TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Elektrotechnik und Informationstechnik
Self-Organizing Systems Lab

Indoor Visual Navigation on Micro-Aerial Drones without External Infrastructure

Master-Thesis
Distributed Software Systems

Eingereicht von

Amos Newswanger

am
22.4.2022

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Kai Cui

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Amos Newswanger, die vorliegende Arbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Arbeit stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

English translation for information purposes only:

Thesis statement pursuant to §22 paragraph 7 and §23 paragraph 7 of APB TU Darmstadt:
I herewith formally declare that I, Amos Newswanger, have written the submitted thesis independently pursuant to §22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

Darmstadt, den 22.4.2022

(Amos Newswanger)

Abstract

For robots to be able to do many tasks in the real world that humans are able to do, they need to be able to navigate unknown, chaotic indoor environments, without the need for any special infrastructure to be installed. Human use vision to navigate indoor environments, and robots can as well. This work explores the possibility for indoor visual navigation on a highly constrained nano-quadrotor drone. This places strict limits on the computational power available, energy usage requirements, and sensors that can be used. The drone uses a single low power grayscale camera for navigation, and a downward facing optical flow based odometer to track its lateral movement. We show that training a constrained end-to-end neural network policy in a photo-realistic simulation can translate into real world performance.

Contents

1 Introduction

Fully autonomous robots that can perform complex tasks in the real world base on high level instructions have the potential to disrupt large segments of the economy. For instance, self driving vehicles have the potential to fundamentally change the way we transport people and goods. Humanoid robots, coupled with general artificial intelligence algorithms and learning processes, could one day serve as a general replacement for human labor. This could eliminate the need for humans to do many tedious, time consuming, and dangerous jobs. However, fully autonomous robots still face many hurdles that will be difficult to overcome. The robot will need to be able to navigate uncontrolled, chaotic environments that it has potentially never seen before. Like a human, the robot should be able to operate with little or no external infrastructure, instead relying on its own internal sensors. In order to be able to operate for extended periods, the robot needs to have a low energy consumption, which limits the processing power and sensors that the robot can carry on-board.

This thesis explores one basic building block needed for many of these fully autonomous systems: visual navigation in an unknown indoor environment without using any external infrastructure, on a highly constrained robot. GPS does not work well indoors, so it cannot be use as a positioning system. Additionally, if we would like robots to be able to operate inside many of the millions of pre-existings indoor environments that have been constructed for humans, we should not rely on special positioning systems to be installed just for robotic navigation as this would be very expensive and limiting. Visual navigation with onboard cameras is thus a promising method to be able to successfully navigate in an uncontrolled an chaotic indoor environment.

The specific robot used is the Crazyfie 2.1 micro-aerial quad-rotor drone. Due to its small size, it has very constrained power and weight limits. The onboard sensors used for navigation are a single forward facing grayscale camera, a downward facing laser ranger for height measurements, and a downward facing optical flow sensor for lateral movement estimation. The available onboard computation is the GAP8 parallel ultra low power processor (PULP). This chip is designed to be able to execute neural networks efficiently, but is still very constrained compared to many other chips.

The task the drone is given is to navigate to a goal point which is defined relative to the drones current position (eg, 3 meters forward and 2 meters to the left) while avoiding obstacles. The AI agent we use to control the drone is an end-to-end neural network that takes a grayscale image and a point goal as an input and outputs an action such as turn left, turn right, move forward or stop. To train this agent, we use reinforcement learning in a photorealistic 3D simulator, AI Habitat [26], which uses 3D scans of real indoor environments as source data.

1.1 Related Work

Visual navigation has a long history in robotics [7, 13]. Broadly, navigation tasks can be categorized as *map-based* and *mapless* techniques, where the map can be metricly acurate map or a topological map. Classic approaches involve building a map with Simultaneous Localization and Mapping (SLAM) by fusing information from multiple views of the environment [43, 18]. The map build with these methods can then be used for navigation and path planing [23]. These classical approaches have inspired learning based methods which attempt to learn how to build spatial representations [10, 33], or a topological map [11]. Classical mapless techniques can be reactive based on appearance-based localization, optical feature tracking, and ground tracking [7]. We use a ground tracking optical flow technique to measure lateral movement of the robot. Learning based mapless techniques can use learned memory representation to build an implicit model of the environment. [28] uses an LSTM to learn how to navigate a maze, and [30] uses an LSTM for semantic target navigation. [15] uses a transformer neural network [44] to build a sequence of embedded scene memory vectors.

There are many different goals for visual navigation. This can range from simply wandering around the environment without colliding with anything [16], exploration [12], following an object [25], getting to a goal position (as in this thesis) [5, 45], following natural language instructions [6], finding a specific image taken from the environment [14, 38], or finding some type of object [17].

In point goal navigation, where the goal is to get to a goal coordinate position. The agent could be given the goal relative to the starting position only [21], or the point goal can be updated at each step based on the agents current position [45]. Under this setup, [45] achieves near perfect point goal navigation with an LSTM network, trained with a massively distributed PPO reinforcement learning algorithm. [21] proposes a metric for evaluating how well a simulator predicts performance in the real world using various point goal navigation algorithms. They suggest that disabling wall sliding in the simulated environment improves real world predictivity significantly.

In the space of visual navigation indoor on drones, [16] trained a drone to fly by collecting data on crashes from a real drone. [32] used a small neural network to steer a Crazyflie nano-quadcopter drone by training it on a large dataset of cars navigating in the real world. They also trained the drone to stop before hitting objects by training the same neural network on a novel dataset of videos leading up to a crash. They were able to execute the network on a parallel ultra low power chip onboard the drone. [27] is another example of a navigation algorithm on the Crazyflie platform. They implemented a minimal bug-like algorithm for exploration with a swarm of drones, each equiped with optical flow sensor for lateral movement, and multi-directional laser rangers to do wall following.

2 Foundations

This chapter provides an overview of the fundamental concepts and building blocks used throughout the thesis.

2.1 Neural Networks

An Artificial Neural Network (or simply a Neural Network) can be thought of as a universal function approximator. We can denote a neural network as a function π_θ, where θ represents the parameters of the neural network (often called weights). The input of the function is a element in the set $\mathcal{X}$, and the output of the function is an element in the set $\mathcal{Y}$. By choosing a good internal structure for π_θ, and tuning the parameters of the network, θ, we can make π_θ approximate some desired mapping of elements in $\mathcal{X}$ to elements in $\mathcal{Y}$.

A common example use case for neural networks is handwritten digit recognition. In this example, elements in $\mathcal{X}$ might be a 28 by 28 array of pixels representing an image of a handwritten digit. Elements in $\mathcal{Y}$ might be a 10-dimensional array where the nth element represents the probability that the input image is the digit n. The goal would then be to find a neural network π_θ which for each input image, would output an array of probabilities where the highest probability corresponds to the digit that the input image represents.

Defining the internal structure of a neural network is critical step in finding a neural network that is capable of approximating a goal function, and it is the source of a large amount of research. Neural networks are typically composed of a series of layers, each of which is typically made up of a linear operation followed by a non-linear operation.

The most basic layer is the fully connected layer. Given the input of the layer $l_{in} \in \mathbb{R}^n$, the output of the layer $l_{out} \in \mathbb{R}^m$, the weight parameters of the layer $W \in \mathbb{R}^{m \times n}$ and bias parameters $B \in \mathbb{R}^n$, and an activation function Act, we can define a fully connected layers using matrix operations

$$l_{out} = \text{Act}(W l_{in} + B). \tag{2.1}$$

The activation function Act is a non-linear function applied pointwise to each element. Common activation functions are sigmoid, tanh and rectified linear ($\max(0, x)$). Fully connected layers are inspired by biological neurons, which take in electrical signals for various other neurons, sum the signals linearly, and then fires its own signal (activates) when the sum of the input signals pass a certain threshold. Chaining together a handful of fully connected layers has proven powerful enough to approximate functions such as handwritten digit classification

as described above, and simple object recognition. In the subsequent sections, we will describe convolutional layers and LSTM recurrent layers which are used in this thesis.

Tuning the parameters θ of the neural network is the next critical step in finding a neural network that approximates a goal function. Neural networks can typically have anywhere from tens of parameters, to hundreds of billions of parameters, such as OpenAI's GPT-3 language model [8]. For almost all interesting tasks, choosing network parameters by hand is not an option. Therefore, a systematic method is required to tune the parameters, such as gradient descent.

Given a function that we would like to approximate, we can define a dataset over this function as some set of points $\{(x, y)|x \in \mathcal{X}, y \in \mathcal{Y}\}$. We can partition this dataset into two disjoint sets, Train and Test. The Train Train set can be used to tune the parameters θ of our network, and Test can be used to evaluate how well our network approximates the goal function. This is a standard setup for supervised learning. To tune the parameters θ, we can use gradient descent. We define an error function $J_{\text{Train}}(\theta)$ on θ for the points in our Train set. A common error function is the mean squared error:

$$J(\theta) = \frac{1}{n} \sum_{(x,y) \in Train} (\pi_\theta(x) - y)^2 \tag{2.2}$$

where n is the number of items in Train. Minimizing J means that the output of π is getting closer to the desired output. As long as π is fully differentiable, we can calculate the gradient of J with respect to θ, $\nabla_\theta J$. To tune the parameters θ, we can minimize J by iteratively updating θ with the formula

$$\theta := \theta - \eta \nabla_\theta J(\theta) \tag{2.3}$$

where η is the update size (also called learning rate). This can be quite expensive though, as a single update requires iterating over the entire Train set. To speed the process up, we can calculate the gradient update for a single element in Train (Stochastic Gradient Descent [35]), or for small batched of elements in Train (Mini-batch Gradient Descent [35]).

In Section 2.2 we describe the reinforcement learning methods use in this thesis for tuning our policy neural network.

2.1.1 Convolutional Neural Networks

Fully connected layers work well for many small tasks. However, when the input to the layer is very large, the number of parameters in the layer quickly becomes intractably large. For example, and fully connected layer with input and output of dimensions of $256x256x3$ would have around 38 billion parameters. This limits the network to having lower resolution inputs, and fewer layers, which limits the discriminative power of the network. Convolutional Neural Networks (CNNs) were developed to tackle these issues, particularly in the field of computer vision which requires high dimensional inputs. A CNN layer reduces the number of parameters required by applying the same small group of parameters across the spacial dimension of the input, thus reusing the same parameters many times.

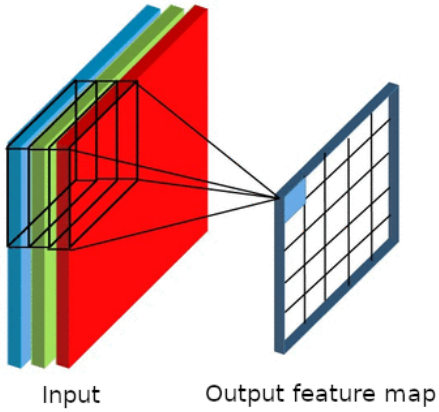


Figure 2.1: Illustration of the cross-correlation operation (also referred to as convolution) of a kernel and an input. A single value in the output feature map is calculated by multiplying the kernel pointwise to a subsection of the input and summing the results

We group the parameters of a convolutional layer into i filters (also commonly called kernels) denoted by $k_i \in \mathbb{R}^{C_{in} \times n \times n}$ and bias $b_i \in \mathbb{R}$ where n is a small number such as 3 or 5. Given the input of the layer $l_{in} \in \mathbb{R}^{C_{in} \times H_{in} \times W_{in}}$, for each k_i, we calculate the output of the layer $l_{out}^{(i)} \in \mathbb{R}^{1 \times H_{out} \times W_{out}}$ with the formula

$$l_{out}^{(i)} = (l_{in} * k_i) + b_i \tag{2.4}$$

where $*$ denotes the cross-correlation along the spatial dimension (height and width), which is illustraed in Figure 2.1. $l_{out}^{(i)}$ can be thought of as a heatmap showing the spatial locations where the filter k_i is highly correlated with the input. $l_{out}^{(i)}$ is also commonly refered to as a feature map. If k_i represents a horizontal edge, then $l_{out}^{(i)}$ would be a heatmap of the location in the input where there are patterns resembling a horizontal edge. The output of the layer, $l_{out} \in \mathbb{R}^{i \times H_{out} \times W_{out}}$, is obtained by concatenating all the $l_{out}^{(i)}$ feature maps and then applying a non-linear activation function. H_{out} and W_{out} are calculated based on the stride of the cross-correlation operation, the size of the filter, and whether the input was padded with extra values.

Chaining multiple convolutional layers together results in a CNN which can perform very complex discriminative tasks. Intuitively, kernels in the early layers of the network represent very simple features, such as edges and corners. Kernels in subsequent layers combine the feature maps produced by previous kernels, so they can represent features of increasing complexity, such as the combination of multiple edges into a square. Downsampling the feature maps along the spatial dimension gives kernels in later layers a larger receptive field (the area in the original input which affects the output in a single position of a kernel's feature map), and ensures that the kernel has some invariance to small spatial shifts in the input.

The power of CNNs was demonstrated over many years on the ImageNet Large Scale Visual Recognition Challenge [36]. Given an image containing an object, the challenge is to correctly

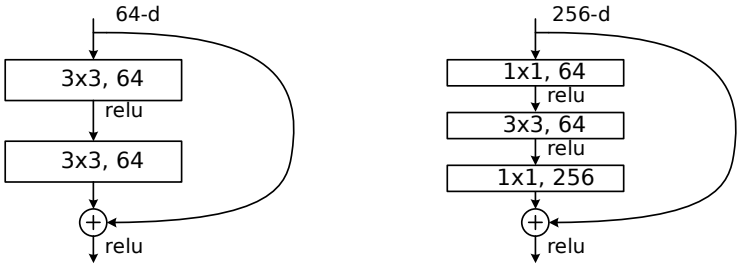


Figure 2.2: Left: A residual block where the input is added to the output of two sequential convolutional layers. Right: A bottleneck residual block where the first 1×1 convolution layer reduces the channel dimension, and the second 1×1 convolution restores the channel dimension. The residual connection allows for a parameter free path for information to bypass the layer.

classify what object is present in the image from a list of 1000 possible object categories. The ImageNet dataset contains over a million images that have been manually annotated with the correct object class. In 2012, a CNN called AlexNet demonstrated that CNNs could beat the previous top methods by a significant margin [24]. Three years later in 2015, ResNet achieved better than the estimated human performance on the challenge [19]. ImageNet has continued to be a benchmark for novel neural network architectures. In the following two sections, we describe two CNNs developed for ImageNet which we use as a visual encoder in our policy neural network. We use these CNNs to encode a high dimensional input image into a low dimensional vector which contains compressed information about the content of the image.

2.1.2 ResNet-50

The ResNet CNN architecture was introduced in 2015 by [19]. It introduced the idea of a residual block, illustrated in Figure 2.2. In a residual block, the input of the block is added to the output of the block. This creates a parameter-free path for information to bypass a layer. With normal convolutional layers, adding an excessive number of layers often results in a degredation of performance as the network becomes more difficult to optimize. However, [19] showed that with residual blocks it becomes easier to optimize a network with a large number of layers, and that the added layers results in better performance.

Figure 2.3 illustrates the full ResNet architecture with 34 layers using residual blocks. We use the ResNet architechture with 50 layers, which is the same as the depicted architecture, but with bottleneck blocks instead of normal residual blocks as illustrated in Figure 2.2. The bottleneck block was introduced as a further optimization to the residual block. It first applies a 1×1 convolution to the input. This can be thought of as a linear combination along the channel dimension of the input, and it is useful to be able to compress or expand the number of channels. The bottleneck block first compresses the channels, then applies a more expensive 3×3 convolution, and then expands the channel dimension with another 1×1 convolution. The residual connection allows information to bypass the bottleneck, which avoids information getting lost in the compression of the channels.

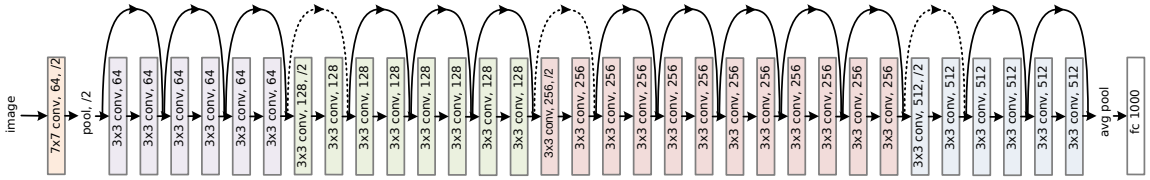


Figure 2.3: ResNet CNN with 34 residual layers. Each layer shows the size of the kernel, the number of kernels, and /2 if the output is downsampled by half. The final fully connected layer predicts the probability of the 1000 object classes in the ImageNet challenge

Input	Operator	t	n
$224^2 \times 3$	conv 3×3		1
$112^2 \times 32$	bottleneck	1	1
$112^2 \times 16$	bottleneck	6	2
$56^2 \times 24$	bottleneck	6	3
$28^2 \times 32$	bottleneck	6	4
$14^2 \times 64$	bottleneck	6	3
$14^2 \times 96$	bottleneck	6	3
$7^2 \times 160$	bottleneck	6	1
$7^2 \times 320$	conv 1×1		1
$7^2 \times 1280$	avgpool 7×7		1
$1^2 \times 1280$	conv 1×1		

Table 2.1: MobileNetV2 architecture. Each layer is repeated n times. The bottleneck blocks are inverted residual blocks with an expansion ratio of t

2.1.3 MobileNetV2

MobileNetV2 [37] was designed with the goal of reducing the size of the network and the total number of operations needed to execute a full forward pass of the network. This is motivated by the desire to execute neural networks on edge or mobile devices which have limited memory or computational power.

MobileNetV2 introduces the inverted residual block as illustrated in Figure 2.4. This block makes use of depthwise separable convolution. The idea of separable depthwise convolution is to replace a single $n \times n$ convolution with a $n \times n$ depthwise convolution followed by a 1×1 convolution. In depthwise convolution, each kernel k_i is in the shape $\mathbb{R}^{1 \times n \times}$, and there is a single kernel for each input channel. Each kernel is convolved with only one channel in the input. The benefit of doing this is that separable depthwise convolution requires significantly fewer parameters, and fewer operations required to execute the layer. For a kernel size of 3×3, this is results in an 8 to 9 times reduction in computational cost over a normal convolutional layer. MobileNetV2 demonstrates that depthwise convolution results in only minimal performance degradation. Table 2.1 gives the details of the MobileNetV2 architecture.

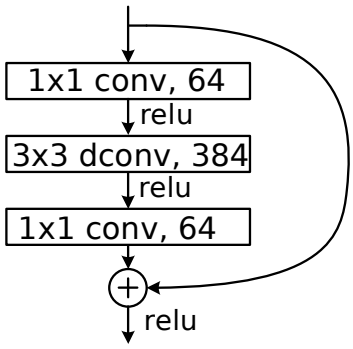


Figure 2.4: Illustration of an inverted residual block. The 1×1 conv layer first expands the channel dimension by a factor of 6, followed by an efficient 3×3 depthwise convolution, and then a 1×1 convolution which compresses the channels again.

2.1.4 LSTM Recurrent Neural Networks

In many cases, the input to a neural network is sequential in nature. For example, it could be a sequence of vectors representing characters in a natural language sentence. It could also be time-series data such as frames of a video, or readings from a sensor. In our case, our input data is a sequence of observations taken from an environment by an agent. The output of the network can also be a sequence, such as a newly generated natural language sentence. In our case, the output is a series of actions that the agent will perform in the environment based on observations it receives. To deal with these types of sequential inputs and outputs, Recurrent Neural Networks (RNNs) were developed.

At time step t, an RNN takes an input vector x_t which represents the t-th item in the input sequence. It also takes a state vector h_{t-1} as an input, which represents an encoding of information about the previous input vectors. The RNN then outputs a corresponding y_t output vector, and an updated state vector, h_t, which will become the input to the network in the next step.

A Long Short Term Memory network (LSTM) is one type of RNN network. Figure 2.5 shows the structure of an LSTM cell. The cell state is broken down into two components, a long term memory C, and a short term memory, h. The long term memory is only operated on by linear operations, so it is capable of holding on to information over many time steps. At each time step, the cell can remove information from C by multiplying it by a mask of values between 0 and 1 produced by a "forget" fully connected layer. It can also insert information into C by adding values pointwise to C from the output of an "update" fully connected layer. Finally, the h_t, which is also the output of the network, y_t, is calculated by "reading" information out of C_t. This is done by multiplying the tanh of C by a mask of values from 0 to 1 calculated by a fully connected layer on h_{t-1}

The inspiration for this architecture is the need to encode information about dependencies in the sequential data over many time steps. The long term memory achieves this by only selectively updating and deleting information at each timestep, and avoiding nonlinear operations

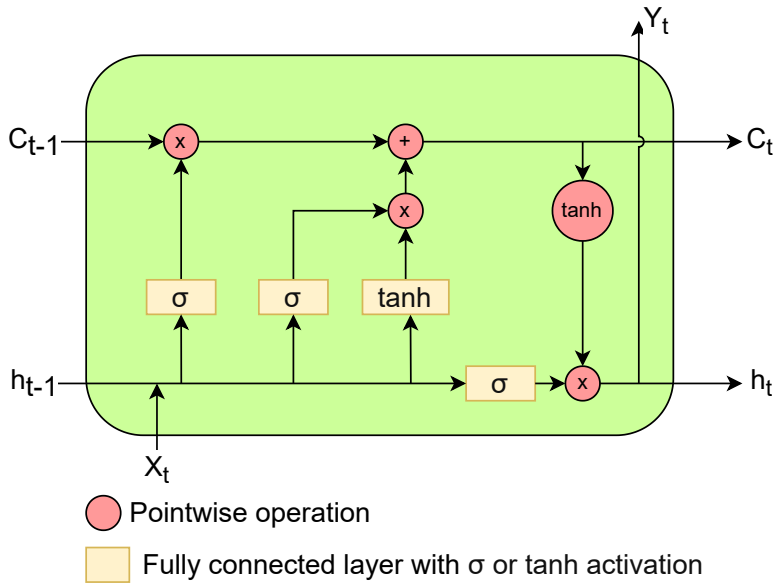


Figure 2.5: Diagram of an LSTM network. Arrows represent vectors. Converging arrows are concatenations, and diverging arrows are copy operations

which tend to destroy information over many repeated applications.

2.2 Reinforcement Learning

Reinforcement learning deals with the problem of how an agent should take actions in an environment in order to maximize the cumulative reward that it gets. The environment could be the real world, or a simulated environment such as videogames, or boardgames like chess and Go. The reward serves as a signal indicating some notion of how well or how poorly the agent is performing on some task in the environment. For instance, in a videogame you might get points for collecting items or clearing levels. In chess you might get some reward for capturing an opponents piece, or checkmating the king. Reinforcement learning deals with trying to teach an agent how to maximize the cumulative reward that it receives.

2.2.1 Formal Definition

Most RL problems can be framed as a Markov Decision Process (MDP). An MDP is a stochastic control process which models the synchronous interaction of an agent with an environment, as shown in Figure 3.1. In an MDP, the outcome of the process is partly random, and partly in control of the agent. An MDP can be defined by a tuple $(\mathcal{S}, \mathcal{A}, T, R)$ where

- $\mathcal{S}$ is a set of states

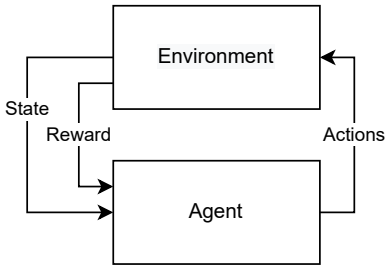


Figure 2.6: Synchronous interaction of an agent with an environment in a Markov Decision Process

- $\mathcal{A}$ is the set of actions
- $T(s'|s,a)$ is the transition probability function of transitioning to state s' given action a is executed in state s
- $R(s,a)$ is the reward function which gives the expected immediate reward for executing action a in state s

The next state and reward depend only on the previous state and action taken. This is known as the Markov Property.

Additionally we can define several other useful terms:

A **policy** π defines the behavior of the agent. It determines which actions the agent will take in any given state. A policy can be deterministic, denoted as $\pi(s) = a$, or stochastic, denoted as $\pi(s|a)$.

The **model** is the transition and reward function of the environment. The agent can have its own model which may be identical to the model of the environment. This is the case for many simple games such as chess, where the agent can perfectly simulate the environment. The agent could also attempt to learn a model which approximates the real model of the environment.

The **return** is the sum of future rewards. We define the return G_t as

$$G_t = \sum_{k=0}^{k=\infty} R_{t+k+1} \tag{2.5}$$

where R_t denotes the reward received at time step t. We can also define a discounted return G_t^γ

$$G_t^\gamma = \sum_{k=0}^{k=\infty} \gamma^k R_{t+k+1} \tag{2.6}$$

where $\gamma \in [0,1]$ is the discount factor. Discounting future rewards is useful because of the inherent stochastic nature of the markov decision process. Rewards that are far in the future are more difficult to accurately predict, so the discount factor reflects the uncertainty about being able to obtain the reward. This is similar to how stock market analysts discount the

future revenue predictions of a company due to uncertainty about how well the company can execute on its plans.

The **state value function**, $V_\pi(s)$, is the expected return in state s under policy π

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.7}$$

The **state action value function**, $Q_\pi(a, s)$, is the expected return of a state and action pair under policy π

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.8}$$

The **advantage function**, $A_\pi(s, a)$, is a measure of how much better action a is in state s compared to all other actions. It is defined as

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{2.9}$$

2.2.2 Optimization Methods

We would like to be able to find an agent that maximizes the expected reward. There are many optimization methods for finding such an agent, and they can broadly be categorized in a few useful ways.

Firstly, methods can be categorized as model-free and model-based methods. In model-based methods, the agent has access to a model of the environment. The model can either be learned, or given to the agent. For example, in chess the agent is given a perfect model of the world. Model-based methods allow the agent to look ahead in the state space without interacting with the environment by unrolling the model. The agent can plan multiple steps ahead. Model-based methods can be more sample efficient at learning than model-free methods, but learning a model of the environment may not always be possible. In this thesis we will not rely on a model.

Secondly, methods can be categorized as value-based or poicy-based. Value-based methods try to learn the state value function or the state-action value function. The agent then takes actions in each state based on the learned value functions. Q-learning is an example of this. The Q function can be represented as a table mapping state-action pairs to expected returns. The Q function can then be updated with dynamic programming through random exploration of the environment. However, if your environment has many states, this method becomes intractable. To solve this, DeepMind introduced Deep Q-learning, where the Q function is learned by a neural network. The neural network can infer information about states it has never visited before based on similarities to sates it has visited, something that a table in Q-learning cannot do. This method was able to beat many Atari games with super-human success [29].

We will use an policy-based method. Policy-based methods focus on directly learning the policy function. Our policy is a neural network which outputs a probability distribution over the action space. We use a policy gradient method to tune the parameters of the network.

Policy Gradient

Policy Gradient methods attempt to directly learn the policy function using gradient optimization. We define our policy as a differentiable function $\pi_\theta(s|a)$ which is parameterized by θ. As described earlier with supervised learning, we define an objective function $J(\theta)$ and use the gradient $\nabla_\theta J(\theta)$ to optimize the parameters θ. Unlike supervised learning, we don't have a set of training points which we can use to define an error function which can be minimized. Instead, we can define J as the the expected cumulative reward that can be received with policy π:

$$J(\theta) = V_\pi(s_0) \tag{2.10}$$

This goal function can then be maximized by gradient ascent to optimize θ. Although we can't calculate $\nabla_\theta J(\theta)$, we can calculate an approximate stochastic estimate $\widehat{\nabla_\theta J(\theta)}$ by sampling episodes from the distribution. By the *policy gradient theorem*, we have that

$$\nabla_\theta J(\theta) \propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} Q_\pi(s,a)\nabla_\theta \pi_\theta(a|s) \tag{2.11}$$

where $\mu(s)$ is the stationary probability of being in state s of the Markov chain of π and $\propto$ means "proportional to". The proof of this can be seen in Section 13.2 of [41]. We can rewrite this formula as an expectation which can be sampled in order to approximate the gradient:

$$\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} Q_\pi(s,a)\nabla_\theta \pi_\theta(a|s) \\
&= \mathbb{E}_\pi\left[\sum_{a\in\mathcal{A}} Q_\pi(S_t,a)\nabla_\theta \pi_\theta(a|S_t)\right] && \text{(replacing s by a sample } S_t\text{)} \\
&= \mathbb{E}_\pi\left[\sum_{a\in\mathcal{A}} \pi_\theta(a|S_t) Q_\pi(S_t,a)\frac{\nabla_\theta \pi_\theta(a|S_t)}{\pi_\theta(a|S_t)}\right] \\
&= \mathbb{E}_\pi\left[Q_\pi(S_t,A_t)\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\right] && \text{(replacing a by sample } A_t\sim\pi\text{)} \\
&= \mathbb{E}_\pi\left[G_t\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\right] && \text{(because } \mathbb{E}_\pi[G_t|S_t,A_t] = Q_\pi(S_t,A_t)\text{)} \\
&= \mathbb{E}_\pi\left[G_t\nabla_\theta \ln \pi_\theta(A_t|S_t)\right] && \text{(because } \nabla \ln x = \frac{\nabla x}{x}\text{)}
\end{aligned}$$

By sampling an episode $(S_0, A_0, R_1, S_1, A_1, R_2...)$, we can then update θ with the stochastic gradient ascent rule:

$$\theta := \theta + \eta G_t \nabla_\theta \ln \pi(A_t|S_t) \tag{2.12}$$

This is the REINFORCE algorithm as described in [41] Section 13.3. This leads to a generalized formulation of the gradient estimation problem:

$$\mathbb{E}_\pi[\Psi_t \nabla_\theta \ln \pi_\theta(A_t|S_t)] \tag{2.13}$$

where Ψ_t represents some credit assignment for taking action A_t in state S_t. The *credit assignment problem* is a fundamental challenge in reinforcement learning. There may be a

long time delay between when an action is taken and when the positive or negative reward that resulted from this action gets revealed. This means that there is no way to know which actions should be encouraged in the policy, and which ones should be discouraged. Instead, we can use the reward gathered over the episode as a general indicator of the quality of all action in the episode. Ψ_t is the specific term used to assign credit, and it can be one of the following as described in [39]:

- $\sum_{t=0}^{\infty} R_t$: the total reward of the episode
- $\sum_{t'=t}^{\infty} R_{t'}$: total reward after action A_t
- $\sum_{t'=t}^{\infty} R_{t'} - b(S_t)$: total reward after A_t minus some baseline function on state S_t
- $Q_\pi(S_t, A_t)$: state action value function
- $A_\pi(S_t, A_t)$: advantage function
- $R_t + V_\pi(S_{t+1}) - V_\pi(S_t)$: TD residual

We use the Generalized Advantage Estimator (GAE) proposed in [39]. We define the TD residual of V with discount factor γ as

$$\delta_t^V = R_t + \gamma V_\pi(S_{t+1}) - V_\pi(S_t) \tag{2.14}$$

GAE(γ, λ) is defined as:

$$\sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{2.15}$$

where $\lambda \in [0, 1]$. Setting λ and γ provide a way to control the tradeoff between bias and variance. The value function is learned by the same neural network that learns the policy function.

Proximal Policy Optimization

It can be tempting to do multiple parameter updates for the same sampled episodes. However, doing this can result in destructively large policy updates that lead to training instability. This is because small changes in the parameters of a policy π can cause a large shift in the steady state distribution of the Markov chain produced by π, and thus a large shift in the cumulative reward received. Intuitively, if an agent takes on average 100 steps to reach a reward state, a policy update that causes the agent to mess up in the first 10 steps might significantly hurt the agents ability to reach a reward state. Failing to reach a reward state means the agent loses a strong training signal that it previously had, leading to the training process collapsing.

Proximal Policy Optimization, proposed in [40], tries to avoid this by optimizing a surrogate function which removes the incentive for large policy updates. The objective function of vanilla

policy gradient can be written as

$$J_t^{PG}(\theta) = \mathbb{E}_t[\nabla_\theta \ln \pi_\theta(A_t|S_t)\Psi_t] \tag{2.16}$$

We define $r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$ as the probability ratio between the policy $\pi_{\theta_{old}}$ which was used to generate sample episodes and π_θ which is the new policy. PPO optimizes the surrogate objective function:

$$J_t^{CLIP}(\theta) = \mathbb{E}_t[\min\left(r_t(\theta)\Psi_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\Psi_t\right)] \tag{2.17}$$

where ϵ is a small value such as 0.2. The clip term clips the probability ratio to a bound defined by ϵ. This ensures that there is no incentive for large parameter updates that would move $r_t(\theta)$ out of the bound. The full objective takes the minimum of the clipped and unclipped objective, so it is bounded by the clipped objective. This clipped objective makes it possible to do multiple policy updates on the same sampled episode without collapsing the training process.

PPO uses GAE as Ψ_t that looks only at a segment $[0, T]$ of the episode. The value function $V_\theta(S_t)$ is learned by the same neural network that learns the policy. In addition to the clipped objective, PPO optimizes $J_t^{VF} = (V_\theta(S_t) - V_t^{targ})^2$ as a squared error on the learned value function and $S[\pi_\theta](S_t)$ as an entropy bonus to promote exploration as suggested in [46]. The final objective function is

$$J_t^{CLIP+VF+S}(\theta) = \mathbb{E}_t[J_t^{CLIP}(\theta) - c_1 J_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{2.18}$$

where c_1 and c_2 are coefficients.

Algorithm 1 shows the optimization algorithm used in PPO.

Algorithm 1 PPO algorithm

```
for iterationg=1,2,... do
    for actor=1,2,...,N do
        Run policy π_{θ_old} in environment for T timesteps
        Compute GAE advantages Â_1, ..., Â_T
    end for
    Optimize J wrt θ with K epochs and minibatch size M ≤ NT
    θ_old ← θ
end for
```

Decentralized Distributed PPO

Decentralized Distributed PPO (DD-PPO) is a method proposed by [45] to massively scale up training with PPO. In order to train a large policy network in a complex environment, a very large number of episodes are needed to achieve good performance. This, coupled with the fact that the policy network and the environment can both be computationally expensive to execute, means that distributing the workload is needed. DD-PPO is a synchronous, meaning all

the worker nodes work synchronously on rolling out episodes using the same policy. It is decentralized, meaning there is no central parameter server that handles optimization. Optimization is done independently by each worker node, and the gradient is shared in a decentralized way.

Given n worker nodes, at time step t each worker node has a copy of the same parameters θ_n^t. Each worker collects episodes of experience (also called rollouts) by executing policy $\pi_{\theta_n^t}$ on the environment and calculating the gradient $\nabla_\theta J^{PPO}$. Once all nodes are done, they share the gradients with a distributed mean AllReduce operation, and use the shared gradient to calculate θ_n^{t+1}. The process can be described functionally as

$$\theta_n^{t+1} = \text{ParamUpdate}\left(\theta_n^k, \text{AllReduce}\left(\nabla_\theta J^{PPO}(\theta_1^k), ..., \nabla_\theta J^{PPO}(\theta_N^k)\right)\right) \tag{2.19}$$

Because rolling out different episodes can take varying amounts of time, the performance could suffer significantly by having to wait for the slowest straggler to finish. To avoid this, DD-PPO will force workers to end early once a certain percent of works have finished, called the preemption threshold. Preemption will only occur once all workers have reached at least one quarter of the maximum steps to make sure all workers contribute to the learning process.

[45] demonstrates that their method scales close to linearly with over 100 GPU nodes running AI Habitat.

3 Hardware

3.1 Crazyflie Drone

The Crazyflie 2.1 is a micro-aerial quadrotor drone produced by Bitcraze. It measures only $92 \times 92 \times 29$ mm in size, and has a takeoff weight of 27 g. This small size makes it suitable for operating safely indoor around humans. Both the software and hardware are open source, which makes it a good platform for research. The Crazyflie has a radio transmitter which can communicate with a USB dongle that can be plugged into a computer. This allows the computer to receive telemetry from the drone, and send commands back to the drone in real time using a high level Python library.

The functionality of the Crazyflie can be expanded with expansion decks that can be mounted either on the top or the bottom of the drone. The decks can communicate with the main controller on the Crazyflie through a set of pins. Bitcraze provides many expansion decks that add features such as positioning, cameras, range sensors, wireless charging and more.

3.2 Positioning

For positioning, we rely on two different expansion decks for different scenarios:

- **Flow Deck** [2]: This deck is mounted on the bottom of the drone. It has a time of flight laser ranger sensor which can measure distances to the ground up to 4 meters. The high precision measurement can be used to maintain stable flight at a specific altitude relative to the floor. The deck also has an optical flow sensor pointed towards the ground. This sensor allows the drone to accurately track lateral movement by tracking the motion of features visible on the ground surface. As long as the ground has enough features for the optical flow algorithm to track, the movement tracking is fairly accurate. Integrating this movement estimation makes it possible to get a position estimate relative to the starting position of the drone. Over short flights, this global position can be fairly accurate, but over longer flights or with poor ground features, the position estimate can drift.

- **Lighthouse Deck** [4]: The Lighthouse positioning deck allows the Crazyflie to accurately calculate it full pose. It is mounted on top of the Crazyflie, and has 4 infrared receivers. It receives a signal from two or more base stations that send out infrared beacons. The base stations can be either the HTC Vive base stations (aka Lighthouse

Figure 3.1: Crazyflie 2.1 micro drone

V1) or the SteamVR Base Station 2.0 (aka Lighthouse V2). Although this positioning system requires external infrastructure, it is still useful to us to evaluate the agents object avoidance capability in a controlled environment.

3.3 Onboard Compute

Onboard compute is provided by the AI Deck v1.1 expansion deck provided by Bitcraze [1]. This deck has a GAP8 parallel ultra low power processor, a low power 320×320 grayscale camera (Himax HM01B0), and a WiFi chip which can be used to send JPEG images to a connected computer. It has 64 MB of HyperFlash and 8 MB of HyberRAM memory.

The GAP8 chip is a RISC-V instruction set processor developed by GreenWaves Technologies [3]. It has one main core for controlling the chip and 8 accelerator cores which can be used to execute vectorized and parallel algorithms. It has a 512 KB L2 memory which is accessible by all of the cores and a 64 KB L1 memory accessible by the 8 accelerator cores. To reduce energy consumption, the chip does not have an automatic data cache system between the levels of memory. Instead, memory must be moved explicitly between the levels.

4 Problem Setup

The problem that we want to solve is Point Goal Navigation. The agent is initiated in a random position in a 3D indoor environment. The environment can be either a real indoor environment or a simulated one. The agent is then give a goal position which is defined relative to the agents current position. At each step, the agent will receive observations from the environment, and must take an action that will bring it closer to the goal position. The agent succeeds if it calls the STOP action within a certain radius of the goal position.

4.1 Formal Definition

We can describe the synchronous interaction between the agent and the environment as a Markov Decision Process (MDP). Because the agent only receives observations from the environment instead of having perfect knowledge about the current state, we will formalize the problem as a Partially Observable Markov Decision Process (POMDP) [22] defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, R(s,a), T(s'|s,a), P(o|s))$ where $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{O}$ are state, action and observation spaces respectively. $R(s,a)$ is the reward function, $T(s'|s,a)$ is the transition function, and $P(o|s)$ is the observation probability. For our purposes, a POMDP has all of the same properties as an MDP. The observations serve as an analog to the state that the agent is it.

For our setup, an $o \in \mathcal{O}$ is made up of the tuple $(\mathcal{I}, g, a_{prev})$ where $\mathcal{I} \in \mathbb{R}^{256\times256\times1}$ is a grayscale image, $g \in \mathbb{R}^2$ is relative distance and angle of the goal position from the agents current position, and a_{prev} is the probability distribution of the actions taken in the previous step. $\mathcal{A}$ is the set of actions {MOVE_FORWARD, TURN_LEFT, TURN_RIGHT, STOP}

4.2 Evaluation Metrics

We use several metrics to evaluate the success of the agent on the task:

- **Success percent**: Given N test episodes, the percent of episodes where the agent succeeds in reaching the goal position.

- **Success weighted by Path Length** (SPL) [5]: Given N test episodes, let S_i be the binary success indicator of episode i, p_i be the distance traveled by the agent and l_i be

the shortest geodesic distance from the start to the goal, SPL is defined as:

$$\frac{1}{N}\sum_{i=1}^{N} S_i \frac{l_i}{\max(p_i, l_i)} \tag{4.1}$$

An SPL of 1 would indicate the agent is performing optimally, while 0 would indicate the agent never succeeds.

- **Average Collisions**: The average number of collisions over N test episodes.

5 Training in Simulation

Humans are able to learn new complex tasks in the real world with very little trial and error. We can learn by receiving and understanding natural language instructions, or by imitating the actions of other humans. We know from many years of experience how to avoid dangerous situations while learning new tasks. These capabilities are still not well understood from an AI standpoint. Our current reinforcement learning systems require millions of episodes of experience to be able to master a new task, and thus it is infeasible to learn new tasks in the real world. Instead, we train our agent in a 3D simulator. This allows the agent to quickly collect millions of episodes of experience, without the need for any real world hardware.

AI Habitat is a photorealistic 3D simulator created primarily by Facebook AI [26]. It is designed from the ground up to be highly efficient, reaching up to 10,000 frames per second on a single GPU. This high efficiency makes it useful for training AI agents using reinforcement learning algorithms, since the agent can accumulate a large amount of experience in a short amount of time. AI Habitat uses scans of real indoor environments as source data. It's data agnostic design allows it to use multiple different datasets of 3D environments.

AI Habitat provides an API on top of the simulation which makes it possible to define new tasks and agents. Some common task definitions are provided, such as point navigation (navigate to a certain point), object navigation (eg, navigate to an instance of a chair), and image navigation (navigate to where the image was taken). The agent can be configured with various different sensors, such as cameras, depth sensors, positioning sensors, semantic sensors, etc.

5.1 Datasets

Table 5.1 compares the various datasets available to use in AI Habitat for navigation tasks. Each dataset is made up of a set of scenes. Each scene is a 3D scan of a real world indoor location, and is made up of the visual appearance and a navigation mesh which can be used to calculate the navegable areas withing the scene. We consider three datasets: Matterport3D (MP3D) [9], Gibson [48] and Habitat-Matterport 3D (HM3D) [34]. For the Gibson dataset, we also distinguish between two subsets: Gibson 4+ and Gibson 0+. The number refers to a rating between 0 and 5 given to each scene by [26] which indicates the quality of the scene. Figure 5.1 explains the rating system and gives examples of each rating.

For our experiments we use Gibson 0+ and Gibson 4+. Because our agent uses visual input only, it is very important that the dataset we use provides enough visual variety for the visual encoder to learn a robust representation. Gibson 0+ provides a good visual variety due to its

Dataset	MP3D [9]	Gibson 0+ [48]	Gibson 4+ [48]	HM3D [34]
Number of scenes	90	571	106	1000
Floor area (m^2)	101.82k	217.99k	17.74k	365.42k
Navigable area (m^2)	30.22k	81.84k	7.18k	112.5k
Navigation Complexity	17.09	14.25	11.9	13.31
Visual Fidelity (FID ↓)	43.8	39.3	27.4	20.5

Table 5.1: Comparison datasets usable for navigation in AI habitat. Each scene is a scan of a real world indoor location. Navigation complexity is the maximum ratio of geodesic path length to euclidean distance between any two navigable locations in the scene. Visual fidelity is measured by the FID metric introduced by [20] and calculated by [42]

large number of scenes (571). However because we are also interested using the agent in the real world, we would like the simulation to look as similar to the real world as possible. Gibson 4+ provides a tradeoff between less visual variety and better visual reconstruction. Although HM3D has the largest number of scenes and the best visual fidelity, it is still relatively new and we were unable to get it to work with our environment configuration. We also do not use MP3D because it has a small number of scenes, low visual fidelity, and high navigation complexity which can make it harder for the agent to learn.

For each dataset, we use the publicly available train/val/test splits [26]. We use a set of episodes that were pre-generated by [26]. Each episode consists of a starting and ending position which have a navigable path between them. [26]

5.2 Environment Configuration

The agent is configured to be 40 cm tall and have a radius of 10 cm. It has two sensors: a grayscale 256×256 camera, and a point-goal sensor. The camera is positioned 40 cm off the ground, and has a 90° vertical and horizontal field of view. The contrast of the camera image is also randomly shifted to imitate the differences in contrast that a real camera can have. The point-goal sensor provides the relative location of the goal point to the agents current location. It consists of two values: the distance to the goal, and the relative angle to the goal.

A common behavior in many 3D simulations is that walking into a wall will cause the agent to slide along the wall. In many cases, this makes it easier for the agent to navigate in the environment because it can "cheat" and bypass an object simply by walking towards it and sliding along its contour. This behavior does not work in the real world, as walking into on object will result in a collision. [21] studied the predictivity of performance of navigation methods in AI habitat of performance in the real world. They found that disabling wall sliding significantly improves predictivity, and so we also disable wall sliding.

The actions are defined as follows:

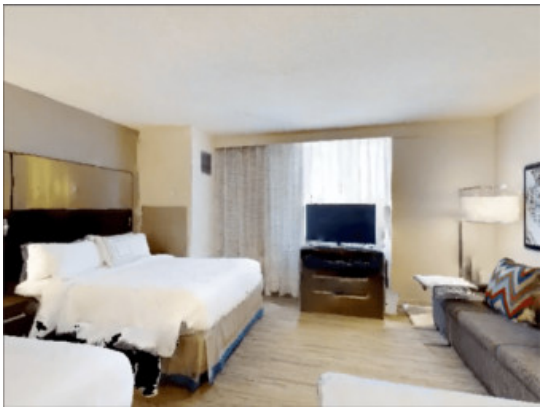
(a) 0: critical reconstruction artifacts, holes, or texture issues

(b) 1: big holes or significant texture issues and reconstruction artifacts

(c) 2: big holes or significant texture issues, but good reconstruction

(d) 3: small holes, some texture issues, good reconstruction

(e) 4: no holes, some texture issues, good reconstruction



(f) 5: no holes, uniform textures, good reconstruction

Figure 5.1: Rating scale created by [26] to create the quality of the 3D texture mesh reconstruction of scenes in the Gibson dataset. We use two subsets: 0+ and 4+.

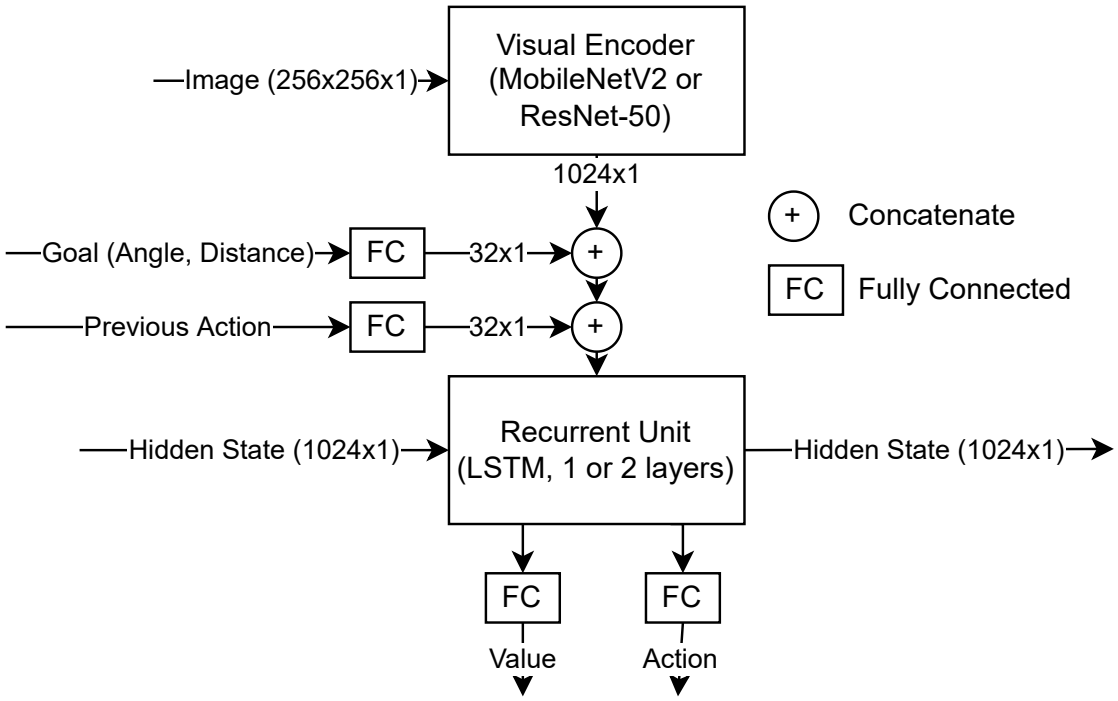


Figure 5.2: Diagram of the policy network. The dimensions of intermediate tensors may be slightly different depending on the configuration.

- MOVE_FORWARD: Move the agent forward by 0.25 m
- TURN_LEFT: Turn the agent left by 10°
- TURN_RIGHT: Turn the agent right by 10°

The actions are not executed deterministically. Some noise is added to the actual displacement by sampling from a gaussian that is derived from a noise model from a motion capture benchmark done on the PyRobot [31].

5.3 Agent

Our agent is defined by a policy neural network $\pi_\theta(a|o, h_{t-1})$ where θ are the parameters of the network. The structure of the network can be seen in Figure 5.2. The policy networks takes as an input the current observations from the environment and the hidden state from the previous step, and computes the probability distribution over the action space. The action chosen for the next step can either be sampled stochastically from the distribution, or deterministically take the action with the highest probability. For training, we sample the action from the distribution to promote exploration.

Network Component	Parameters
ResNet-50	7.07M
MobileNetV2	2.18M
LSTM-1	2.23M
LSTM-2	4.33M

Table 5.2: Comparison of the number of parameters for each network component. LSTM-x indicates an LSTM with x layers.

The network structure is base on the policy used in [45]. It is composed of two basic parts, the visual encoder and the recurrent unit.

The visual encoder is a convolutional neural network that takes the image observation as input and outputs a 1024×1 encoding. We use two different off-the-shelf networks, ResNet-50 [19] and MobileNetV2 [37]. Both of these networks were designed for object detection on the ImageNet dataset. The network structure is kept the same except for two changes. First, the last fully connected layers are removed because we are not doing object detection. Second, batch normalization layers are replaced with group normalization [47] layers. This is because the high correlation between the inputs of the network are not well suited for batch normalization. Group normalization normalizes over groups of channels in the convolution layers.

ResNet-50 is the encoder used by [45], so we include it as a baseline. The network is modified from its standard configuration by adding a 2×2 average pooling layer as the first layer, and halving the channels of all the convolution layers. Because we would like to execute the policy network on the GAP8 chip onboard the Crazyflie drone, we also consider MobileNetV2 as a visual encoder. MobileNetV2 was designed for edge computing devices with limited resources. Compared to other similar networks, it reduces the total number of parameters significantly, and also reduces the number of operations needed to do a full forward pass of the network. Table 5.2 compares the total number of parameters needed for each network component.

For the recurrent unit, we use and LSTM with 2 layers as a baseline since this is what [45] used. We also consider an LSTM with 1 layer in order to further reduce the number of parameters and operations of the network.

5.4 Training

We use DD-PPO to train the agent using Generalized Advantage Estimation [39]. The pre-emption threshold is 60%. The GAE is parameterized by the discount factor γ of 0.99 and λ of 0.95. The PPO loss function has a clip factor ϵ of 0.2, a value loss with a coefficient of 0.5, and an entropy bonus with a coefficient of 0.01. Each worker node simulates 14 different environments simultaneously, and unrolls each environment for up to 128 steps and then does 2 epochs of PPO optimization with a mini-batch size of 2. We use the Adam optimizer with a learning rate of $2.5e^{-4}$. We use 8 worker nodes for a total of 112 simultaneously simulated

Policy	DS	SPL ↑	Success (%)	Collision Avg	Collision Free (%)
ResNet50-LSTM2	0+	0.6829	97.89	13.00	38.84
ResNet50-LSTM2	4+	0.6838	95.87	12.00	50.40
MobileNetV2-LSTM1	0+	0.6723	95.77	16.31	41.75
MobileNetV2-LSTM2	0+	0.6780	96.47	14.00	42.15
ResNet50-LSTM2-CollFT	0+	0.5867	85.92	3.17	60.06

Table 5.3: Evaulation results on the 994 training episodes. CollFT refers to extra fine-tuning training done with an additional negative reward for collisions.

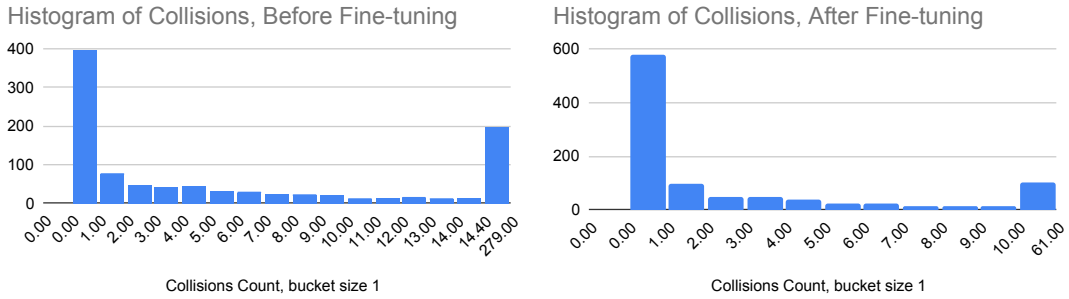


Figure 5.3: Collision histogram for the 994 test episodes before and after fine-tuning the ResNet50-LSTM2 agent with a negative reward for collisions.

environments. Each worker node is an NVIDIA DGX A100 with 40GB of memory, contained in a DGX node with 8 GPUs provided by the Lichtenberg High Performance Computer of TU Darmstadt. Each policy variant is trained for 150 million environment steps, taking on average around 16 hours.

The agent is given a reward of 2.5 for for getting within 30 cm of the goal point. The agent is also given a reward of $-\triangle_{dist} - 0.01$ for each step it takes. The -0.01 is a slack reward to penalize longer paths over shorter paths. $\triangle_{dist}$ is the change in the shortest geodesic distance to the goal point. This gives a reward to the agent for getting closer to the goal, even if it doesn't reach it. We also experiment with fine-tuning an already trained agent for 30 million steps with an additional reward of -0.5 each time it collides with an object.

5.5 Results

Table 5.3 shows the evaluation results of the various agent configuration. All of the variants achieve a high success rate of 95%, with variants using ResNet-50 and LSTM2 components performing sightly better. An SPL of around 0.68 indicates that the agents are not learning a perfectly optimal path. Figure 5.4 show three of the test episodes and the path the agent took. This partly because the shortest geodesic path hugs corners as closely as possible. With sliding disabled in the environment, the agent needs to avoid getting too close to walls to avoid colliding and getting stuck.

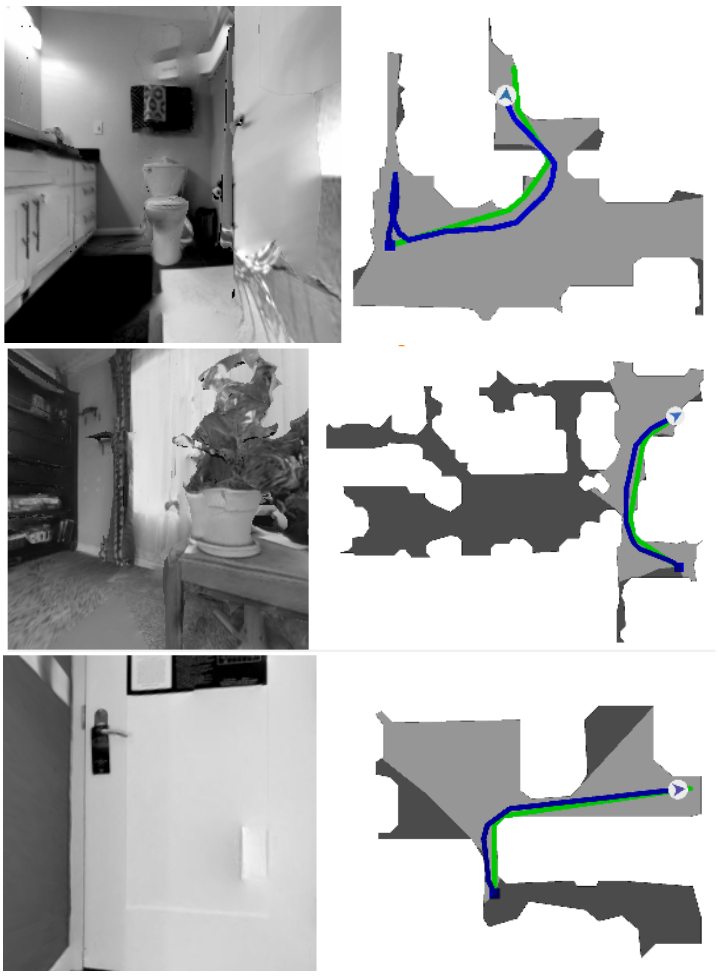


Figure 5.4: A selection of episodes from the evaluation set. The green line represents the shortest geodesic path, and the blue line represents the path taken by the agent with the ResNet50-LSTM2 policy. The gray area is the navigable area.

The average number of collisions is quite high, but this is because the distribution is heavily skewed as show in Figure 5.3. However, the percentage of episodes with no collisions is still quite low for a policy that we want to deploy on a quadcopter. Fine-tuning with a negative reward for collisions increases the percentage of collision free episodes from 38.84% to 60.06%, though it causes the success rate and SPL to drop.

6 Real World Experiments

We evaluate the performance of the agent in the real world in two different ways. First, we evaluate the agents ability to avoid objects in a lab setting with accurate positioning provided by the Lighthouse system. Secondly, we evaluate the agent outside of the lab setting using the Flow Deck as a positioning system. The goal of this experiment for a human to fly the drone some distance from the starting position, and then have the agent autonomously return to the starting position.

6.1 Agent Control of the Quadcopter

The agent can control the Crazyflie either by remote control or by on-board control. In the remote control mode, images taken from the camera on the Crazyflie are compressed to JPEG images by the GAP8 chip, and then streamed over WiFi to a stationary computer. Position estimates from the drone are streamed separately over the Crazyflie radio to a dongle connected to the same computer. The relative position between the Crazyflie and the goal position is calculated, and fed into the policy network along with the most recent image. The action taken is sampled from the action distribution produced by the policy network. The action is executed by either sending a position setpoint relative to the Crazyflies previous position, or by sending a velocity command that is executed for a certain amount of time. Position setpoints are more appropriate use with the Lighthouse positioning system since the positioning framework is more steady. For the Flow Deck, velocity commands are more appropriate since the tight control loop between the optical flow sensor and the Crazyflie can control the velocity accurately.

The agent's policy network can also be executed directly on the drone on the GAP8 chip. The smallest network variant that we considered, using MobileNetV2 and a one layer LSTM, has 5.46 million parameters. If these parameters are quantized to 8 bit integers, the network would take approximately 5.46 MB of memory to store. This is small enough to fit on the HyperRAM on the AI deck. C code that is optimized to execute on the GAP8 can be generated from the network graph using the NNTool provided by Greenwaves. The tool generates all of the necessary memory management operations which move network parameters and intermediate values between the different memory levels. Unfortunately, due to time constraints we were not able to implement this yet, so all of the real world experiments are done with remote control.

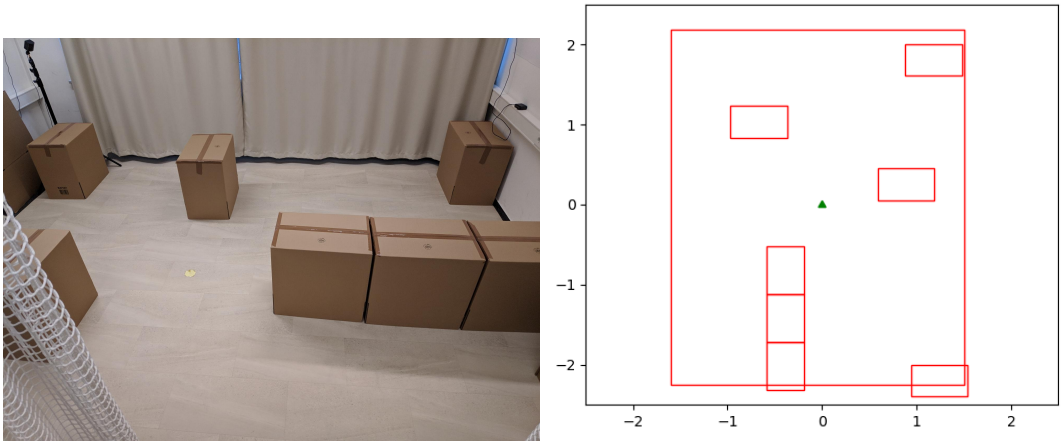


Figure 6.1: Layout of the obstacle course.

6.2 Obstacle Avoidance

With this experiment, the goal is to show that the agents can avoid obstacles using vision only. An accurate position is provided by the Lighthouse system, with two external beacons in oposite corners of the room. Cardboard boxes are used as obstacles. Seven $60 \times 40 \times 40$ cm boxes are placed in a room that is measures around 3×5 meters. The location of each box is mapped. The layout of the obstacle course can be seen in Figure 6.1. The Crazyflie is then placed at a starting location and tasked with navigating to the goal position. We define six different test episodes withing the obstacle course.

Figure 6.2 shows the resulting trajectories over the six episodes. In five out of the six cases, the Crazyflie successfully reaches the goal, although not in the most efficient way possible. In one of the episodes, the Crazyflie flies directly into an obstacle. On inspection, it appears that this was caused by the agent operating on a slightly outdated image which showed the obstacle as being farther away than it was. The image used by the agent can be up to 0.4 seconds old. Figure 6.3 shows an example view of the obstacles from the Crazyflie's camera.

Unfortunately, these results can only qualitatively demonstrate that the policy learned in training does transfer into the real world. To show the effectiveness quantitatively would require a larger sample size, which we were unable to collect due to time constraints. Only the ResNet50-LSTM2 policy was test.

6.3 Return to Home

The second real world experiment performed is return to home. In this experiment, we use the Flow Deck as the positioning system. The Crazyflie is placed in a starting position somewhere in an indoor environment. The starting position is given the coordinate of $(0, 0)$. The Flow Deck tracks the relative position from this point by integrating the lateral movement. The

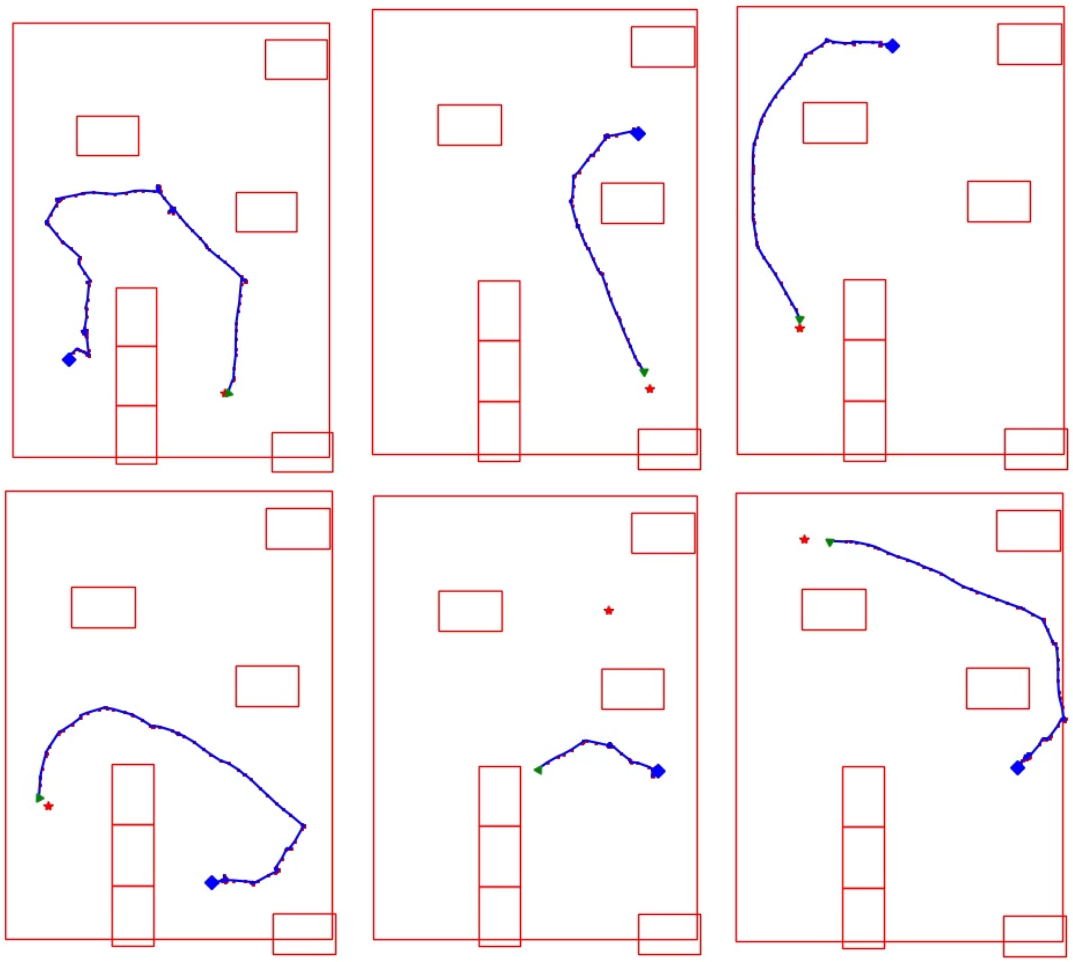


Figure 6.2: Resulting trajectories on 6 test episodes using the ResNet50-LSTM2 policy network. The red star shows the goal position. The Crazyflie reaches the goal in 5 of the episodes, and collides with an obstacle in one of them.

Figure 6.3: View of the obstacles from the Crazyflie's camera.

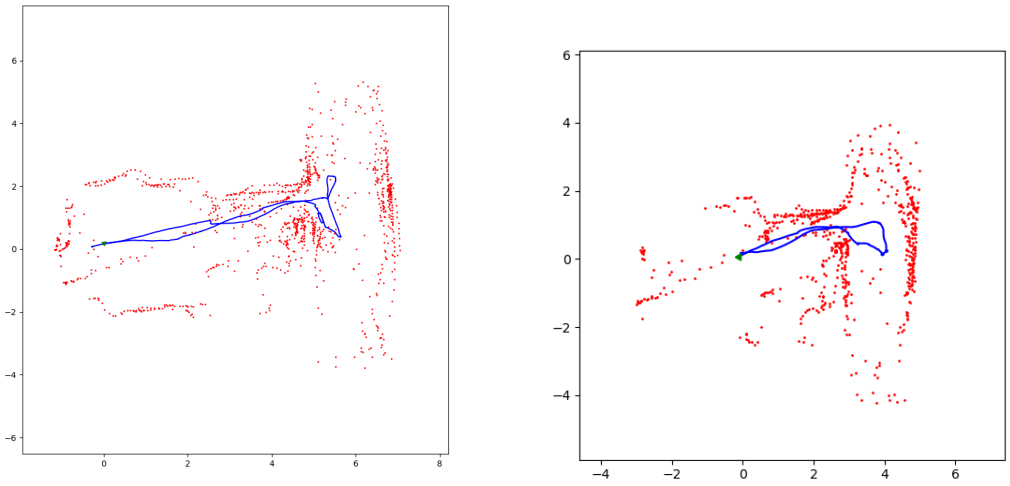


Figure 6.4: Resulting map produced by the return to home experiment. The red dots are generated from the Multiranger deck which has four laser rangers pointed forward, backward, left and right. The Crazyflie is flow out into a hallway by a human, and then flow back into the room by the agent.

Crazyflie is then flown manually by a human some distance away from the starting position. The agent is then tasked with autonomously flying the Crazyflie back to the starting position.

Figure 6.4 shows the resulting map generate by two runs of the experiment. The red dots are produced by the Multiranger Deck from Bitcraze, which has four lazer range sensor pointed forwards, backwards, left and right. This experiment demonstrates how the agent can operate fully autonomously in an indoor environment without using any external infrastructure for positioning. The main limitation of this method is that the floor must have sufficient texture for the optical flow sensor to operate well throughout the entire flight. Over long fights, even with good ground texture, the positioning estimate can drift due to accumulation of error in the integration of the lateral movement.

7 Conclusion and Outlook

In this thesis, we explored the potential for neural networks that are small enough to execute on a nano-quadcopter drone to do visual point goal navigation. We demonstrated that such a network, when trained in a simulation and given only very limited information from a monocular low resolution grayscale camera, can perform this task in the real world. This is exciting because the completely decentralized execution and control would allow for the operation of a swarm of nano-quadcopters. Each drone in the swarm could operate independently of any infrastructure, so the swarm could operate completely autonomously in an unknown environment. This could be useful for quickly searching a large indoor space.

This also opens the door to exploring more complex navigation tasks on nano-quadcopters that rely on visual navigation. For example, the agent could be tasked with exploring the environment and seeing as much as it can of the environment around the starting position in the shortest amount of time.

Although point goal navigation is a useful task for evaluating the effectiveness of visual navigation, it has some limitations that make it less useful in practice. The agent needs to be able to accurately track its position over time to know that it is getting closer to the goal. The Flow Deck does this well over short flights, but doing this with no external infrastructure and over longer flights is not easy. If the drone loses its position at any time, or the position estimate drifts too far, it cannot recover. To overcome this issue, we could define a task that does not rely on exact positioning over a long time horizon. For instance, the agent could be tasked with navigating to an image of a particular location in the environment. The agent could also be given some time to explore the environment and generate a sequence of memory vectors that encode semantic information at various locations, and the relative location of nearby memories. It could then be commanded to navigate to the location of any of it previous memories.

Finally, it is also important to note the limitations of using reinforcement learning for this task. All of the learning process must take place inside of a simulation. If the real world deviates too far from what was present in the simulation, then the method may not perform well. Simulations are expensive and time consuming to create, and their complexity often pales in comparison to the real world. Additionally, training an agent on a new task takes a significant amount of computational resources and time. These are fundamental limitation on how reinforcement learning can be use in the real world, and it is an open question how they can be overcome.

Bibliography

[1] Ai-deck 1.1. https://store.bitcraze.io/products/ai-deck-1-1?variant=32485907890263. Accessed: 202-04-21.

[2] Flow deck v2. https://www.bitcraze.io/products/flow-deck-v2/. Accessed: 202-04-21.

[3] Gap8 manual. https://greenwaves-technologies.com/manuals/BUILD/HOME/html/index.html. Accessed: 202-04-21.

[4] Lighthouse positioning deck. https://www.bitcraze.io/products/lighthouse-positioning-deck/. Accessed: 202-04-21.

[5] Peter Anderson, Angel Chang, Devendra Singh Chaplot, Alexey Dosovitskiy, Saurabh Gupta, Vladlen Koltun, Jana Kosecka, Jitendra Malik, Roozbeh Mottaghi, Manolis Savva, et al. On evaluation of embodied navigation agents. *arXiv preprint arXiv:1807.06757*, 2018.

[6] Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid, Stephen Gould, and Anton Van Den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3674–3683, 2018.

[7] Francisco Bonin-Font, Alberto Ortiz, and Gabriel Oliver. Visual navigation for mobile robots: A survey. *Journal of intelligent and robotic systems*, 53(3):263–296, 2008.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[9] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niebner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3d: Learning from rgb-d data in indoor environments. In *2017 International Conference on 3D Vision (3DV)*, pages 667–676. IEEE Computer Society, 2017.

[10] Devendra Singh Chaplot, Dhiraj Gandhi, Saurabh Gupta, Abhinav Gupta, and Ruslan Salakhutdinov. Learning to explore using active neural slam. In *International Conference on Learning Representations*, 2019.

[11] Devendra Singh Chaplot, Ruslan Salakhutdinov, Abhinav Gupta, and Saurabh Gupta. Neural topological slam for visual navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12875–12884, 2020.

[12] Tao Chen, Saurabh Gupta, and Abhinav Gupta. Learning exploration policies for navigation. In *International Conference on Learning Representations*, 2018.

[13] Guilherme N DeSouza and Avinash C Kak. Vision for mobile robot navigation: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 24(2):237–267, 2002.

[14] Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[15] Kuan Fang, Alexander Toshev, Li Fei-Fei, and Silvio Savarese. Scene memory transformer for embodied agents in long-horizon tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 538–547, 2019.

[16] Dhiraj Gandhi, Lerrel Pinto, and Abhinav Gupta. Learning to fly by crashing. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3948–3955. IEEE, 2017.

[17] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2616–2625, 2017.

[18] R Hartley and A Zisserman. Multiple view geometry in computer vision (cambridge university, 2003). *C1 C3*, 2, 2013.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[21] Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. Sim2real predictivity: Does evaluation in simulation predict real-world performance? *IEEE Robotics and Automation Letters*, 5(4):6670–6677, 2020.

[22] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[23] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic

roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[25] Alex X Lee, Sergey Levine, and Pieter Abbeel. Learning visual servoing with deep features and fitted q-iteration. *arXiv preprint arXiv:1703.11000*, 2017.

[26] Manolis Savva*, Abhishek Kadian*, Oleksandr Maksymets*, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[27] KN McGuire, Christophe De Wagter, Karl Tuyls, HJ Kappen, and Guido CHE de Croon. Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment. *Science Robotics*, 4(35):eaaw9710, 2019.

[28] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[30] Arsalan Mousavian, Alexander Toshev, Marek Fišer, Jana Košecká, Ayzaan Wahid, and James Davidson. Visual representations for semantic target driven navigation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8846–8852. IEEE, 2019.

[31] Adithyavairavan Murali, Tao Chen, Kalyan Vasudev Alwala, Dhiraj Gandhi, Lerrel Pinto, Saurabh Gupta, and Abhinav Gupta. Pyrobot: An open-source robotics framework for research and benchmarking. *arXiv preprint arXiv:1906.08236*, 2019.

[32] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini. A 64mw dnn-based visual navigation engine for autonomous nano-drones. *IEEE Internet of Things Journal*, 2019.

[33] Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*, 2017.

[34] Santhosh Kumar Ramakrishnan, Aaron Gokaslan, Erik Wijmans, Oleksandr Maksymets, Alexander Clegg, John M Turner, Eric Undersander, Wojciech Galuba, Andrew Westbury, Angel X Chang, et al. Habitat-matterport 3d dataset (hm3d): 1000 large-scale 3d envi-

ronments for embodied ai. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[35] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[37] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[38] Nikolay Savinov, Alexey Dosovitskiy, and Vladlen Koltun. Semi-parametric topological memory for navigation. In *International Conference on Learning Representations*, 2018.

[39] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[41] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[42] Andrew Szot, Alex Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Chaplot, Oleksandr Maksymets, Aaron Gokaslan, Vladimir Vondrus, Sameer Dharur, Franziska Meier, Wojciech Galuba, Angel Chang, Zsolt Kira, Vladlen Koltun, Jitendra Malik, Manolis Savva, and Dhruv Batra. Habitat 2.0: Training home assistants to rearrange their habitat. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[43] S Thrun, W Burgard, and D Fox. Probabilistic robotics. ma, 2005.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[45] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *International Conference on Learning Representations (ICLR)*, 2020.

[46] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[47] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[48] Fei Xia, Amir R Zamir, Zhiyang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson env: Real-world perception for embodied agents. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9068–9079, 2018.