced the camera horizontal FOV from 90 degrees to 84, which exacerbates observability challenges.

G Additional Results

Table 3 shows simulation results on the full LANI test and development data.

H Common Questions

Do you assume any alignment between simulated and real environments? Our learning approach does not assume that simulated and real data is aligned.

What is the benefit of SUREAL over reinforcement learning with an auxiliary loss as a means of utilizing annotated data? There are a number of reasons to prefer SUREAL:

- The 2-stage decomposition means that there is no gradient flow from second to first stage, and so the policy gradient loss will not update Stage 1 parameters anyway.
- With SUREAL, only the second stage needs to be computed during the PPO updates, which drastically improves training speed.
- In SUREAL, we do not need to send new Stage 1 parameters to actor processes at every iteration, since these parameters are not optimized with reinforcement learning.

Why did you select this particular set of baselines? The baselines have been developed for instruction following in a very similar environment to ours, allowing us to focus on evaluating domain transfer performance, exploration performance, and the effect of our learning method. Instruction-following tasks have unique requirements for sample complexity and utilizing limited annotated data. Comparisons against general sim-to-real models may not yield useful insights, or yield insights that have already been demonstrated in prior work [34].

Why do you learn control, rather than using deterministic control? The predicted visitation distributions often have complex and unpredictable shapes, depending on the instruction and environment layout. Designing a hand-engineered controller that effectively follows the predicted distributions reduces to either (a) a challenging optimal control problem at test time or (b) a difficult engineering and testing challenge. We instead opt to learn a solution at training time. We find that learning Stage 2 behavior is more straightforward, and is in line with our general strategy of reducing engineering effort and use of hand-crafted representations.

How noisy are your pose estimates? We use a Vicon motion capture system to obtain pose estimates. The poses have sub-centimeter positional accuracy, but often arrive at the drone with delay due to network latency.

Hyperparameter	Value
Environment Settings	
Maximum yaw rate	$\omega_{\max} = 1m/s$
Maximum forward velocity	$v_{\max} = 0.7m/s$
Image and Feature Dimensions	
Camera horizontal FOV	84°
Input image dimensions	$128 \times 72 \times 3$
Feature map $\mathbf{F}^C$ dimensions	$32 \times 18 \times 32$
Semantic map $\mathbf{S}^W$ dimensions	$64 \times 64 \times 32$
Visitation distributions d^g and d^p dimensions	$64 \times 64 \times 1$
Environment edge length in meters	$4.7m$
Environment edge length in pixels on $\mathbf{S}^W$	32
Model	
Visitation prediction interval timesteps	$T_d = 1$
`STOP` action threshold	$\kappa = 0.5$
General Learning	
Deep Learning library	PyTorch 1.0.1
Classification auxiliary weight	$\lambda_{\text{percept}} = 1.0$
Grounding auxiliary weight	$\lambda_{\text{ground}} = 1.0$
Language auxiliary weight	$\lambda_{\text{lang}} = 1.0$
Supervised Learning	
Optimizer	ADAM
Learning Rate	0.001
Weight Decay	10^{-6}
Batch Size	1
Reinforcement Learning (PPO)	
Num supervised epochs before starting RL (K^B_{iter})	25
Num epochs ($K^{\text{RL}}_{\text{epoch}}$)	400
Iterations per epoch ($K^{\text{RL}}_{\text{iter}}$)	50
Number of parallel actors	4
Number of rollouts per iteration N	20
PPO clipping parameter	0.1
PPO gradient updates per iter ($K^{\text{RL}}_{\text{steps}}$)	8
Minibatch size	2
Value loss weight	1.0
Learning rate	0.00025
Epsilon	1e-5
Max gradient norm	1.0
Use generalized advantage estimation	False
Discount factor (γ)	0.99
Entropy coefficient	0.001
Entropy schedule	multiply entropy coefficient by 0.1 after 200 epochs
Reward Weights	
Stop reward weight (λ_s)	0.5
Visitation reward weight(λ_v)	0.3
Exploration reward weight (λ_e)	1.0
Negative per-step reward (λ_{step})	-0.04

Table 4: Hyperparameter values.

Is the model tolerant to noise? Prior work has tested the semantic mapping module that we use against noisy poses and found that it can tolerate moderate amounts of noise [34] even without explicit probabilistic modelling. This is not the focus of this paper, but we find that our model is able to recover from erroneous pose-image pairs caused by network latency. Evaluating the robustness of the model to more types of noise, including in pose estimates, is an important direction for future work.

I Hyperparameters

Table 4 shows the hyperparameter assignments. The hyperparameters were manually tuned on the development set to trade off use of computational resources and development time. We do not claim the selected hyperparameters are optimal, but we did observe that they consistently resulted in learning convergence and stable test-time behavior.