TECHNISCHE UNIVERSITÄT BERLIN

Bachelor Thesis

Autonomous, indoor operation of Small Unmanned Aircraft Systems using Reinforcement Learning and LSTM

Max Mühlefeldt

Matriculation number 382737

Technische Universität Berlin

Faculty V – Mechanical Engineering and Transport Systems

Institute of Aeronautics and Astronautics

Chair of Flight Guidance and Air Traffic

Supervisors

Prof. Dr. ir. Maarten Uijt de Haag

Prof. Dr. Wolfgang Hönig

Mats Martens, M.Sc.

Berlin 2024

Statement of Independent Work

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin, August 18, 2024

[illegible]

Max Mühlefeldt

Zusammenfassung

Der autonome Einsatz von small Unmanned Aerial Systems in unzugänglichen Innenräumen ist aufgrund ihrer Größe naheliegend. Die geringe Größe von small Unmanned Aerial Systems begrenzt sowohl die Flugzeit als auch die Kapazität, Sensoren zu transportieren. Zudem steht in Innenräumen regelmäßig kein Global Navigation Satellite System zur Positionsbestimmung zur Verfügung.

Diese Arbeit entwickelt einen Ansatz, Innenraumnavigation mittels Reinforcement Learning umzusetzen. Mit der Unity Engine und dem ML-Agents Paket wird eine 3D-Simulation aufgebaut, um einen Agenten in dynamischen Innenräumen zu trainieren. Der Agent ist nach einem small Unmanned Aerial System modelliert und verfügt über eine eingeschränkte Sensorausstattung. Hauptsächlich verfügt der Agent über Entfernungssensoren. Das eingesetzte Neuronale Netzwerk wird um Long Short-Term Memory erweitert, um dem Netzwerk ein Gedächtnis zu ermöglichen. Ein robustes Belohnungssystem wird entwickelt, um die Aufgabe des Agenten, die kollisionsfreie Navigation zu einem Zielobjekt, zu kodieren.

Die Navigationsaufgabe wird erfolgreich durch den Agenten absolviert, auch wenn dieser nur über eine geringe Anzahl von Sensoren verfügt. Weiterhin wird untersucht, welchen Einfluss die Anzahl der Sensoren auf die Leistung des Agenten hat. Die Ergebnisse lassen den Schluss zu, dass eine größere Sensoranzahl dem Training des Agenten hilft, gemessen an verschiedenen Metriken. Jedoch ist die Erhöhung der Sensoranzahl nicht in jedem Fall uneingeschränkt empfehlenswert. Die Ergebnisse werden diskutiert und Ansätze für zukünftige Arbeiten in diesem Feld dargelegt.

Autonomous, indoor operation of Small Unmanned Aircraft Systems using Reinforcement Learning and LSTM

Max Mühlefeldt*
Technische Universität Berlin, Germany

This thesis investigates the application of Reinforcement Learning for the training of agents modelled after small Unmanned Aerial Systems to navigate indoor environments using limited sensor suites. The implementation of simulated environments and agents using Unity and the Machine Learning package ML-Agents is examined. The agent's Neural Network is augmented with memory capability through the incorporation of Long Short-Term Memory. Key challenges addressed include the collision-free navigation and the selection of an appropriate reward system. Furthermore, the influence of the sensor count onboard the agent is studied in regard to the agent's performance. The findings demonstrate the feasibility of achieving effective navigation, even in randomised environments without advanced sensor suites.

I. Nomenclature

a	=	specific action
$\mathcal{A}$	=	action space
A_t	=	action by the agent at time step t
$\hat{A}_t$	=	estimator of the advantage function at time step t
b	=	bias
c_j	=	LSTM memory cell
d	=	number of dimension
dp	=	ML-Agents decision period
$\mathbb{E}$	=	expected value
$\mathbb{E}_t$	=	expected value at time step t
$\mathbb{E}_\pi$	=	expected value given a policy π
f_a	=	activation function
f_t	=	forget gates of the entire LSTM layer at time step t
f_{jt}	=	forget gate of the LSTM cell c_j at time step t
g_t	=	input gates of the entire LSTM layer at time step t
G_t	=	expected reward
g_{jt}	=	input gate of the LSTM cell c_j at time step t
H	=	number of range sensors onboard the agent
i_t	=	input of the entire LSTM layer at time step t
i_{jt}	=	input of the LSTM cell c_j at time step t
L^{CLIP}	=	PPO objective function
o_t	=	output gates of the entire LSTM layer at time step t
o_{jt}	=	output gate of the LSTM cell c_j at time step t
p	=	probability
$\Pr$	=	probability
q_π	=	action-value function
r	=	specific reward
$\mathcal{R}$	=	reward space
r_t	=	probability ration between current and old policy
R_t	=	reward at time step t
R_{goal}	=	reward component, goal reached

*max.muehlefeldt@tu-berlin.de

R_{col}	=	reward component, collision
R_{door}	=	reward component, door passage
R_{action}	=	reward component, action penalty
R_{limit}	=	expected reward limit for the environment
s, s'	=	specific states
$\mathcal{S}$	=	state space
s_t	=	internal state of the entire LSTM layer at time step t
S_t	=	state at time step t
s_{jt}	=	internal state of the LSTM cell c_j at time step t
t	=	discrete time step
T	=	final time step
v_*	=	optimal value function
v_x	=	agent velocity in x axis
v_z	=	agent velocity in z axis
v_π	=	value function
$\overline{W}$	=	weight matrix
$\overline{X}$	=	input vector
$\hat{y}$	=	calculated prediction of neuron
$\hat{y}_t$	=	output of the entire LSTM layer at time step t
$\hat{y}_{jt}$	=	output of the LSTM cell c_j at time step t
γ	=	discount rate
δ	=	range measurements
δ_{goal}	=	distance threshold to consider the target reached
ϵ	=	clipping hyperparameter of the PPO algorithm
θ	=	parameter of policy π
π, π'	=	policies
ρ_y	=	rotation around y axis
σ	=	sigmoid function

II. Introduction

A. Motivation

Small Unmanned Aerial Systems (sUAS) are, due to their size and ability to overcome obstacles, well suited for indoor operations. Search and rescue operations in unknown, hazardous environments is a possible deployment scenario for sUAS [1, p. 1036]. These operational scenarios pose different challenges, such as the lack of *Global Navigation Satellite Systems* (GNSS) or the limited *maximum take-off mass* (MTOM) of sUAS, which allows only for restricted sensor suites and energy storage, thereby enabling only short-duration flights. In these scenarios, operator intervention is often neither possible nor desirable, making autonomous operations of sUAS necessary.

Navigational tasks are traditionally a two-stage process of exploration and exploitation. During the exploration a model of the surrounding world is created from sensor inputs and if available pre-compiled maps. The world model is used during the exploitation phase to reach the navigation objective. SUAS rarely have access to pre-compiled map data of the operating environment and must rely solely on their onboard sensor suite [2, p. 47].

Reinforcement Learning (RL) presents a *Machine Learning* (ML) approach leveraging goal-directed learning from the interaction of a decision-making agent with its environment [3, pp. 1–5]. Compared to the traditional navigation approach the two separate stages of navigation are replaced by leveraging a reward function describing the navigational task [4, p. 1]. To perform a navigational task memory capability must be available [5, p. 2]. *Long Short-Term Memory* (LSTM) allows the storage of recent input events within *Neural Networks* (NN) and thus provides the desired memory [6, p. 1].

The readily available software Unity supports a general-purpose engine on different hardware and operating system platforms. Unity provides a suitable engine to create 2D and 3D environments and corresponding agents in an RL context without limiting the type of environment [7, pp. 8–9]. In addition to creating environments, Unity offers an integrated RL framework to train agents using various algorithms and features with the *Unity ML-Agents Toolkit* (ML-Agents) [7, p. 11].

B. Goal and Structure

The goal of this thesis is to demonstrate a successful application of RL to achieve autonomous indoor operation of an sUAS. The following points outline the brief of this thesis.

1) *Simulated environment setup*: A simulated indoor environment is constructed to mimic real-world conditions using the Unity engine. The overall floor plan of the environment remains fixed, with a randomised inner wall. The agent represents an sUAS equipped with Light Detection and Ranging (LiDAR) range sensors.
2) *RL*: The RL agent is trained within the simulated environment to navigate towards a 3D-modelled target while avoiding collisions. An NN with LSTM is utilised.
3) *Sensor input analysis*: The agent's input is limited due to the sensor capacity constraints of an actual sUAS. The impact on NN performance is analysed as the number of sensors increases.

After presenting related work in this field, the theoretical background to RL is laid out, followed by the presentation of the used method and the discussion of the results.

C. Related Work

The subject of autonomous navigation of sUAS using RL and LSTM has gained a great deal of attention in the literature. Mirowski et al. [5] used RL in conjunction with LSTM to train an agent to navigate a dynamic 3D maze. The agent observations consisted of visual inputs and the agents velocity. The agent showed good performance in navigational tasks compared against NN without LSTM and simple feed forward networks. Auxiliary tasks were used to increase learning feedback to the agent beside the primary goal feedback. Depth prediction and loop closures were incorporated as auxiliary tasks. The experiments were conducted using the DeepMind Lab software which is specifically designed for ML applications with limited options for observations and actions by the agent [8]. The ML approach proofed not suitable to determine shortest paths [5, p. 9].

Dhiman et al. [4] evaluated the performance of RL using a suite of navigational tasks by using random maps. The agent navigates a dynamic maze only by visual inputs. Based on the work of Mirowski et al. [5] a NN featuring LSTM was implemented [4, pp. 2–3]. Again DeepMind Lab was used to create the 3D environment with random layouts. Dhiman et al. [4] found the navigational performance on previously unseen maps to be trailing behind traditional navigational approaches [4, p. 6].

Wang et al. [9] utilised very sparse reward feedback from the environment to train an agent modelled after sUAS. The agents observation capabilities were limited to range finders and the relative position to the target using GNSS [9, p. 6182]. Thus this represents a comparable sUAS to the one used in this thesis. An algorithm complementing the RL training was proposed to provide non-expert prior policies and dynamic training goals to handle the sparse reward system [9, p. 6183].

Youn et al. [10] adopted a classic algorithm for collision-free path finding for sUAS in GNSS denied environments [10, p. 958–959]. Distinct mapping and path finding phases are required to perform the navigational task [10, p. 962]. Their experimental sUAS featured a wide ranging sensor suite of 2D LiDAR, 3D camera, accelerometers, gyroscope and an altimeter [10, p. 960]. Simulation of the environment was used in conjunction with real-life tests and verified the approach without ML.

Hodge et al. [11] investigated aerial drone navigation in grid based environments. The drone received input regarding the contents of the adjacent grid spaces and the distance to the target along two axes. The agents actions were limited to movements to the next grid cells along two axes without rotation capability [11, p. 2020]. RL with the addition of LSTM was used. Additionally Hodge et al. [11] provided knowledge learned from previously achieved simpler tasks by using curriculum learning. The dynamic 3D environment was created in Unity.

This thesis falls within this group of sources but focuses on the available sensor capability of the sUAS. Unity, a general-purpose simulation engine, is used to create an indoor environment. This approach contrasts with [5] and [4], but it aligns with their use of ML and NN with LSTM. While [11] is more similar to this thesis, it is considered that the limitations of a grid-based navigation system are too restrictive to allow for realistic sUAS movement through the environment. The proposed sensor capability primarily utilises LiDAR sensors, similar to [10], but without additional sensing capabilities.

III. Theoretical Background

This section provides the theoretical background necessary to utilise ML for training an agent to perform a navigation task. ML algorithms can learn to perform tasks based on experiences from a dataset [12, pp. 97–105]. Specifically, NNs featuring LSTM are introduced. Additionally, RL and associated learning algorithms are described.

A. Neural Networks

Artificial Neural Networks or more commonly known as *Neural Networks* (NN) are algorithms with their origin in concepts derived from neuroscience [13, p. 13]. A NN is a directed acyclic graph with edges parameterised by individual weights. The weights affect the function calculated in the nodes. Based on neuroscience the nodes are called *neurons*. Within each node, excluding fixed input nodes, a function is computed based on the parameterised input. Overall, the NN performs a cascading function calculation through the nodes. NNs can compute almost any function by setting the weights correctly [14, p. 4], [15, p. 225].

In its simplest form, NNs are known as *perceptron* and feature a single input layer followed by an output node as shown in Figure 1. Given an input vector $\overline{X} = \{x_1, x_2, \ldots, x_d\}$ with dimensions d and a single output $\hat{y}$. The weight vector $\overline{W} = \{w_1, w_2, \ldots, w_d\}$ also has d dimensions [14, p. 5]. The calculated prediction, i.e. the output, takes the form

$$\hat{y} = f_a f_{\overline{W}}(\overline{X})) = f_a \left(\sum_{n=1}^{d} w_n x_n + b \right). \tag{1}$$

With f_a being the non-linear activation function of the combination of the input and weights by the sum $\overline{W}^T \cdot \overline{X}$ [14, p. 6]. Often the input to a node is complemented by the *bias b* with the aim to provide a variable to adjust for thresholds within a given dataset by providing a constant input modified by an individual weight [14, p. 8].

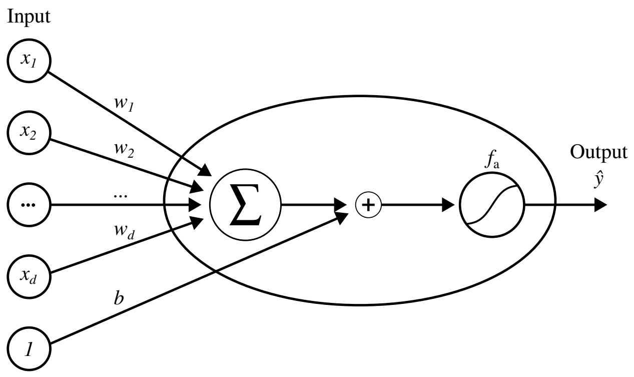

Fig. 1 Layout of a perceptron with bias and sigmoid activation function. Figure by the author, adapted from [14, Fig. 1.3].

The choice of the activation function f_a is an important design decision of NNs [14, p. 10]. The nature of the dataset and distribution of target variables influence the choice of the activation function [15, pp. 227–228]. Typical candidates for activation functions are the sigmoid σ and tanh functions [14, pp. 11–12]. While σ computes a value in $]0, 1[$ tanh calculates a value in $]-1, 1[$ which enables the interpretation of the result as probabilities [14, p. 11], [16, p. 87], [17]:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{3}$$

Multi-layer NNs feature nodes structured in layers. Input neurons in the first layer are followed by any number of hidden layers and output neurons in the last layer complete the multi-layer NN. The input is fed in forward direction from the input through the hidden layers to the output layer. Usually all nodes in a layer are connected with all nodes in the successor layer. These networks, illustrated on the left side of Figure 2, are referred to as *feed-forward neural networks* [14, pp. 13–14].

Multi-layer feed-forward NNs have the ability to learn functions by setting the weights of the edges. This training of the network associates inputs with outputs of the NNs. By utilising nonpolynomial activation functions, like the sigmoid function, the network can approximate any function provided a bias is fed to the nodes [18, pp. 863–864].

B. Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a group of NNs that is particularly well suited for sequential input data such as a sequence of data representing individual states of an environment [12, p. 367]. In contrast to feed-forward networks cyclic connections are the core feature of RNNs [19, p. 18]. Elman [20] laid out the concept of using the influence of prior internal states of the network to affect the current internal representation. A recurrent network structure is employed to share parameters across parts of the input sequence. In theory, an RNN can map the entire history of previous inputs from the sequence to each output. Recurrency provides dynamic memory for the network [20, p. 207]. Inputs can persist in the network and influence any output [19, pp. 18–19]. Figure 2 shows a simple RNN with output at each time step. The hidden units feature recurrent connections as self-loops and between each other, though other forms of recurrent connections are feasible [19, p. 18]. The RNN accesses the same weights across multiple time steps, and the output is therefore influenced by the previous output. RNNs usually operate only on a subset of the entire sequence, called minibatches [12, pp. 367–368].

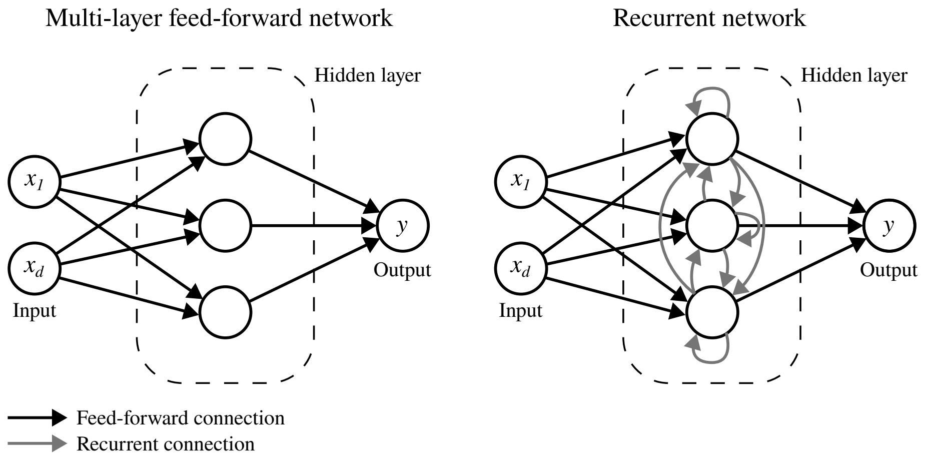

Fig. 2 Layouts of a Multi-layer feed-forward NN and a RNN. Biases and weights of the edges omitted. Figure by the author, adapted from [19, Fig. 3.3], [14, Fig. 1.9, 8.2].

RNNs feature very deep computational graphs due to performing repeated operations at each time step. When applied to long input time sequences, vanishing or exploding gradients may occur. In case of vanishing gradients, the direction of parameter change becomes difficult to determine. Exploding gradients cause unstable learning progress [12, p. 286], [19, p. 31]. Especially long-term interactions can cause exponentially small weights and cause gradient issues [12, p. 396]. Strategies to tackle these issues when applying RNNs to long-term dependencies include, for instance, models operating at multiple time scales. Part of a model may manage small details in fine grained time scales while another part works on the broader time scale, storing past information for later use in the RNN [12, p. 402].

C. Long Short-Term Memory

Another strategy to deal with the aforementioned issues involves the introduction of gates to RNNs [12, p. 402]. The LSTM model developed by Hochreiter and Schmidhuber [6] introduced *Memory cells* c_j as replacement or addition to the typical nodes in the hidden layers of RNNs [6, pp. 6–7].

A memory cell is shown in Figure 3. The input feature i_{jt} is calculated by a typical unit with an activation function as described by Equation 1. The input gate g_{jt} allows to protect the memory state in c_j from irrelevant inputs. The input is fed into a linear unit with a recurrent self-connection. This unit represents the internal state s_{jt} of the memory cell [6, p. 7]. Gers et al. [21] introduced the forget gate f_{jt} to control the recurrent connection of s_{jt} to prevent the gradient of c_j to vanish or the output of c_j to gradually equal the activation function of the output gate, which was a common problem of LSTM memory cells without forget gates [21, p. 851]. The forget gate f_{jt} enables individual resets of irrelevant memory in LSTM networks [21, p. 855]. The output gate o_{jt} prevents the flow of irrelevant memory from the cell to further downstream memory cells or units in the network [6, p. 7].

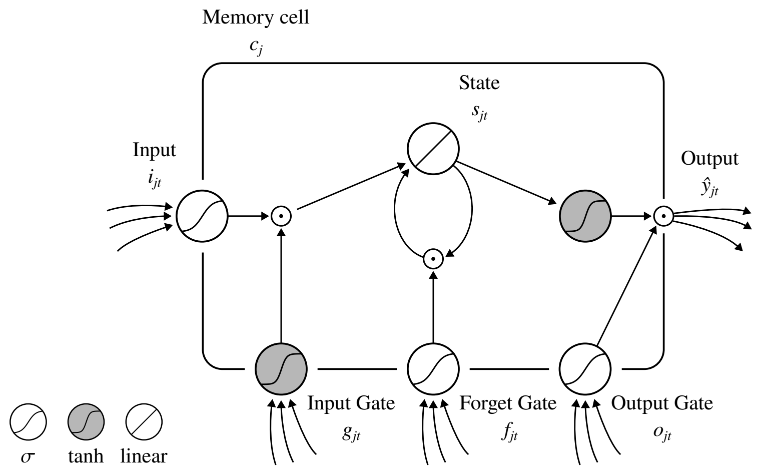

Fig. 3 Single LSTM memory cell c_j with σ and tanh **activation functions. Figure by the author, adapted from [6, Fig. 1], [21, Fig. 1], [19, Fig. 4.2].**

The overall network of the RNN with recurrent connections between units remains intact but the LSTM memory cells feature extended parameters and internal recurrent connections [12, p. 406]. An excerpt of an RNN with LSTM is shown in Figure 4. Networks utilising LSTM are able to learn to bridge time lag far beyond the capabilities of standard RNNs [21, p. 850].

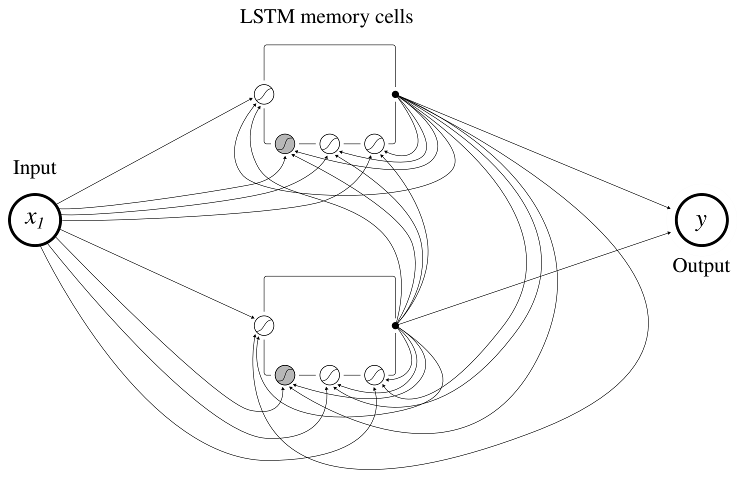

Fig. 4 RNN with two LSTM memory cells in the hidden layer. Weighted edges from input x_1, recurrent edges and output edges are shown. Figure by the author, adapted from [6, Fig. 2], [21, Fig. 3], [19, Fig. 4.3].

An LSTM layer performs a series of calculations for each element of the input sequence. For each time step t the following calculations are computed using weight matrices $\overline{W}$ and biases $\overline{b}$. The index j corresponds to the memory cell c_j. The gates of the LSTM layer are covered by

$$f_t \coloneqq \sigma(\overline{W}_{jf}x_t + \overline{b}_{jf} + \overline{W}_{yf}y_{t-1} + \overline{b}_{yf}), \tag{4}$$
$$g_t \coloneqq \tanh(\overline{W}_{jg}x_t + \overline{b}_{jg} + \overline{W}_{yg}y_{t-1} + \overline{b}_{yg}), \tag{5}$$
$$o_t \coloneqq \sigma(\overline{W}_{jo}x_t + \overline{b}_{jo} + \overline{W}_{yo}y_{t-1} + \overline{b}_{yo}). \tag{6}$$

The calculations for input i_t, state s_t and output y_t are shown below with $\odot$ being the Hadamard product [22].

$$i_t \coloneqq \sigma(\overline{W}_{ji}x_t + \overline{b}_{ji} + \overline{W}_{yi}y_{t-1} + \overline{b}_{yi}), \tag{7}$$
$$s_t \coloneqq f_t \odot s_{t-1} + i_t \odot g_t, \tag{8}$$
$$y_t \coloneqq o_t \odot \tanh(s_t). \tag{9}$$

The individual cell states are calculated on the basis of the gated input and the gated recurrent connection, Equation 8. The output of the layer y_t is gated as shown in Equation 9. The dimension of y_t is a crucial hyperparameter of LSTM implementations. The feature count of y_t dictates the size of the LSTM layer and in some respects the memory size of the overall NN [23].

D. Reinforcement Learning

RL presents one of the approaches of ML with the focus on goal-directed learning. RL is basically learning how to map situations to possible actions in order to maximise a numerical reward. The actions are performed by an agent, which has to discover the best action for any situation by trying different available actions during the learning process. The action taken may not yield a reward immediately but only in the future. The agent has the ability to receive information about the state of the environment (sensation) and can take actions that affect the state of the environment (action). The agent also has an objective within the environment (goal) [3, p. 1].

The trial-and-error nature of the learning process presents a trade-off challenge for the agent. To discover actions that yield a better reward, the agent has to explore. To receive a reward, the agent must take advantage of the knowledge gained previously. Both exploration and exploitation must be executed and weighed against each other in order to get the best possible long-term reward [3, p. 3].

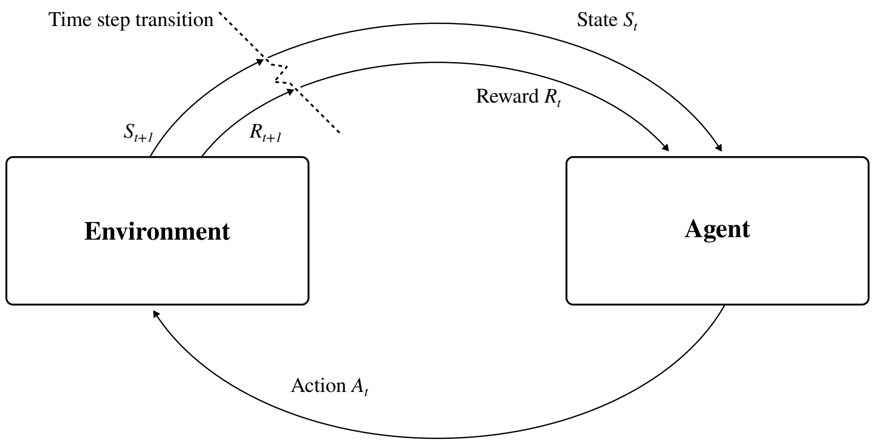

Fig. 5 Interaction between the agent and the environment. Figure by the author, adapted from [3, p. 48].

1. Markov Decision Process

Markov decision processes or MDPs are a formalisation of sequential decision making and present an idealised form of the RL problem [3, p. 47]. The elements and properties of MDPs are presented in the following.

The agent learns and chooses the actions to perform. Everything apart from the agent is the environment. The agent interacts with the environment continuously by receiving a state from the environment and then performing an action in turn as shown in Figure 5. The environment also gives rewards to the agent in the form of numerical values. The interaction of agent and environment occurs as a series of discrete time steps t. At each time step, the agent receives a state of the environment $S_t \in \mathcal{S}$ and an action $A_t \in \mathcal{A}$ is performed. In turn, the environment presents the agent with a reward $R_{t+1} \in \mathcal{R}$ and a new state S_{t+1}. This interaction is repeated and a sequence of interactions between agent and environment is created.

Here only finite MDPs are discussed with finite elements in all of $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ [3, pp. 47–48]. This holds true for the floating-point representation of $\mathbb{R}$. $S \in \mathcal{S}$ may be a vector of real numbers but is only represented with floating-point numbers with a limited range of values [16, pp. 938–939]. The used $\mathcal{A}$ and $\mathcal{R}$ are also finite.

The states $S \in \mathcal{S}$ of MDPs must maintain the *Markov property*, by including all information about all past interactions between the agent and the environment that can influence future interactions [3, p. 49]. Therefore, any given state is only dependent on its previous state and the action performed in the previous state.

R_t and S_t are random variables with well defined discrete probability distributions, only depending on the previous state and action as prescribed by the Markov property. At a specific time step t the state $s' \in \mathcal{S}$ and the reward $r \in \mathcal{R}$ can occur with the probability $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ [3, p. 49]:

$$p(s', r \mid s, a) \coloneqq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}. \tag{10}$$

The goal of the agent is formalised by the reward signal presented by the environment to the agent in form of $R_t \in \mathbb{R}$ at every time step t. The reward shall define the desired goal without providing instructions on how to achieve the goal. The expected rewards for a given triple of state, action and state is the function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ [3, p. 49]:

$$r(s, a, s') \coloneqq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} \frac{p(s', r \mid s, a)}{p(s' \mid s, a)}. \tag{11}$$

The agent's aim is to maximise the expected value of the cumulative sum of a received scalar. The expected reward beyond the current time step t is the sum of the rewards up to the final time step T. To the expected reward the discount rate $\gamma \in [0, 1]$ is applied to influence the far sight of the agent with regard to the potential future rewards [3, p. 55]:

$$G_t \coloneqq \gamma^0 R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{12}$$

With $\gamma = 0$ the agent only takes the very next reward into account, $G_t = R_{t+1}$, but with γ closer to 1 more future rewards are taken into account. No discounting would be applied with $\gamma = 1$ [3, p. 55].

The starting state and terminal state mark the start and end of an *episode*, respectively. Following the termination of an episode a reset to a specific starting state or a selected state from possible options is performed. A terminal state is especially practical when the particular application presents an inherent final goal like reaching a target. The interaction of agent and environment is a sequence of episodes each with a distinct start and terminal state [3, p. 54].

The policy π determines how the agent behaves by mapping the state of the environment to the probabilities of taking each possible action [3, p. 58]. In the broadest sense, RL methods specify how the agent's experiences influence the policy. Most RL algorithms use a *value function* v_π, also known as *state-value function*, to estimate the expected reward G_t based on policy π and starting in a specific state s, $v_\pi : \mathcal{S} \to \mathbb{R}$:

$$v_\pi(s) \coloneqq \mathbb{E}_\pi[G_t \mid S_t = s], \forall s \in \mathcal{S}. \tag{13}$$

In addition the *action-value function* q_π estimates the expected return when taking action a in state s under policy π, $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$:

$$q_\pi(s, a) \coloneqq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]. \tag{14}$$

Both value functions, v_π and q_π, can be estimated from gathered experience. One possible option is to average across actual returns. Maintaining an average for many states is expensive. A less expensive option is to approximate v_π and q_π through parameterised functions [3, pp. 58–59].

The *Bellman equation* for v_π defines the relationship between the value of a state and the value of all successor states weighing each by its probability of occurring according to policy π [3, p. 59]:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right], \forall s \in \mathcal{S}. \tag{15}$$

The value of the start state equals the value of the most likely next state and the reward along the way taking the discount factor γ into account. The value function v_π is the unique solution of the Bellman equation. In addition, the Bellman equation forms the basis of many approaches to determine v_π [3, pp. 59–60].

For finite MDP an *optimal policy* π_* and corresponding *optimal state-value function* v_* can be defined. Based on the value function v all policies are partially ordered. For at least one policy π_* $v_\pi(s) \geq v_{\pi'}(s), \forall s \in \mathcal{S}$, holds for all other policies π'. The optimal value function is defined as [3, pp. 62–63]

$$v_*(s) \coloneqq \max_\pi v_\pi(s), \forall s \in \mathcal{S}. \tag{16}$$

Under an optimal policy π_* the value of a state equals the expected return for the best action choosen in that state. This is reflected in the Bellman optimality equation for v_*:

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')]. \tag{17}$$

The Bellman optimality equation has exactly one solution for finite MDPs [3, pp. 63–64]. Solving the Bellman optimality equation provides a direct route to solving the RL problem by determining the optimal policy. This route proves impractical most of the time because of insufficient computational resources. Usually, approximate solutions are the only viable solutions to the Bellman optimality equation and RL in general [3, p. 66].

2. Reward Function

Designing the reward is a fundamental part of RL and has a profound impact on the performance of the RL algorithm [24, p. 228]. The reward function must frame the goal of the RL problem overall and also indicate the progress towards reaching the goal. The reward takes the shape of the scalar R_t at every time step t and is sent from the environment to the agent as shown in Figure 5. The agent must be enabled by the reward function to learn a behaviour that eventually reaches the desired goal. In the worst case, the agent may adopt undesirable or dangerous behaviours to receive rewards from the environment [3, p. 469]. This phenomenon is called *reward hacking* and can be caused by varying reasons, such as complex reward functions potentially being open in unforeseen ways to provide unwanted rewards to the agent [25, pp. 7–8].

In general, the reward function itself presents an additional parameter of the learning algorithms introduced in III.E. As a parameter, the reward function is subject to trial and error or even algorithmic optimisation to find a suitable candidate [3, pp. 469, 471]. A natural source of reward may be inherent to the problem itself. For instance, goal-directed RL problems like navigation towards a target should reward reaching the target. Adding more rewards in addition to the goal provides more information, but risks introducing distractions from the goal [26, pp. 1104–1105].

The reward function can take the form of a sparse or dense function. Sparse reward functions return mostly zero and a differentiating value is only sparsely available [27, p. 2]. Non-zero returns often correspond to physical events in the environment. For example, reaching a target can generate the reward of +1 [28, p. 3405]. Hodge et al. [11] followed this approach and additionally rewarded collision by -1. This presents a challenge to the agent, as progress may be difficult to observe [3, p. 469]. Sub-goals can be introduced to the sparse function to increase the feedback of information to the agent. Additionally, the sparse reward can be dynamically adapted during the training progress. This shaping of the reward can include training in various RL problems that become increasingly more difficult and complement each other in the curriculum to achieve the ultimate goal [3, p. 470]. However, this approach has a high likelihood of influencing the optimal policy [27, p. 3]. Sparse reward systems are often complimented by negative penalties for each action taken [29, p. 3]. The *minimum-time* approach focuses on fast goal completion by only providing a fixed negative reward for each action and no differentiating value. This is a denser reward system but still considered to be sparse [27, p. 4], [24, p. 233].

Dense reward functions return mostly non-zero values and provide more information to the agent, but are recognised to be difficult to design without interfering with the ultimate goal [28, p. 3405], [3, p. 470]. Especially the faster learning progress of dense reward is acknowledged, but the quality of the resulting behaviour may be compromised [27, pp. 4–6]. Both forms of the reward function can be combined [30, p. 4].

E. Learning Algorithms

Proximal Policy Optimisation (PPO) and *Soft Actor-Critic* (SAC) are two common algorithms to solve RL problems [7, p. 11]. PPO is a member of the *Policy Gradient Methods* while SAC can be counted to the *Actor-Critic Methods*. Both algorithms present viable options and are introduced in the following.

1. Actor-Critic Methods

Actor-Critic Methods learn approximations of the policy (actor) and the value function (critic). Usually a state-value function is learnt and applied to the second state of a state-action-state transition. The state-value function is used to assess the action [3, p. 331].

Part of this family of methods is the SAC algorithm developed by Haarnoja et al. [31]. In contrast to on-policy methods, SAC does not require to get new samples for each gradient step. SAC efficiently reuses past experiences and optimises a policy with regard to reward return and entropy. Thus, policy updates favour wide-ranging exploration. This RL algorithm is designed specifically for continuous state and action spaces [31, pp. 1–3]. Due to the complexity of RNN and LSTM networks, the use of discrete action spaces is commonly applied [17]. Following this recommendation, the SAC algorithm was dismissed for this thesis.

2. Policy Gradient Methods

Policy Gradient Methods learn a parameterised policy for action selection without the need to access the value function. To learn the vector of policy parameters $\theta \in \mathbb{R}^{d'}$ the gradient of the performance measure is used. θ is integrated into the policy $\pi(a \mid s, \theta) \coloneqq \Pr\{A_t = a \mid S_t = s, \theta_t = \theta\}$ [3, p. 321].

The PPO algorithm was first introduced by Schulman et al. [32]. The algorithm alternates between sampling data through environment interactions and performing several epochs of optimisation using the sampled data [32, p. 1]. Schulman et al. proposed an objective function to be maximised [32, p. 3]:

$$L^{CLIP}(\theta) \coloneqq \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \operatorname{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]. \tag{18}$$

With

$$r_t(\theta) \coloneqq \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{19}$$

being the probability ratio between the current updated policy and the old policy. $\hat{A}_t$ estimates the advantage based on the current value function in an algorithm that alternates between sampling and optimisation [32, p. 2]. Equation 18 updates the policy, usually across multiple steps, to maximise the objective. The hyperparameter ϵ, in conjunction with the clipping in Equation 18, limits the new policy from deviating too far from the old policy to ensure reasonable policy updates [33].

IV. Method

This section details the conducted experimental investigation. The elements of the RL problem are introduced, and the implementation in Unity is outlined.

A. Elements of the Reinforcement Learning Problem

The agent and the environment form the core components of the navigational RL problem discussed in this thesis. Both the agent and the environment, including their sub-elements, are described in detail hereafter.

1. Agent

The *Bitcraze Crazyflie 2.1* sUAS (Figure 6) is commercially available and is the basis for the agent model. The flight time is limited to 7 minutes. The recommended payload weight is restricted to 15 g. An *inertial measurement unit* (IMU) with a 3 axis accelerometer is available [34].

Fig. 6 Bitcraze Crazyflie 2.1 sUAS [34].

The available actions for the agent $\mathcal{A} = \{1, \ldots, 6\}$ are: Forward (1), backward (2), right (3), left (4), rotate right (5) and rotate left (6). The available movement options are illustrated in Figure 7. In line with Wang et al. [9] the movement of the sUAS is simplified to operate only at a fixed height [9, p. 6182]. Further, the action space $\mathcal{A}$ is discrete, in line with the recommendations of ML-Agents when using LSTMs [23]. Therefore, only one discrete action is selectable at every time step t, $|A_t| = 1$.

The limited sensor capability is comparable to the drone modelled by Wang et al. [9], but it lacks GNSS capability. The IMU onboard the sUAS is used to calculate the velocity in two axes and the rotation around the y axis. The single-point range sensors employed onboard the sUAS perform LiDAR-based measurements. The 5 g *Benewake TFmini-S* sensor served as basis to devise a possible ranging suite. Due to its size, weight and operating range this sensor type was selected. However, these sensors have operating constraints to consider, including operating range, resolution, and potential malfunctions due to encountered materials [35]. The implementation in this thesis assumes the sensors work at any distance, without malfunctions, and with infinite resolution, limited only by the resolution of floating-point numbers [16, pp. 938–939]. Assuming a sensor weight of 5 g, a sensor count H of 3–4 may be realistic considering the payload limit of the base sUAS. The number of range sensors H is dynamically set with the minimum being one sensor. There is always one sensor pointing in the forward direction of the agent. The rest of the range sensors is distributed equiangular to cover the entire 360° around the agent. Possible interference of range sensors and flight hardware is disregarded. A setup with four range sensors is also shown in Figure 7.

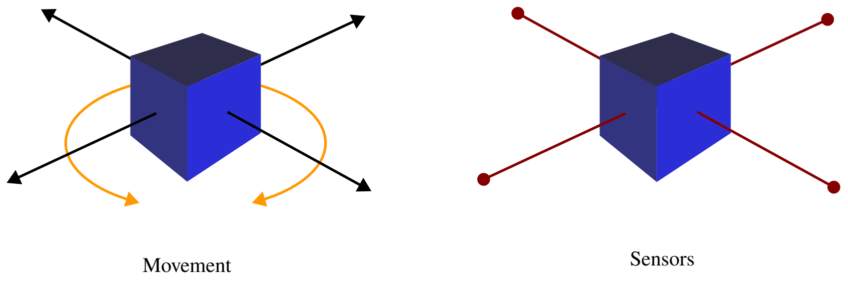

Fig. 7 Drone movement options and sensor positions for an agent featuring four range sensors.

2. Environment Layouts

Two different environment layouts were implemented to train sUAS agents. Both environments feature overarching properties. The elements of the first environment, as described below, are also part of the second environment.

The floor plan of *environment 1* consists of outer walls, a ceiling, and a floor. The floor plan is shown in Figure 8. The agent can move freely at a fixed height within the confines of the walls. Collisions of the agent and other objects are detected. The target rests on the floor and is detectable by the range sensors of the agent. The positions of the drone and the target are reset at the start of each episode. δ_{goal} defines the threshold distance to the target and a minimum distance of $3\,\delta_{\text{goal}}$ between the agent and the target is ensured at the start of the episode. The minimum distance is set to ensure a minimum length of the episode. The agent and target can not be placed in the outer perimeter of the environment. This represents a simplification of the RL problem by ensuring a minimum distance from the walls, which enables a collision-free approach path for the agent to the target and results in larger differences in sensor readings between the target and the surrounding walls. This *no-placement zone* stretches across the outer 10 % of the distance between opposing walls.

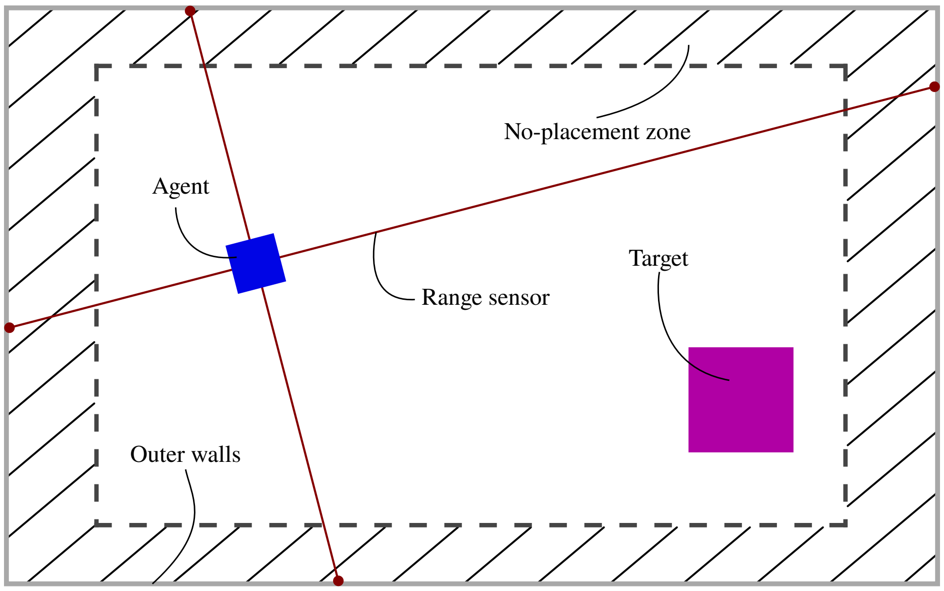

Fig. 8 Floor plan of environment 1 with the agent, the range sensors, the target, and the no-placement zone shown.

The layout of *environment 2* is shown in Figure 9. The elements of environment 1, especially the no-placement zone, are carried over to the second environment but a dynamic inner wall with a door is introduced. The door is represented by a simple opening in the inner wall. The position of the inner wall and the placement of the door within the wall is randomly assigned within the highlighted door placement zone in Figure 9. The area is chosen to create two rooms of suitable size for the sUAS to move throughout the rooms while exploring and avoiding collisions. The door frame is never in direct contact with the outer wall. The agent and the target are placed at least $3\,\delta_{\text{goal}}$ from the door. Both objects can be placed in separate rooms or in the same room. In 50 % of the episodes the objects are in the same room at the start. If the objects occupy the same room, no passage of the agent through the door is necessary to complete the navigation task.

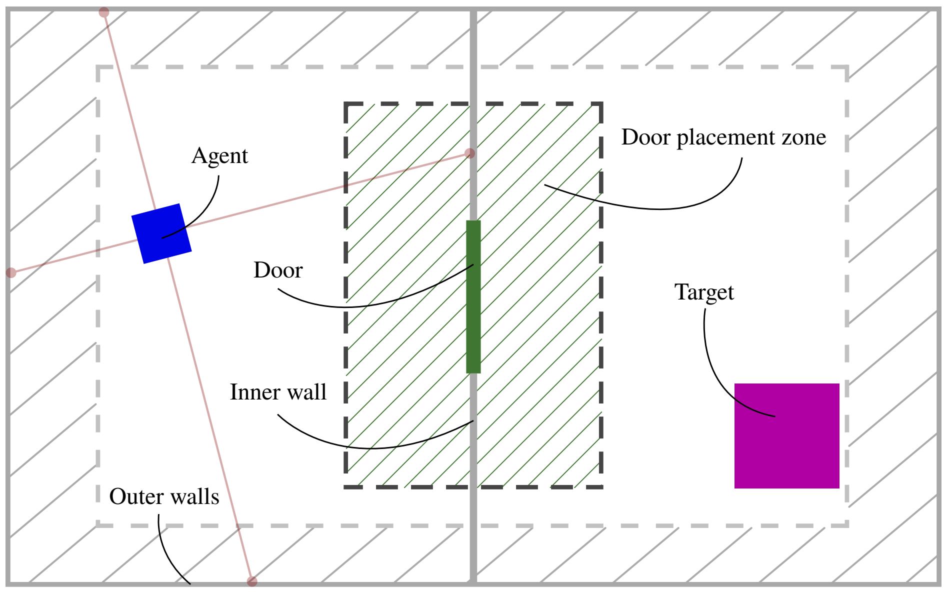

Fig. 9 Floor plan of environment 2 featuring a dynamic inner wall with a randomly placed door within the placement zone. Elements of environment 1 are still present.

3. State

The state $S_t \in \mathcal{S}$ provided by the environment to the agent corresponds to sensor capability of the agent and takes the form

$$S_t := (\delta_1, \dots, \delta_H, v_x, v_z, \rho_y). \tag{20}$$

$\delta_1, \dots, \delta_H$ are the range sensor measurements with H being the range sensor count currently onboard the agent. δ_h is normalised by dividing it by the maximum possible distance in the environment, thereby scaling δ_h to $[0, 1]$. The observations of velocity in two axes v_x and v_z as well as rotation around the y axis ρ_y complete the state information. ρ_y is also normalised to the range $[0, 1]$ by dividing it by 360.

4. Reward

The implemented reward system is relatively sparse and takes the form:

$$R_t := R_{\text{goal}} + R_{\text{col}} + R_{\text{door}} + R_{\text{action}}. \tag{21}$$

A sparse reward system was chosen because of clearly defined physical interactions of the agent and the environment such as collisions. The robustness and simplicity of sparse reward systems as discussed in section III.D.2 support this decision.

The sub-goal R_{door} was added to aid the training process without providing constant feedback to the agent. The system combines goal-reward, collision-penalty, sub-goal-reward and action-penalty. The choosen values of the reward components were initially selected based on ML-Agents' recommendations and literature sources. Subsequently these values were subject of a hyperparameter study. The values presented below represent a selection of possible values, determined through experimental hyperparameter search.

R_{goal} encodes the goal-reward representation. When the distance from the agent to the target drops below the threshold δ_{goal} the target is considered reached by the agent. The ultimate goal of the navigational problem is reached and a positive reward is given. Otherwise R_{goal} remains 0:

$$R_{\text{goal}} := \begin{cases} +1 & \text{distance to target} < \delta_{\text{goal}}, \\ 0 & \text{target not reached.} \end{cases} \tag{22}$$

Collision-free navigation is a key requirement of this thesis as stipulated in task 2). During the experiments, the continuation of an episode even in the event of a collision proved beneficial to the learning progress. To account for collision avoidance, a penalty for collisions is required. Initial contact with an object is considered the initiation of the collision event and incurs a penalty. Continued contact is penalised less severely, but a penalty is still applied to encourage fast disengagement from the object. Collisions of the agent with objects other than the target are penalised through the component R_{col}. Derived from the physical event between the agent and the environment, R_{col} encodes a sparse feedback only on contact:

$$R_{\text{col}} \coloneqq \begin{cases} -0.5 & \text{initial collision with object,} \\ -0.3 & \text{continued collision with object,} \\ 0 & \text{otherwise.} \end{cases} \tag{23}$$

A single sub-goal is employed in the reward system to aid in the learning progress. The door passage is used as sub-goal in the layout of environment 2. R_{door} describes the passage through the door. In practice, the door is represented by a door frame in the wall and forms a checkpoint. Upon departure from the checkpoint, a check is performed to verify whether a passage from one room to the other room was achieved and whether the passage through the door was in the correct direction, i.e. towards the target. R_{door} is set to discourage repeated passages through the checkpoint to gain an overall positive reward.

$$R_{\text{door}} \coloneqq \begin{cases} +0.5 & \text{door passage in correct direction,} \\ -0.6 & \text{door passage in wrong direction,} \\ 0 & \text{otherwise.} \end{cases} \tag{24}$$

Each performed action is penalised in line with recommendations laid out in section III.D.2. Especially given the flight time of the original sUAS the desire to achieve a minimum flight time solution gains additional attention. The relatively small penalty for the action is set in combination with T in mind. For each episode

$$\sum_{t}^{T} R_{\text{action}} \geq -1 \tag{25}$$

should hold true as per ML-Agents recommendations [23]. The action penalty was set to:

$$R_{\text{action}} \coloneqq -0.0005. \tag{26}$$

Based on the environment, the expected reward limit for an episode R_{limit} varies. In environment 1, the agent cannot pass through the door, making R_{door} irrelevant. The influence of R_{action} on R_{limit} is limited and decreases with shorter episode length. Therefore, the reward in environment 1 should converge towards $R_{\text{limit}} = 1.0$. In environment 2, the passage through the door becomes another factor, reflected by R_{door}. The expected reward limit rises to $R_{\text{limit}} = 1.25$, taking into account that the agent and the target are located in the same room in half of the episodes, making no door passage necessary.

B. Implementation in Unity

The actual implementation in Unity, as described in task 1), is outlined in the following. Unity was used in version 2021.3.11f1. The code used by the author is publicly available [36].

The agent moves at a fixed height and has no pitch or roll ability, which limits the RL problem to a 2D space. However, the implementation in Unity was done purposefully in 3D for future extendability, with provisions for action and sensor capabilities for 3D movements of the agent.

1. Agent in Unity

The agent is represented by a single cube game object. The Rigidbody component of the game object allows the control of physical properties of the agent. Especially the drag component can be set. The influence of gravity on the agent was deactivated to have the agent always move at a constant, pre-set height. The rotation of the game object was restricted to be only possible through actions $A_t \in \{5, 6\}$ of the agent. A collision between the agent and another object can not cause the rotation of the agent. These simplifications were deemed reasonable in context of the highly abstract environment. The Collider component manages the detection of collisions between the agent and other objects.

The implementation of the agents movement is following the behaviour of real-life sUAS. Based on the input the `AddForce()` function is used to exert a force onto the agent. This leads to inertia being introduced to the movement of the agent. This implementation reflects better real-life sUAS and ensures a correct detection of collisions [37].

2. Environment in Unity

The environment described in section IV.A.2 is constructed in Unity from basic game objects as shown in Figure 10. The floor and ceiling are Plane objects. The outer walls are Cube objects. All objects are equipped with their respective collider components to detect collisions between the environment objects and the agent. The Unity engine detects collisions and triggers functions `OnCollisionEnter()` and `OnCollisionStay()`. These function calls are used to apply R_{col} accordingly to R_t.

In case of environment 2 the inner wall is dynamically created. The inner wall is also constructed from a number of Cube objects. An actual door frame is build dynamically. The passable area of the area is a Collider trigger making the object traversable for the agent. Entering the trigger calls `OnTriggerEnter()`. `OnTriggerExit()` is called on leaving the trigger by the agent. The latter is used to check the quality of the passage through the door in terms of direction and apply R_{door}.

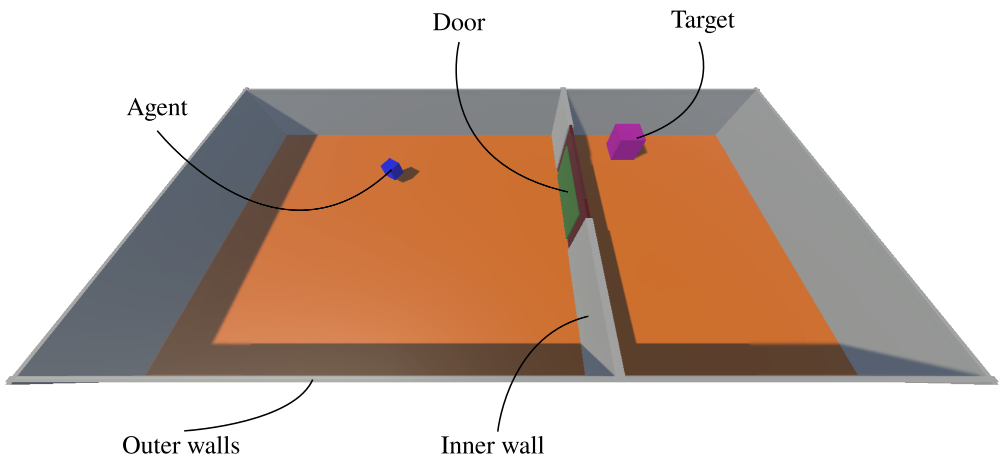

Fig. 10 Environment 2 as used in Unity. Agent (blue) and target (violet) situated in different rooms. The door checkpoint (green) is surrounded by the solid door frame (brown). The ceiling is transparent but features the same properties as the floor (orange).

C. Machine Learning with Unity

The Unity ML package ML-Agents is used to train an agent within the described environments. Aspects of the implementation of ML-Agents, the training and hyperparameter setup are discussed below. ML-Agents version 0.30.0 was utilised with Python 3.9. PyTorch in version 1.7.1 was used to provide ML implementations for ML-Agents.

1. Neural Networks in ML-Agents

The main consideration of any ML application is the employed NN and its structure. ML-Agents provides various options to configure an NN via the training configuration file. An example file with commentary is provided in the appendix. The fundamental options for a feed-forward network consist of the number of hidden layers and number of hidden units. Minimum one hidden layer is always present.

The memory capability of networks is implemented via an LSTM implementation. ML-Agents provides the option to append a single LSTM layer after the hidden layers of the vanilla NN. The dimension of the LSTM layer output y_t and size of the LSTM layer is defined via the configuration option `memory_size`. The option `sequence_length` defines the size of the minibatch of the input sequence for the training of the RNN [23]. The LSTM layer implementation of ML-Agents reflects the LSTM structure discussed in section III.C [22].

ML-Agents supports three algorithms to solve RL problems. PPO, SAC and *Multi-Agent Posthumous Credit Assignment* (MA-POCA) are the available algorithms [7, p. 11] [17]. The MA-POCA algorithm is used in cooperative multi-agent RL and is able to handle a variable number of agents working together throughout an episode [38, p. 2]. Since the RL problem in this thesis is limited to a single agent, MA-POCA is not applicable. As briefly discussed in section III.E the PPO algorithm is the only suitable candidate to train neural networks with LSTM.

2. Time Step Implementation

A crucial area of attention needs to be the implementation of time steps. The Unity engine itself treats the execution of `FixedUpdate()` as a discrete time step. Within a call of `FixedUpdate()` physics calculations of the engine are executed. These function calls occur independently of the frame rate [37].

ML-Agents introduces the *Decision Requester* Component to the agent. The component allows to set the *Decision Period* dp to a numerical value. With $dp = 1$ the ML time step t corresponds directly to the calls of `FixedUpdate()`. With every `FixedUpdate()` call, a decision is requested from the NN for the agent to execute. This results in limited time for applied forces to act on the agent. Thus the distance between states of the environment is minimal. This results in unstable learning progress. $dp = 5$ is a more common setting [23]. One time step t now comprises fives updates to the physics engine. With every five calls of `FixedUpdate()` a new NN decision is requested. This allows for more updates to the physics caluclations to take place and the distance between states S_t and S_{t+1} increases. Tests with further settings for dp were performed, but $dp = 5$ proved to be reasonable.

A further setting of the Decision Requester is whether decisions of the NN should be repeated during the selected period. When selected and with $dp = 5$, the agent executes the same action as previously selected by the NN with every `FixedUpdate()` call. This also proved to be a reasonable setting during the experiments.

The last important aspect of the time step implementation is the setting for the maximum number of time steps per episode T. This setting is known as `MaxStep` in ML-Agents and limits the number of `FixedUpdate()` calls. With `MaxStep = 1000` and $dp = 5$ only 200 individual actions can be requested from the NN before the episode automatically terminates [23]. These settings of `MaxStep` were determined to be viable to limit the possible episode length while allowing enough time for exploration. The implementation of a maximum episode length T is only loosely associated to the possible flight time of the real sUAS.

3. Hyperparameter Search

As hinted at previously, a large number of parameters in ML are not necessarily obvious to determine. These parameters require a trial-and-error or an algorithmic search. Different hyperparameter optimiser options are available. PyTorch offers options to automatically optimise hyperparameters [39]. As far as the author is aware, there does not exist a hyperparameter optimiser with direct integration into ML-Agents. For this thesis a simple hyperparameter search was implemented. The developed script runs pre-compiled builds of the Unity environment automatically. User selected sets of values for hyperparameters are run in all possible permutations. Due to hardware limitations this approach is time consuming. Also, the approach lacks the ability to automatically narrow a search grid down and requires constant user evaluation of the results.

The main parameters, which were subject to the hyperparameter search, are briefly introduced in table 1. These were selected in line with [23] and through experimentation. Especially the structure of the NN was subject of extensive evaluation. In addition to the listed parameters, the components of the reward system were studied.

Hyperparameter	Description
time_horizon	Number of steps (state-action pair) to be collected before being added to the buffer. Needs to take sparse reward sytem into account.
batch_size	Size of experience pool for each gradient descent iteration.
buffer_size	Number of collected experiences before updating the policy.
learning_rate	Learning rate for gradient descent.
hidden_units	Size of the hidden layers.
num_layers	Number of hidden layers.
memory_size	Size of the LSTM layer.
sequence_length	Minibatch size to train the RNN.
γ	Reward discount factor.

Table 1 Hyperparameter.

D. Experiments

Brief of this thesis is to evaluate the use of Unity and ML-Agents to train an agent to navigate indoor environments. Three experiments were conducted as part of this thesis and are outlined below. A hyperparameter optimisation covering NN parameter is conducted for all experiments.

1. Experiment 1: Environment 1 Verification

Initial work to verify the created environments and agent, as described in task 2), was conducted using Environment 1 without the dynamic inner wall. The discussed reward structure is employed, without the R_{door} component. The goal of this first experiment is to establish whether the devised NN is capable of performing the navigational task with very few sensors at all. The performance of the agent is monitored using the listed indicators. The indicators, except the total number of performed steps (state-action pairs), are calculated as a mean over the last 1e+6 steps of the training run.

Due to the random nature of the environments, the evaluation of the NN performance is carried out during the final phase of the training rather than in separate test environments. This evaluation uses different scenarios from those used during the training beforehand, as the start positions of the agent and the target are randomised. The assessment of different models and training runs based on randomised scenarios of the environments is limited. To mitigate this issue, the evaluation is conducted over a large number of training steps and scenarios.

- *Cumulative reward*: The reward is accumulated over the number of steps of the episode. The cumulative reward should rise and converge towards the expected reward limit $R_{\text{limit}} = 1$.
- *Standard deviation (STD) of the cumulative reward*: The standard deviation is taken to gauge the stability of the learning progress in the final phase of the training.
- *Episode length*: The episode length should decrease from the maximum step count of 200 per episode.
- *Target rate*: Percentage of the episodes terminated on reaching the target.
- *Collision rate*: The percentage of the actions leading to an initial contact between the agent and an object. Excluding actions associated with a continued collision event. Perfect collision free navigation would result in a value of 0 %.
- *Total performed steps*: Total number of performed steps during the training run.

2. Experiment 2: Environment 2 Verification

To further verify the created environments in accordance with task 2) further training was performed using environment 2. The complexity of the dynamic inner wall required an increase in the duration (max step count) of the training. Beside the indicators used during the course of experiment 1, further indicators are introduced to gauge how the agent manages the door passage:

- *Door passage quality*: Gauge the ratio between passage in the correct and the incorrect direction. The quality is listed in the interval $[0, 1]$. A value of 1.0 would indicate only correct passages were performed by the agent.
- *Door passages per episode*: In cases of very few passages, the previous indicator can be misleading. A single registered door passage per 10,000 steps might suggest excellent passage quality, yet door passages occur extremely

rarely. To offer better insight, the number of door passages per episode is also provided. An agent needs to navigate through the door if the target is in another room. However, if both the target and the agent are in the same room, a door passage is not necessary. Assuming a perfect passage quality of 1.0 the number of passages per episode is going to be at about 0.5.

3. Experiment 3: Sensor Count Study

The limited payload capacity of the agent led to the study of how the sensor count H influences the performance of the agent as described in task 3). To provide a wide field of options and a sUAS model independent study the upper limit of sensor count was chosen to be well beyond the payload capability of the base sUAS.

After verifying the proposed environment and overall implementation through experiment 2, the same environment 2 is used. The NN parameter are also carried over from the previous experiment without further hyperparameter search. The training duration was reduced compared to experiment 2. This decision was taken to limit the anyways extensive computational time needed to carry out the detailed experiments. The same performance indicators as before are available.

V. Results

Based on the described method the results of the experiments are outlined and discussed. Points of interest and suggestions for future work on this topic are given.

A. Environment 1 Verification

As introduced in section IV.D.1 experiment 1 was used to verify the general feasibility of the setup for the given task. The base parameters of the used NN and the performance indicators are presented in Table 2. The structure of the NN and further hyperparameter were subject to a hyperparameter search using the described approach. Only the best training run is discussed. The sensor count $H = 3$ was selected to resemble a likely candidate for the base sUAS.

Parameter	Value	Performance Indicator	Value
Range sensor count	3	Cumulative reward	0.86
Hidden layers	1	Cumulative reward STD	0.01
Hidden units	512	Episode length	53.0
LSTM layer size	64	Target rate [%]	97.7
		Collision rate [%]	0.02
		Total performed steps	2e+07

Table 2 NN parameter and performance indicators of experiment 1.

Figure 11 shows the progress of the training in regard to reward, episode length and collisions. The cumulative reward increases over the course of the training run and is trending towards the expected limit R_{limit}. The episode length decreases as expected. In 97.7% of the episodes the agent reaches the target. Initial collisions are limited to below 25 instances per 10000 steps very early in the training. During the final training stages an action by the agent is collision-free in 99.98% of all actions. Continued collisions are also decreasing during the course of the training, i.e. the agent disengages faster from an object in case of a collision. The trained NN performs well in regard to collision-free navigation as stipulated in task 2).

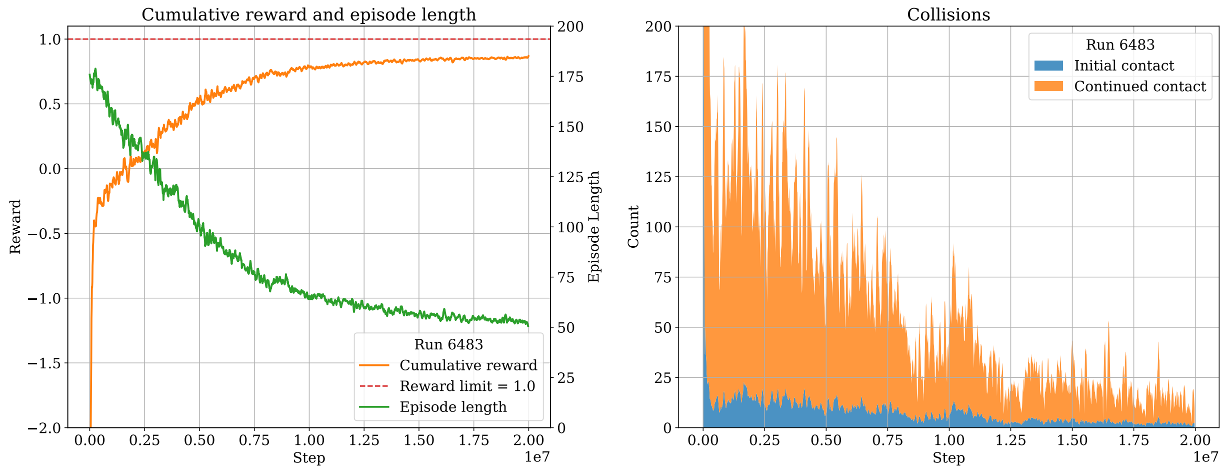

Fig. 11 Cumulative reward and episode length of a single training run in environment 1. Initial and continued collisions illustrated.

B. Environment 2 Verification

Experiment 2, as introduced in section IV.D.3, was conducted next. The sensor count of the agent was increased to 32. This number of sensors is beyond the upper limit of the sUAS payload capability. To aid in training progress, this high number of sensors was selected to aid the agent in identifying its environment faster. Previous training runs during hyperparameter search favoured a higher numbers of sensors. Further investigation of the influence of the numbers of sensors is the subject of the final experiment. Table 3 provides the overview of the NN and the performance indicators.

Parameter	Value	Performance Indicator	Value
Range sensor count	32	Cumulative reward	0.82
Hidden layers	1	Cumulative reward STD	0.02
Hidden units	512	Episode length	63.68
LSTM layer size	64	Target rate [%]	96.27
		Collision rate [%]	0.09
		Total performed steps	1e+08
		Door passage quality	0.8
		Door passages per episode	0.8

Table 3 Parameter overview and final performance indicators of experiment 2.

Similar to the training in environment 1, the episode length decreases, and the cumulative reward increases over the course of the training toward the expected R_{limit} (Figure 12). Correspondingly, initial and continued collisions decrease. In comparison to experiment 1 the collision rate increases by a factor of 5. The collision rate is still deemed to be acceptable at below 0.1% of steps.

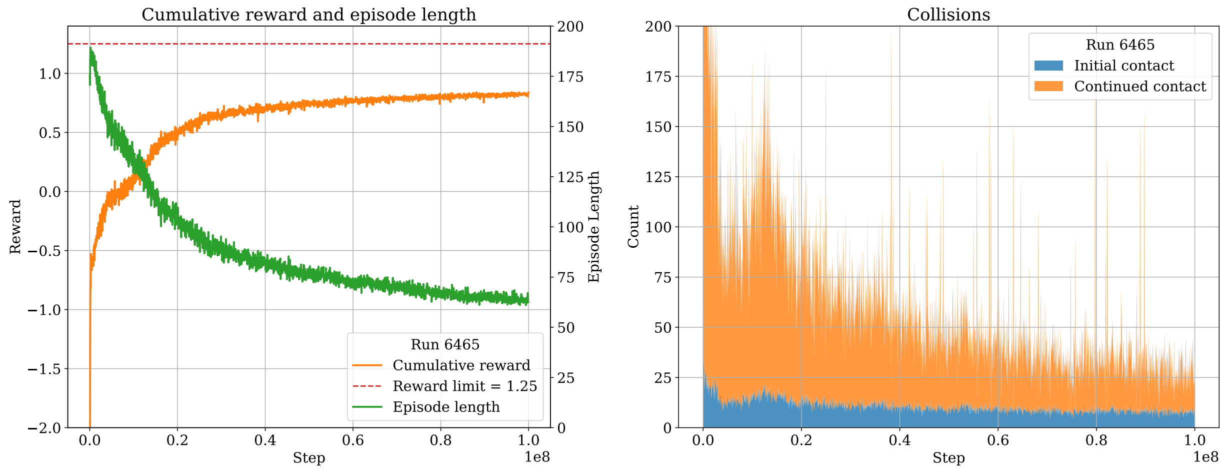

Fig. 12 Cumulative reward and episode length of a single run in environment 2. Initial and continued collisions. Featuring a LSTM enhanced NN.

The passage through the door proves to be a hurdle for the agent, as plotted in Figure 13. Incorrect passages through the door (away from the target) rise to about 30 incorrect passages per 10000 steps. This point is reached early during the training at 0.2e+8 steps. After this point the number of incorrect passages remains nearly constant, decreasing slightly to 25 incorrect passages in the final stages of the training. The correct passages show a consistently increasing trend. With a quality score of 0.8, the results are usable, and the data suggests that a longer training run would yield even better quality. The 0.8 door passages per episode is above the optimal value of 0.5, but again a longer training duration would most likely improve this metric.

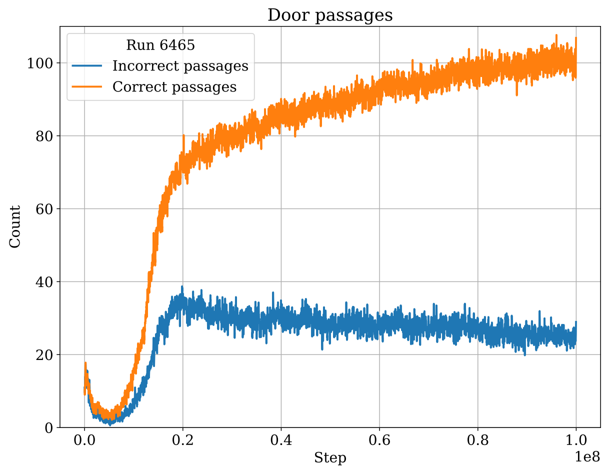

Fig. 13 Door passages by the agent in correct and incorrect direction.

C. Sensor Count Study

As introduced in section IV.D.3 experiment 3 was used to conduct a study of the influence of the sensor count on the NN performance. The NN structure of the previous section V.B was maintained. Individual training runs were conducted with the sensor count $H \in \{1, 2, 3, 4, 8, 16, 32, 64, 128\}$. Each training run was limited to 2e+7 steps. As such, this dataset only provides an initial overview. Training runs to verify environment 2 were usually conducted over 1e+8 steps and still showed active learning progress during the final phase of the training.

Table 4 shows the final performance indicators for each sensor count. The colour scale from white to green (worse to better) in the table indicates the relative performance of the sensor count for a given indicator compared to the other runs.

	Sensor count								
Performance indicator	1	2	3	4	8	16	32	64	128
Cumulative reward	-0.11	-0.11	-0.02	-0.06	0.04	0.55	0.6	0.62	0.68
Cumulative reward STD	0.09	0.07	0.11	0.1	0.15	0.06	0.05	0.05	0.05
Episode length	158.76	158.93	148.96	148.36	138.3	101.88	100.14	93.11	85.43
Target rate [%]	37.42	33.76	40.67	39.33	47.06	82.12	86.86	85.93	90.2
Collision rate [%]	0.11	0.07	0.07	0.1	0.08	0.08	0.08	0.07	0.08
Door passage quality	0.59	0.7	0.61	0.62	0.54	0.7	0.7	0.7	0.71
Door passages per episode	0.34	0.01	0.03	0.03	0.47	1.05	1.08	1.08	1.04

Table 4 Performance indicators of increasing sensor counts during experiment 3.

Increasing the number of sensors, considerably aids the learning process. The final cumulative reward increases while the standard deviation of the reward decreases, indicating a more stable learning progress. Figure 14 highlights the distinct grouping of sensor counts.

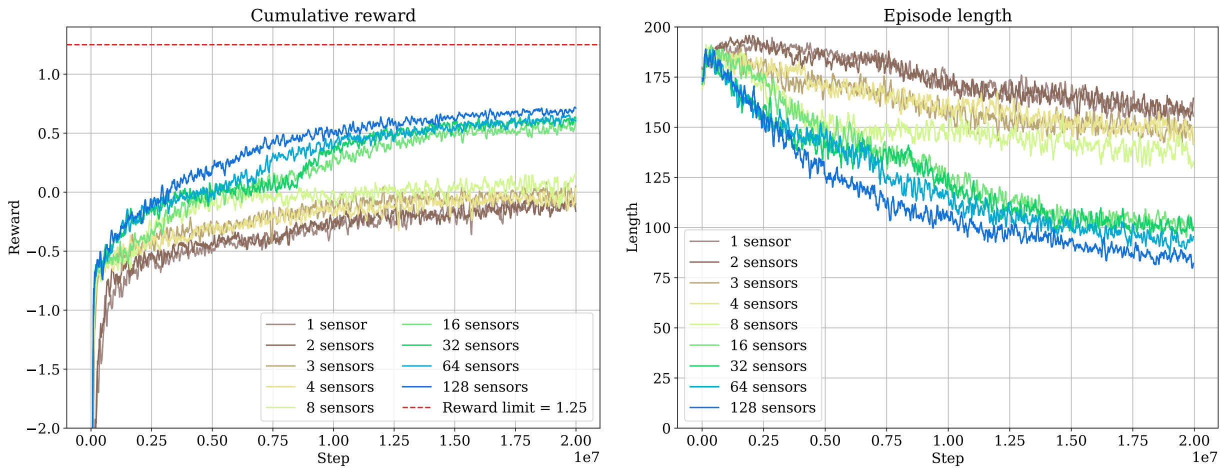

Fig. 14 Cumulative reward and episode length of runs with varying sensor counts.

1–8 sensors reach during this short training all very similar final rewards around at about 0. While sensor counts 16–128 fall again in a group reaching about 0.6 reward each. This observation carries over to the episode length. Shorter episode lengths are achieved by sensor counts 16 and above. The target rate is in line with this observation. The collision rate is for all sensor counts well below 0.2% of agent actions. The collision rates of these runs are so close within each other, a qualified assessment with regard to collision-free navigation is not possible. These short training runs indicate very good collision avoidance for all sensor counts.

In line with the other indicators, the quality of the door passage profits from a higher number of sensors. This indicator has to be seen together with the passages per episode. Figure 15 presents the behaviours of the agents across the training for correct and incorrect door passages.

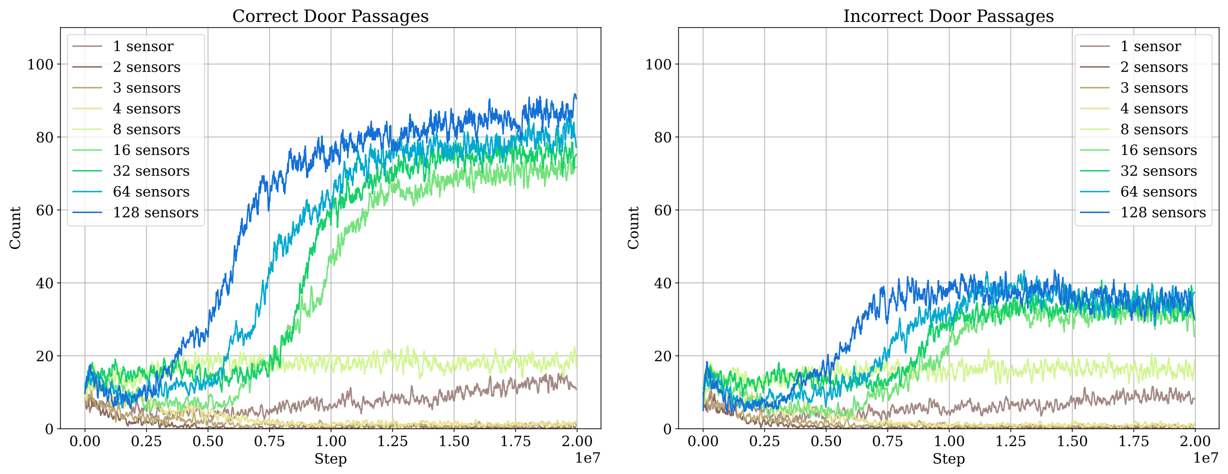

Fig. 15 Correct and incorrect door passages for different sensor counts.

The setups with 1–8 sensors exhibit below 0.5 door passages per episode with medium passage qualities. Counts 2–3 manage nearly no door passages and as such the passage quality of these sensor counts is obscuring. Sensor count 8 is the first run to combine a relatively good collision rate with acceptable passage performance, but still lacks an acceptable level of reaching the target. In fact, the training progress of this run was in the final stages the most unstable, based on the STD of the reward. The remaining counts of 16 and above show higher door passage quality but double the requiered door passages are performed per episode. In general, the sensor counts 16–128 show overall increasing performance in relation to the number of sensors onboard.

The training runs featuring 16 and 32 sensors show very close performance. This indicates a caveat to simply increasing the sensor count. The increase in sensor number is favourable, but must not yield better results per se. The weight penalty of more sensors might be not worth taking when comparable results with fewer sensors are possible.

D. Limitations and Future Work

The presented method and results are discussed with a focus on potential future work in this field.

1. Unity and ML-Agents

Unity as a simulation platform is suitable for the type of RL application discussed in this thesis. It is not necessary to rely on purpose-built ML engines such as DeepMind Lab, as used by Mirowski et al. [5] and Dhiman et al. [4].

Especially the integration of ML-Agents into Unity provides a practical way to implement agent-environment interaction. However, the structure of the NN is severely limited by ML-Agents. Considering the network used by Dhiman et al. [4] with two LSTM layers, it would be advantageous to use a separate network implementation to gain full control over the network. This includes the use of optimiser algorithms to search for hyperparameters without constant operator input. The hyperparameter search strategy used in this thesis is practical but presents only a limited approach.

2. Agent

The representation of the agent in the Unity engine is minimalist but represents a sufficient approach. The range sensors are currently assumed to operate perfectly. Future work should consider operational limitations of the sensors, such as malfunctions due to interaction with reflective or transparent materials. This includes the positioning of the sensors onboard the sUAS, as the interference of flight hardware and sensors is currently not considered. The simplification of the agent to not allow height changes is a further point of interest for future work.

3. Environment

The 3D environments used are highly abstract but represent a first step to implement random indoor environments in Unity. Currently, the environments hardly reflect a real-life indoor environment. Decoy objects, for instance, would greatly increase the complexity of the environment. Openings in the outer walls should be introduced to mimic windows.

Environment 2 uses a dynamic inner wall and as such presents varied challenges to the agent. This variation is still very limited. Fully dynamic floor plans present a large work package, but would help to train more robust NNs.

The reliance on collision detection by the Unity Engine presents an unforeseen challenge. Function calls are triggered whenever a collision event is detected during each physics update in the Unity Engine. A potential consequence is that R_{col} may be applied multiple times during the course of one step (state-action pair). This would lead to very low negative rewards and shift the focus of the reward function towards collision prevention and away from the core navigational task.

4. Reward System

The devised sparse reward system in form of R_t proved robust and simple. The navigation task and sub tasks were encapsulated in the reward function without the need for a dense reward system. The introduction of a sub-goal in form of R_{door} was a success. There were no signs of reward hacking observable. As discussed above, the implementation R_{col} may be problematic.

5. Navigation Task

The overall navigation task of the agent was achieved in a random layout environment. The agent achieved the requirements of task 2) by reaching the target without collision. The target and the collision rate in the crucial environment 2 were acceptable despite increased complexity of the environment.

6. Number of Sensors

The increase of the sensor count onboard the agent is an interesting dilemma. The agent is able to perform the navigation task with relatively few sensors. Especially is there no need to use visual inputs or other more complex sensing technologies as used by [4], [5], [9] and [10]. Type-aware object detection as used by Hodge et al. [11] was also not required to achieve the training goal.

While additional sensors can enhance the learning process, they may not always yield the desired performance increase. Considering payload limitations, a range of 3–8 sensors is a practical selection. Future work could focus on this range, assuming the base sUAS remains consistent, and perform individual hyperparameter optimisation for each sensor count.

7. Complexity of Real World Applications

Transferring the derived NN directly onto a real life sUAS would not yield any useful results due to the abstract nature of the environment and the agent itself. Only an increase of environment complexity during the training would allow to train a NN that may be able to safely navigate in a real-life scenario.

The analysis of the safety of the navigation solution is not part of this thesis. Hodge et al. [11] described a possible approach to identify safety requirements and evaluate the navigational approach accordingly [11, pp. 2027–2029].

8. Imitation Learning

The ML-Agents package supports different implementations of imitation learning [23]. This learning approach builds a model of recorded behaviour [40, p. 120]. Imitation learning requires a relatively high quality of the input data. Producing a viable dataset from recording user inputs proofed difficult in practise. It stands to reason that the time step implementation in Unity and ML-Agents makes the recording of human input impracticable. An approach may be to use pre-programmed algorithmic solvers of the navigation task to generate a suitable dataset. The derived solution to the navigational problem may be influenced by the base data so strongly that the quality of the solutions remains low.

VI. Conclusion

This thesis demonstrates the successful training of an sUAS agent using RL. The Unity engine, along with ML-Agents, effectively allows the creation of an indoor environment and an sUAS agent. The utilised NN featured an LSTM layer to incorporate memory ability. However, ML-Agents limits the NN structure and hyperparameter optimisation possibilities.

In a randomised environment with a dynamic floor plan—where the room dividing wall is random and the outer walls are fixed—the agent was able to navigate to a target object while avoiding collisions. Despite the constraints of an sUAS, achieving the navigational task with only range sensors and an IMU is feasible. The sensor count study highlights the importance of carefully selecting the number of sensors.

The abstract nature of the environment and the agent makes real-life deployment of this navigation solution impractical. On the other hand, the developed approach represents a starting point for furthering the use of RL in indoor navigation applications while relying on limited sensor suites.

Appendix

A. ML-Agents configurations

Below the configurations of ML-Agents are listed, as they were used during the training runs highlighted in the Results section V. The basic configuration of ML-Agents is initially described.

1. Configuration structure

Below a training configuration used by ML-Agents is shown with comments describing key parameters. This file conforms to the YAML file standard. For further configuration options see the documentation [23].

```
# ML-Agents configuration.
behaviors:
  # ID of the agent.
  RollerAgent:
    # Common Trainer Configurations.
    # Algorithm to solve the RL problem.
    trainer_type: ppo
    # Number of checkpoints to keep.
    keep_checkpoints: 5
    # Number of performed steps.
    # One step equals a state-action pair.
    max_steps: 100e6
    time_horizon: 2000
    # Number of cumulated steps in ML-Agents.
    # For every 10000 steps one datapoint will be saved.
    summary_freq: 10000

    hyperparameters:
      # Common Trainer Configurations.
      batch_size: 356
      buffer_size: 51200
      learning_rate: 1e-3
      learning_rate_schedule: linear

      # PPO-specific Configurations.
      beta: 1e-2
      # Clipping factor.
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3

    # Configure the NN.
    network_settings:
      # Can be none, because no goal observation is used.
      conditioning_type: none
      normalize: false
      # Size of the hidden layers.
      hidden_units: 512
      # Number of hidden layers.
      num_layers: 1
      # LSTM settings.
      # Remember: sequence_len <= batch_size.
      memory:
        # memory_size // 2 = hidden_size.
        # Hidden size equals the size of the LSTM layer.
        # Dimension of the LSTM layer output.
        memory_size: 16
        # Minibatch size of the sequence.
        sequence_length: 4
    # Reward signal settings.
    reward_signals:
      extrinsic:
        # Discount factor.
        gamma: 0.99
        strength: 1.0
```

2. Environment 1 verification

The selected training configuration of training run 6483 as used in the verification of environment 1 in section V.A:

```
behaviors:
  RollerAgent:
    hyperparameters:
      batch_size: 356
      beta: 1e-2
      buffer_size: 51200
      epsilon: 0.2
      lambd: 0.95
      learning_rate: 1e-3
      learning_rate_schedule: linear
      num_epoch: 3
    keep_checkpoints: 5
    max_steps: 20e6
    network_settings:
      conditioning_type: none
      hidden_units: 512
      memory:
        memory_size: 128
        sequence_length: 64
      normalize: false
      num_layers: 1
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    summary_freq: 10000
    time_horizon: 2000
    trainer_type: ppo
```

3. Environment 2 verification

The selected training configuration of training run 6465 as used in the verification of environment 2 in Section V.B.

```
RollerAgent:
    hyperparameters:
      batch_size: 356
      beta: 1e-2
      buffer_size: 51200
      epsilon: 0.2
      lambd: 0.95
      learning_rate: 1e-3
      learning_rate_schedule: linear
      num_epoch: 3
    keep_checkpoints: 5
    max_steps: 100e6
    network_settings:
      conditioning_type: none
      hidden_units: 512
      memory:
        memory_size: 128
        sequence_length: 64
      normalize: false
      num_layers: 1
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    summary_freq: 10000
    time_horizon: 2000
    trainer_type: ppo
```

4. Sensor count study

This configuration was used for all sensor counts as lined out in Section V.C. The different sensor counts are not reflected in this configuration.

```
behaviors:
  RollerAgent:
    hyperparameters:
      batch_size: 356
      beta: 1e-2
      buffer_size: 51200
      epsilon: 0.2
      lambd: 0.95
      learning_rate: 1e-3
      learning_rate_schedule: linear
      num_epoch: 3
    keep_checkpoints: 5
    max_steps: 20e6
    network_settings:
      conditioning_type: none
      hidden_units: 512
      memory:
        memory_size: 128
        sequence_length: 64
      normalize: false
      num_layers: 1
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    summary_freq: 10000
    time_horizon: 2000
    trainer_type: ppo
```

Acknowledgements

I would like to express my gratitude to the Chair of Flight Guidance and Air Transport at Technische Universität Berlin for providing me the opportunity to write my thesis. I am also grateful for the support, assistance and access to hardware provided by my colleagues and friends at the Chair for Fluid System Dynamics at Technische Universität Berlin. I especially thank Christina Hofmann for all the support and help throughout the course of my thesis.

References

[1] Krátký, V., Petráček, P., Báča, T., and Saska, M., "An autonomous unmanned aerial vehicle system for fast exploration of large complex indoor environments," *Journal of field robotics*, Vol. 38, No. 8, 2021, pp. 1036–1058. https://doi.org/10.1002/rob.22021.

[2] Elfes, A., "Using occupancy grids for mobile robot perception and navigation," *Computer (Long Beach, Calif.)*, Vol. 22, No. 6, 1989, pp. 46–57. https://doi.org/10.1109/2.30720.

[3] Sutton, R. S., and Barto, A. G., *Reinforcement Learning*, 2nd ed., MIT Press, Cambridge, USA, 2018.

[4] Dhiman, V., Banerjee, S., Griffin, B., Siskind, J. M., and Corso, J. J., "A Critical Investigation of Deep Reinforcement Learning for Navigation," *arXiv e-prints*, 2019. https://doi.org/10.48550/arXiv.1802.02274.

[5] Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., Kumaran, D., and Hadsell, R., "Learning to Navigate in Complex Environments," *arXiv e-prints*, 2017. https://doi.org/10.48550/arXiv.1611.03673.

[6] Hochreiter, S., and Schmidhuber, J., "Long Short-Term Memory," *Neural Computation*, Vol. 9, No. 8, 1997, pp. 1735–1780.

[7] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., and Lange, D., "Unity: A general platform for intelligent agents," *arXiv e-prints*, 2020. https://doi.org/10.48550/arXiv.1809.02627.

[8] Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S., "DeepMind Lab," *arXiv e-prints*, 2016. https://doi.org/10.48550/arXiv.1612.03801.

[9] Wang, C., Wang, J., Wang, J., and Zhang, X., "Deep-Reinforcement-Learning-Based Autonomous UAV Navigation With Sparse Rewards," *IEEE Internet of Things Journal*, Vol. 7, No. 7, 2020, pp. 6180–6190. https://doi.org/10.1109/JIOT.2020.2973193.

[10] Youn, W., Ko, H., Choi, H., Choi, I., Baek, J.-H., and Myung, H., "Collision-free Autonomous Navigation of A Small UAV Using Low-cost Sensors in GPS-denied Environments," *International Journal of Control, Automation and Systems*, Vol. 19, 2021, pp. 953–968. https://doi.org/10.1007/s12555-019-0797-7.

[11] Hodge, V. J., Hawkins, R., and Alexander, R., "Deep reinforcement learning for drone navigation using sensor data," *Neural Computing and Applications*, 2021, pp. 2015–2033. https://doi.org/10.1007/s00521-020-05097-x.

[12] Goodfellow, I., Bengio, Y., and Courville, Y., *Deep Learning*, The MIT Press, 2016.

[13] Müller, B., Reinhardt, J., and Strickland, M. T., *Neural Networks: An Introduction*, 2nd ed., Springer, 1995.

[14] Aggarwal, C. C., *Neural Networks and Deep Learning*, Springer Cham, 2023. https://doi.org/10.1007/978-3-031-29642-0.

[15] Bishop, C. M., *Pattern recognition and machine learning*, Springer, New York, NY, 2006.

[16] Bronshtein, I. N., Muehlig, H., Musiol, G., and Semendyayev, K. A., *Handbook of Mathematics*, 5th ed., Springer, Berlin, Heidelberg, 2007.

[17] Unity Technologies, "ML-Agents," https://github.com/Unity-Technologies/ml-agents/tree/release_20_branch, 2021. Last visited 2024-07-01.

[18] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S., "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural Networks*, Vol. 6, No. 6, 1993, pp. 861–867. https://doi.org/10.1016/S0893-6080(05)80131-5.

[19] Graves, A., *Supervised Sequence Labelling with Recurrent Neural Networks*, Springer, 2012. https://doi.org/10.1007/978-3-642-24797-2.

[20] Elman, J. L., "Finding structure in time," *Cognitive Science*, Vol. 14, No. 2, 1990, pp. 179–211. https://doi.org/10.1016/0364-0213(90)90002-E.

[21] Gers, F. A., Schmidhuber, J., and Cummins, F., "Learning to Forget: Continual Prediction with LSTM," *Neural Computation*, Vol. 12, No. 10, 2000, pp. 2451–2471. https://doi.org/10.1162/089976600300015015.

[22] PyTorch Foundation, "PyTorch 1.7.1 documentation, LSTM," https://pytorch.org/docs/1.7.1/generated/torch.nn.LSTM.html, 2019. Last visited 2024-07-22.

[23] Unity Technologies, "ML-Agents Documentation," https://github.com/Unity-Technologies/ml-agents/tree/release_20_branch/docs, 2021. Last visited 2024-07-16.

[24] Koenig, S., and Simmons, R. G., "The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms," *Machine Learning*, Vol. 22, 1996, pp. 227–250.

[25] Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D., "Concrete Problems in AI Safety," *arXiv e-prints*, 2016. https://doi.org/10.48550/arXiv.1606.06565.

[26] Wiewiora, E., "Reward Shaping," *Encyclopedia of Machine Learning and Data Mining*, Springer US, Boston, MA, 2017, 2$^{\text{nd}}$ ed., pp. 1104–1106. https://doi.org/10.1007/978-1-4899-7687-1_966.

[27] Vasan, G., Wang, Y., Shahriar, F., Bergstra, J., Jagersand, M., and Mahmood, A. R., "Revisiting Sparse Rewards for Goal-Reaching Reinforcement Learning," *arXiv e-prints*, 2024. https://doi.org/10.48550/arXiv.2407.00324.

[28] Smart, W. D., and Pack Kaelbling, L., "Effective reinforcement learning for mobile robots," *Proceedings 2002 IEEE International Conference on Robotics and Automation*, Vol. 4, 2002, pp. 3404–3410. https://doi.org/10.1109/ROBOT.2002.1014237.

[29] Sowerby, H., Zhou, Z., and Littman, M. L., "Designing Rewards for Fast Learning," *arXiv e-prints*, 2022. https://doi.org/10.48550/arXiv.2204.11076.

[30] Botteghi, N., Sirmacek, B., Mustafa, K. A. A., Poel, M., and Stramigioli, S., "On Reward Shaping for Mobile Robot Navigation: A Reinforcement Learning and SLAM Based Approach," *arXiv e-prints*, 2002. https://doi.org/10.48550/arXiv.2002.04109.

[31] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *arXiv e-prints*, 2018. https://doi.org/10.48550/arXiv.1801.01290.

[32] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O., "Proximal Policy Optimization Algorithms," *arXiv e-prints*, 2017. https://doi.org/10.48550/arXiv.1707.06347.

[33] OpenAI, "Proximal Policy Optimization," https://spinningup.openai.com/en/latest/algorithms/ppo.html, 2018. Last visited 2024-07-09.

[34] Bitcraze AB, "Crazyflie 2.1," https://store.bitcraze.io/collections/kits/products/crazyflie-2-1, 2024. Last visited 2024-07-15.

[35] Benewake (Beijing) Co., Ltd., "TFmini-S12m LiDAR Ranging Module," https://en.benewake.com/TFminiS/index_proid_325.html, 2024. Last visited 2024-07-28.

[36] Mühlefeldt, M., "Autonomous, indoor operation of Small Unmanned Aircraft Systems using Reinforcement Learning and LSTM," https://github.com/muehlefeldt/unity-machine-learning, 2024.

[37] Unity Technologies, "Unity User Manual 2021.3 (LTS)," https://docs.unity3d.com/2021.3/Documentation/Manual/index.html, 2021. Last visited 2024-07-19.

[38] Cohen, A., Teng, E., Berges, V.-P., Dong, R.-P., Henry, H., Mattar, M., Zook, A., and Ganguly, S., "On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning," *arXiv e-prints*, 2022. https://doi.org/10.48550/arXiv.2111.05992.

[39] PyTorch Foundation, "PyTorch 1.7.1 documentation , torch.optim," https://pytorch.org/docs/1.7.1/optim.html, 2019. Last visited 2024-07-22.

[40] Sammut, C., "Behavioral Cloning," *Encyclopedia of Machine Learning and Data Mining*, Springer US, Boston, MA, 2017, 2$^{\text{nd}}$ ed., pp. 120–124. https://doi.org/10.1007/978-1-4899-7687-1_69.